PROPERTIES OF AGE-BASED AUTOMATIC MEMORY RECLAMATION ALGORITHMS

A Dissertation Presented

by

DARKO STEFANOVIĆ

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 1999

Department of Computer Science

© Copyright by Darko Stefanović 1999

All Rights Reserved

PROPERTIES OF AGE-BASED AUTOMATIC MEMORY RECLAMATION ALGORITHMS

A Dissertation Presented

by

DARKO STEFANOVIĆ

Approved as to style and content by:

J. Eliot B. Moss, Chair

Kathryn S. McKinley, Member

Jack C. Wileden, Member

C. Mani Krishna, Member

James F. Kurose, Department Chair
Department of Computer Science

To my parents

ACKNOWLEDGEMENTS

Throughout my graduate studies, it has been an honor and an intellectual challenge to have Eliot Moss as my advisor. The imprint of his scientific method on my development is deep, and I trust that it is reflected in the present work.

I have been fortunate to have Kathryn McKinley as my unofficial second advisor. I am grateful to her, not only for her incisive technical commentary, but also for her unflagging enthusiasm and encouragement.

I owe a debt of gratitude to Steve Heller and his group at Sun Microsystems Laboratories for allowing me to use an experimental Java virtual machine, and especially to David Detlefs, who tirelessly explained to me the intricacies of garbage collection in that system, and ran many weeks of trace generation.

This project could not have been realized without the past efforts of members of the Object Systems Laboratory: Tony Hosking, Amer Diwan, Rick Hudson, and Eric Brown, who designed and implemented the garbage collector toolkit and the Smalltalk system I have used. I should also like to acknowledge the support of the current group members: John Cavazos, Asjad Khan, Todd Wright, Abhishek Chandra, and Sharad Singhai; their hard system-building work provided the impetus to carry on with this study. John Ridgway has always found the time to guide me through the hidden corners of typesetting, and I thank him for that. It has been a great pleasure to work with them, and with many other members of the laboratory over the years.

I should like to acknowledge the benefit of discussions I have had during the development of the present work with Cathy McGeoch, Andrew Appel, and Amer Diwan. The dissertation itself has been greatly improved by the comments of my examining committee members Jack Wileden and Mani Krishna.

Upon my arrival in the town of Amherst I had the good fortune to come to know Mrs Joann C. Scott, and I am grateful for the friendship and wisdom she has shown me ever since.

Lastly, I am grateful to my parents and my sister for their persistent encouragement and unquestioning confidence. I thank my parents for their support throughout the years of my studies, which made this work possible, and for having instilled in me the spirit of free inquiry, which made it worthwhile.

———

This work has been supported by the National Science Foundation under grant IRI-9632284, and by gifts from Digital Equipment Corporation (now Compaq), Sun Microsystems, and Hewlett-Packard.

ABSTRACT

PROPERTIES OF AGE-BASED AUTOMATIC MEMORY RECLAMATION ALGORITHMS

FEBRUARY 1999

DARKO STEFANOVIĆ

Dipl.Ing., UNIVERSITY OF BELGRADE

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor J. Eliot B. Moss

Dynamic memory management enables a programmer to allocate objects for arbitrary preiods of time. It is an important feature of modern programming languages, and is fundamental to object-oriented languages. Automatic reclamation, also known as garbage collection, automates the detection of the time when a data item can no longer be used.

The work described herein considers garbage collection algorithms that base their decisions solely upon the relative age of data. This age-based class of algorithms generalizes previously defined generational garbage collection algorithms and includes promising new algorithms. The work identifies relevant performance factors and reports them for a set of object-oriented benchmark programs, establishing a fair comparison by imposing uniform maximum space constraints. A precise tracing and garbage collection algorithm evaluation framework provides accurate results and thus meaningful comparisons.

The results indicate, contrary to assumptions in the literature, that the new algorithms copy less data than the generational algorithms, even though they retain a higher percentage of reclaimable data. In agreement with the assumptions in the literature, the results indicate that generational algorithms do less pointer-tracking work. Thus, given suitable relative costs of copying and pointer-tracking, the new algorithms perform better. Estimations of the relative costs for a typical modern processor suggest that the new algorithms are usually superior.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS . . . . . **v**

ABSTRACT . . . . . **vii**

LIST OF TABLES . . . . . **xiii**

LIST OF FIGURES . . . . . **xv**

Chapter

1. INTRODUCTION **1**

2. BACKGROUND **7**

2.1 Dynamic memory management 7

2.1.1 Liveness and reachability 8
2.1.2 Preserving live data 10
2.1.3 Incremental collection 11
2.1.4 Generational copying collection 11

2.1.4.1 Pointer maintenance 13

2.1.5 Mature spaces 14

2.2 Object survival and mortality 15
2.3 Theoretical models of memory management 17

3. AGE-BASED GARBAGE COLLECTION ALGORITHMS **19**

3.1 Conceptual design 19

3.1.1 True-age schemes 21
3.1.2 A renewal-age scheme 29
3.1.3 Towards optimal schemes 30

3.1.4 Collection failure and recovery . . . 31
3.1.5 Garbage cycles . . . 31
3.1.6 Locality . . . 32

3.2 Implementation . . . 33

3.2.1 Blocks . . . 33
3.2.2 Remembered sets . . . 35
3.2.3 Write barrier . . . 38
3.2.4 A write barrier design for large address spaces . . . 39
3.2.5 An estimation of copying and pointer-maintenance costs . . . 43

3.3 Summary . . . 46

4. EVALUATION: SETTING . . . 52

4.1 Evaluation using program traces . . . 53

4.1.1 Trace content . . . 54
4.1.2 Trace format . . . 54
4.1.3 Trace generation . . . 55

4.2 Benchmarks and languages . . . 57

4.2.1 Languages . . . 57
4.2.2 Benchmarks . . . 58

4.2.2.1 Collecting benchmarks . . . 59
4.2.2.2 Selecting benchmarks . . . 60
4.2.2.3 The final benchmark set . . . 61

4.2.3 Properties of benchmarks . . . 63

4.3 Simulating garbage collection . . . 66

4.3.1 General simulation algorithm . . . 68
4.3.2 Age-based simulation . . . 69
4.3.3 Block-based simulation . . . 69

5. EVALUATION: COPYING COST . . . 112

5.1 Method for evaluating different configurations of a collector . . . 112

5.1.1 Results . . . 113

5.1.1.1 The FC-TOF collector . . . 114
5.1.1.2 The FC-ROF collector . . . 115

5.1.1.3 The FC-TYF collector . . . 115
5.1.1.4 The FC-DOF collector . . . 115
5.1.1.5 The FC-DYF collector . . . 116
5.1.1.6 The GYF collectors . . . 116

5.2 Method for comparing various collection schemes . . . 117
5.3 Method for recognizing and measuring the excess retention cost . . . 118

5.3.1 Results . . . 119

5.4 Methods for examining the lower limits of copying cost and the feasibility of adaptation . . . 121

5.4.1 Locally-optimal copying cost . . . 121
5.4.2 Window motion . . . 122
5.4.3 Demise point analysis . . . 122

5.4.3.1 Visualization: demise maps . . . 123
5.4.3.2 Visualization: heap profiles . . . 124
5.4.3.3 Visualization: position mortality . . . 125

5.4.4 Case studies . . . 126

5.4.4.1 Benchmark Richards . . . 126
5.4.4.2 Benchmark Lambda-Fact5 . . . 127

5.4.5 Adaptive FC collection . . . 128

5.4.5.1 Eliminating the "static" data in Richards . . . 128
5.4.5.2 Choosing the collected region in Lambda-Fact5 . . . 131

5.5 Summary . . . 136

6. EVALUATION: COSTS OTHER THAN COPYING . . . 276

6.1 Method for evaluating the pointer maintenance cost . . . 276

6.1.1 Results . . . 277
6.1.2 Block size . . . 280
6.1.3 Comparison between collectors . . . 281
6.1.4 Additional observations . . . 286

6.2 Methods for examining the directions of pointers in the heap . . . 288

6.2.1 Pointer stores in the ideal heap . . . 288
6.2.2 Pointer stores and remembered sets in age-based collectors . . . 290

6.3 Method for evaluating the collection start-up cost 291

6.3.1 Results 293

6.4 Summary 293

7. CONCLUDING REMARKS 460

BIBLIOGRAPHY 463

LIST OF TABLES

Table		Page
4.1	Properties of benchmarks used	62
5.1	Copying cost dependence on the adaptation threshold	130
5.2	Copying cost dependence on external bias	134
6.1	Properties of benchmarks used: run-time allocation and pointer stores	278
6.2	Costs of block FC-DOF for StandardNonInteractive	295
6.3	Costs of block FC-DOF for HeapSim	301
6.4	Costs of block FC-DOF for Lambda-Fact5	310
6.5	Costs of block FC-DOF for Richards	322
6.6	Costs of block FC-DOF for JavaBYTEmark	326
6.7	Costs of block FC-DOF for Bloat-Bloat	327
6.8	Costs of block FC-DOF for Toba	331
6.9	Costs of block 2GYF for StandardNonInteractive	335
6.10	Costs of block 2GYF for HeapSim	343
6.11	Costs of block 2GYF for Lambda-Fact5	354
6.12	Costs of block 2GYF for Richards	370
6.13	Costs of block 2GYF for JavaBYTEmark	375
6.14	Costs of block 2GYF for Bloat-Bloat	376

6.15 Costs of block 2GYF for Toba 382

LIST OF FIGURES

Figure **Page**

1.1 Surviving objects in a newly allocated area. 2

1.2 A collection queue. 3

2.1 A typical object mortality function. 16

2.2 A typical object survivor function. 16

3.1 Viewing the heap as an age-ordered list. 20

3.2 Pseudocode for the conceptual design. 22

3.3 TYF: True youngest-first collection (general case). 23

3.4 Pseudocode for GYF collectors. 24

3.5 Generational youngest-first collection (2GYF). 25

3.6 Generational youngest-first collection with flexible generation boundaries (NGYF). 26

3.7 FC-TYF: True youngest-first collection (fixed collected region). 27

3.8 FC-TOF: True oldest-first collection (fixed collected region). 27

3.9 FC-DOF: Deferred older-first collection. 28

3.10 Deferred older-first collection, window motion example. 29

3.11 FC-DYF: Deferred younger-first collection. 30

3.12 FC-ROF: Renewal-oldest-first collection. 30

3.13 Heap organized as an age-ordered list of blocks. 33

3.14 Filling preceding block to avoid fragmentation. 35

3.15 Directional filtering of pointer stores at run-time (DOF). 36

3.16 Directional filtering of pointer stores at collection time (DOF). 37

3.17 DOF heap organization in a large address space. 40

3.18 Directional filtering for DOF in a large address space. 42

3.19 Write barrier code. 47

3.20 Write barrier assembly code. 48

3.21 Remembered set processing code. 49

3.22 Remembered set processing assembly code. 50

3.23 Age-based scheme name abbreviations. 51

4.1 Non-generational collection performance: Interactive-AllCallsOn. 71

4.2 Non-generational collection performance: Interactive-TextEditing. 72

4.3 Non-generational collection performance: StandardNonInteractive. 73

4.4 Non-generational collection performance: HeapSim. 74

4.5 Non-generational collection performance: Lambda-Fact5. 75

4.6 Non-generational collection performance: Lambda-Fact6. 76

4.7 Non-generational collection performance: Swim. 77

4.8 Non-generational collection performance: Tomcatv. 78

4.9 Non-generational collection performance: Tree-Replace-Binary. 79

4.10 Non-generational collection performance: Tree-Replace-Random. 80

4.11 Non-generational collection performance: Richards. 81

4.12 Non-generational collection performance: JavaBYTEmark. 82

4.13 Non-generational collection performance: Bloat-Bloat. 83
4.14 Non-generational collection performance: Toba. 84
4.15 Object size distribution: Interactive-AllCallsOn. 85
4.16 Object size distribution: Interactive-TextEditing. 86
4.17 Object size distribution: StandardNonInteractive. 87
4.18 Object size distribution: HeapSim. 88
4.19 Object size distribution: Lambda-Fact5. 89
4.20 Object size distribution: Lambda-Fact6. 90
4.21 Object size distribution: Swim. 91
4.22 Object size distribution: Tomcatv. 92
4.23 Object size distribution: Tree-Replace-Binary. 93
4.24 Object size distribution: Tree-Replace-Random. 94
4.25 Object size distribution: Richards. 95
4.26 Object size distribution: JavaBYTEmark. 96
4.27 Object size distribution: Bloat-Bloat. 97
4.28 Object size distribution: Toba. 98
4.29 Object lifetime distribution: Interactive-AllCallsOn. 99
4.30 Object lifetime distribution: Interactive-TextEditing. 99
4.31 Object lifetime distribution: StandardNonInteractive. 100
4.32 Object lifetime distribution: HeapSim. 100
4.33 Object lifetime distribution: Lambda-Fact5. 100
4.34 Object lifetime distribution: Lambda-Fact6. 101
4.35 Object lifetime distribution: Swim. 101

4.36 Object lifetime distribution: Tomcatv. 101

4.37 Object lifetime distribution: Tree-Replace-Binary. 102

4.38 Object lifetime distribution: Tree-Replace-Random. 102

4.39 Object lifetime distribution: Richards. 102

4.40 Object lifetime distribution: JavaBYTEmark. 103

4.41 Object lifetime distribution: Bloat-Bloat. 103

4.42 Object lifetime distribution: Toba. 103

4.43 Live profile: Interactive-AllCallsOn. 104

4.44 Live profile: Interactive-TextEditing. 105

4.45 Live profile: StandardNonInteractive. 105

4.46 Live profile: HeapSim. 106

4.47 Live profile: Lambda-Fact5. 106

4.48 Live profile: Lambda-Fact6. 107

4.49 Live profile: Swim. 107

4.50 Live profile: Tomcatv. 108

4.51 Live profile: Tree-Replace-Binary. 108

4.52 Live profile: Tree-Replace-Random. 109

4.53 Live profile: Richards. 109

4.54 Live profile: JavaBYTEmark. 110

4.55 Live profile: Bloat-Bloat. 110

4.56 Live profile: Toba. 111

5.1 Window motion in the FC-DOF scheme. 132

5.2 Window survivor ratios in the FC-DOF scheme. 133

5.3 Window motion in the FC-OPT scheme. 134

5.4 Window survivor ratios in the FC-OPT scheme. 135

5.5 Window motion in the adaptive scheme. 136

5.6 Window survivor ratios in the adaptive scheme. 137

5.7 Copying cost comparison: Interactive-AllCallsOn, $V = 764$. 138

5.8 Copying cost comparison: Interactive-AllCallsOn, $V = 931$. 138

5.9 Copying cost comparison: Interactive-AllCallsOn, $V = 1135$. 139

5.10 Copying cost comparison: Interactive-AllCallsOn, $V = 1384$. 139

5.11 Copying cost comparison: Interactive-AllCallsOn, $V = 1687$. 139

5.12 Copying cost comparison: Interactive-AllCallsOn, $V = 2057$. 140

5.13 Copying cost comparison: Interactive-TextEditing, $V = 1338$. 141

5.14 Copying cost comparison: Interactive-TextEditing, $V = 1631$. 141

5.15 Copying cost comparison: Interactive-TextEditing, $V = 1988$. 142

5.16 Copying cost comparison: Interactive-TextEditing, $V = 2424$. 142

5.17 Copying cost comparison: Interactive-TextEditing, $V = 2955$. 142

5.18 Copying cost comparison: StandardNonInteractive, $V = 1414$. 143

5.19 Copying cost comparison: StandardNonInteractive, $V = 1723$. 143

5.20 Copying cost comparison: StandardNonInteractive, $V = 2101$. 144

5.21 Copying cost comparison: StandardNonInteractive, $V = 2561$. 144

5.22 Copying cost comparison: StandardNonInteractive, $V = 3122$. 144

5.23 Copying cost comparison: StandardNonInteractive, $V = 3805$. 145

5.24 Copying cost comparison: StandardNonInteractive, $V = 4639$. 145

5.25 Copying cost comparison: StandardNonInteractive, $V = 5655$. 145

5.26 Copying cost comparison: StandardNonInteractive, $V = 6894$. 146

5.27 Copying cost comparison: HeapSim, $V = 113397$. 147

5.28 Copying cost comparison: HeapSim, $V = 138231$. 147

5.29 Copying cost comparison: HeapSim, $V = 168503$. 148

5.30 Copying cost comparison: HeapSim, $V = 205405$. 148

5.31 Copying cost comparison: HeapSim, $V = 250387$. 148

5.32 Copying cost comparison: HeapSim, $V = 305221$. 149

5.33 Copying cost comparison: HeapSim, $V = 372063$. 149

5.34 Copying cost comparison: Lambda-Fact5, $V = 7673$. 150

5.35 Copying cost comparison: Lambda-Fact5, $V = 9354$. 150

5.36 Copying cost comparison: Lambda-Fact5, $V = 11402$. 151

5.37 Copying cost comparison: Lambda-Fact5, $V = 13899$. 151

5.38 Copying cost comparison: Lambda-Fact5, $V = 16943$. 151

5.39 Copying cost comparison: Lambda-Fact5, $V = 20654$. 152

5.40 Copying cost comparison: Lambda-Fact5, $V = 25177$. 152

5.41 Copying cost comparison: Lambda-Fact6, $V = 16669$. 153

5.42 Copying cost comparison: Lambda-Fact6, $V = 20320$. 153

5.43 Copying cost comparison: Lambda-Fact6, $V = 24770$. 154

5.44 Copying cost comparison: Lambda-Fact6, $V = 30194$. 154

5.45 Copying cost comparison: Lambda-Fact6, $V = 36807$. 154

5.46 Copying cost comparison: Lambda-Fact6, $V = 44868$. 155

5.47 Copying cost comparison: Lambda-Fact6, $V = 54693$. 155

5.48 Copying cost comparison: Lambda-Fact6, $V = 66671$. 155

5.49 Copying cost comparison: Lambda-Fact6, $V = 81272$. 156

5.50 Copying cost comparison: Swim, $V = 21383$. 157

5.51 Copying cost comparison: Swim, $V = 26066$. 157

5.52 Copying cost comparison: Swim, $V = 31774$. 158

5.53 Copying cost comparison: Swim, $V = 38733$. 158

5.54 Copying cost comparison: Swim, $V = 47215$. 158

5.55 Copying cost comparison: Swim, $V = 57555$. 159

5.56 Copying cost comparison: Swim, $V = 70160$. 159

5.57 Copying cost comparison: Swim, $V = 85524$. 159

5.58 Copying cost comparison: Swim, $V = 104254$. 160

5.59 Copying cost comparison: Tomcatv, $V = 37292$. 161

5.60 Copying cost comparison: Tomcatv, $V = 45459$. 161

5.61 Copying cost comparison: Tomcatv, $V = 55414$. 162

5.62 Copying cost comparison: Tomcatv, $V = 67550$. 162

5.63 Copying cost comparison: Tomcatv, $V = 82343$. 162

5.64 Copying cost comparison: Tomcatv, $V = 100376$. 163

5.65 Copying cost comparison: Tomcatv, $V = 122358$. 163

5.66 Copying cost comparison: Tomcatv, $V = 149154$. 163

5.67 Copying cost comparison: Tomcatv, $V = 181818$. 164

5.68 Copying cost comparison: Tree-Replace-Binary, $V = 11926$. 165

5.69 Copying cost comparison: Tree-Replace-Binary, $V = 14538$. 165

5.70 Copying cost comparison: Tree-Replace-Binary, $V = 17722$. 166

5.71 Copying cost comparison: Tree-Replace-Binary, $V = 21603$. 166

5.72 Copying cost comparison: Tree-Replace-Binary, $V = 26334$. 166

5.73 Copying cost comparison: Tree-Replace-Random, $V = 15985$. 167

5.74 Copying cost comparison: Tree-Replace-Random, $V = 19486$. 167

5.75 Copying cost comparison: Tree-Replace-Random, $V = 23754$. 168

5.76 Copying cost comparison: Tree-Replace-Random, $V = 28956$. 168

5.77 Copying cost comparison: Tree-Replace-Random, $V = 35297$. 168

5.78 Copying cost comparison: Tree-Replace-Random, $V = 43027$. 169

5.79 Copying cost comparison: Tree-Replace-Random, $V = 52450$. 169

5.80 Copying cost comparison: Tree-Replace-Random, $V = 63936$. 169

5.81 Copying cost comparison: Richards, $V = 1826$. 170

5.82 Copying cost comparison: Richards, $V = 2225$. 170

5.83 Copying cost comparison: Richards, $V = 2713$. 171

5.84 Copying cost comparison: Richards, $V = 3307$. 171

5.85 Copying cost comparison: Richards, $V = 4032$. 171

5.86 Copying cost comparison: Richards, $V = 4914$. 172

5.87 Copying cost comparison: Richards, $V = 5991$. 172

5.88 Copying cost comparison: Richards, $V = 7303$. 172

5.89 Copying cost comparison: Richards, $V = 8902$. 173

5.90 Copying cost comparison: JavaBYTEmark, $V = 72807$. 174

5.91 Copying cost comparison: JavaBYTEmark, $V = 88752$. 174

5.92 Copying cost comparison: JavaBYTEmark, $V = 108188$. 175

5.93 Copying cost comparison: JavaBYTEmark, $V = 131881$. 175

5.94 Copying cost comparison: Bloat-Bloat, $V = 246766$. 176

5.95 Copying cost comparison: Bloat-Bloat, $V = 300808$. 176

5.96 Copying cost comparison: Bloat-Bloat, $V = 366682$. 177

5.97 Copying cost comparison: Bloat-Bloat, $V = 446984$. 177

5.98 Copying cost comparison: Bloat-Bloat, $V = 544872$. 177

5.99 Copying cost comparison: Bloat-Bloat, $V = 664195$. 178

5.100 Copying cost comparison: Bloat-Bloat, $V = 809650$. 178

5.101 Copying cost comparison: Bloat-Bloat, $V = 986959$. 178

5.102 Copying cost comparison: Toba, $V = 353843$. 179

5.103 Copying cost comparison: Toba, $V = 431335$. 179

5.104 Copying cost comparison: Toba, $V = 525794$. 180

5.105 Copying cost comparison: Toba, $V = 640941$. 180

5.106 Copying cost comparison: Toba, $V = 781303$. 180

5.107 Copying cost comparison: Toba, $V = 952404$. 181

5.108 Copying cost comparison: Toba, $V = 1160976$. 181

5.109 Copying cost comparison: Toba, $V = 1415223$. 181

5.110 Copying cost comparison: Toba, $V = 1725148$. 182

5.111 Best configuration comparison: Interactive-AllCallsOn. 183

5.112 Best configuration comparison: Interactive-TextEditing. 184

5.113 Best configuration comparison: StandardNonInteractive. 184

5.114 Best configuration comparison: HeapSim. 185

5.115 Best configuration comparison: Lambda-Fact5. 185

5.116 Best configuration comparison: Lambda-Fact6. 186

5.117 Best configuration comparison: Swim. 186

5.118 Best configuration comparison: Tomcatv. 187

5.119 Best configuration comparison: Tree-Replace-Binary. 187

5.120 Best configuration comparison: Tree-Replace-Random. 188

5.121 Best configuration comparison: Richards. 188

5.122 Best configuration comparison: JavaBYTEmark. 189

5.123 Best configuration comparison: Bloat-Bloat. 189

5.124 Best configuration comparison: Toba. 190

5.125 Best configuration comparison: Interactive-AllCallsOn. 191

5.126 Best configuration comparison: Interactive-TextEditing. 192

5.127 Best configuration comparison: StandardNonInteractive. 192

5.128 Best configuration comparison: HeapSim. 193

5.129 Best configuration comparison: Lambda-Fact5. 193

5.130 Best configuration comparison: Lambda-Fact6. 194

5.131 Best configuration comparison: Swim. 194

5.132 Best configuration comparison: Tomcatv. 195

5.133 Best configuration comparison: Tree-Replace-Binary. 195

5.134 Best configuration comparison: Tree-Replace-Random. 196

5.135 Best configuration comparison: Richards. 196

5.136 Best configuration comparison: JavaBYTEmark. 197

5.137 Best configuration comparison: Bloat-Bloat. 197

5.138 Best configuration comparison: Toba. 198

5.139 Excess retention ratios: Interactive-AllCallsOn, $V = 764$. 199

5.140 Excess retention ratios: Interactive-AllCallsOn, $V = 931$. 199

5.141 Excess retention ratios: Interactive-AllCallsOn, $V = 1135$. 200

5.142 Excess retention ratios: Interactive-AllCallsOn, $V = 1384$. 200

5.143 Excess retention ratios: Interactive-AllCallsOn, $V = 1687$. 200

5.144 Excess retention ratios: Interactive-AllCallsOn, $V = 2057$. 201

5.145 Excess retention ratios: Interactive-TextEditing, $V = 1338$. 202

5.146 Excess retention ratios: Interactive-TextEditing, $V = 1631$. 202

5.147 Excess retention ratios: Interactive-TextEditing, $V = 1988$. 203

5.148 Excess retention ratios: Interactive-TextEditing, $V = 2424$. 203

5.149 Excess retention ratios: Interactive-TextEditing, $V = 2955$. 203

5.150 Excess retention ratios: StandardNonInteractive, $V = 1414$. 204

5.151 Excess retention ratios: StandardNonInteractive, $V = 1723$. 204

5.152 Excess retention ratios: StandardNonInteractive, $V = 2101$. 205

5.153 Excess retention ratios: StandardNonInteractive, $V = 2561$. 205

5.154 Excess retention ratios: StandardNonInteractive, $V = 3122$. 205

5.155 Excess retention ratios: StandardNonInteractive, $V = 3805$. 206

5.156 Excess retention ratios: StandardNonInteractive, $V = 4639$. 206

5.157 Excess retention ratios: StandardNonInteractive, $V = 5655$. 206

5.158 Excess retention ratios: StandardNonInteractive, $V = 6894$. 207

5.159 Excess retention ratios: HeapSim, $V = 113397$. 208

5.160 Excess retention ratios: HeapSim, $V = 138231$. 208

5.161 Excess retention ratios: HeapSim, $V = 168503$. 209

5.162 Excess retention ratios: HeapSim, $V = 205405$. 209

5.163 Excess retention ratios: HeapSim, $V = 250387$. 209

5.164 Excess retention ratios: HeapSim, $V = 305221$. 210

5.165 Excess retention ratios: HeapSim, $V = 372063$. 210

5.166 Excess retention ratios: Lambda-Fact5, $V = 7673$. 211

5.167 Excess retention ratios: Lambda-Fact5, $V = 9354$. 211

5.168 Excess retention ratios: Lambda-Fact5, $V = 11402$. 212

5.169 Excess retention ratios: Lambda-Fact5, $V = 13899$. 212

5.170 Excess retention ratios: Lambda-Fact5, $V = 16943$. 212

5.171 Excess retention ratios: Lambda-Fact5, $V = 20654$. 213

5.172 Excess retention ratios: Lambda-Fact5, $V = 25177$. 213

5.173 Excess retention ratios: Lambda-Fact6, $V = 16669$. 214

5.174 Excess retention ratios: Lambda-Fact6, $V = 20320$. 214

5.175 Excess retention ratios: Lambda-Fact6, $V = 24770$. 215

5.176 Excess retention ratios: Lambda-Fact6, $V = 30194$. 215

5.177 Excess retention ratios: Lambda-Fact6, $V = 36807$. 215

5.178 Excess retention ratios: Lambda-Fact6, $V = 44868$. 216

5.179 Excess retention ratios: Lambda-Fact6, $V = 54693$. 216

5.180 Excess retention ratios: Lambda-Fact6, $V = 66671$. 216

5.181 Excess retention ratios: Lambda-Fact6, $V = 81272$. 217

5.182 Excess retention ratios: Swim, $V = 21383$. 218

5.183 Excess retention ratios: Swim, $V = 26066$. 218

5.184 Excess retention ratios: Swim, $V = 31774$. 219

5.185 Excess retention ratios: Swim, $V = 38733$. 219

5.186 Excess retention ratios: Swim, $V = 47215$. 219

5.187 Excess retention ratios: Swim, $V = 57555$. 220

5.188 Excess retention ratios: Swim, $V = 70160$. 220

5.189 Excess retention ratios: Swim, $V = 85524$. 220

5.190 Excess retention ratios: Swim, $V = 104254$. 221

5.191 Excess retention ratios: Tomcatv, $V = 37292$. 222

5.192 Excess retention ratios: Tomcatv, $V = 45459$. 222

5.193 Excess retention ratios: Tomcatv, $V = 55414$. 223

5.194 Excess retention ratios: Tomcatv, $V = 67550$. 223

5.195 Excess retention ratios: Tomcatv, $V = 82343$. 223

5.196 Excess retention ratios: Tomcatv, $V = 100376$. 224

5.197 Excess retention ratios: Tomcatv, $V = 122358$. 224

5.198 Excess retention ratios: Tomcatv, $V = 149154$. 224

5.199 Excess retention ratios: Tomcatv, $V = 181818$. 225

5.200 Excess retention ratios: Tree-Replace-Binary, $V = 11926$. 226

5.201 Excess retention ratios: Tree-Replace-Binary, $V = 14538$. 226

5.202 Excess retention ratios: Tree-Replace-Binary, $V = 17722$. 227

5.203 Excess retention ratios: Tree-Replace-Binary, $V = 21603$. 227

5.204 Excess retention ratios: Tree-Replace-Binary, $V = 26334$. 227

5.205 Excess retention ratios: Tree-Replace-Random, $V = 15985$. 228

5.206 Excess retention ratios: Tree-Replace-Random, $V = 19486$. 228

5.207 Excess retention ratios: Tree-Replace-Random, $V = 23754$. 229

5.208 Excess retention ratios: Tree-Replace-Random, $V = 28956$. 229

5.209 Excess retention ratios: Tree-Replace-Random, $V = 35297$. 229

5.210 Excess retention ratios: Tree-Replace-Random, $V = 43027$. 230

5.211 Excess retention ratios: Tree-Replace-Random, $V = 52450$. 230

5.212 Excess retention ratios: Tree-Replace-Random, $V = 63936$. 230

5.213 Excess retention ratios: Richards, $V = 1826$. 231

5.214 Excess retention ratios: Richards, $V = 2225$. 231

5.215 Excess retention ratios: Richards, $V = 2713$. 232

5.216 Excess retention ratios: Richards, $V = 3307$. 232

5.217 Excess retention ratios: Richards, $V = 4032$. 232

5.218 Excess retention ratios: Richards, $V = 4914$. 233

5.219 Excess retention ratios: Richards, $V = 5991$. 233

5.220 Excess retention ratios: Richards, $V = 7303$. 233

5.221 Excess retention ratios: Richards, $V = 8902$. 234

5.222 Excess retention ratios: JavaBYTEmark, $V = 72807$. 235

5.223 Excess retention ratios: JavaBYTEmark, $V = 88752$. 235

5.224 Excess retention ratios: JavaBYTEmark, $V = 108188$. 236

5.225 Excess retention ratios: JavaBYTEmark, $V = 131881$. 236

5.226 Excess retention ratios: Bloat-Bloat, $V = 246766$. 237

5.227 Excess retention ratios: Bloat-Bloat, $V = 300808$. 237

5.228 Excess retention ratios: Bloat-Bloat, $V = 366682$. 238

5.229 Excess retention ratios: Bloat-Bloat, $V = 446984$. 238

5.230 Excess retention ratios: Bloat-Bloat, $V = 544872$. 238

5.231 Excess retention ratios: Bloat-Bloat, $V = 664195$. 239

5.232 Excess retention ratios: Bloat-Bloat, $V = 809650$. 239

5.233 Excess retention ratios: Bloat-Bloat, $V = 986959$. 239

5.234 Excess retention ratios: Toba, $V = 353843$. 240

5.235 Excess retention ratios: Toba, $V = 431335$. 240

5.236 Excess retention ratios: Toba, $V = 525794$. 241

5.237 Excess retention ratios: Toba, $V = 640941$. 241

5.238 Excess retention ratios: Toba, $V = 781303$. 241

5.239 Excess retention ratios: Toba, $V = 952404$. 242

5.240 Excess retention ratios: Toba, $V = 1160976$. 242

5.241 Excess retention ratios: Toba, $V = 1415223$. 242

5.242 Excess retention ratios: Toba, $V = 1725148$. 243

5.243 Copying cost, with FC-OPT: StandardNonInteractive, $V = 1414$. 244

5.244 Copying cost, with FC-OPT: StandardNonInteractive, $V = 1723$. 244

5.245 Copying cost, with FC-OPT: StandardNonInteractive, $V = 2101$. 245

5.246 Copying cost, with FC-OPT: StandardNonInteractive, $V = 2561$. 245

5.247 Copying cost, with FC-OPT: StandardNonInteractive, $V = 3122$. 245

5.248 Copying cost, with FC-OPT: StandardNonInteractive, $V = 3805$. 246

5.249 Copying cost, with FC-OPT: StandardNonInteractive, $V = 4639$. 246

5.250 Copying cost, with FC-OPT: StandardNonInteractive, $V = 5655$. 246

5.251 Copying cost, with FC-OPT: StandardNonInteractive, $V = 6894$. 247

5.252 Copying cost, with FC-OPT: Lambda-Fact5, $V = 7673$. 248

5.253 Copying cost, with FC-OPT: Lambda-Fact5, $V = 9354$. 248

5.254 Copying cost, with FC-OPT: Lambda-Fact5, $V = 11402$. 249

5.255 Copying cost, with FC-OPT: Lambda-Fact5, $V = 13899$. 249

5.256 Copying cost, with FC-OPT: Lambda-Fact5, $V = 16943$. 249

5.257 Copying cost, with FC-OPT: Lambda-Fact5, $V = 20654$. 250

5.258 Copying cost, with FC-OPT: Lambda-Fact5, $V = 25177$. 250

5.259 Copying cost, with FC-OPT: Tree-Replace-Random, $V = 15985$. 251

5.260 Copying cost, with FC-OPT: Tree-Replace-Random, $V = 19486$. 251

5.261 Copying cost, with FC-OPT: Tree-Replace-Random, $V = 23754$. 252

5.262 Copying cost, with FC-OPT: Tree-Replace-Random, $V = 28956$. 252

5.263 Copying cost, with FC-OPT: Tree-Replace-Random, $V = 35297$. 252

5.264 Copying cost, with FC-OPT: Tree-Replace-Random, $V = 43027$. 253

5.265 Copying cost, with FC-OPT: Tree-Replace-Random, $V = 52450$. 253

5.266 Copying cost, with FC-OPT: Tree-Replace-Random, $V = 63936$. 253

5.267 Demise graphs: Interactive-AllCallsOn. 254

5.268 Demise graphs: Interactive-TextEditing. 255

5.269 Demise graphs: StandardNonInteractive. 256

5.270 Demise graphs: HeapSim. 257

5.271 Demise graphs: Lambda-Fact5. 258

5.272 Demise graphs: Lambda-Fact6. 259

5.273 Demise graphs: Swim. 260

5.274 Demise graphs: Tomcatv. 261

5.275 Demise graphs: Tree-Replace-Binary. 262

5.276 Demise graphs: Tree-Replace-Random. 263

5.277 Demise graphs: Richards. 264

5.278 Demise graphs: JavaBYTEmark. 265
5.279 Demise graphs: Bloat-Bloat. 266
5.280 Demise graphs: Toba. 267
5.281 Demise position as mortality: Interactive-AllCallsOn. 268
5.282 Demise position as mortality: Interactive-TextEditing. 269
5.283 Demise position as mortality: StandardNonInteractive. 269
5.284 Demise position as mortality: HeapSim. 270
5.285 Demise position as mortality: Lambda-Fact5. 270
5.286 Demise position as mortality: Lambda-Fact6. 271
5.287 Demise position as mortality: Swim. 271
5.288 Demise position as mortality: Tomcatv. 272
5.289 Demise position as mortality: Tree-Replace-Binary. 272
5.290 Demise position as mortality: Tree-Replace-Random. 273
5.291 Demise position as mortality: Richards. 273
5.292 Demise position as mortality: JavaBYTEmark. 274
5.293 Demise position as mortality: Bloat-Bloat. 274
5.294 Demise position as mortality: Toba. 275

6.1 Copying and pointer cost tradeoff: Interactive-AllCallsOn. 389
6.2 Copying and pointer cost tradeoff: Interactive-TextEditing. 390
6.3 Copying and pointer cost tradeoff: StandardNonInteractive. 390
6.4 Copying and pointer cost tradeoff: HeapSim. 391
6.5 Copying and pointer cost tradeoff: Lambda-Fact5. 391

6.6 Copying and pointer cost tradeoff: Lambda-Fact6. 392

6.7 Copying and pointer cost tradeoff: Swim. 392

6.8 Copying and pointer cost tradeoff: Tomcatv. 393

6.9 Copying and pointer cost tradeoff: Tree-Replace-Binary. 393

6.10 Copying and pointer cost tradeoff: Bloat-Bloat. 394

6.11 Copying and pointer cost tradeoff: Toba. 394

6.12 Pointer store heap position: Interactive-AllCallsOn. 395

6.13 Pointer store heap position: Interactive-TextEditing. 396

6.14 Pointer store heap position: StandardNonInteractive. 397

6.15 Pointer store heap position: HeapSim. 398

6.16 Pointer store heap position: Lambda-Fact5. 399

6.17 Pointer store heap position: Lambda-Fact6. 400

6.18 Pointer store heap position: Swim. 401

6.19 Pointer store heap position: Tomcatv. 402

6.20 Pointer store heap position: Tree-Replace-Binary. 403

6.21 Pointer store heap position: Tree-Replace-Random. 404

6.22 Pointer store heap position: Richards. 405

6.23 Pointer store heap position: JavaBYTEmark. 406

6.24 Pointer store heap position: Bloat-Bloat. 407

6.25 Pointer store heap position: Toba. 408

6.26 DOF pointer position distributions: StandardNonInteractive. 409

6.27 DOF pointer position distributions: HeapSim. 410

6.28 DOF pointer position distributions: Richards. 411

6.29 DOF pointer position distributions: Lambda-Fact5. 412
6.30 DOF pointer position distributions: Bloat-Bloat. 413
6.31 DOF pointer position distributions: JavaBYTEmark. 414
6.32 Relative invocation counts: Interactive-AllCallsOn, $V = 764$. 415
6.33 Relative invocation counts: Interactive-AllCallsOn, $V = 931$. 415
6.34 Relative invocation counts: Interactive-AllCallsOn, $V = 1135$. 416
6.35 Relative invocation counts: Interactive-AllCallsOn, $V = 1384$. 416
6.36 Relative invocation counts: Interactive-AllCallsOn, $V = 1687$. 416
6.37 Relative invocation counts: Interactive-AllCallsOn, $V = 2057$. 417
6.38 Relative invocation counts: Interactive-TextEditing, $V = 1338$. 418
6.39 Relative invocation counts: Interactive-TextEditing, $V = 1631$. 418
6.40 Relative invocation counts: Interactive-TextEditing, $V = 1988$. 419
6.41 Relative invocation counts: Interactive-TextEditing, $V = 2424$. 419
6.42 Relative invocation counts: Interactive-TextEditing, $V = 2955$. 419
6.43 Relative invocation counts: StandardNonInteractive, $V = 1414$. 420
6.44 Relative invocation counts: StandardNonInteractive, $V = 1723$. 420
6.45 Relative invocation counts: StandardNonInteractive, $V = 2101$. 421
6.46 Relative invocation counts: StandardNonInteractive, $V = 2561$. 421
6.47 Relative invocation counts: StandardNonInteractive, $V = 3122$. 421
6.48 Relative invocation counts: StandardNonInteractive, $V = 3805$. 422
6.49 Relative invocation counts: StandardNonInteractive, $V = 4639$. 422
6.50 Relative invocation counts: StandardNonInteractive, $V = 5655$. 422
6.51 Relative invocation counts: StandardNonInteractive, $V = 6894$. 423

6.52 Relative invocation counts: HeapSim, $V = 113397$. 424

6.53 Relative invocation counts: HeapSim, $V = 138231$. 424

6.54 Relative invocation counts: HeapSim, $V = 168503$. 425

6.55 Relative invocation counts: HeapSim, $V = 205405$. 425

6.56 Relative invocation counts: HeapSim, $V = 250387$. 425

6.57 Relative invocation counts: HeapSim, $V = 305221$. 426

6.58 Relative invocation counts: HeapSim, $V = 372063$. 426

6.59 Relative invocation counts: Lambda-Fact5, $V = 7673$. 427

6.60 Relative invocation counts: Lambda-Fact5, $V = 9354$. 427

6.61 Relative invocation counts: Lambda-Fact5, $V = 11402$. 428

6.62 Relative invocation counts: Lambda-Fact5, $V = 13899$. 428

6.63 Relative invocation counts: Lambda-Fact5, $V = 16943$. 428

6.64 Relative invocation counts: Lambda-Fact5, $V = 20654$. 429

6.65 Relative invocation counts: Lambda-Fact5, $V = 25177$. 429

6.66 Relative invocation counts: Lambda-Fact6, $V = 16669$. 430

6.67 Relative invocation counts: Lambda-Fact6, $V = 20320$. 430

6.68 Relative invocation counts: Lambda-Fact6, $V = 24770$. 431

6.69 Relative invocation counts: Lambda-Fact6, $V = 30194$. 431

6.70 Relative invocation counts: Lambda-Fact6, $V = 36807$. 431

6.71 Relative invocation counts: Lambda-Fact6, $V = 44868$. 432

6.72 Relative invocation counts: Lambda-Fact6, $V = 54693$. 432

6.73 Relative invocation counts: Lambda-Fact6, $V = 66671$. 432

6.74 Relative invocation counts: Lambda-Fact6, $V = 81272$. 433

6.75 Relative invocation counts: Swim, $V = 21383$. 434

6.76 Relative invocation counts: Swim, $V = 26066$. 434

6.77 Relative invocation counts: Swim, $V = 31774$. 435

6.78 Relative invocation counts: Swim, $V = 38733$. 435

6.79 Relative invocation counts: Swim, $V = 47215$. 435

6.80 Relative invocation counts: Swim, $V = 57555$. 436

6.81 Relative invocation counts: Swim, $V = 70160$. 436

6.82 Relative invocation counts: Swim, $V = 85524$. 436

6.83 Relative invocation counts: Swim, $V = 104254$. 437

6.84 Relative invocation counts: Tomcatv, $V = 37292$. 438

6.85 Relative invocation counts: Tomcatv, $V = 45459$. 438

6.86 Relative invocation counts: Tomcatv, $V = 55414$. 439

6.87 Relative invocation counts: Tomcatv, $V = 67550$. 439

6.88 Relative invocation counts: Tomcatv, $V = 82343$. 439

6.89 Relative invocation counts: Tomcatv, $V = 100376$. 440

6.90 Relative invocation counts: Tomcatv, $V = 122358$. 440

6.91 Relative invocation counts: Tomcatv, $V = 149154$. 440

6.92 Relative invocation counts: Tomcatv, $V = 181818$. 441

6.93 Relative invocation counts: Tree-Replace-Binary, $V = 11926$. 442

6.94 Relative invocation counts: Tree-Replace-Binary, $V = 14538$. 442

6.95 Relative invocation counts: Tree-Replace-Binary, $V = 17722$. 443

6.96 Relative invocation counts: Tree-Replace-Binary, $V = 21603$. 443

6.97 Relative invocation counts: Tree-Replace-Binary, $V = 26334$. 443

6.98 Relative invocation counts: Tree-Replace-Random, $V = 15985$. 444

6.99 Relative invocation counts: Tree-Replace-Random, $V = 19486$. 444

6.100 Relative invocation counts: Tree-Replace-Random, $V = 23754$. 445

6.101 Relative invocation counts: Tree-Replace-Random, $V = 28956$. 445

6.102 Relative invocation counts: Tree-Replace-Random, $V = 35297$. 445

6.103 Relative invocation counts: Tree-Replace-Random, $V = 43027$. 446

6.104 Relative invocation counts: Tree-Replace-Random, $V = 52450$. 446

6.105 Relative invocation counts: Tree-Replace-Random, $V = 63936$. 446

6.106 Relative invocation counts: Richards, $V = 1826$. 447

6.107 Relative invocation counts: Richards, $V = 2225$. 447

6.108 Relative invocation counts: Richards, $V = 2713$. 448

6.109 Relative invocation counts: Richards, $V = 3307$. 448

6.110 Relative invocation counts: Richards, $V = 4032$. 448

6.111 Relative invocation counts: Richards, $V = 4914$. 449

6.112 Relative invocation counts: Richards, $V = 5991$. 449

6.113 Relative invocation counts: Richards, $V = 7303$. 449

6.114 Relative invocation counts: Richards, $V = 8902$. 450

6.115 Relative invocation counts: JavaBYTEmark, $V = 72807$. 451

6.116 Relative invocation counts: JavaBYTEmark, $V = 88752$. 451

6.117 Relative invocation counts: JavaBYTEmark, $V = 108188$. 452

6.118 Relative invocation counts: JavaBYTEmark, $V = 131881$. 452

6.119 Relative invocation counts: Bloat-Bloat, $V = 246766$. 453

6.120 Relative invocation counts: Bloat-Bloat, $V = 300808$. 453

6.121 Relative invocation counts: Bloat-Bloat, $V = 366682$. 454

6.122 Relative invocation counts: Bloat-Bloat, $V = 446984$. 454

6.123 Relative invocation counts: Bloat-Bloat, $V = 544872$. 454

6.124 Relative invocation counts: Bloat-Bloat, $V = 664195$. 455

6.125 Relative invocation counts: Bloat-Bloat, $V = 809650$. 455

6.126 Relative invocation counts: Bloat-Bloat, $V = 986959$. 455

6.127 Relative invocation counts: Toba, $V = 353843$. 456

6.128 Relative invocation counts: Toba, $V = 431335$. 456

6.129 Relative invocation counts: Toba, $V = 525794$. 457

6.130 Relative invocation counts: Toba, $V = 640941$. 457

6.131 Relative invocation counts: Toba, $V = 781303$. 457

6.132 Relative invocation counts: Toba, $V = 952404$. 458

6.133 Relative invocation counts: Toba, $V = 1160976$. 458

6.134 Relative invocation counts: Toba, $V = 1415223$. 458

6.135 Relative invocation counts: Toba, $V = 1725148$. 459

(This page intentionally left blank)

CHAPTER 1

INTRODUCTION

Garbage collection, the automatic reclamation of dynamically allocated memory, is a mature memory-management technique. It has been in use since 1958 [McCarthy, 1960] and is now accepted as a customary element for dynamic memory management in run-time systems of modern programming languages, especially object-oriented ones, for reasons of safety and for its clear software engineering benefits [Gosling *et al.*, 1996]. The increasing acceptance of object-oriented languages makes it more important than ever for garbage collection to work as fast as possible.

Multi-generational copying collectors are the most popular memory reclamation algorithms in use today. Generational collectors have two real advantages over original non-generational collectors: pause times are shorter (on average), and total cost is lower.

There are, however, three tenets that are commonly held as explanations why generational collectors have these advantages, and that have discouraged exploration of alternative strategies. The first has to do with a property, empirically confirmed in object-based systems, that young objects die fast (more precisely, that object mortality is a decreasing function). The popular wisdom is that generational collectors exploit this property by concentrating effort on collecting young objects. Furthermore, whenever only a part of the heap is collected at once, the remainder must be assumed live for the purposes of the collection, and all pointers from outside the region collected into it are roots of the collection. This approximation is conservative, and considers more data live than is actually the case, and thus copying cost is greater. The second tenet is that this excess copying effort is small in generational collection because pointers from outside the region into it are, by construction, pointers from older objects to

younger objects, and such pointers are few. The third tenet is that for the same reason it is easy to maintain the write barrier (actions taken during the execution of the program when a pointer is stored) and remembered sets (the sets of pointers crossing into a region).

Taken together, these three tenets do more than explain the good performance of generational collection. They have been understood in the past to rule out using any strategy that might collect older objects in preference to younger. Any such strategy would apparently concentrate effort on regions of lower mortality. It would also cause the collected region to have many incoming pointers from outside, thus from younger objects, and have both a high excess copying penalty and a high maintenance cost for the write barrier and remembered sets. Our study shows that, although each of the three tenets is intuitively and plausibly based on truthful observations of particular garbage collectors' behavior, alternatives to youngest-first generational collection are nevertheless feasible. The several plausible inferences drawn from the observations do not in fact rule out good incremental copying collectors other than generational: as we shall discover, they merely set the stage for a performance trade-off.

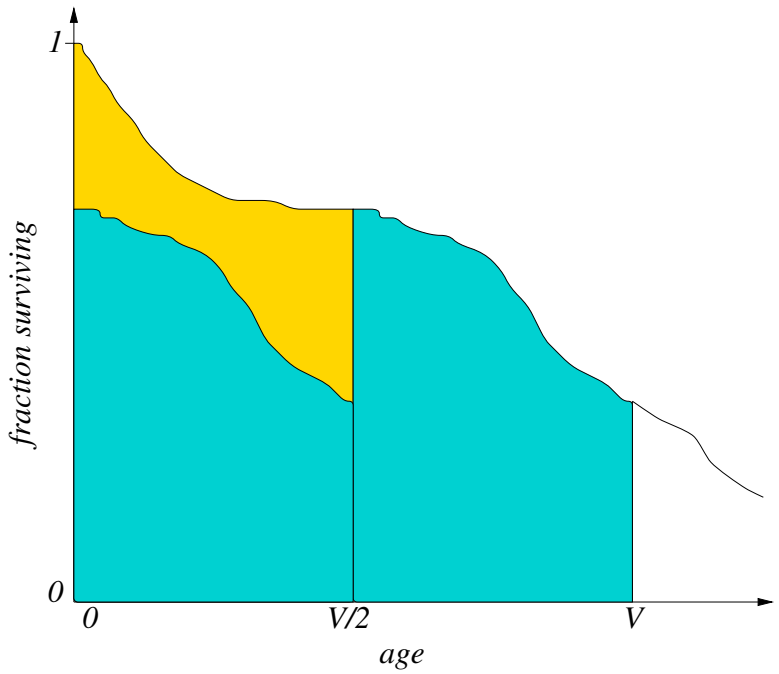


Figure 1.1. Surviving objects in a newly allocated area.

First, the intuition that effort should go to regions with high mortality is countered by the intuition that within a given area that has been newly allocated, its older half has less live data than its younger half, so it ought to be better to collect the older half.[1] A plot of the survivor function of the distribution of object lifetimes (Figure 1.1) graphically illustrates this fact—the survivor function is always a decreasing function, thus the area under the curve, which is the amount of surviving data, is smaller in the segment $[V/2, V]$ than in the segment $[0, V/2]$. In fact, it would be best to collect just the oldest object in the area. Figure 1.2 illustrates the point: imagine that newly allocated objects are placed in a queue, which plays the role of the youngest generation in generational collectors (also known as the *nursery*). When objects emerge from the queue, they are tested for liveness. Ideally, the length of the queue should be equal to the lifetime of each object, so that no object exits the queue alive. However, objects have diverse lifetimes, which a queue of fixed length cannot match. Yet if objects allocated in close succession have similar lifetimes, then the queue can manage them well, and if it can adjust the queue length to the objects' lifetimes, so much the better. Our design for the deferred older-first collector in Section 3.1.1, p.26, can be seen as an attempt to realize an approximation to a collection queue, within the restrictions of a finite and fixed space available for the queue and its survivors.

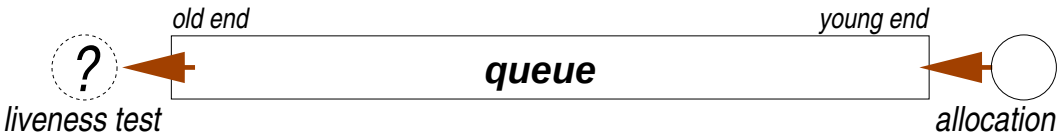


Figure 1.2. A collection queue.

With regard to the empirical observations on which the first tenet was founded, we should point out that, even though a decreasing mortality has been attested in all measured systems

[1]This point is subtle. For instance, in discussing garbage collection in the implementation of SML, Appel states both that *the longer one waits before a collection, the more records have turned into garbage* [Appel, 1992, p. 228], and that *newer cells have a higher proportion of garbage, which is why the garbage collection efforts should be concentrated on them* [Appel, 1992, p. 208]. Both of these statements are true, but not in the same context. If the meaning of characterizations such as these is improperly or simplistically understood, it is easy to derive incorrect inferences.

for relatively young objects, the behavior for relatively older objects, and certainly for very long-lived objects in long-running programs, remains unexplored.

Concerning the second tenet, excess copying: we empirically examined the difference in copying cost between a collector that assumes that everything outside the collected region is live, and one that has an oracle to tell apart the live from the dead. For the collection schemes we considered, we found that, as long as the scheme preserves the objects in a logical order of allocation (thus always having data of a contiguous age group in the region collected), the excess copying is usually not excessive. We then measured the statistics of pointer direction and found that neither older-to-younger nor younger-to-older direction dominates across all studied traces (Smalltalk and Java). But, more importantly, we found that pointer distances (differences in age between the object containing the pointer and the object pointed to) are very small indeed, whether positive or negative.[2] While this property works in favor of generational collection, it also works in favor of any other age-order-preserving collection scheme, by reducing the number of pointers that must be maintained (Section 3.2.2).

Concerning the third tenet, the implementation overheads: it withstood our validation. Indeed, the circumstance that neither pointer direction dominates is not significant, in light of another empirical fact: most pointer stores are into new, or very recently allocated objects. Generational collectors do not need to record these stores, regardless of direction. The other age-based collectors we introduce here must record the younger-to-older pointer stores into new objects, and pay a significant price for it. Nevertheless, we shall see that for many programs, it is worth paying this price to receive the benefit of lower copying cost.

In this study, we explore generational and other collection schemes among those that preserve the relative order of objects, using simulation studies and a prototype collector imple-

[2]DeTreville reports that in Taos, an operating system written in Modula-2+, the younger-to-older pointer stores barely outnumber (by $1.96 \cdot 10^6$ to $1.17 \cdot 10^6$) older-to-younger pointer stores [DeTreville, 1990b, p. 29]. The reported temporal distances are also very short; since they were obtained with respect to a particular garbage collector execution, they are not directly comparable to our measurements of Smalltalk and Java distances in Section 6.2.

mentation. We examine the obvious opposite of youngest-first collection, which is a strict oldest-first: choosing a fixed amount of oldest data. It sometimes works well, but if the chosen amount is small, the collector works poorly. Similarly, a strict youngest-first collector, which always chooses a fixed amount of youngest data (i.e., the nursery), works poorly. These observations lead us to explore strategies in which the collector chooses the collected region more flexibly, and visits both older and younger regions. A deferred older-first collector does so in the simplest possible manner, by moving a window of collection across the heap, but looking at older objects first. It performs remarkably well, sometimes close to the performance of a locally-optimal collector. The intuitive reason for this good performance is also remarkably simple: as the window of collection moves towards younger objects, it finds itself eventually at a position corresponding to an age by which most objects have died; and, since few objects survive in the window, the window does not move very fast. Thus, this collector homes in on the region where it is most profitable to collect, and stays there for a long time. A welcome property is that the best configurations observed in our experiments are those with a small window size, and, since the amount copied per collection is limited by window size, the low total copying cost is accompanied by predictably short pause times. Pointer-maintenance costs are inevitably higher in this collector than they are in generational collection. Nevertheless, the trade-off between these two cost components favors the newly proposed collector for a number of programs we examine.

In Chapter 2, we present the state of scholarship and introduce the terms used in the study of garbage collection algorithms. We discuss a classification of age-based collectors and the design and implementation of new algorithms in Chapter 3. We set the stage for the experimental evaluation of different collectors by examining the intrinsic properties of benchmark programs in Chapter 4. The copying cost of collection is the subject of Chapter 5, while the pointer-maintenance cost is discussed in Chapter 6.

Note on organization. Chapters 4–6 include long series of figures and tables. These are printed *en masse* at the end of each chapter, so as not to disrupt the flow of text, yet keep them close to it.

CHAPTER 2

BACKGROUND

This chapter introduces the concepts and the terminology used in the subsequent discussion while surveying the vast field of literature related to memory management and garbage collection.

2.1 Dynamic memory management

Under the appellation of dynamic memory management, we study the provision in the runtime system of a programming language of mechanisms for allocation of contiguous blocks of memory of arbitrary size and unrestricted temporal extent at the request of the program. These blocks of memory are known as *objects*. In object-oriented languages, there is usually a one-to-one correspondence between a linguistic object and its representation as a dynamically allocated block of memory. In other contexts, the representations of a C `struct` created using a call to `malloc`, a Pascal record created using `new`, or a LISP `cons` cell, are all objects.

A significant amount of research has gone into algorithms for managing dynamically allocated data under the restriction that the objects must not be moved, and that the program must explicitly request object deallocation following its last use of an object. These algorithms concentrate on fast allocation, typically using free lists, and efficient memory usage, avoiding fragmentation. Wilson wrote an excellent survey of the field [Wilson *et al.*, 1995].

Our focus, however, is on algorithms that automatically detect that the last use of an object must have occurred, by proving that no further use of the object is possible. These algorithms

are called *garbage collectors*. They remove the burden of object deallocation from the programmer, and in so doing eliminate an important source of programming errors.[1]

The garbage collector considers an object to be garbage (or *dead*) if it is not *reachable*: the program cannot access it by any chain of pointer traversals. Any object that is potentially reachable is considered *live*. Although the name implies that garbage is collected, it is usually the live objects that the collector works with: it preserves them, while reusing the space of the garbage objects.

We shall briefly introduce the concepts needed to understand the arguments made later; for a comprehensive treatment, the reader should consult Cohen's survey of the early development of garbage collection techniques [Cohen, 1981], Wilson's survey, which concentrates on newer algorithms [Wilson, 1992], as well as Jones and Lins's textbook presentation of the field [Jones and Lins, 1996].

2.1.1 Liveness and reachability

Let us first look at simple (non-generational) collection, in which the entire heap is subject to collection. Ideally, any object that will not be used by the program after the time of collection can be discarded as garbage. Unfortunately, this criterion is not effectively computable. Instead, the consensus is to use *pointer reachability* as a conservative approximation. Given a set of root pointers into the heap, owned by the program in its data outside the heap (stack, registers, etc.), everything reachable from this set is considered live. In other words, the transitive closure of the points-to relation, starting with the roots, defines the survivors of the collection. This approximation is safe, since no computation can use an object unless it can refer to it by a pointer, and if it does not already have that pointer it can only obtain it by following some chain of pointers starting from some other pointer it does have. Pointer reachability is simple to compute and is found in the core of all garbage collectors. Between the approximation of

[1] A study of software errors in IBM MVS ascribes almost one half to pointer and array access errors [Sullivan and Chillarege, 1991].

liveness by reachability and the unachievable exact liveness according to future use, there can be a gradation of techniques to make the approximation more accurate using different static analyses of the program (compile-time garbage collection, live variable analysis, etc.). We do not consider these here; hereafter we equate liveness with reachability from the root set of pointers.

We are concerned, however, with another distinction that arises only with region-based collection, which divides the heap H into a collected region C and a remainder U. Live objects in the collected region are those reachable from the set of root pointers. The goal of region-based collection is to avoid having to look at the (normally large) remainder. Instead, a write-barrier ensures that all pointers crossing from U to C are readily available at time of collection. All such pointers, called the *remembered set* for region C, are added to the root set, and then the transitive closure of pointer reachability is computed only within C. Let us call this result S_r, for *survivors by remembered-set reachability*. The actual set of live data is $S \subseteq S_r$. Calculation using remembered-set reachability treats all objects in the uncollected remainder U as if live, since it does not examine U, and thus potentially overestimates the set S. In *any* region-based collector, there is potential for such excess retention or "nepotism" [Ungar and Jackson, 1988] on each garbage collection.

How much excess retention is there? Clearly, a collection scheme could have dramatically high amounts of excess retention if it were the case that (1) pointers tended to cross region boundaries predominantly into the collected region, thus there were many pointers in the remembered-set, *and* (2) many of these pointers were from objects in the remainder that are actually dead, *and* (3) the pointer structure caused the transitive closure operation falsely to mark very large amounts of data as live, starting from such false pointers. The effects on performance would be profound, since both the cost of each collection and the frequency of collections would be high.

We have assumed in the preceding that the set of external roots in the stack and registers is accurately known. This is predicated on the ability of the collector to examine and interpret

the stack and registers. A system may opt for self-descriptive data, which incurs the space overhead of tagging each word, or a more compact form using stack maps describing stack contents at admissible garbage collection points. Additionally, a data-flow liveness analysis may reduce the number of registers in the root set [Agesen *et al.*, 1998].

On the contrary, if the system is uncooperative, then it may be the case that neither the values in the stack and registers, nor the values in the heap can be accurately interpreted: it may be impossible to distinguish some pointer representations from some non-pointer representations. Such is the situation with languages such as C and C++, which were designed without garbage collection in mind. *Conservative* garbage collectors (or *ambiguous-roots* collectors) have been developed to cope with this problem [Boehm and Weiser, 1988; Demers *et al.*, 1990; Boehm, 1993]. Although the analyses in this study have been designed and experimentally tested with accurate-roots collection in mind, they may be adapted to an ambiguous-roots setting.

2.1.2 Preserving live data

The collector must preserve the live objects for future use by the program. However, a distinction is made in collector design among techniques that keep objects in place, and those that move them. Mark/sweep collectors do not move objects [McCarthy, 1960]. Instead, free lists are used to manage space, as in manual dynamic storage allocation. The mark phase automates the detection of objects that have become garbage, while the sweep phase invokes the deallocation routine of the underlying dynamic storage allocator for each garbage object in turn. Copying collectors, which are the focus of our study, move the live objects into a new memory area, the "to-space," allowing the entire old area, or "from-space," to be reused for allocation [Minsky, 1963; Fenichel and Yochelson, 1969]. The copying is accompanied by the update of pointers that point to the moved objects. The detection of live objects, by transitive closure for pointer reachability, and the copying into to-space are usually interleaved in a *Cheney scan* [Cheney, 1970] which also eliminates the need for an auxiliary stack to compute the closure. With the use of two *semispaces* allocation is from a contiguous area,

hence simple and fast. The collector visits only live objects, which are typically fewer than garbage objects. However, the memory requirement of the copying collector is, in the worst case, double that of the mark/sweep collector.

2.1.3 Incremental collection

Collectors that collect entire heaps at once incur very long pause times. In applications tolerating only short and predictable pauses, incremental collection is needed. The coordination between the collector and the running program to allow the collector to make progress piecemeal, or even concurrently with the program, is not simple. It is accomplished either using a *read-barrier* mechanism, or a *write-barrier* mechanism. With a read barrier, the program is alerted when it tries to follow a pointer to an object that the collector has moved [Baker, 1978]. Because read operations are extremely frequent, read barriers have used hardware support for efficiency [Zorn, 1989].[2] With a write barrier, when the program stores a pointer into an object, the write is recorded. Whereas hardware support for the write barrier can be provided as well, as in the SOAR Smalltalk system [Ungar, 1986], pointer update operations are typically much fewer in number than pointer reads, so a software barrier in the form of in-line code is generally thought to be acceptable.

2.1.4 Generational copying collection

Generational copying collection divides the heap according to the age of objects into multiple areas, or generations, and collects younger generations more frequently than older generations [Lieberman and Hewitt, 1983; Moon, 1984; Ungar, 1984]. There are subtle differences among these algorithms as described in the literature, because most descriptions combine the organization of the heap into generations with particular policies for promotion of survivors of garbage collections. Even when these algorithms are designed as non-incremental ("stop-and-copy"), they can be effectively incremental, by virtue of the fact that most collections are only

[2] Zorn finds that a software read barrier could be implemented with a CPU overhead of not more than 20%, however [Zorn, 1990a].

of the youngest generation, and thus most pauses are short. Numerous practical implementations and studies of predictive and adaptive management of generations have reported good performance [Caudill and Wirfs-Brock, 1986; Courts, 1988; Shaw, 1988; Sobalvarro, 1988; Ungar and Jackson, 1988; Wilson, 1989; Wilson and Moher, 1989; Appel, 1989b; Wilson *et al.*, 1991; Hudson *et al.*, 1991; Stefanović, 1993b; Stefanović and Moss, 1994; Diwan *et al.*, 1995; Barrett and Zorn, 1995].

The performance of a collector is affected by a number of factors: the number of generations in the system; the promotion policy, or how long an object must remain in one generation before it is advanced into the next older; when to initiate collection; and how to track the pointers from older to younger generations [Wilson, 1992]. This space of design choices is immense. Moreover, the allocation characteristics of languages and individual programs are varied, and they strongly affect the performance of collectors. The performance metrics—time overhead, pause time, and space overhead—are insidiously hard to define and measure. Does the time overhead include the write barrier, and how can the cost of the write barrier be extricated from other run-time costs? Is maximum pause time more important than average pause time, or some percentile of the distribution thereof? Should one measure the peak memory usage or the average memory usage? Should peak usage reflect the space needed only during copying to hold the to-space? Is it reasonable to let the oldest generation be "tenured" (allowed to grow indefinitely and never collected) or should measurements be taken with respect to a constant available space? The answers to these questions must depend on the intended application. As a result of the inherent complexity of generational collector design and of the methodological difficulties in evaluation, the understanding of generational garbage collectors has remained incomplete.

Aside from remarks in Wilson's survey [Wilson, 1992] and Jones and Lins' book [Jones and Lins, 1996, p.151], which admit the possibility of collecting an older generation independently of a younger one, generational collectors are youngest-first: when collecting an older generation, all younger ones must also be collected. The number of generations is typ-

ically small, so the choice of the size of the collected region is limited. Barrett and Zorn examined an adaptive scheme which dynamically chose an age boundary for each collection [Barrett and Zorn, 1995]. Our work instead proposes to collect among older (but not necessarily oldest) objects preferentially, postponing the youngest objects until they have had time to become older and die. The work closest to ours in this respect is Clinger and Hansen's "non-predictive" collector, which organizes objects according to the time elapsed since last collected and performs well with a hypothetical exponential distribution of lifetimes [Clinger and Hansen, 1997]. However, as we shall see in our empirical study in Chapter 5, that particular organization does not work remarkably well.

2.1.4.1 Pointer maintenance

In order to collect one region (a set of youngest generations) without collecting the remainder of the heap, the set of pointers crossing the boundary from the remainder and into the collected region must be known. Pointers installed in an object at the time of its creation can only point to older objects and can never cross the boundary in the interesting direction, only pointers installed by *updates* can. Each updating pointer store must be checked: if the stored pointer is $p \rightarrow q$, and p is in an older generation than q, then a record of this store will allow the pointer to be found at the time of collection of q. Early generational collectors used indirect pointers and region entry and exit tables, mechanisms that are very expensive without hardware support for pointer operations, and not transparent to the language. Ungar's collector used a *remembered set* of updated objects in the old (tenured) generation [Ungar, 1984]. A set can alternatively record the precise location of the updated object fields ("slots"), trading extra space for reduced time to find the updated fields at garbage collection time [Hosking *et al.*, 1992]. If there are few repeated stores into the same location between collections, a data structure simpler than a set may be preferable, such as a store list [Appel, 1989b] or a sequential store buffer [Hosking *et al.*, 1992].

Virtual memory mechanisms can be used to record the approximate location of updated object fields to within a page. Pages are write-protected, the first update of a page causes a page protection fault, and the software fault handler makes a note of the page (in an auxiliary bit array) and unprotects it. This scheme is called *page marking*. Unfortunately, virtual memory primitives in current operating systems are not well adapted to the task (non-pointer stores and null pointer stores equally cause protection faults) and are slow (because of user/kernel mode switching) [Zorn, 1990a]; moreover, the granularity of pages is too coarse (the ratio of page size to the typical number of pointer updates within it is high) [Hosking *et al.*, 1992]. Software surrogates of page marking are known as *word marking* [Sobalvarro, 1988], or more generally, *card marking*, in which the address space is conceptually divided into sub-page units (cards). Each pointer store is augmented with code to index into an auxiliary card table and set the bit corresponding to the card of the updated field. Whereas word marking records the precise location of updated fields, card marking with cards larger than a word saves auxiliary table space at the cost of scanning individual cards at garbage collection time [Hosking *et al.*, 1992]. Card tables as described tend to retain scattered marked entries over time, and it is better to summarize them at collection time (into a concise remembered-set representation) so that the card table reflects only incremental updates [Hosking and Hudson, 1993].

2.1.5 Mature spaces

Bishop [Bishop, 1977] and later Hudson and Moss [Hudson and Moss, 1992] proposed dividing the heap into spaces managed by different algorithms. Bishop allows several independent areas, not necessarily all garbage-collected, and uses lists of incoming and outgoing inter-area links. Hudson and Moss divide the heap by age into a young and a mature space, which is in agreement with the intuition that the patterns of pointer structure linking the objects lose the strong directional bias (see Section 6.2) after a certain age, hence the generational algorithms, which exploit this bias, lose efficiency. The mature space is divided into areas of bounded size (or blocks of fixed size in an implementation in the Beta language [Grarup and

Seligmann, 1993]), which are processed one at a time. It is assumed that a generational algorithm manages the young space and engages the incremental mature space algorithm only when promoting out of the oldest generation of the young space.

2.2 Object survival and mortality

A recent book on garbage collection states [Jones and Lins, 1996, p.144]:

> *The insight behind generational garbage collection is that storage reclamation can be made more efficient and less obtrusive by concentrating effort on reclaiming those objects most likely to be garbage, i.e., young objects.*

Hayes and others observe that youngest objects have the highest *mortality* [Hayes, 1991; Baker, 1993; Hayes, 1993; Stefanović and Moss, 1994], which is a refinement of the somewhat vague observation that most objects have a very short lifetime while some live much longer [Lieberman and Hewitt, 1983; Ungar, 1984; Shaw, 1988; DeTreville, 1990a; Zorn, 1990b]. Figure 2.1 graphs a typical mortality function using Tomcatv written in Smalltalk as an example (see Section 4.2.2.3), with the age of objects on the horizontal axis, and for a given age, the rate at which they die on the vertical axis. Both axes use a logarithmic scale.

Is high mortality the same as being most likely to be garbage? No, since the very newest objects are patently not garbage. In statistical terms, it is the *survivor* function, and not the mortality, that tells us what is live, and the survivor function is always 1 at age 0. Formally, the survivor function $s(t)$ is the fraction of original allocation that is still live at age t. The mortality $m(t) = -\frac{s'(t)}{s(t)}$ is the age-specific death rate, or likelihood that an object will die in the next instant provided it has survived until age t [Cox and Oakes, 1984; Elandt-Johnson and Johnson, 1980; Baker, 1993]. Figure 2.2 plots a typical survivor function, again using Tomcatv. The horizontal axis is object age on a logarithmic scale. The vertical axis is the fraction of original allocation still live at a given age.

Given a region of just-allocated data of size V, is it more profitable to collect its younger half, ages 0 to $V/2$, or its older half, ages $V/2$ through V? Statistically speaking, the answer

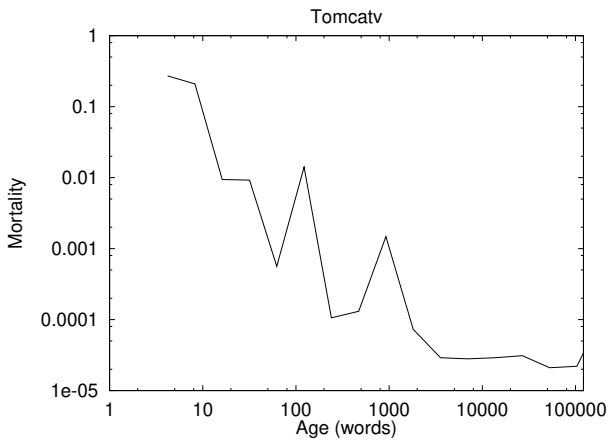


Figure 2.1. A typical object mortality function.

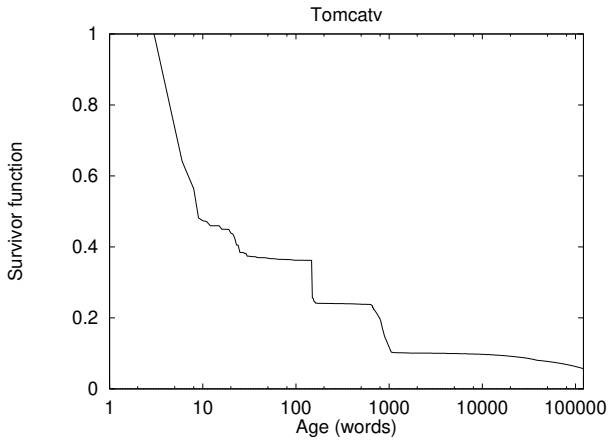


Figure 2.2. A typical object survivor function.

must be to collect the older half, which is clear from the survivor function diagram, Figure 1.1, or the simple calculation: $\int_{V/2}^{V} s(t)dt \leqslant \int_{0}^{V/2} s(t)dt$, because the survivor function $s(t)$ is non-increasing.

In practice, we have to decide where to put the survivors, which complicates this simple analysis. The analysis suggests, however, the following revision of the preceding quotation:

> *Generational collection is efficient, provided its generations are well configured, because it concentrates on a set of young objects that includes the objects most likely to be garbage, albeit along with objects least likely to be garbage.*

If the collector can concentrate *only* on the objects most likely to be garbage, then it may be able to achieve even lower copying cost. This discussion was entirely independent of the

actual shape of the survivor and mortality curves, but the relative merits of generational and other collectors will certainly depend on them.

Assumed or measured object lifetime distributions can be used to predict the performance of collectors [Clinger and Hansen, 1997]. However, the implicit assumption that object lifetimes are independent identically distributed random variables is far from true: phase behavior of programs must lead to highly correlated object lifetimes. The collector performance calculations are tied to the assumption of a steady state, but this assumption restricts the class of admissible distributions. The questions of the feasibility and the utility of analytical modelling of object lifetimes therefore remain open [Stefanović *et al.*, 1998a; Appel, 1997; Stefanović *et al.*, 1998b].

2.3 Theoretical models of memory management

Object-based computation appears in its purest form in the Storage Modification Machine, or pointer machine [van Emde Boas, 1990; Kolmogorov and Uspenskii, 1958; Schönhage, 1980], a model of computation based on a finite directed graph structure of storage instead of an array of memory registers as in the more commonly used RAM model. The graph structure is dynamic. The model incorporates the notion of new node creation. The requirement for what we call garbage collection is expressed abstractly in the following dictum [van Emde Boas, 1990, p. 32]: "The map p^* does not have to be surjective; however, nodes which can not be reached by tracing a word w in Δ^* starting from the center a will play no subsequent role during the computations of the SMM, and therefore nodes may be assumed to have disappeared when they become unreachable."

A complementary aspect of theoretical approaches is the development of semantic models of memory management: deriving the properties of dynamically allocated data by linguistic means, usually in the context of typed functional languages [Morrisett *et al.*, 1995]. These models indicate which objects are garbage and why, and additionally allow one to express memory management actions syntactically, and then reason about their correctness. (Unfor-

tunately, these models do not answer quantitative questions of the kind posed in our study.) Similar models are desirable for memory management in object-oriented languages, but their semantic modelling in general remains less settled [Abadi and Cardelli, 1996]. A related discipline is that of compile-time garbage collection, or type and data-flow analyses to improve the run-time performance of collection in (usually) functional languages [Appel, 1989a; Harrison, 1989; Baker, 1990; Goldberg and Gloger, 1992; Boehm and Shao, 1993; Fradet, 1994; Aditya *et al.*, 1994; Mohnen, 1995]. The primary idea is to eliminate the need for heap allocation in the first place, by proving that certain objects can be allocated on the stack instead, which is considered (although not universally, cf. [Appel, 1987]) to be faster. More ambitious analyses aim to provide hints to a generational collector about object lifetimes [Yi and Harrison, 1992].

CHAPTER 3

AGE-BASED GARBAGE COLLECTION ALGORITHMS

In this chapter we establish a classification of garbage collection algorithms that preserve the age ordering of objects in the heap. We introduce new algorithms within this classification. Unlike generational algorithms, the new algorithms do not always collect a youngest subset of heap objects. We proceed to discuss the repercussions of this design difference on the implementation of copying and pointer maintenance in the collector.

3.1 Conceptual design

The garbage collection algorithms we consider are all "generational" in the broadest sense of the word: when a collection happens, each scheme partitions the heap H into two regions: the collected region C, in which the collector examines the objects for liveness, and if live, they survive the collection; and the uncollected remainder region U, in which the collector assumes the objects to be live and does not examine them. The non-generational collector is a degenerate case in which the latter region is empty. The collector further partitions the set C into the set of survivor objects S and the set of garbage objects G, by computing root pointers into C and the closure of the points-to relation within C. To make the freed space conveniently available for future allocation, a copying collector manipulates the survivors S by moving them to the "to-space". This manipulation is a primary cost of collection [Jones and Lins, 1996].

Denoting by $||X||$ the total volume of objects in a set X, we use as a measure of the copying cost of collection the "mark/cons"[1] ratio, the ratio of the volume copied to the volume freed (i.e., available for allocation in the subsequent cycle): $\mu = \frac{||S||}{||G||}$.

It is desirable to minimize μ. If the initial partition into C and U is such that most of C falls into G, the copied volume of survivors $||S||$ will be small. However, if *arbitrary* schemes are considered, then clearly the best scheme is one that always makes the initial partition so that $G = C$, and thus $\mu = 0$. But is it realistic? The computational effort to perform the initial partition ideally is as great as to perform a non-generational collection—except if an oracle were available. We consider as realistic only those schemes which compute the initial partition very cheaply.

We restrict our attention to a class of schemes that keep objects in a linear order according to a certain *object timestamp*. Imagine objects in the heap as if arranged from left to right, with the oldest timestamp on the left, and the youngest timestamp on the right, as in Figure 3.1. The region collected, C, is restricted to be a contiguous subsequence of this sequence of heap objects, thus the cost of the initial partition, simply choosing the two boundaries, in some prescribed manner, is virtually nil. We call these schemes *age-based* collection.

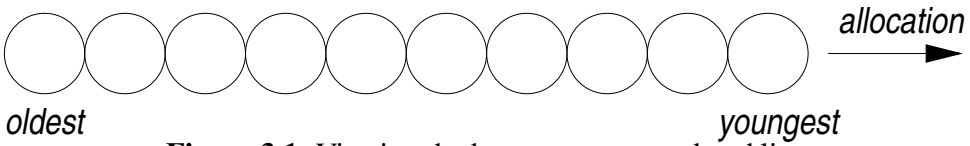


Figure 3.1. Viewing the heap as an age-ordered list.

We consider two notions of timestamp: *true age* and *renewal age*. In the first case, each object is given a permanent timestamp once, at the time of allocation. The relative order of objects in the sequence of heap objects never changes. In the second case, upon collection, the collector gives all survivors fresh, renewed, timestamps, as if they were newly allocated. It then moves these survivors to the young end of the sequence of heap objects.

[1]The name is used for historical reasons: in the original mark/sweep collector for LISP [McCarthy, 1960] the live objects were *marked* during collection, whereas the allocation consisted solely of LISP pairs, known as `cons` cells.

True-age timestamps permit the collector to focus attention on particular age groups, i.e., on sets of objects allocated consecutively. They arise naturally from considerations of object lifetime behavior—clusters of objects allocated together tend to expire together [Hayes, 1993]. On the other hand, the notion of renewal-age timestamps can arise from considerations of an *a priori* exponential lifetime distribution [Clinger and Hansen, 1997].

Traditional generational collection schemes are, in the main, age-based. They are based on true age, and the region collected is some subsequence that always includes youngest (most recently allocated) objects.[2] In the following, we introduce and categorize alternative collection schemes according to their choice of objects for collection. The age-based collection algorithm is also presented in the form of code in Figure 3.2.

3.1.1 True-age schemes

The simplest age-based collector and the base point of our comparisons is the non-generational collector. It will be labelled NONGEN in the discussion and figures below. The region of collection in NONGEN is always the entire heap: $C = H, U = \emptyset$.

We now consider age-based schemes, distinguished by different choices they make for the collected region. A *true youngest-first* (TYF) collector always chooses some youngest (rightmost) subsequence of the sequence of heap objects (Figure 3.3).

Generational collector schemes are variants of true youngest-first collection, differing in the size of C and in how they trigger collections [Barrett and Zorn, 1995]. In the basic design [Jones and Lins, 1996, p.147], new allocation is into one fixed-size part of the heap (the *nursery*), and the remainder is reserved for older objects (the *older generation*). Whenever the nursery fills up, it is collected, and the survivors promoted to the older generation (Figure 3.5). When the older generation fills up, then the following collection collects it together with the

[2] A slight divergence from the age-based criterion may arise in *copying* collectors, because the order of the objects in to-space is determined by the heap traversal (typically breadth-first, as in the standard Cheney scan [Jones and Lins, 1996; Cheney, 1970]), and is in general different from original order in from-space. Hence, the age order of objects is perturbed slightly within S for each collection. In *compacting* collectors reordering does not occur at all.

initialize:

available_space := *heap_size*

current_timestamp := 0

allocate (*num_words*):

if *available_space* $\geqslant$ *num_words* then

create new object o with

$o.size$:= *num_words* and $o.timestamp$:= *current_timestamp*

current_timestamp := *current_timestamp* + *num_words*

available_space := *available_space* - *num_words*

else

Garbage collection:

set timestamp boundaries t_{min}, t_{max} according to scheme

$C := \{o | o.timestamp \in [t_{min}, t_{max}]\}$

$U := H \setminus C$

$S_0 := \{o | o \in C \wedge (\exists r \in Roots \cup U) r \rightarrow o\}$

$S := TC(\rightarrow, C, S_0)$, where transitive closure is computed as:

$S_n = \{o | o \in C \wedge (\exists o' \in S_{n-1}) o' \rightarrow o\}$

$S = \bigcup_{n=0}^{\infty} S_n$(union finite)

$G := C \setminus S$

available_space := *available_space* + $\|G\|$

discard objects in G

if $\|G\| = 0$ then fail

fi

Figure 3.2. Pseudocode for the conceptual design.

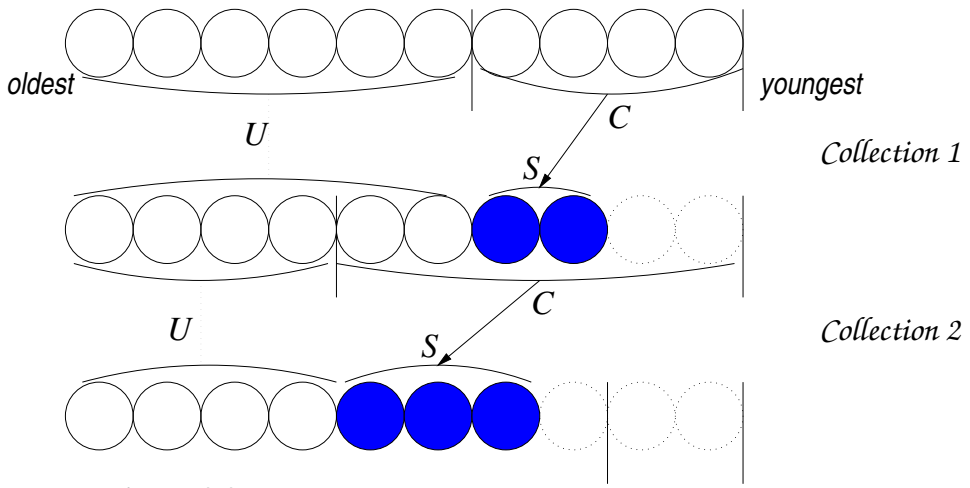


Figure 3.3. TYF: True youngest-first collection (general case).

$\|C\|$: collected region	U: remainder (region not collected)
S ●: region of survivors	○: area freed for new allocation

Two successive collections are shown. For sake of illustration, the surviving amount is 2 on collection 1, and 3 on collection 2. The collected region is enclosed in vertical bars.

Legend for Figures 3.3–3.12.

nursery. In a two-generation collector, that collection considers the entire heap. The collector deliberately reserves space, so that, unlike in TYF, the region chosen for collection contains exactly the objects allocated since the last collection (except for full heap collections). The operation of the GYF collector is summarized in the code of Figure 3.4.

We study two- and three-generation schemes: 2GYF (2 Generations; Youngest-First), and 3GYF. We stipulate that the size of each generation is strictly greater than 0, and therefore 3GYF never degenerates into 2GYF. When discussing generational schemes generally, we use the label GYF. We also examine a scheme in which the older generation is allowed to grow and use all the space of the nursery, and vice versa [Appel, 1989b], and no space is held in reserve; this scheme is denoted NGYF, and its operation is shown in Figure 3.6.

The simplest youngest-first collector is one that always chooses a constant amount of youngest objects to collect; we call this scheme FC-TYF (Fixed size of Collected region; True Youngest-First). Note, as is clear from the diagram in Figure 3.7, that survivors of the preceding collection remain in the collected region. One can expect this collector to be inefficient,

initialize:

Generations 1–N:

set $\sum_{g=1}^{N} nominal_size_g = heap_size$

for $g = 1, ..., N$ do $Gen_g := \emptyset$

$reserve := heap_size$

$next_gen := 1$

allocate (*num_words*):

if $\min(nominal_size_1 - current_size_1, reserve) \geqslant num_words$ then

create new object o in Gen_1 with

$o.size := num_words$

$current_size_1 := current_size_1 + num_words$

$reserve := reserve - num_words$

else

Garbage collection:

$gen := next_gen$

$C := \bigcup_{g=1}^{gen} Gen_g$

$U := H \setminus C$

$S_0 := \{o | o \in C \wedge (\exists r \in Roots \cup U) r \rightarrow o\}$

$S := TC(\rightarrow, C, S_0)$

for $g = 1, ..., gen$ do $S^{[g]} := S \cap Gen_g$

$Gen_1 := \emptyset$

for $g = 2, ..., gen{+}1$ do $Gen_g := Gen_g \cup S^{[g-1]}$,

except if $gen = N$ then $Gen_N := S^{[N]} \cup S^{[N-1]}$

$reserve := heap_size - \sum_{g=1}^{N} \|Gen_g\|$

$next_gen := \min g' : (\forall g \geq g') \|Gen_g\| \leqslant nominal_size_g$

fi

Figure 3.4. Pseudocode for GYF collectors.

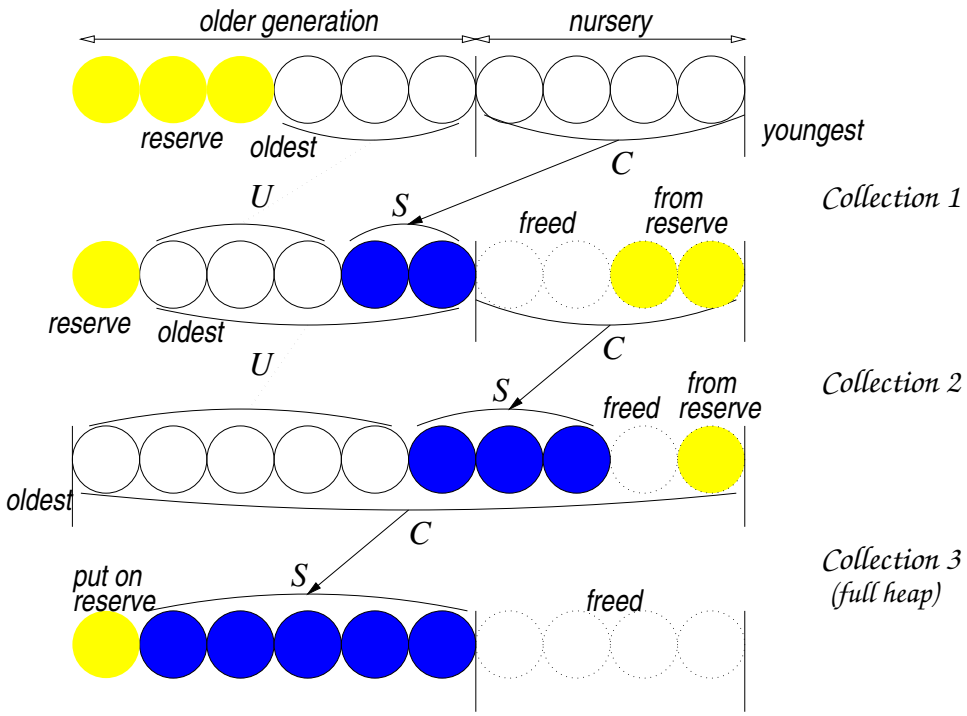


Figure 3.5. Generational youngest-first collection (2GYF).

because objects are repeatedly copied, and collections are frequent, because only the space freed is available for new allocation. Indeed, the older region remains untouched after it is initially allocated. Contrast this with the GYF collector, which does not use all the available space, putting some on reserve, but is then able to collect at regular intervals that correspond to the nursery size, and does not copy survivors of minor collections again until the next full collection.

A *true oldest-first* (TOF) collector always chooses an oldest (leftmost) subsequence of the sequence of heap objects (Figure 3.8). An FC-TOF collector considers a constant amount of oldest objects. It is clear that survivors of the collection will be collected repeatedly. If the oldest objects in the heap survive indefinitely, as in many programs, and the collected region is small and only encompasses such objects, FC-TOF necessarily fails because it finds no garbage in the collected region. However, we encountered some programs where FC-TOF, with a moderate size of the collected region, works better than other age-based collectors: intuitively, it gives objects enough time to die before they enter the collected region.

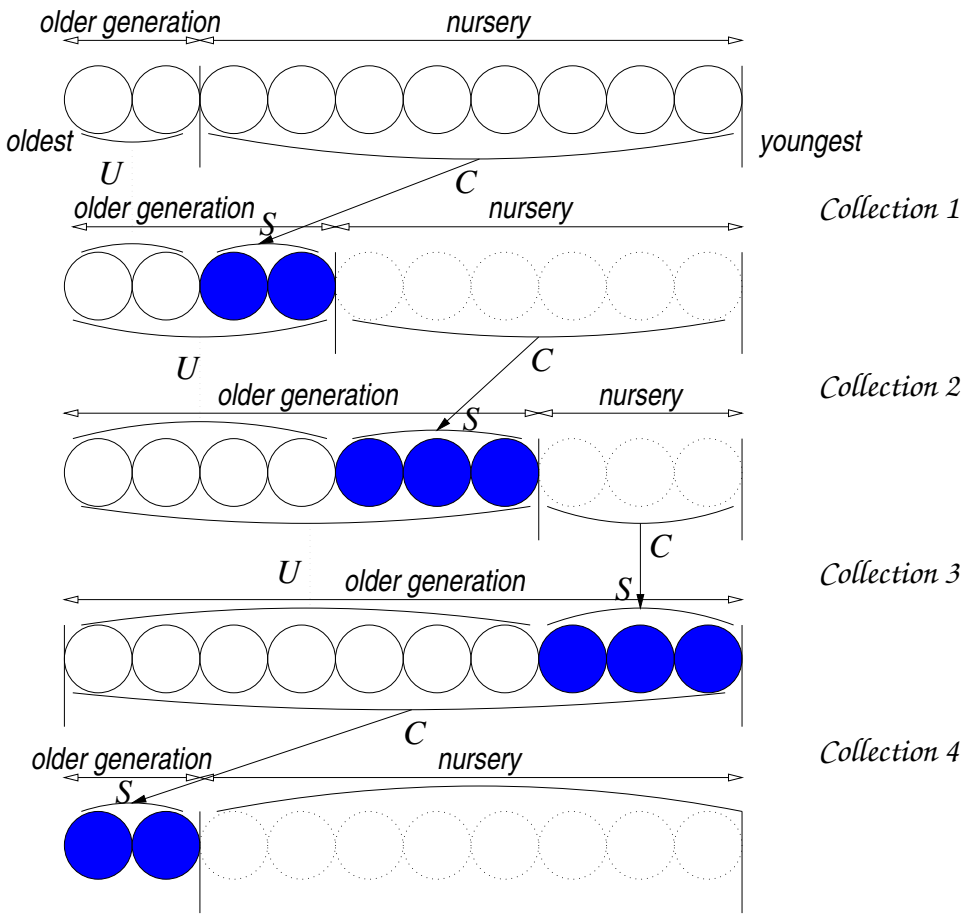


Figure 3.6. Generational youngest-first collection with flexible generation boundaries (NGYF).

The simplicity of FC-TYF and FC-TOF schemes is in the fact that collected regions are fixed. However, it is intuitively clear that performance suffers because some objects are subjected to collection repeatedly, while others are not subjected to collection soon enough. Thus, we must look for schemes that consider all parts of the heap at appropriate intervals. Note that GYF, which is known to work well in practice, revisits the entire heap, and at regular intervals, by means of full collections. Let us instead examine ways to revisit all of the heap piecemeal.

A *deferred older-first* (DOF) collector chooses a middle subsequence of heap objects, which is immediately to the right of the survivors of the previous collection (Figure 3.9). Thus the region of collection sweeps the heap rightwards, as a sliding window of collection. If the remaining area to the right is smaller than the window, the collector completes the sweep by

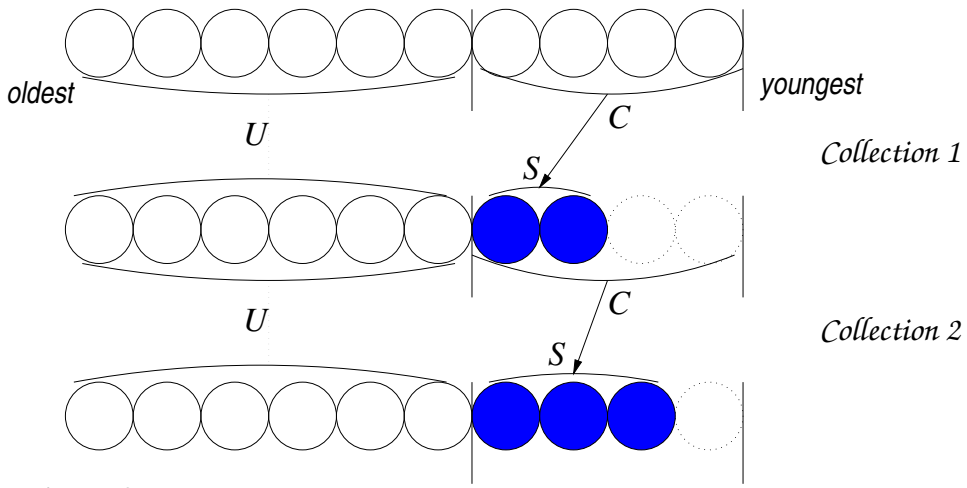


Figure 3.7. FC-TYF: True youngest-first collection (fixed collected region).

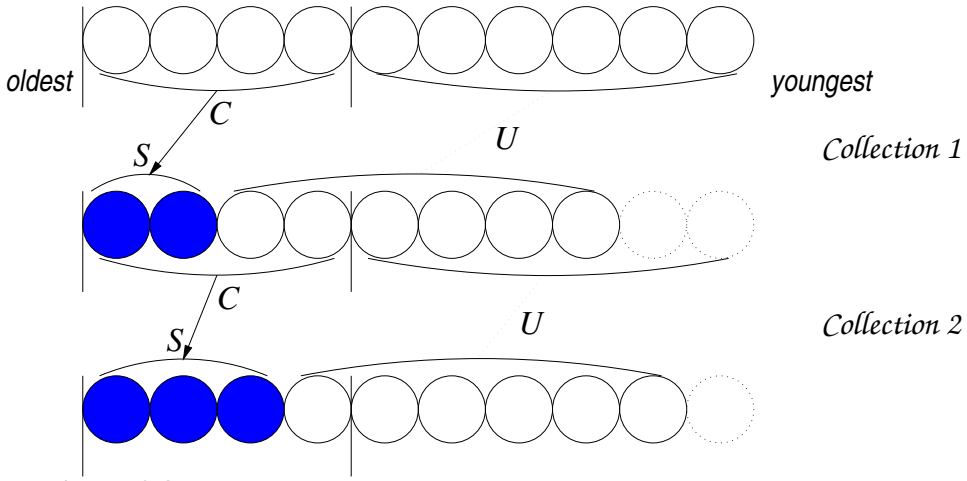


Figure 3.8. FC-TOF: True oldest-first collection (fixed collected region).

collecting the remaining area of youngest objects.[3] The FC-DOF collector employs a window of constant size.

The intuition for the potentially good performance of the FC-DOF collector can be gleaned from the diagram in Figure 3.10, which shows a series of eight collections, and indicates how the window of collection might move across the heap. The most important aspect is that if the window is in a position that results in small survivor sets, then the window moves by only a

[3]In the original description of the DOF algorithms, we proposed that as soon as the remaining area to the right is smaller than the window, the window is reset to the left end. However, we subsequently found that better worst-case copying cost is achieved with the current description. Additionally, full sweep is required for certain pointer filtering techniques (see below, Section 3.2.2).

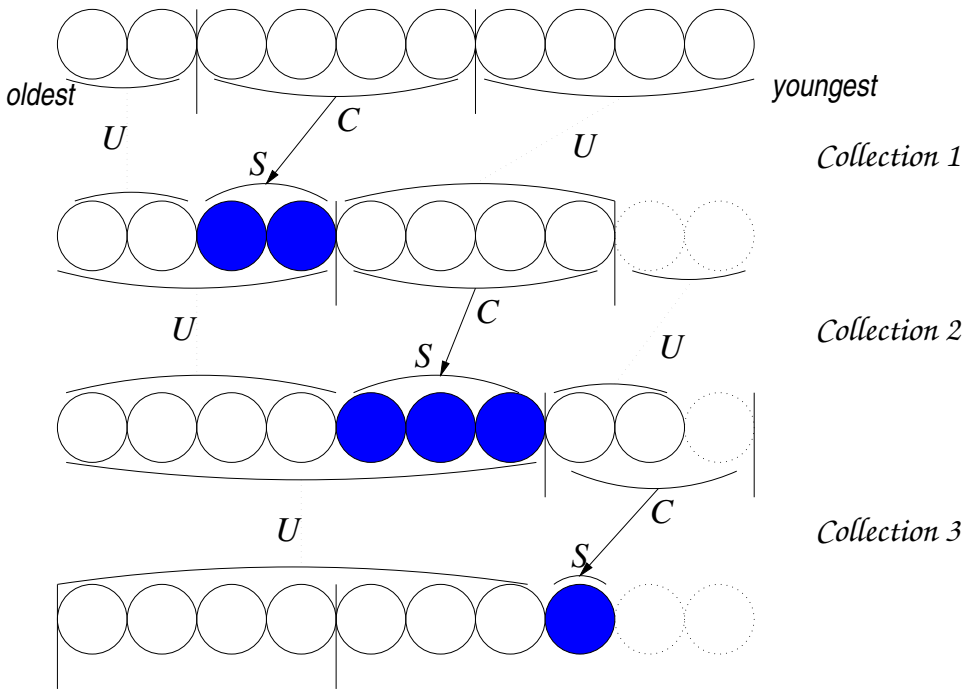


Figure 3.9. FC-DOF: Deferred older-first collection.

small amount from one collection to the next. If it continues to move slowly, then it is likely to remain for a long time in the same region, corresponding to the same age of objects, and it continues to enjoy small survivor sets (Collections 4–8). The outcome is that considerable progress is made in allocation while but a small amount is copied: μ is low. The question then is how long the window can remain in such a good position, and, once it has left it, how costly is the return, which involves finishing the rightward sweep, resetting to the left end, and sweeping over older objects back to the "sweet spot."

A *deferred younger-first* (DYF) collector similarly sweeps the heap, but leftwards (Figure 3.11). As soon as the remaining area to the left is smaller than the window, the window is reset to the right end. The FC-DYF collector employs a window of constant size. Despite the apparent symmetry of definition, DYF does not enjoy the window-motion benefit of DOF: the window moves leftward by one window size even when there are no survivors. We shall find that for this reason DYF never performs well.

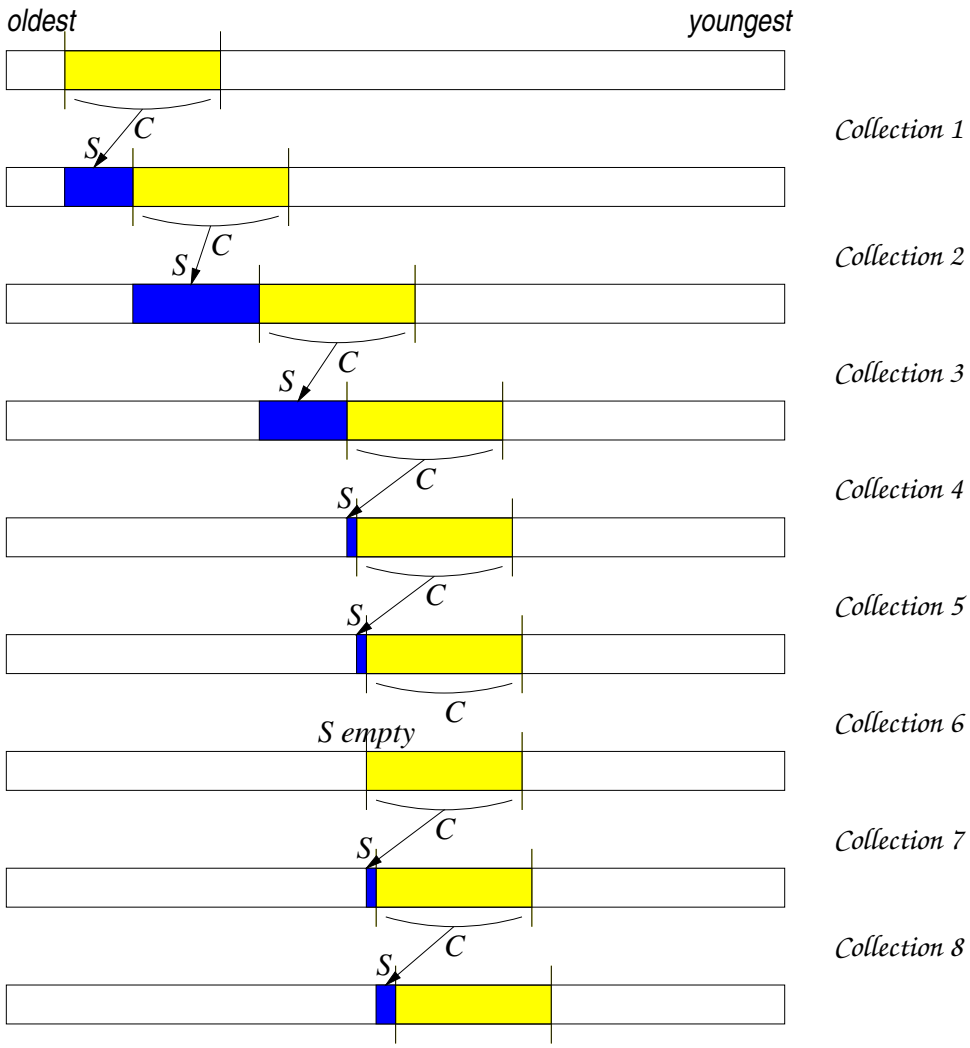


Figure 3.10. Deferred older-first collection, window motion example.

3.1.2 A renewal-age scheme

The collector proposed recently by Clinger and Hansen [Clinger and Hansen, 1997] and its generalization, the "oldest-first" collector [Stefanović *et al.*, 1998a], are also age-based, but according to renewal age, and older objects form the collected region. Similar ideas for circular management of heaps can be traced to earlier literature [Baker, 1978; Bekkers *et al.*, 1986; Lang and Dupont, 1987]. A *renewal-oldest-first* collector always chooses a leftmost subsequence of the sequence of heap objects—but the survivors are placed to the right of the uncollected remainder (Figure 3.12). Therefore, even though the collection window remains at the left, "old" end, the collector does not process the survivors of the collection again, until

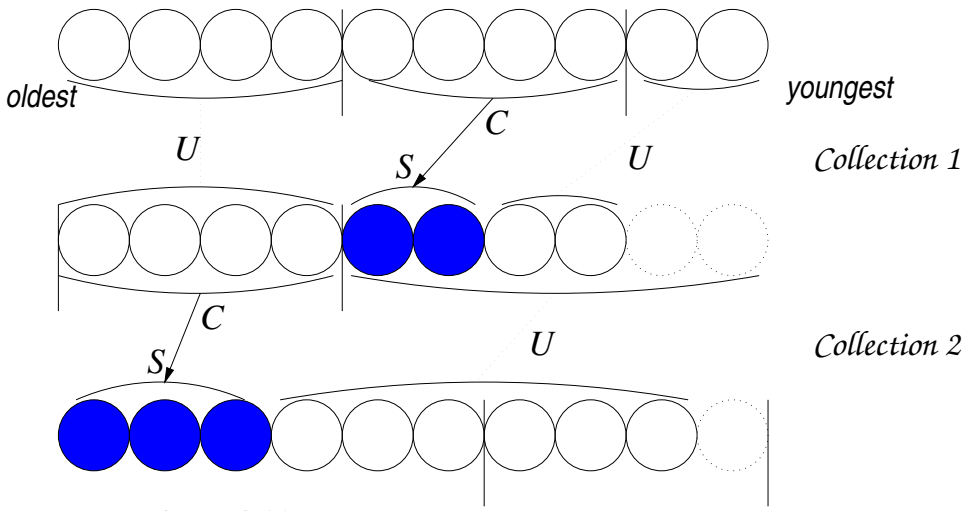


Figure 3.11. FC-DYF: Deferred younger-first collection.

they have been pushed back to the left by new allocation after several succeeding collections. Note, however, the conspicuous crossover of the logical paths of survivor and remainder objects, which results in irreversible mixing of true object ages in the heap, even in compacting collection.

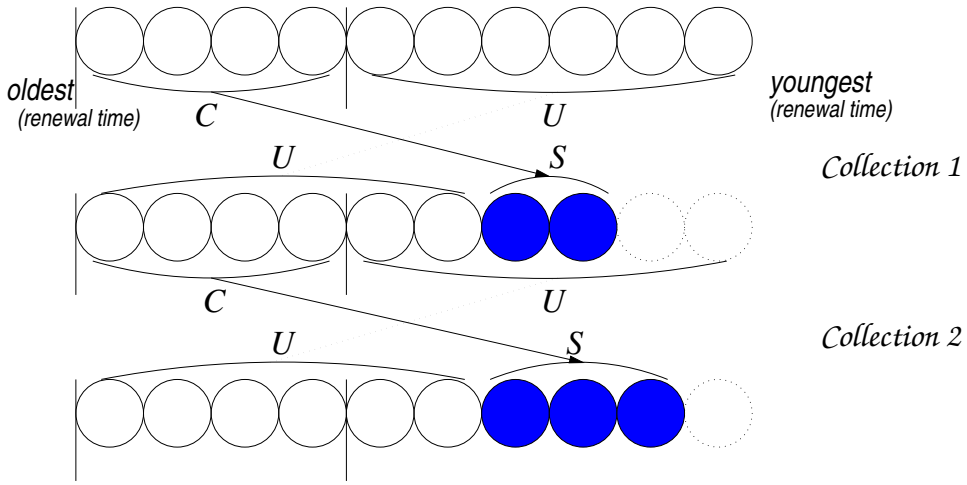


Figure 3.12. FC-ROF: Renewal-oldest-first collection.

3.1.3 Towards optimal schemes

There is a final category of schemes, clearly unrealistic for actual implementations, that we consider. Recall that a deferred oldest-first collector collects a middle subsequence of

age-ordered heap objects. Contemplate for a moment a generalization: let us choose any subsequence of a given size, not necessarily starting from a known point, but instead whichever subsequence yields the lowest mark-cons ratio on this collection. Thus we arrive at the *locally-optimal* collector scheme FC-OPT.[4]

Is there a globally optimal scheme? The global optimum need not coincide with the aggregate outcome of locally optimal decisions. The overall copying cost can be reduced by selecting the amount of garbage released in such a way that the next collection comes at a particularly propitious time, namely when there are few live objects. However, the consideration of globally optimal schemes is beyond the scope of this investigation. Instead, we shall only use the results of the locally-optimal collector schemes as an upper bound on the minimum copying cost, for comparisons with realistic schemes.

3.1.4 Collection failure and recovery

It is possible, especially for an FC collector with a small setting for C, to find no garbage in the collected region. If that happens, we let the FC collector fail for the purposes of this study. An implementation could increase the heap size temporarily, or retry collection on another region. A moving-window scheme can retry collection at the next window position. Alternatively, the whole heap can be collected. Note that GYF schemes by design occasionally consider the whole heap, hence they enjoy an advantage over the FC schemes as simulated here, with the disadvantage that their maximum pause times are no lower than with non-generational collection.

3.1.5 Garbage cycles

A cycle of garbage objects can be so large that it cannot fit in the collected region. Even if the cycle is smaller than the size of the collected region, it can consist of objects of sufficiently

[4]Similarly, we could explore the limits of true oldest-first and true youngest-first collection: instead of fixing the amount to be collected, let the collector choose that amount which results in the lowest mark/cons ratio μ *for the current collection*, but we leave this exploration for future work.

diverse age that it is never entirely within the collected region.[5] In either case, the cycle cannot be reclaimed. Traditional generational collection, at the risk of long pause times, finds such cycles by collecting the whole heap occasionally. Since the presence of such cycles is liable to lead to poor performance and ultimately failure, the same remedy, suggested in the preceding section, applies. Alternatively, the purely age-based scheme can be modified, with provisions similar to the train algorithm for the *mature object space*, which tackles the problem of cycles while maintaining incrementality [Hudson and Moss, 1992; Seligmann and Grarup, 1995]. Note that the results presented here are based on a simulator and a prototype implementation which do not implement any mechanism to enforce reclamation of cycles.

3.1.6 Locality

Our study is based on extensive simulations, in order to explore a large number of collector configurations, and indeed a number of altogether different schemes. This focuses attention on the primary costs, namely copying and pointer maintenance. However, we cannot measure certain other costs, without integrating our implementation with a live object system. The chief of these costs is the effect on locality [Zorn, 1991; Reinhold, 1993; Gonçalves, 1995; Diwan *et al.*, 1995; Appel and Shao, 1996]. The schemes that we found to perform well with respect to copying cost also share the property that they *do not change* the relative order of objects; think of them as compacting. Therefore, they do not adversely affect the locality of access within the mutator. Unlike traditional generational collection, which repeatedly collects from the same memory region of young objects and *occasionally* the whole heap, some of the proposed schemes visit different parts of the heap more regularly. Although it is tempting to predict the memory hierarchy behavior of these schemes by observing the patterns of collected region motion, the complexity of interaction of allocation, copying and pointer maintenance

[5]This scenario is made unlikely by the fact that an unreclaimed cycle of garbage objects migrates to and concentrates in the oldest end of the heap. There its adverse effect on collector performance is indistinguishable from that of truly live but permanent objects (Section 5.4.3).

accesses with the memory hierarchy requires that the behavior should be measured in a fully implemented system instead.

3.2 Implementation

In preceding sections, we outlined several age-based garbage collection algorithms that we use to explore reducing copying cost beyond generational youngest-first collectors, by prescribing which portions of the heap should constitute the collected region. The following chapter will show that the promise of the DOF collector is actually delivered, but now we consider how to implement in practice a collector with a collected region that ranges outside the space of youngest objects.

If we look at all the age-based schemes together, we find two requirements for implementation. First, it must be possible to identify any set of objects according to their age, and use it as the collected region. Second, it must be possible to enumerate pointers into the collected set from outside, in order to collect it correctly.

3.2.1 Blocks

The first requirement can be fulfilled efficiently by dividing the heap into blocks and linking blocks into an age-ordered list (Figure 3.13). The collected region must consist of a number of blocks. This restriction on the freedom to choose the collected region is acceptable: constant overheads of collection will surely render too expensive any scheme that chooses extremely small collected regions. Therefore, imposing a block as the minimum size of the collected region, as well as the measure of granularity of window positioning, must be appropriate for some block size.

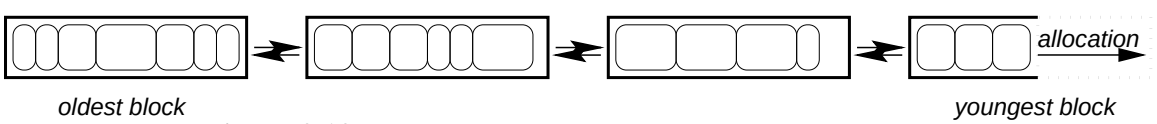


Figure 3.13. Heap organized as an age-ordered list of blocks.

Hence, the heap address space is divided into blocks. Each block consists of 2^b bytes, and is aligned on a 2^b-byte boundary. Assume for the time being that no object is larger than a block. Each object is entirely contained within one block.

The block table is an auxiliary data structure, an array that lists all the blocks available in the address space and records the status of each (free, or in use). Additional information for blocks in use is accessed indirectly, to avoid the space overhead for unused blocks.

Given the address of a heap object, or a field within an object, a constant b-bit-shift operation calculates an index into the block table corresponding to the block in which the object lies.

Blocks in use are doubly linked in age order, so that age-based collectors can easily follow their collection policies. Each block records the next free address within it, so that objects can be added to it: by the mutator in allocation, or by the garbage collection in survivor copying. During collection, each block records whether it is in the collected region or not. During the Cheney scan phase of garbage collection, a block also records the next address to be scanned. In generational schemes, the block additionally records the number of the generation to which it belongs.

The stipulation that whole blocks are subject to collection raises concern about possible fragmentation: if the survivors from an entire collected region must be placed in blocks themselves, then we can expect, on average, half a block of wasted space. The situation is easily remedied for the DOF collector, if the collector promotes survivors into any available space in the adjacent block to the left (youngest among uncollected blocks older than the collected region), as shown in Figure 3.14. Because of the rightward sweep of the collected region, any incomplete block left at the the end of one collection is filled in subsequent collections, and fragmentation is eliminated.

Objects larger than one block are allocated in a special area, the *large object space*, but each large object is logically assigned to a block. The collector never physically copies large objects, but rather logically moves them from a particular block to another by manipulating

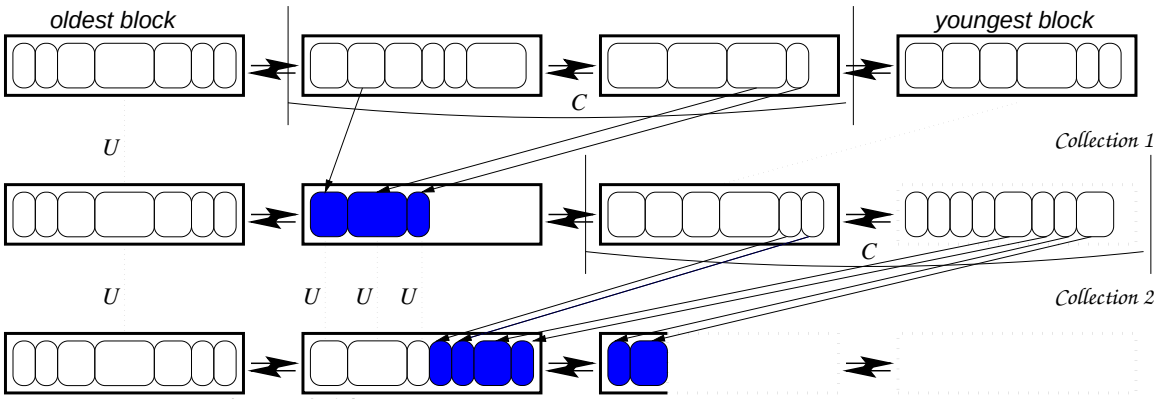


Figure 3.14. Filling preceding block to avoid fragmentation.

efficient indexing data structures. The implementation of large object space can be adapted from the one in the GC Toolkit [Hudson *et al.*, 1991]. The handling of objects in large object space is more expensive than of those in ordinary blocks [Hicks *et al.*, 1998]; an examination of the size distributions of objects (Section 4.2.3) shows, however, that large objects are a rare occurrence for our benchmarks.

3.2.2 Remembered sets

The second requirement is to be able to find pointers into the collected region, no matter which blocks, or how many blocks, are in the region. The most general solution is to track the set of pointers into each block (the remembered set), and then compute the union of the sets for all blocks in the collected region.

In contrast, note that a garbage collection algorithm that defines fixed boundaries for the collected region can safely record just the pointers that cross that boundary. For instance, a generational collector maintains one remembered set per generation, no matter how large the generation is. The need to maintain remembered sets for each block can cause significantly larger space overhead, as well as time overhead for recording and processing the pointers.

The generational collector does not even maintain truthful remembered sets for each generation: by design, a collection of generation *n* always collects all younger generations as well,

therefore it is only necessary to know the pointers into generation n from *older* generations.[6] As we shall see in Section 6.1.3, this structure greatly reduces the number of pointers that must be tracked.

Therefore, while the design using per-block remembered sets achieves the desired functionality for the fully-general age-based schemes, we must also consider the implications of the particular age-based scheme on the sets of pointer stores that the system can safely ignore.

Note that the majority of stored pointers have the source and target within the same block, as we shall see in Section 6.2. Since a block is always entirely within the collected region, or entirely outside it, such local pointers do not need to be recorded in the remembered sets. This optimization applies to all age-based schemes. It eliminates a majority of pointer stores for reasonable block sizes at very low cost.

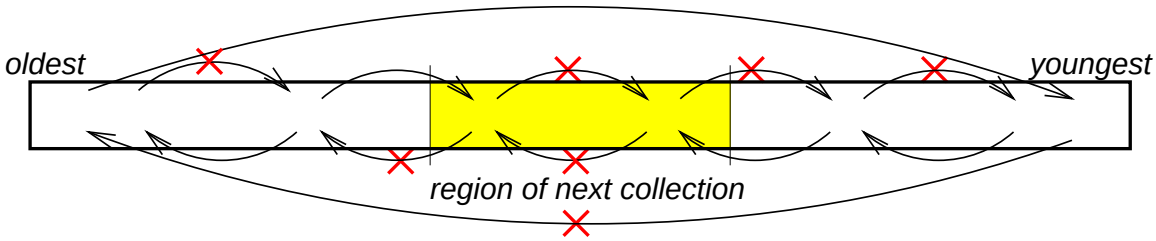


Figure 3.15. Directional filtering of pointer stores at run-time (DOF).

A more aggressive elimination criterion is possible. Suppose that the stored pointer is $p \rightarrow q$, and that the object p is subject to collection *before* object q. Then the pointer will be invalidated before q is collected, either because p is moved if live, or because p is dead. Hence the pointer $p \rightarrow q$ should not be recorded in the remembered set of q. Thus the general rule is to filter out those stores for which the source will be collected before the target, or at the same time as the target. Some age-based schemes have strict policies for the positioning of the collection window in the heap, and it is therefore possible to establish whether the source or the target of a pointer store will be collected first. Among them is DOF, with the full, cyclic,

[6] Note as a special case that in the two-generation collector the (single) remembered set is always empty following a collection, thus the write barrier at collection time can be elided.

sweep of the heap. The directional filtering for the DOF scheme is shown in Figure 3.15 for pointer stores in the mutator.

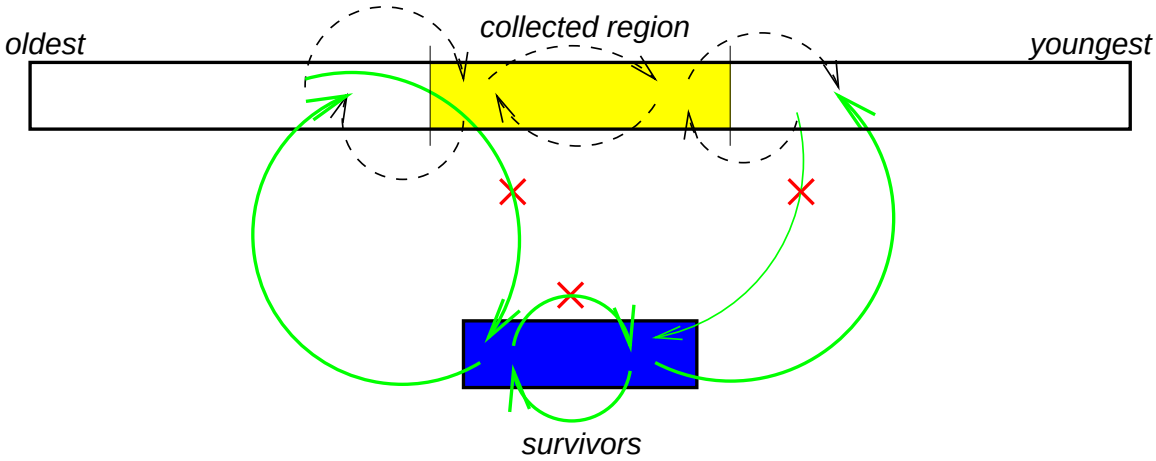


Figure 3.16. Directional filtering of pointer stores at collection time (DOF).

Distinguishing the region that will be collected next, and the regions older and younger than it, there are 12 cases of pointer stores at run-time. One half are eliminated by observing that the source will be collected before the target, and additionally, the case with both source and target within the region of the following collection is eliminated.

In Figure 3.16 we show the filtering scheme for pointer stores during collection. The dashed lines indicate the pointers as they existed before the collection, and the full lines indicate the pointers established during the collection, as survivor objects are copied into to-space. Here the filter eliminates 3 out of 6 cases. The actual benefits of directional filtering depend on the distribution of different pointer store directions, and we shall return to this question in Section 6.2.

In the design of remembered sets for the newly proposed collectors, there is an additional consideration that did not arise for generational collectors. Namely, when a collection copies objects from the collected set C into the survivors set S, then any pointers from C into the uncollected remainder U, recorded in the remembered sets of the blocks of U, must be discarded prior to collection. Otherwise, the remembered sets would contain entries for source addresses in the from-space of the collection, which, following the collection, are invalid. The

collection then reestablishes remembered set entries for the surviving pointers from S into U. Therefore, the data structure for remembered sets must support deletion efficiently. (In generational collection, the pointers from C to U are younger-to-older, hence they are not recorded in remembered sets to begin with.)

3.2.3 Write barrier

We now consider how to construct remembered sets. In between collections, new pointers come into being by means of initialization of pointer-typed fields of newly allocated objects, and by updates of these fields in older objects. Pointer updates are tracked using a write-barrier mechanism. As for pointer initialization, either the initializing stores are tracked using the same write-barrier mechanism, as it is, in effect, done in the Smalltalk system, or the newly allocated area is scanned for pointers at garbage collection time.

We consider two write-barrier mechanisms. The first write-barrier mechanism immediately inserts stores into remembered sets. When a store occurs in the mutator, the following filtering actions occur. First, it is established that the store is of a pointer as opposed to a non-pointer value. In dynamically typed languages such as Smalltalk, this check is at run-time. In statically typed languages, such as Java and Modula-3, the write barrier code is only emitted for pointer stores, so this check disappears. Second, the write barrier verifies that the stored pointer value (the target) is not nil, since such stores do not create new pointers. Third, the write barrier verifies that the pointer crosses block boundaries. This check involves the comparison for equality of the high-order bits (full address length less b) of the pointer source and target. Fourth, if the particular age-based scheme permits further elimination of pointers according to the age of the source and target, then the write barrier may eliminate some additional stores. Fifth and last, the pointer source is inserted into the remembered set of the block corresponding to the target.

The second write-barrier mechanism is a hybrid of *card marking* and remembered sets [Hosking and Hudson, 1993]. The heap is logically divided into cards of size 2^c bytes, where

$c \leqslant b$. Experience with generational collectors suggests that cards are typically much smaller than blocks: e.g., while blocks of 64 KB were used in a Smalltalk implementation, the most efficient card sizes were in the range 256 bytes – 1 KB [Hosking *et al.*, 1992; Hosking and Hudson, 1993]. A card table is an array with an entry for each card of the heap. At the end of a collection, all entries of the card table are *clean*. In between collections, at every pointer update (the store of a pointer $p \rightarrow q$) the write barrier marks the entry for the card containing p as *dirty*. At the beginning of each collection, the collector scans the card table to find dirtied cards. For each dirtied card table entry, it then scans the corresponding card in memory, i.e., it examines all pointer-valued fields in all the objects in the card, and updates appropriate remembered sets—in our example, it is the remembered set of the block containing the address q. When cards are examined and summarized into remembered sets, the same filtering criteria apply as discussed above for the case of immediate insertion into remembered sets. One benefit of card marking is that *duplicate* entries are immediately eliminated. The price is paid instead in scanning the cards, since the location of the updated pointer source is recorded only imprecisely. (When scanning a card, the collector may examine many more pointers, depending on the size of the card, in addition to the one (or ones) that caused the card table entry to be dirtied.) We shall see, however, that there are few duplicates *within* a block (Section 6.1.3), thus card marking is not likely to be profitable.

3.2.4 A write barrier design for large address spaces

Returning to the write-barrier mechanism without card marking, we observe that it can be implemented especially efficiently if a large virtual address space is available. The goal is to make the sequence of write-barrier instructions short for the common case, namely that of stores that the barrier does *not* record. A large address space allows a heap organization that avoids the need to index into a block table and is able to make the pointer update filtering decisions solely from the two addresses of pointer source and target, which are already available in hardware registers for the store itself.

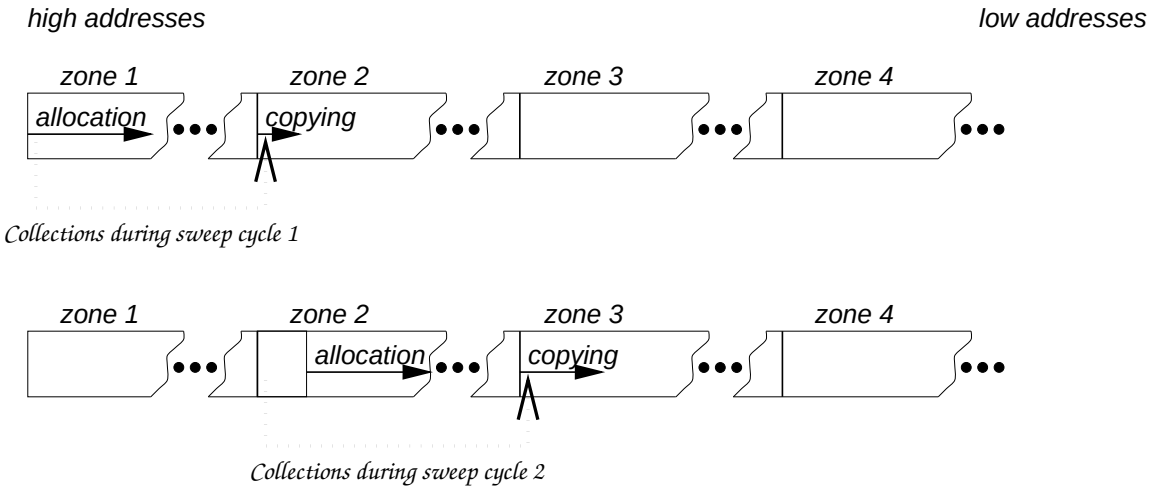


Figure 3.17. DOF heap organization in a large address space.

We illustrate the heap organization in Figure 3.17.[7] The address space is divided into zones, each zone consisting of a large number of blocks. At any moment exactly two adjacent zones are in use for the heap. Initially, all allocation is into zone 1, at the high end of the address space, whereas all copying by the collector is into the next lower zone 2. Both allocation and copying proceed from higher to older addresses within a zone. As soon as the amount allocated in zone 1 becomes equal to the specified heap size (expressed as a number of blocks), garbage collection begins in the DOF manner, with the collection window sliding over zone 1, from high addresses to low. The survivors of collection are copied over into zone 2. Subsequent collections are triggered when the total amount in the heap (i.e., in zones 1 and 2) again equals the specified heap size. This process continues until either: (a) the window of collection catches up with allocation in zone 1, or (b) the allocation in zone 1 reaches the limit, i.e., runs into zone 2. Case (a) corresponds to the natural end of the window sweep of the DOF collector (Figure 3.9). Case (b) is coerced into case (a) by enlarging the window to encompass the entire remainder of zone 1. Thus in both cases a full sweep of all objects allocated in zone 1 is completed. At this point, which corresponds to the resetting of the window in Figure 3.9, zone

[7]Note that the address-ordered layout of Figure 3.17 is logically equivalent to the age-ordered layout of Figure 3.9.

1 is abandoned, and its role is assumed by zone 2: allocation proceeds in zone 2, from the last copied object on towards lower addresses. Zone 3 is introduced for the survivors of subsequent collections.[8] In this fashion, allocation and collection can proceed until all zones have been exhausted. Under the assumption that we have a large virtual address space, most programs will not exhaust all available zones; if that does occur, however, the remedy is to move the data back into the high addresses, at the price of one non-generational collection (or relocation).

Choosing a good size for the zones is challenging. On the one hand, a zone should accommodate the amount allocated during one sweep of the collection window: if it does not, then the premature end of sweep will be forced (case (b) above), which will result in a longer-than-normal pause time and disturb the desirable operation regime of the DOF collector. On the other hand, when the DOF collector is in its most desirable regime, Figure 3.10, then the window moves slowly, and the amount allocated during a sweep is large. Therefore, since both the number of zones, and the size of a zone must be large, a large virtual address space is truly necessary for this organization to be effective.

The directional filtering at run-time, shown in the age-order layout in Figure 3.15, becomes very simple in this address layout, as shown in Figure 3.18: if the address of the pointer source is higher than the address of the target, the pointer store need not be recorded. Thus the directional filter can be implemented in a single comparison instruction. Note that the store of a null pointer, normally represented as a machine 0, is also caught by this filter, hence a separate prior check is not needed. The block-local filter remains as described above, but it can be coalesced with the directional check: one of the values compared in the directional check is first rounded to the nearest block boundary. In all, the filters require only a handful of instructions (a bit-masking operation, a comparison, and a conditional branch) and involve no memory accesses (Figure 3.20, p. 48, on the left).

[8]The operating system must cooperate by allowing the garbage collector to acquire and explicitly dispose of segments of the address space.

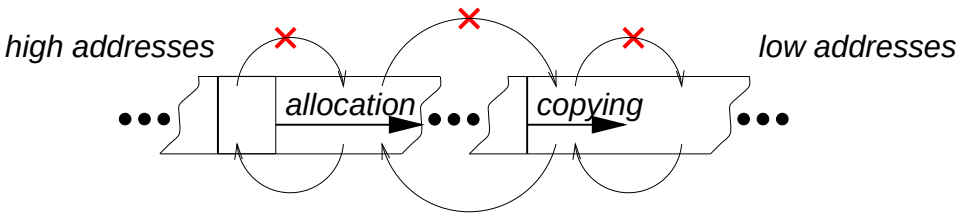


Figure 3.18. Directional filtering for DOF in a large address space.

Thus, the majority of pointer stores, those that do not need to be recorded, are handled quickly. We now consider the remainder. We have observed (as we report in Section 6.1.3) that there are very few duplicate pointer stores for the remembered set of a block in general, and, in particular, between two collections. Hence card marking is not advisable. However, if we wish to keep the cost of the write barrier predictable and low, we should not immediately insert the pointer store, that is, the address of the pointer source, into the remembered set of the pointer target. Instead we can write the same address into a sequential store buffer [Hudson *et al.*, 1991], and subsequently flush the store buffer into the remembered set. There will be a sequential store buffer for each block in current use.

At collection time, we form the set of remembered pointers into the collected region by considering the "union" of the remembered sets of all blocks in the region. We chose to maintain the remembered set of a block between collections as a sequential store buffer; now, rather than construct the remember set explicitly and then compute the union, we simply merge the store lists for the blocks involved, and eliminate pointers that go between two blocks both in the collected region.[9]

Recall that when remembered sets are explicitly maintained, it is necessary to purge the sets of uncollected blocks of entries for pointers from collected blocks, prior to collection. With the address layout proposed here, together with directional filtering, that is no longer necessary,

[9]Note that each store buffer is likely to be a nearly sorted list (with decreasing addresses), because most pointer stores are into the newly allocated area, and thus the sequence of pointer source addresses is correlated with the direction of allocation. This circumstance may influence the choice of the merge algorithm. Also note that an actual merge operation is not absolutely required: it is possible, and it may be less expensive, to recognize duplicate entries on the fly.

since entries cannot become stale: the collected region lies within the highest valid addresses, and no pointers from higher, now invalid addresses, were recorded in the remembered sets for the collected region, since such pointers would have been higher-to-lower address pointers.

The directional filtering described here is cheaper than the traditional generational test [Hosking *et al.*, 1992], which requires indexing into a block table to determine the generations of the pointer source block and the pointer target block, at the cost of two memory accesses. For fairness, however, under the assumption of a large address space, the generational test can also be simplified. We divide the address space into a number of very large blocks, and assign each generation to one block. We place the nursery at the highest addresses, and older generations at progressively lower addresses. With this heap layout, an address comparison acts as a generational test to eliminate younger-to-older stores, and simultaneously handles the null pointer store; a block-local filter is needed to eliminate older-to-younger stores within a generation.

Thus the write barrier cost components—the cost incurred for each pointer store and the additional cost incurred for each pointer store that must be recorded—will be comparable in implementations of the collectors studied. Similarly, the costs incurred at the time of processing each remembered set entry will be comparable. Therefore, in this study we focus on measuring the number of times different collectors invoke each of these operations. We find significant differences between the collectors. Chapter 6 describes these results in detail.

3.2.5 An estimation of copying and pointer-maintenance costs

In subsequent chapters we shall obtain accurate operation counts for the copying and pointer-maintenance actions of garbage collection. If in addition to the number of operations, we know the cost of each operation, we can calculate total collection costs, and by comparing with an actual implementation, validate the cost model.

We performed preliminary experiments with the proposed write barrier on a 292 MHz Alpha 21164 processor. The implementation is presented in Figure 3.19 as source code and in

Figure 3.20 as Alpha assembly code compiled using `gcc -O2`. The null-pointer, block-local, and directional filtering are accomplished using a single test. The estimated overhead of the barrier if the test eliminates the store is 2 cycles.[10] Stores that are not eliminated are recorded, without duplicate removal, in a sequential store list corresponding to the target block. A sequential store list holds a maximum of 15 entries. When a store list fills up, a new one is allocated from a large linear arena; thus each block owns a linked list of sequential store lists. The measured overhead of the barrier, including the writing into a sequential store list and management of sequential store lists, is 11 cycles per store.

Figures 3.21 and 3.22 present the code executed at garbage collection time to scan the list of sequential store lists of a block. The collector retrieves each entry, interprets it as a pointer source, and then reads the current target value of this pointer. (The target may have changed since the time when this store was recorded, so the collector verifies that the current target is within the collected region.) Finally, the collector retrieves the header word of the target object. If the header word is in fact a forwarding pointer, then the object need not, and must not, be copied again. If the header word is a true header, then this is the first time the collector has encountered the object in this collection, and therefore the collector copies the object into to-space. (A forwarding pointer is an object address and its least significant bit is 0, because objects are word-aligned, while addresses are expressed in bytes. We can define a true header to have 1 in its least significant bit to make headers and forwarding pointers distinguishable.) The cost of this entire sequence, sans copying, is 13 cycles per processed entry.

Thus, assuming that filtering succeeds in eliminating 95% of pointer stores on average, and assuming that all recorded stores are eventually processed, the average cost per pointer store is estimated at $0.95 \cdot 2 + 0.05 \cdot (11 + 13) = 3.1$ cycles.

[10]The measured overhead on the four-way issue Alpha 21164 processor ranges from 1 to 3 cycles for the three instructions in the test, depending on the surrounding code; the average and median of 2 is used in the absence of any assumptions about the surrounding mutator code.

In addition to the break-down of pointer-maintenance costs, we performed an experiment to measure the components of copying cost in detail. Whereas in the aggregate, the copying cost is roughly proportional to the amount copied, here we examine the contribution of individual operations to that cost. Our model assumes a copying collector, in which a first phase of root processing (processing of global roots and remembered sets) is followed by a Cheney scan of to-space. For each object copied, the collector pays the following three costs: the cost of processing the pointer that keeps the object reachable, the overhead of installing a forwarding pointer and preparation for copying the words of the object, and the overhead of preparation for scanning the pointer fields of the object. Together these three are the per-object cost α_{obj}. For each word copied, there is a cost α_{word}. Finally, in the scanning phase there is an additional cost for each scanned pointer that does not cause an object to be copied (whereas scanning a pointer that does cause an object to be copied is subsumed in α_{obj}). There are three circumstances when this happens: the pointer is null; the pointer points outside the collected region; or the pointer points to an object that has already been copied. In our large address space model, these three circumstances are identified as follows: category *null/lower* encompasses null pointers and pointers to addresses below the collected region (addresses above the collected region are not possible at collection time), and category *duplicate* encompasses pointers into the collected region to objects that have already been copied. The per-pointer-field costs for these two categories are $\alpha_{null/lower}$ and $\alpha_{duplicate}$, respectively. The full copying cost is then

$$c = \alpha_{obj} n_{obj} + \alpha_{word} n_{word} + \alpha_{null/lower} n_{null/lower} + \alpha_{duplicate} n_{duplicate}.$$

We measured the operation costs α in a synthetic copying experiment, in which heaps of different size, and with different object sizes and pointer contents were constructed and copied by a collector. We used an object format applicable to a typed language such as Java: an object header word points to a class record, which points to class-specialized methods to copy and scan the objects. We estimate the following values for the operation costs on the Alpha 21164, for operation within the primary cache: $\alpha_{obj} = 65$ cycles, $\alpha_{word} = 2.5$ cycles,

$\alpha_{null/lower} = 15$ cycles, and $\alpha_{duplicate} = 17$ cycles. These estimates could be used in conjunction with an instrumented garbage collector to validate the detailed model of copying cost.

One must be cautious with these numbers for processor cycles spent on particular operations. They were obtained in a tight loop, and may not reflect actual costs accurately. On the one hand, the cost of memory accesses is discounted, since block table entries are cache-resident. On the other hand, cycle counts may be overestimated, since the processor has no opportunity to interleave the write barrier instructions with other mutator instructions.

3.3 Summary

The proposed class of age-based collectors represents a logical generalization of the concept of generational collection. Age is the simplest criterion for choosing the collection region, and there are established techniques for encoding object age with sufficient accuracy and reasonable overhead; we have also suggested new techniques applicable to large address spaces that promise to reduce overheads further even in the more complex context of age-based collection.

The collectors as described are all of the stop-the-world variety. No explicit provisions for incrementality are suggested. However, we believe that collectors with a small fixed size collected region can provide sufficiently low maximum pause times that they can be considered truly incremental (even if they have to do so at the price of increased total time).

We described the collectors as operating within a unified heap, in which all objects lie. Hence, the same basic policy is applied to newly allocated objects as to older objects, and even long-lived semi-permanent objects. Surely the motivation that has led researchers in the past to consider dividing the heap into ephemeral and mature spaces remains valid, and different age-based schemes could be applied within each division.

We described simple oldest-first, youngest-first, and moving window schemes (Figure 3.23 summarizes the employed abbreviations). The space of other possible designs is vast, and

```
#define BLOCK_SHIFT 16
#define BLOCK_BYTES (1 << BLOCK_SHIFT)
#define BLOCK_MASK (~(BLOCK_BYTES - 1))
#define BLOCK_ADDR(addr) (addr & BLOCK_MASK)
#define SB_SHIFT 7
#define SB_BYTES (1 << SB_SHIFT)
#define SB_WORDS (1 << (SB_SHIFT-3))
#define SB_MASKLOW (SB_BYTES - 1)
typedef struct bte
{
      long int ** store_buffer_pointer; /* where to insert the next entry */
      ... other block information ...
} BTE;
#define BTE_SHIFT (BLOCK_SHIFT - log2 (sizeof(BTE)))
/* the store: */
*reg_source = reg_target;
/* the write barrier: */
if (reg_source < BLOCK_ADDR(reg_target)
{
     /* record the pointer: */
     btep = block_table_offset + (BLOCK_ADDR(reg_target)>>BTE_SHIFT);
     p = btep→store_buffer_pointer;
     p --;
     *p = reg_source;
     if ((p & SB_MASKLOW) == 0)
     {
          /* allocate new store list, set p to beginning of list: */
          previous_ssb_arena_pointer = ssb_arena_pointer;
          ssb_arena_pointer -= SB_WORDS;
          if (ssb_arena_pointer < ssb_arena_low_limit)
               {
               ... allocate new arena ...
               }
          *ssb_arena_pointer = previous_ssb_arena_pointer;
          p = ssb_arena_pointer;
     }
     btep→store_buffer_pointer = p;
}
```

Figure 3.19. Write barrier code.

```
register usage:
$0 is reg_target
$2 is BLOCK_ADDR(reg_target)
$2 is also p
$2 is also previous_ssb_arena_pointer
$3 is btep
$5 is block_table_offset
$10 is ssb_arena_pointer
$11 is reg_source
$13 is ssb_arena_low_limit

the store and the write barrier:
        zapnot $0,252,$2 ;; clear low 16 bits
        cmplt $11,$2,$1
        stq $0,0($11)
        bne $1,$42
$43: continuation of mutator
```

```
$42: record the pointer:
        sra $2,13,$1 ;; >>BTE_SHIFT
        addq $5,$1,$3
        ldq $2,0($3)
        subq $2,8,$2
        and $2,127,$1
        stq $11,0($2)
        bne $1,$46
        allocate new store list:
        bis $10,$10,$2
        subq $10,128,$10
        cmple $13,$10,$1
        ...conditional branch to allocate
        new arena (not shown)...
        stq $2,0($10)
        bis $10,$10,$2
$46:
        stq $2,0($3)
        br $31,$43
```

Figure 3.20. Write barrier assembly code.

may offer potential for better performance; for example, hybrid schemes can be envisaged to combine generational collectors' reserve space management with sweeping windows.

```
#define SB_SHIFT 7
#define SB_BYTES (1 << SB_SHIFT)
#define SB_WORDS (1 << (SB_SHIFT-3))
#define SB_MASKLOW (SB_BYTES - 1)
#define SB_CHUNK_ENTRIES (SB_WORDS-1);
#define SB_LINKMASK_VALUE (SB_CHUNK_ENTRIES*8);
#define FWD_POINTER_MASK 1
#define FWD_POINTER_TAG 0
p = actual_block_table [1] . store_buffer_pointer;
while (p != 0)
{
    while ((p & SB_MASKLOW) != SB_LINKMASK_VALUE)
    {
        reg_source = *p;
        reg_target = *reg_source;
        if (reg_target > reg_collection_window_low)
        {
            /* target is in the collected region */
            if ((*reg_target & FWD_POINTER_MASK) != FWD_POINTER_TAG)
            {
                /* the word in the object header is not a forwarding pointer,
                 because it does have the low bit set */
                reg_target_new = ...new address of target object after copying...;
                *reg_source = reg_target_new;
            }
        }
        p++;
    }
    p = *p;
}
```

Figure 3.21. Remembered set processing code.

```
register usage:
$2 is p
$11 is reg_source
$10 is reg_target
$14 is reg_collection_window_low

        lda $5,actual_block_table
$60:
        ldq $2,8($5)
        beq $2,$59
        br $31,$73
$66:
        ldq $11,0($2) ;; reg_source = *p
        ldq $10,0($11)
        cmpule $10,$14,$1 ;; target check
        bne $1,$67
        ldl $1,0($10) ;; load header word
        blbc $1,$67 ;; branch if low bit is 0
        ... forward pointer (not shown); new address is in $0 ...
        stq $0,0($11)
$67:
        addq $2,8,$2 ;; p++
$73:
        and $2,127,$1
        cmpeq $1,120,$1
        beq $1,$66
        ldq $2,0($2) ;; p = *p
        bne $2,$73
$59:
```

Figure 3.22. Remembered set processing assembly code.

- NONGEN: non-generational collector
- TYF: true youngest-first collector
- TOF: true oldest-first collector
- ROF: oldest-first collector with the renewal-age modification
- DOF: old-to-young sweeping collector (deferred-older-first)
- DYF: young-to-old sweeping collector (deferred-younger-first)
- OPT: locally-optimal collector
- FC: prefix for age-based schemes with a collection region of fixed size
- GYF: generational collectors in general
- 2GYF, 3GYF: generational collectors with generations of fixed size, with two and three generations, respectively
- NGYF: generational collector with two variable-sized generations

Figure 3.23. Age-based scheme name abbreviations.

CHAPTER 4

EVALUATION: SETTING

The evaluation of garbage-collection algorithms requires us to pose a large number of questions in order to expose the strengths and weaknesses of algorithms in different circumstances. We developed a number of different experimental methods to answer them and we present these methods in detail here and in the following two chapters: the discussion of each method is immediately followed by a presentation of the experimental results.

The three main approaches to a comparative performance evaluation of garbage collector performance, offering different perspectives, are analytical modelling, trace-based simulation, and prototype implementation. *Analytical modelling* consists of constructing a mathematical description of the operation of the garbage collection algorithm, and applying it to a mathematical description of the load, i.e., the allocation and lifetime properties of objects and pointers. When successful, analytical modelling provides succinct models, and has useful predictive power with respect to the choice of collector configuration. Unfortunately, the premise of having reliable realistic mathematical descriptions of lifetime properties of objects, and especially properties of pointers, is not easily met with current knowledge.

Trace-based simulation consists of constructing a software simulation of the operation of the garbage collection algorithm, and applying it to an empirically obtained trace of the load, i.e., the sequence of allocations and other object-related operations of a real program. Simulation isolates the memory management tasks from other (computational) tasks that a real program performs, and allows a variety of configurations to be tried. It is even possible to try collection algorithms that would be excluded in practice, such as those using backtracking, or oracles, in their decisions. The analysis of simulation results provides explanatory and predic-

tive power, but is not as succinct as analytical modelling. It is usually not difficult to extract traces when a working garbage collector is given, thus one can reasonably obtain traces from more than one object system and analyze them within the same simulation framework.

Prototype implementation consists of constructing complete garbage collection algorithms and implanting them in a working object-based system, in order to measure how each performs in operation. Implementation has the advantage that it accounts for the full cost of memory management; on the other hand, it is difficult to extract the contribution of each cost factor. Furthermore, that cost is necessarily a reflection of a number of implementation decisions. From a practical standpoint, it is a usually a difficult task to retrofit (a number of) garbage collectors onto a working system.

Our study primarily uses trace-based simulation using specially built object-level simulators. A prototype implementation was also completed according to the design of Section 3.2, and integrated with a trace simulator, which allows us to gather many of the performance statistics as in a working object system.

In the remainder of this chapter, we discuss our experimental setting: the design of the garbage collection simulators and the set of benchmark programs used. The following two chapters present the findings: Chapter 5 examines the copying cost of collection, and Chapter 6 examines non-copying costs of collection, viz., the cost of collector invocation, and the pointer management costs.

4.1 Evaluation using program traces

The primary method used in this study is evaluation by trace-based simulation. The experiment proceeds in two phases. First, by instrumenting an object-based system and executing a program, a trace is produced. Second, the trace is used to direct the execution of different garbage collection algorithms and to derive statistical measures of the program.

4.1.1 Trace content

The information contained in the trace must permit accurate simulation of each garbage collection algorithm. To the point, the trace must identify all objects that the program allocated in the heap, providing for each object its time of allocation, its time of demise, and its size.[1] Furthermore, all stores of pointer values into the objects in the heap must be present in the trace.

This information is sufficient for a garbage collector simulator to determine which objects are reachable in any chosen collection area at any chosen time. A real collector first identifies global roots by scanning the stack, and then computes the transitive closure of the points-to relation to find the reachable, hence live, objects. As it will be described below, the simulated collector relies instead on accurate allocation and demise records: all objects for which an allocation record has been seen but not a demise record, are live. In addition, the trace allows the collector to maintain an accurate image of the pointers among heap objects. In particular, the pointers from the uncollected region into the collected region are accurately known. Thus the closure of the points-to relation can be computed in the simulator to complete the set of collection survivors.

4.1.2 Trace format

The common format for the traces is as follows. Each trace is a text file, in which each line records an event. Three kinds of events are represented: object allocation (A-records), object demise (D-records), and field update (U-records). An allocation record has the form:

`A` *allocation-address* *size*

where *allocation-address* is the hexadecimal byte-address of the starting word of the object, and size is the number of words in the object. A demise record has the form:

---

[1] These are the properties meaningful in *every* object-based system. In some systems it may be worthwhile to report more detailed information, when it is defined in the language at hand, such as the type (or class) of the object, or the allocation point. However, since age-based garbage collection algorithms do not make use of such information, we shall assume that only the general properties, applicable to all object-based systems, are recorded.

`D` *allocation-address*

where *allocation-address* is the hexadecimal byte-address of the starting word of the object. An update record has the form:

`U` *field-address* *value-address*

where *field-address* is the hexadecimal byte-address of the field which is being updated, and *value-address* is the pointer value stored. (The special value -1 represents the store of a *null pointer*.) The form of the traces reflects the assumption that an object is a contiguous sequence of fields, each field being an integer number of words. A pointer-valued field takes one word. A word is the smallest quantum of allocated space, and therefore all sizes and amounts are reported in words. In the systems studied (Smalltalk and Java 32-bit implementations), a word is four bytes long. Another assumption implicit in the trace format is that the location of an object is immutable. This assumption was made for simplicity: rather than have a separate notion of object identity, the address (whether allocation address or a field address within the object) uniquely identifies the object. Without loss of generality, it may also be specified that allocation addresses of objects in the trace follow consecutively, as if the objects were allocated from an infinite allocation area starting at zero.

4.1.3 Trace generation

Having described the content of the trace, let us consider how to generate a trace. For this study, the method is to augment an existing object-based system and its garbage collector with code to produce the required trace records. The existing system is assumed to have the functionality to identify each object allocation, object demise, and pointer store.

Every object-based system can identify object allocations.

Every system with a garbage collector can also identify object demise. However, if we rely on the garbage collector existing in the system (often a state-of-the-art generational collector), we must ensure that the collections consider the entire heap, as in a non-generational collector,

to avoid the inaccuracies of excess retention (Section 5.3).[2] Following each such full collection, the reachability of objects is accurately known; any objects that are now not reachable, but were reachable at the previous full collection have died in the intervening period. If the time of demise must be known with accuracy, the period between full collections must be short. In fact, it is critical for the correctness of *simulated* collections to be able to determine reachability at every possible garbage collection instant (in simulation), i.e., *following every object allocation* (in simulation). Therefore, a full collection (in trace generation) must also follow every object allocation. As a result, the trace generation mechanism is very expensive: full collections are expensive, and many are needed. In fact, the cost is roughly proportional to the integral $\int_0^T v(t)dt$ of the live data amount over the duration of the program.[3]

Update records, the final component of the trace, are easily generated by instrumenting the write barrier in a system with a generational collector, since the collector must be informed of pointer stores. One must only ensure that in the existing system the write barrier is indeed invoked for each pointer store, that is, the system does not optimize away some pointer stores (e.g., initializing stores or null pointer stores), which it is allowed to do for operation of a generational collector, but not for trace generation.

Let us consider some questions about traces generated in this manner. Any measurement disturbs the system measured, and generating object traces is no exception. If the traced program relies on timing properties, such as by querying a real-time clock, its behavior will be severely disturbed by the heavy overhead of tracing. We do not have such programs among our benchmarks. If an application is multi-threaded, as many interactive Java programs are, and

[2]It is additionally assumed throughout that the collector is accurate (as opposed to conservative). It is certainly possible to obtain traces with a conservative collector, such as the one found in the Kaffe virtual machine for Java, but the utility of such imprecise traces is dubious: the garbage collection schemes in this study are meant to be accurate, not conservative.

[3]Note that the underlying assumption is that the existing object-based system and its garbage collector are to be exploited with minimally intrusive modifications—as dictated by technical considerations when dealing with large and complex systems, and also dictated by legal and commercial considerations when systems are not available for inspection and modification in their entirety. If, on the other hand, a system can be redesigned for the express purpose of generating our traces, then better solutions are possible than repeated full collections, in the form of incremental computation of object reachability.

if garbage collection requires synchronization of all threads, then the concurrency behavior of the application will be changed under tracing. The observed order of execution will be a valid order, but not a likely, or the "intended" order.

Finally, recall that in the common trace format the object addresses are immutable. If the tracing object-based system uses moving collection, it is necessary, but easy, to translate object addresses into the absolute form.

4.2 Benchmarks and languages

We first consider the object-based systems ("languages") used in the study, and then the benchmark programs, detailing their general properties.

4.2.1 Languages

We used two object-based systems. The first, **Smalltalk**, is an older purely object-oriented language [Goldberg and Robson, 1983], which consists of three parts: the virtual machine, implemented in an earlier effort in the Object Systems Laboratory [Moss, 1987; Hudson *et al.*, 1991; Hosking *et al.*, 1992], the basic virtual image of Smalltalk-80 from PARC Place/Digitalk, and the additional classes and methods of the benchmark programs. The virtual machine includes the language-independent garbage collector toolkit, also developed in the Object Systems Laboratory [Hudson *et al.*, 1991]. The virtual machine provides the execution (by interpretation) of the bytecodes in the methods contained in the basic virtual image and the benchmark classes.

In the Smalltalk system, all of these classes and methods are present as objects in a benchmark virtual image, which is loaded at program start-up to build an initial heap state. When generating the trace, we distinguish between these objects and objects subsequently allocated as a result of program execution. The main results reported here reflect the program execution objects alone. This choice is justified by the observation that image objects could better be treated as static, instead of as part of the garbage-collected heap [Clinger and Hansen, 1997].

Another property of this implementation of Smalltalk is that method activation records (frames) are internally treated somewhat like objects [Moss, 1987]. However, frames are not treated as objects for the purposes of trace generation.

The second system used, **Java**,[4] is a contemporary object-oriented language that has found much use in Internet-based downloadable applications. It consists of a virtual machine, and a set of basic and benchmark classes and methods, loaded on demand. The Java virtual machine modified to generate traces for this study is an experimental machine, called the ExactVM, developed by Sun Microsystems Laboratories, which is equipped with a garbage collector that satisfies all the requirements outlined above. The Java virtual machine used has the option of running as interpreter, or as a just-in-time compiler. The same objects are allocated, and in the same order, in either case, and pointer updates, and, consequently, object demise points, also happen in the same order—as long as the compilation does not interleave the compiled code from successive bytecodes. Since any differences in behavior, if at all present, are incidental and not material to this study, they can be ignored. The interpreted mode of operation was then chosen for reasons of simplicity.[5]

4.2.2 Benchmarks

The selection of benchmark programs for evaluating garbage collector performance is not an easy task. There is no widely-accepted suite of benchmarks, unlike for general computer performance evaluation, where there are several (e.g., SPEC95). It was therefore necessary to collect and select useful benchmarks from various sources, and it was also necessary to develop the selection criteria.

---

[4] Java is a trademark of Sun Microsystems.

[5] To generate traces using the just-in-time compilation mode, it is necessary to modify the code-generation phase of the compiler so that it will produce appropriate code at object allocations and at pointer stores. Our experience with the Kaffe virtual machines shows this to be a relatively straightforward exercise if the compiler is cleanly written. It is, however, a futile exercise, because the performance advantage of compiled code is entirely annulled by the overhead of trace generation.

4.2.2.1 Collecting benchmarks

Collecting a large preliminary set of benchmarks posed distinct challenges in both languages under consideration. For Smalltalk, although the language is no longer in vogue, a number of public-domain programs (i.e., classes) implementing interesting applications exist and can be obtained on the Internet. Unfortunately, many of these programs depend on the presence of some vendor-specific classes, or are otherwise incompatible with the virtual image used in our Smalltalk-80 system. To the programs we could find, we added some that we have written ourselves and used previously in a study of Smalltalk garbage collection performance [Hosking *et al.*, 1992]. We also translated two SPECfp95 programs from FORTRAN into Smalltalk.

That version incompatibilities should be encountered in Smalltalk, a language which in its present form dates back eighteen years, is perhaps not surprising. However, we found that significant incompatibilities exist even in Java, a much younger language. They are primarily caused by changes in the class library between versions 1.0 and 1.1, assumed by most applications, and version 1.2, which the tracing Java virtual machine uses. A more significant difficulty in collecting Java benchmarks is the fact that the vast majority of publicly available Java code is in the form of Java *applets*, partial programs which execute under the control of a Web-browser and depend on it for the user interface. A browser must contain a Java virtual machine, but that machine is not accessible for instrumentation. We can only use full-fledged Java applications, which run directly on a virtual machine. It is perhaps not a great loss, since the majority of observed applets appear to be graphical widgets and similar toy programs, which neither run long nor allocate much data. On the other hand, the resulting benchmark set is skewed in favor of language-processing tools (compilers and optimizers). Additionally, we had access to the traces (although not the sources) of a preliminary set of Java programs considered for inclusion in the SPEC JVM98 suite.

4.2.2.2 Selecting benchmarks

Having collected a larger set of programs (circa 30 Smalltalk and 15 Java) we considered which objective criteria should decide their utility as benchmarks of garbage collector performance. The first criterion must be that the program exercises the memory management functions at all. For instance, some Java graphics applications are CPU-intensive, but allocate a negligible amount of data, and are of no use in this study.

Even programs that do allocate an appreciable amount of data do not necessarily exercise the garbage collector. Many programs, in particular most of the SPEC JVM98 programs, display a monotone-increasing heap profile: they slowly allocate pieces of some large data structure from start of execution to the end. Since the heap must be at least as large as the maximum amount of live data ever in it (in our cost model and simulation, the heap size is fixed), the program never fills the heap and the garbage collector never runs. Thus, loosely speaking, we need the criterion that the garbage collector runs at all.

In fact, we need a stronger criterion. Consider that, over the execution of the program, the garbage collector will run at different moments depending on the garbage collection algorithm. The amount actually live depends, somewhat unpredictably, upon the precise moment of collection. (Perhaps the most obvious effect is at the end of program execution—since no copying cost is charged past the last collection.) It is thus necessary to reduce the "random" variation in measured collection performance that the "random" collection moments introduce, so as to expose underlying trends. It is therefore desirable to have a large number of collections to even out the "randomness"; in other words, it is desirable to have benchmarks that allocate a very large amount of data relative to their average *live* amount.

We can formalize this requirement by observing that the number of collections performed by a non-generational collector is approximately proportional to the ratio of the amount allocated to the average live amount (see below). We require this number to be more than 10 to admit a measurement. Furthermore, we admit a benchmark only if admissible measurement of non-generational collection is possible for a heap size three times larger than minimum. Thus,

if a benchmark is admitted, the range of admissible heap sizes is at least 1:2, non-generational collector requires more than 10 collections, and our investigated schemes typically require hundreds of collections, so the aforementioned "randomness" is largely avoided.

4.2.2.3 The final benchmark set

Our selection process resulted in a set of 11 Smalltalk benchmarks and 3 Java benchmarks. Their basic properties are listed in Table 4.1. Execution time was measured in normal operation, without any instrumentation and with default generational garbage collector configurations, on a 50 MHz SPARCstation 20 workstation. The amount of data allocated by the program is expressed in words (each word being 4 bytes). The remaining columns give the number of objects allocated, the maximum live amount in words (which is also the minimum required heap size to execute the program), and the number of pointer stores.

The results of non-generational collection for these programs are reported in Figures 4.1–4.10, pp. 71–80 (at the end of this chapter).

Let us take as an example Figure 4.5. In the graph in Figure 4.5(a), we have plotted the absolute mark/cons ratio $\mu_{\text{non-generational}}$ against total heap size V. It is easy to check that the shape of the μ curve closely fits the theoretical prediction, $\mu_{\text{non-generational}} = \frac{1}{\frac{V}{v}-1}$ [Appel, 1987; Jones and Lins, 1996], where v is the effective live data amount. In the graph in Figure 4.5(b), we have plotted the number of garbage collections n against total heap size V. The horizontal dotted line is drawn at $n = 10$, thus the region below the horizontal dotted line (and to the right of the dotted vertical) is the region of heap sizes so large that they result in 10 or fewer non-generational collections. Because of inaccuracies that may be introduced under such conditions, we do not compare results for this region.

We now describe individual benchmarks, providing where possible details of their structure.

Table 4.1. Properties of benchmarks used

Benchmark	Ex. time (s)	Words alloc.	Objects alloc.	Max. live	Ptr. stores
Smalltalk					
Interactive-AllCallsOn	1.91	18 359	1 722	627	855
Interactive-TextEditing	2.30	22 747	3 556	1 098	3 509
StandardNonInteractive	6.11	204 954	46 157	1 160	7 251
HeapSim	67625	3 791 306	1 084 650	93 026	30 298
Lambda-Fact5	6.88	277 940	53 580	6 295	91 877
Lambda-Fact6	55.86	1 216 247	241 864	13 675	404 670
Swim	11.65	1 533 642	444 908	17 542	134 355
Tomcatv	3.92	2 117 849	624 612	30 593	286 032
Tree-Replace-Binary	3.32	209 600	35 785	9 784	39 642
Tree-Replace-Random	4.55	925 236	189 549	13 114	168 513
Richards	10819	4 400 543	652 954	1 498	763 626
Java					
JavaBYTEmark	353.76	1 161 949	109 896	59 728	49 061
Bloat-Bloat	387.54	37 364 458	3 429 007	202 435	4 927 497
Toba	178.05	38 897 724	4 168 057	290 276	3 027 982

- Interactive-AllCallsOn and Interactive-TextEditing. Two tests from the standard sequence specified in the Smalltalk-80 image [Goldberg and Robson, 1983], comprising only the macro-tests. Previously used in Ref. [Hosking *et al.*, 1992].

- StandardNonInteractive. A subset of the standard sequence of tests as specified in the Smalltalk-80 image [Goldberg and Robson, 1983], comprising the tests of basic functionality.

- HeapSim. Program to simulate the behavior of a garbage-collected heap, not unlike the simplest of the tools used in this study. It is however instructed to simulate a heap in which object lifetimes follow a synthetic (exponential) distribution, and consequently the objects of the simulator itself exhibit highly synthetic behavior.

- Lambda-Fact5 and Lambda-Fact6. An untyped lambda-calculus interpreter, evaluating the expressions 5! and 6! in the standard Church numerals encoding [Barendregt, 1984, p.140]. Previously used in Ref. [Hosking *et al.*, 1992]. Both input sizes are used in order to explore the effects of scale.

- Swim. The SPEC95 benchmark `102.swim`, translated into Smalltalk by the author: shallow water model with a square grid.

- Tomcatv. The SPEC95 benchmark `101.tomcatv`, translated into Smalltalk by the author: a mesh-generation program.

- Tree-Replace-Binary. A synthetic program that builds a large binary tree, then repeatedly replaces randomly chosen subtrees at fixed height with newly built subtrees. (This benchmark was named Destroy in Ref. [Hosking *et al.*, 1992; Hosking and Hudson, 1993].) Tree-Replace-Random is a variant which replaces subtrees at randomly chosen heights.

- Richards. The well-known operating-system event-driven simulation benchmark. Previously used in Ref. [Hosking *et al.*, 1992].

Our set of Java programs is as follows:

- JavaBYTEmark. A port of the BYTEmark benchmarks to Java, from the BYTE Magazine Web-site.

- Bloat-Bloat. The program Bloat, version 0.6, [Nystrom, 1998] manipulating the class files from its own distribution.

- Toba. The Java-bytecode-to-C translator Toba working on Pizza [Odersky and Wadler, 1997] class files [Proebsting *et al.*, 1998].

4.2.3 Properties of benchmarks

We now examine some properties of the benchmark programs. We have already seen their behavior under non-generational collection. Before we proceed to their behavior under different age-based collectors, we look at those properties that are *independent* of any collection algorithm. We examine the distribution of object sizes, the distribution of object lifetimes, and the time-varying live amount profile. We shall later refer to these properties when we look for

causes of particular performance behavior, but we shall find that their predictive value is not as great as expected.

We first look at the distribution of object sizes: the typical sizes of objects, and how much sizes deviate from the typical. The size of a *block* must be substantially larger than the typical object size in order to avoid fragmentation. More to the point, the fraction of objects larger than the block size must be small, to minimize the overhead of allocating in the Large Object Space, and the overhead of subsequent collections of objects in Large Object Space [Hicks *et al.*, 1998]. The distribution we measure is a time-averaged property: we do not consider how the numbers of small and large object in the heap change over time.

The size distributions for our set of benchmarks are presented in Figures 4.15– 4.28 (pp. 85–98). The distributions are computed according to four different weighting criteria: (a) by count of objects alone; (b) by volume, which is the object size itself (to reflect the fraction of the heap occupied by objects of a particular size); (c) by lifetime (to take into account that a longer-lived object is more likely to be encountered in collection); and (d) by both volume and lifetime. Each row of a figure corresponds to a weighting criterion. The left column gives the distribution in raw form, as a histogram, whereas the right column gives the cumulative distribution, normalized to 1, so that fractions can be read off easily.

Most programs have several dominant object sizes that constitute most of their allocated volume. Weighting is important: whereas, for instance, 99.5% of allocated objects in Swim are 10 words or smaller, these objects account for 95% weighted by volume, 92.5% weighted by lifetime, and just 50% weighted by both volume and lifetime (Figure 4.21, p. 91). The latter is the most realistic measure of the distribution as seen by a garbage collector,[6] and as we see, if small objects have short lifetimes, their participation in the load can be much less significant than their *numbers* may suggest. We note similar behavior—that smaller objects

[6]We note that age-based collectors that examine less than the whole heap are able to bias the distribution of the objects they see in particular ways with respect to the lifetime, though not with respect to size. Other (non-age-based) collectors are free to *use* size in their allocation decisions.

have shorter lifetimes than larger objects—in all Smalltalk and Java programs except three. The three exceptions are HeapSim, Lambda-Fact5, and Lambda-Fact6, in which a single size represents such an overwhelming fraction of allocation that no differentials can be observed, hence the cumulative distribution curves for different weightings are indistinguishable.[7]

As we have noted, many arguments about garbage collection performance invoke the properties of object lifetime distributions [Baker, 1993; Clinger and Hansen, 1997], sometimes assuming *a priori* distributions in order to derive tractable mathematical models. Even though it is not our goal to develop analytical models, it is proper to evaluate our benchmark set with respect to object lifetime distributions. We shall see that the relationship of the distribution to actual performance is tenuous.

We present the lifetime distributions by means of the survivor function and the mortality function (Section 2.2), shown for our benchmark set in Figures 4.29– 4.42 (pp. 99–103). For each benchmark, the lifetimes of all allocated objects were taken together, weighted by volume (i.e., by object size) and a cumulative distribution was found. Thus, the lifetime distribution is another time-averaged property. Each figure (a) shows the survivor function, plotting object age on a logarithmic scale to show the interesting behavior at young ages more clearly. Each figure (b) shows the mortality function, obtained by numerical differentiation and suitable smoothing from the survivor function. Because of the inherent error of this numerical process, applied to data that are not intrinsically smooth, the mortality curves should be taken *cum grano salis*, as merely indicative of the lifetimes.

The fact that the survivor function is rapidly decreasing, even with a logarithmic scale for the age axis, confirms the observation made many times in the literature, that object lifetimes

[7]The most striking observation is that Java programs have large maximum object sizes, whereas only one Smalltalk program, HeapSim, has objects larger than 140 words. We note that block size in implementation is restricted to be a power-of-two multiple of the virtual memory page size, and that page sizes are of the order 1000–2000 words (8 KB, or 2048 words for Sun Solaris 2.5.1 running on an UltraSparc). Therefore, there is no additional constraint upon block size to make a large object space *unnecessary for these Smalltalk program executions*. The same program may allocate larger objects on a different input. Whether the potential reduction in memory management overhead, by not having a large object space, is significant, and worth the expense of compile-time analysis needed to prove that no large objects are allocated, is beyond the scope of this work.

tend to be short. A few programs exhibit significant fractions of allocated data living to ages close to the duration of the program (Swim, Tomcatv, HeapSim, Tree-Replace-*, Bloat-Bloat, Toba). In others, the survivor function approaches negligible values at an age two orders of magnitude below the duration of the program.

It is tempting to make performance predictions by extrapolating from lifetime distributions. We can speculate that the presence of sharp peaks in the mortality curve (as in StandardNonInteractive, Figure 4.31, and Richards, Figure 4.39) indicates an opportunity for good performance of the FC-DOF collector, provided that the *age* of high mortality translates into a *heap region*, compatible with the collection window, with low copying cost. On the other hand, there is little to distinguish the survivor functions of Lambda-Fact5 (Figure 4.33) and Richards (Figure 4.39), yet we shall see that their behavior under age-based collection is radically different.

In addition to the preceding time-averaged properties, we look at the live profile: the amount of live data in the heap as a function of time. Our traces already express exact liveness information, thus it is easy to produce the live profiles, given in Figures 4.43– 4.56. The live profiles reveal periodic patterns, as in Figures 4.48 and 4.49, which result from iterative or recursive control structure of the program.

4.3 Simulating garbage collection

The primary goal of garbage collection algorithm simulation in this study is to estimate the copying cost of collection. The simulators accurately compute the amount of data copied in collection; this amount is a good estimate of copying cost. In addition, the simulators compute the number of objects copied, the number of garbage collections invoked, and other secondary statistics.

We first describe object-level simulators. They do not model the costs realistically incurred in remembered set maintenance, although they themselves do maintain per-object remembered sets. These costs will be addressed in Section 4.3.3 using block-level simulation based on the prototype implementation. The simulators do not model the locality effects, namely the lo-

cality of access within the collection process, the locality from one collection to another, the locality interference between the mutator and the collector, and lastly, instruction locality. The organization of the simulator is entirely different from that of a realistic garbage collector, thus any timings or locality measurements of the simulated collector are meaningless; moreover, the computational actions of the mutator are not available at all in the trace. These effects can be of interest on modern hardware, and they should be considered in future implementation experiments. However, the primary cost, which is that of copying, is accurately measured by simulation. Simulation has additional benefits, such as the freedom to explore configuration spaces thoroughly and at low cost since computation is eliminated and only memory management is simulated, the freedom to explore new collection algorithms *in abstracto*, and the ability to operate with a common trace format uniformly for various source languages.

The number of garbage collections invoked is easily and accurately obtained by simulation. It is of some importance in evaluating the performance of an algorithm. Each invocation of the collector entails a considerable overhead: the collector must find the global roots of the collection, typically by scanning the execution stack (or multiple stacks in multiple threads of computation). For this reason, the scheme with fewer collector invocations would be preferred, *ceteris paribus*. However, the number of invocations can be viewed from a different angle: the fewer collections there are, the more work is done by each, and the longer the pauses are. In an interactive or real-time setting especially, it is desirable to limit pause times. More precisely, a garbage collection algorithm that imposes an upper bound on the amount of work at each collection may be preferable, in spite of a larger number of collector invocations, to a collector that performs a smaller number of collections, each of longer duration.[8] In summary, it is not appropriate to use the number of garbage collections invoked as a performance measure without reference to the intended application.

[8]Traditional generational collection performs a larger number of collections than non-generational collection, and these collections are on average shorter. However, it *occasionally* collects the entire heap, and thus its upper bound on the amount copied in a collection is not better than with non-generational collection.

4.3.1 General simulation algorithm

The simulation algorithm resembles in its outline the operation of an actual garbage collector (Section 3.1), but differs in important details. We begin with an informal account.

Let us first consider the actions of the simulator while it is simulating the actions of the mutator given in the trace. The simulator sets up a certain, immutable, heap size and an initial allocation area size, and then begins to read the trace. For each `A` record, the allocation area decreases by the size of the allocated object, a running total of volume allocated increases by the same amount, and a data structure is created to represent the object. The simulator maintains a global set of data structures representing all objects present in the heap. A splay-tree data structure [Sleator and Tarjan, 1983] is used for the purpose, because of its good temporal-locality properties. For each `D` record, the simulator marks the corresponding data structure as *known-dead*, but does not discard it immediately. For each `U` record, the simulator first identifies the object in which the updated field is and the object to which the pointer points, and then it records the newly stored pointer in the data structures of the two objects. (The data structure for an object contains both the outgoing pointers and the incoming pointers.)

Eventually, following the allocation of an object, the allocation area will be exhausted, and at that point the simulator begins simulating a collection. The first task is to choose the subset of objects that will be subjected to collection (the collected region C). How this choice is made differs between collection algorithms, as described in Chapter 3. The remaining objects will be the uncollected set U.

The simulator finds the roots of the collection as follows. First, all objects in C that are not *known-dead* are roots. These objects would be reachable from the stack and globals at this point in actual execution. Second, all objects in C which have an incoming pointer from U are roots. These objects must be assumed live because *all* of U is assumed live in region-based collection.

The simulator then computes the transitive closure of the points-to relation over the root set within C, using a simple worklist mechanism. The result is the survivor set S. Data structures

corresponding to the garbage set $G = C \setminus S$ can now safely be discarded from the global set. Finally, the sum of the sizes of objects in S is added to the running sum of volume copied, and a collection counter is incremented.

The available allocation area increases by the amount freed, that is, the sum of the sizes of objects in G. If this amount is still not greater than zero, the simulator announces that the collector has failed.

As we have seen, the collection simulator keeps track of the total volume of allocated objects, as well as the total volume of survivor objects over all performed collections. By definition, the total volume of survivor objects is the measure of the copying cost of collection. The mark/cons ratio μ is also directly available as the ratio of the total volume of survivor objects to the total volume of allocated objects.

4.3.2 Age-based simulation

In age-based simulation, the relative order of object allocation remains the (logical) order of objects in the heap, and determines collection decisions. Therefore, the global set is implemented as a (doubly-linked) list. New allocation adds objects to the young (right) end of the list. The choice of the collected region is restricted to selecting a contiguous sublist C of the global list. The collection then finds the sublist of survivors S, which is spliced back into the global list in the place which C occupied.

When simulating age-ordered schemes, the time of allocation serves as a unique timestamp, and membership in a sublist can be determined by timestamp comparison instead of a full-fledged set-membership test; other manipulations are simplified as well.

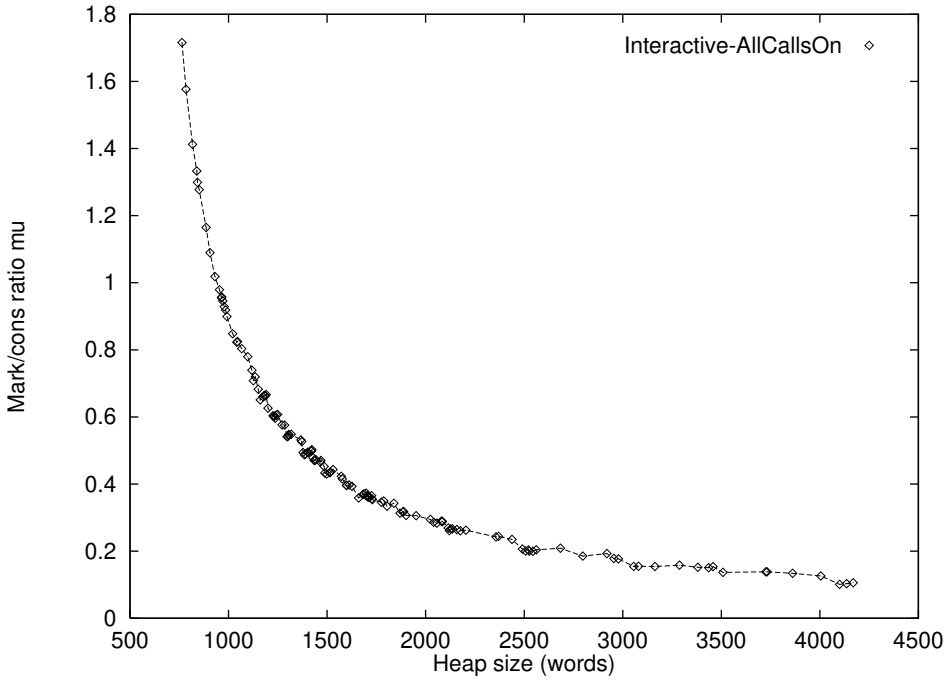
For renewal-age schemes, S is attached past the young end instead. The objects in S receive *renewed* timestamps.

4.3.3 Block-based simulation

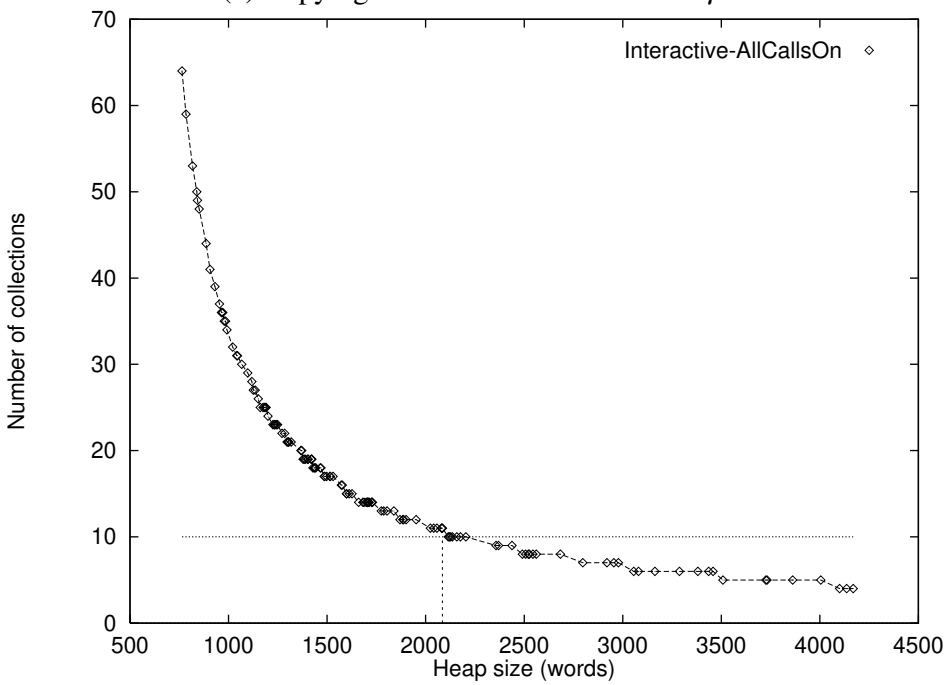
We implemented a prototype version of the block heap organization for age-based garbage collection, as outlined in Section 3.2. This prototype runs on 32-bit Sun SPARC machines and

does not use the design for large address spaces of Section 3.2.4. It is heavily instrumented in order to record all collector actions, as well as all the write barrier actions.

In order to examine how the block implementation works for the same traces that were examined using object-level simulation, we construct a *pseudomutator*: a program that accepts a trace input in our standard format, and performs the same heap-related actions that the original traced program did. We then integrate the pseudomutator with the (various) garbage collectors in the block implementation prototype. Because the computational aspects of the program are replaced with table lookup in the pseudomutator, the combined system remains a simulator, rather than an accurate replica of the original program. Therefore we are content with obtaining thorough statistics of copying costs (as counts of words copied) and pointer-management costs (as counts of different write-barrier and remembered-set processing operations).

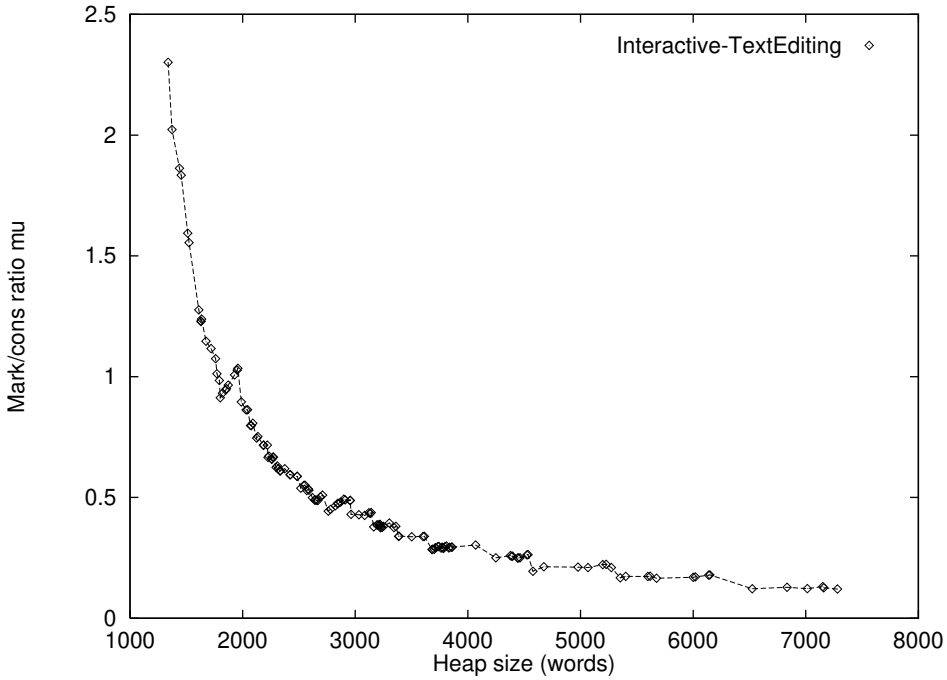


(a) Copying cost as the mark/cons ratio μ

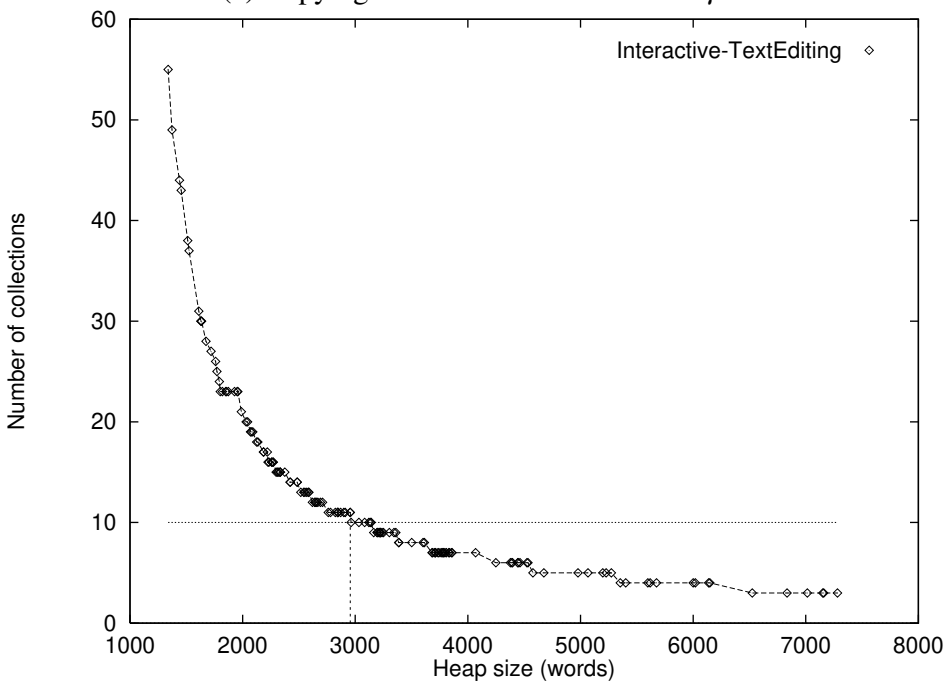


(b) Collector invocation count

Figure 4.1. Non-generational collection performance: Interactive-AllCallsOn.

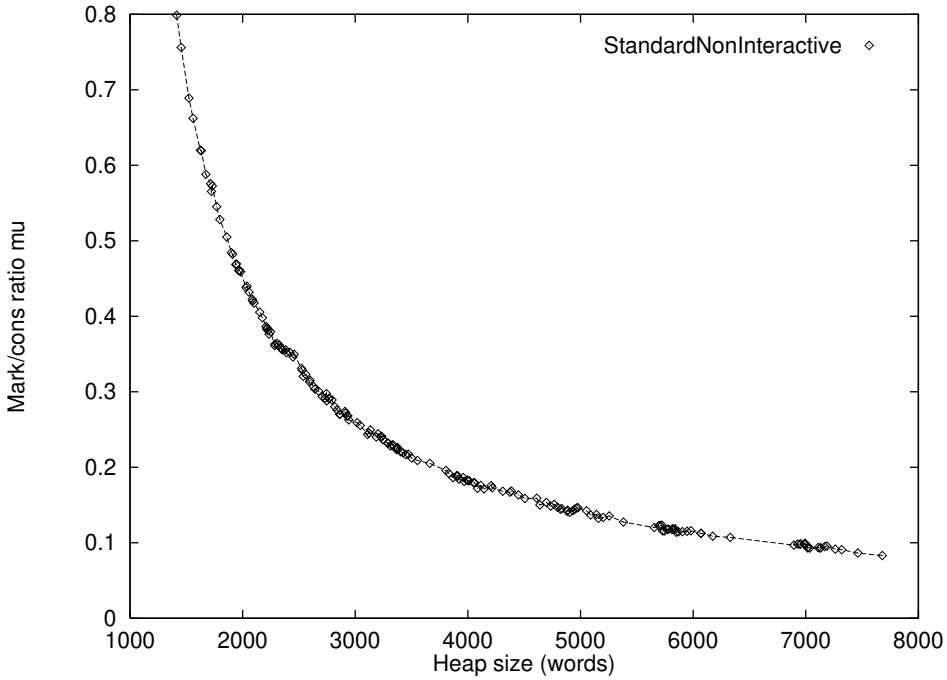


(a) Copying cost as the mark/cons ratio μ

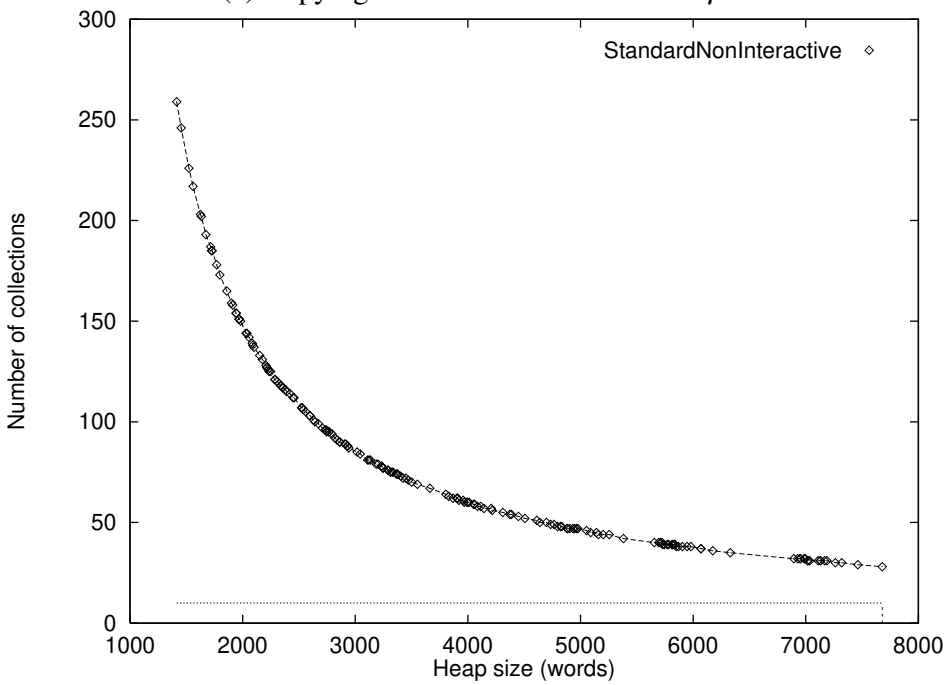


(b) Collector invocation count

Figure 4.2. Non-generational collection performance: Interactive-TextEditing.

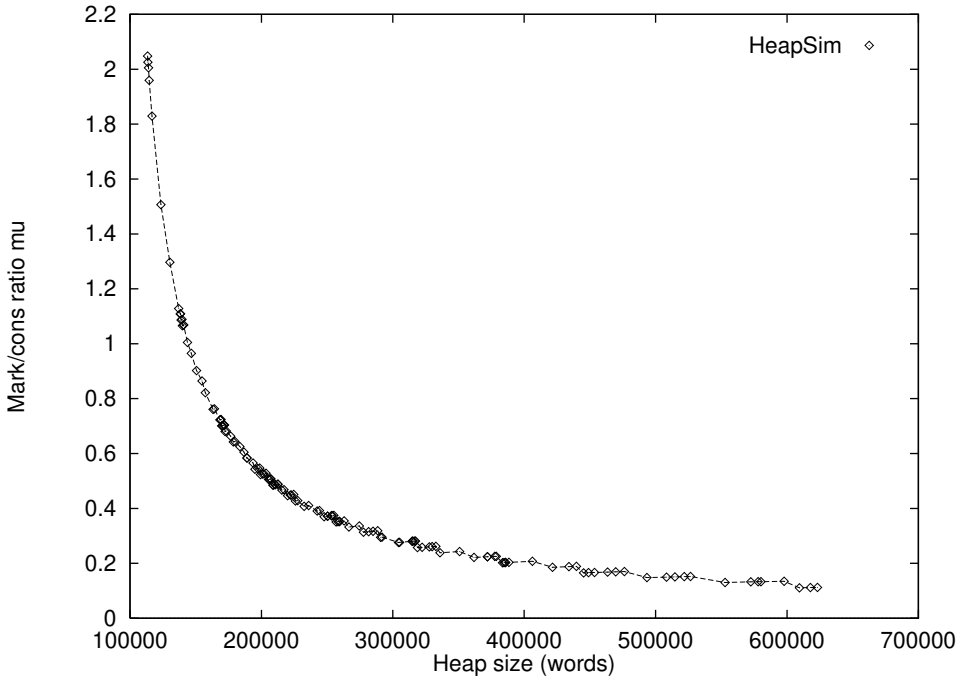


(a) Copying cost as the mark/cons ratio μ

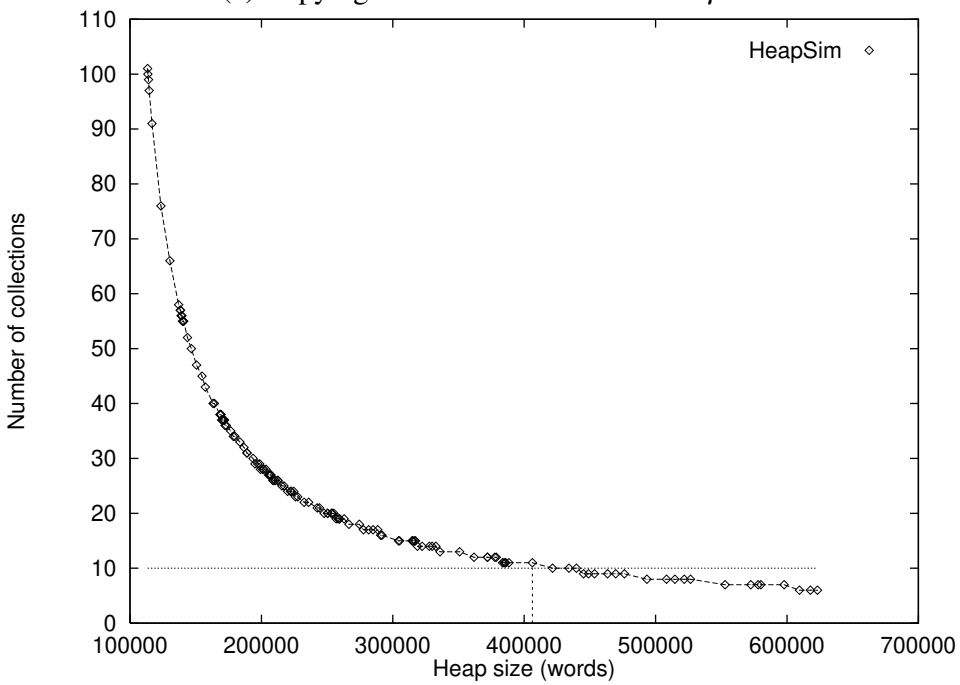


(b) Collector invocation count

Figure 4.3. Non-generational collection performance: StandardNonInteractive.

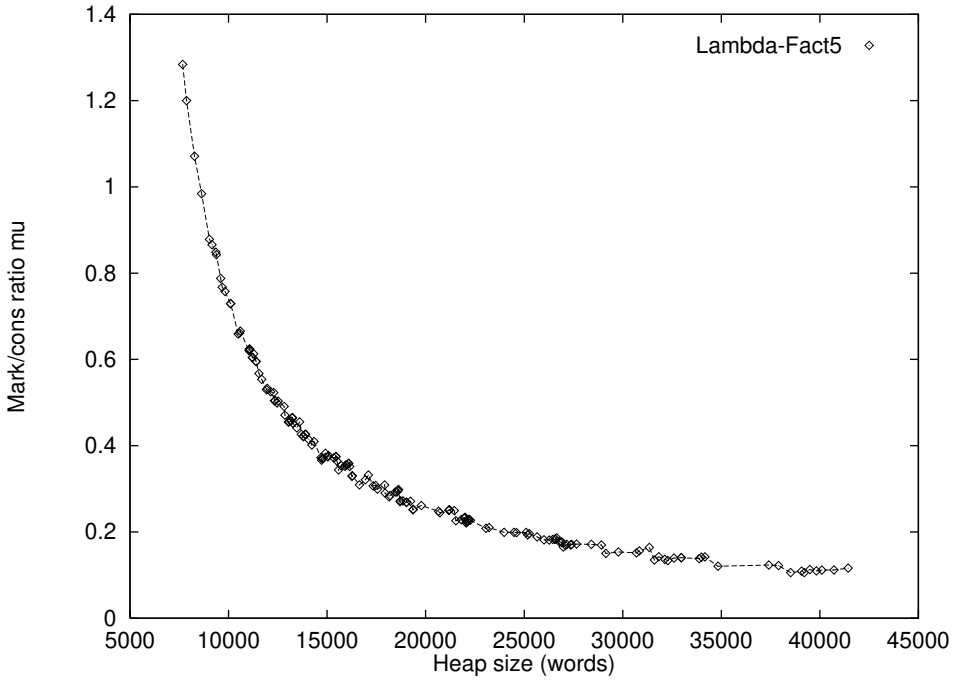


(a) Copying cost as the mark/cons ratio μ

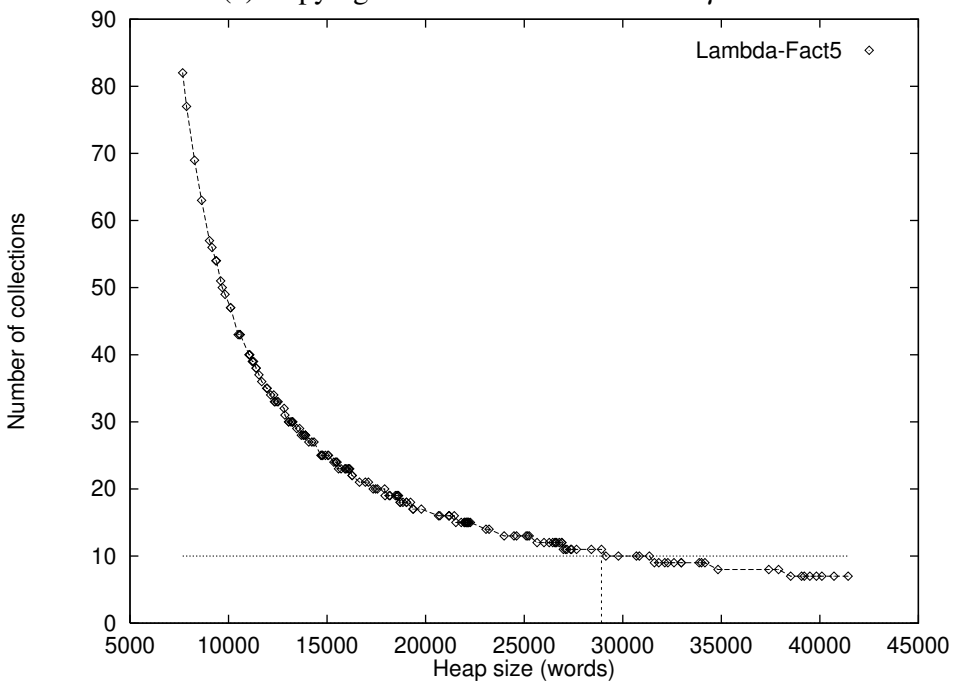


(b) Collector invocation count

Figure 4.4. Non-generational collection performance: HeapSim.

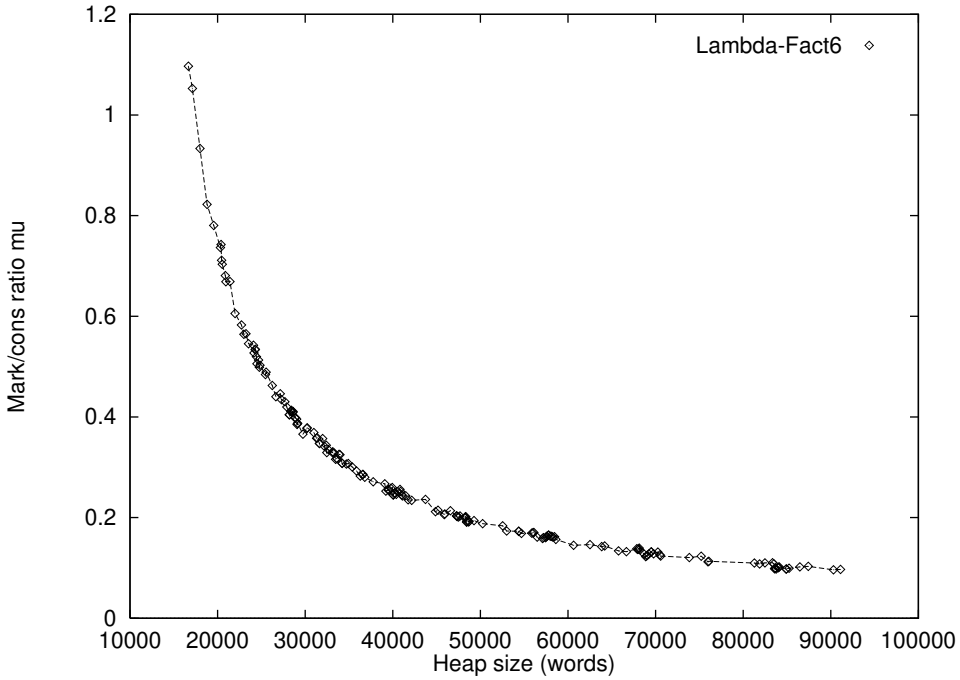


(a) Copying cost as the mark/cons ratio μ

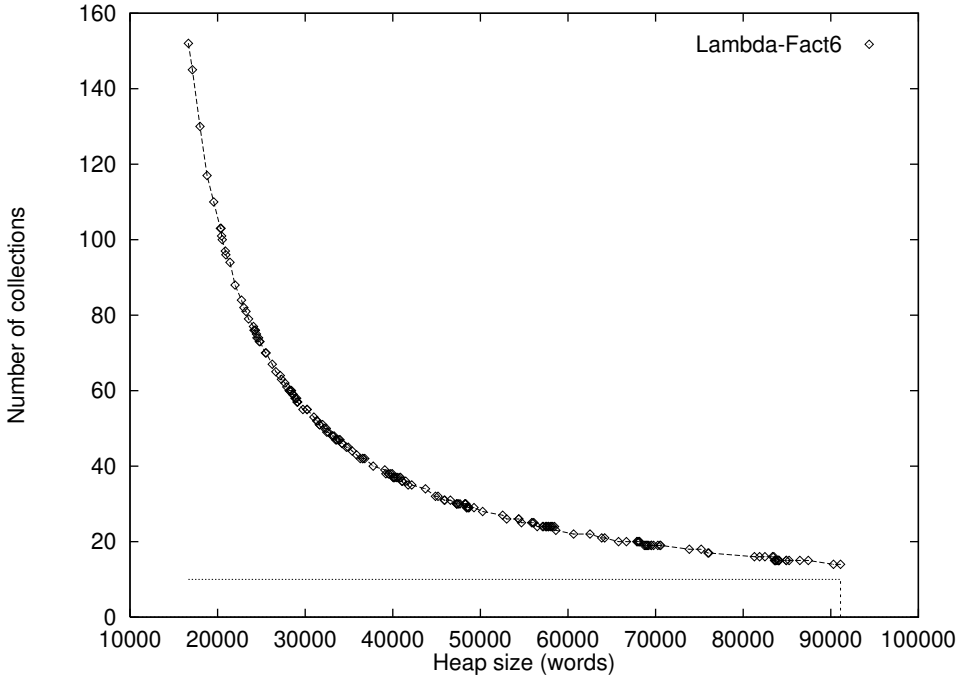


(b) Collector invocation count

Figure 4.5. Non-generational collection performance: Lambda-Fact5.

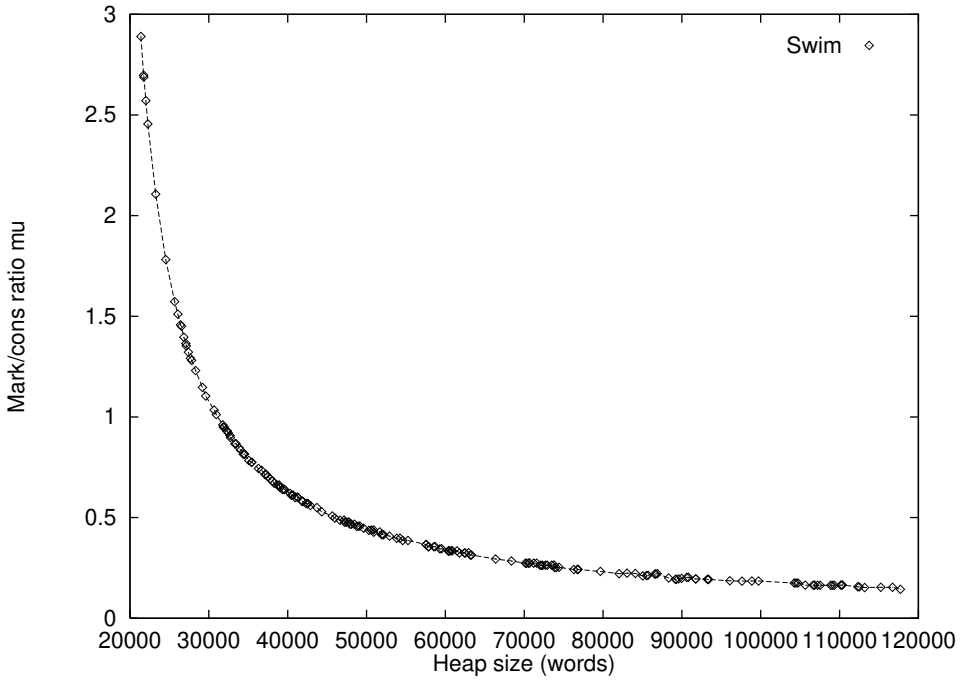


(a) Copying cost as the mark/cons ratio μ

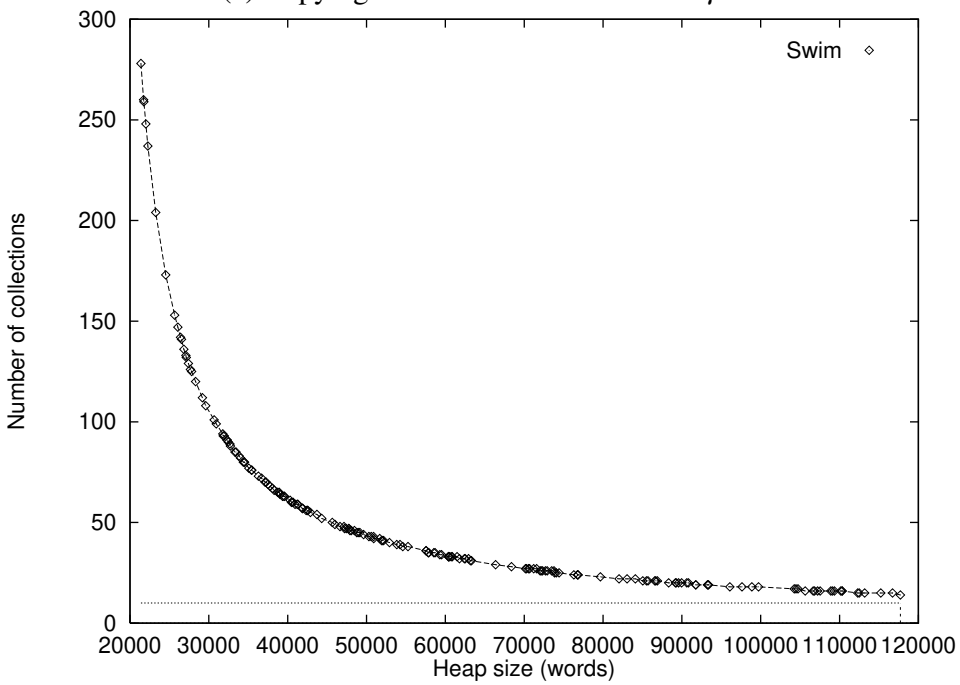


(b) Collector invocation count

Figure 4.6. Non-generational collection performance: Lambda-Fact6.

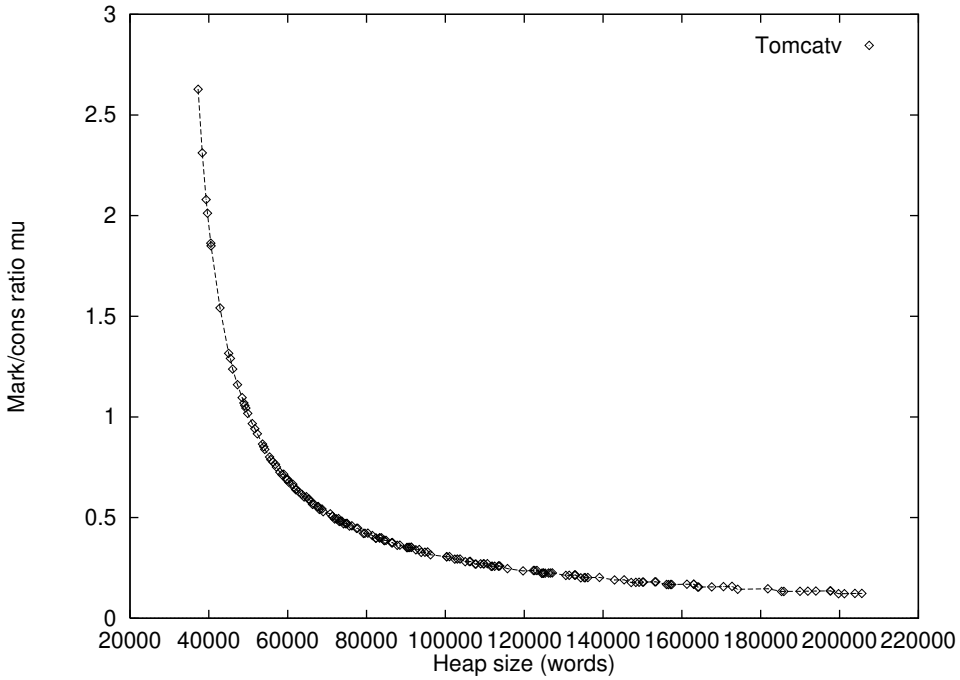


(a) Copying cost as the mark/cons ratio μ

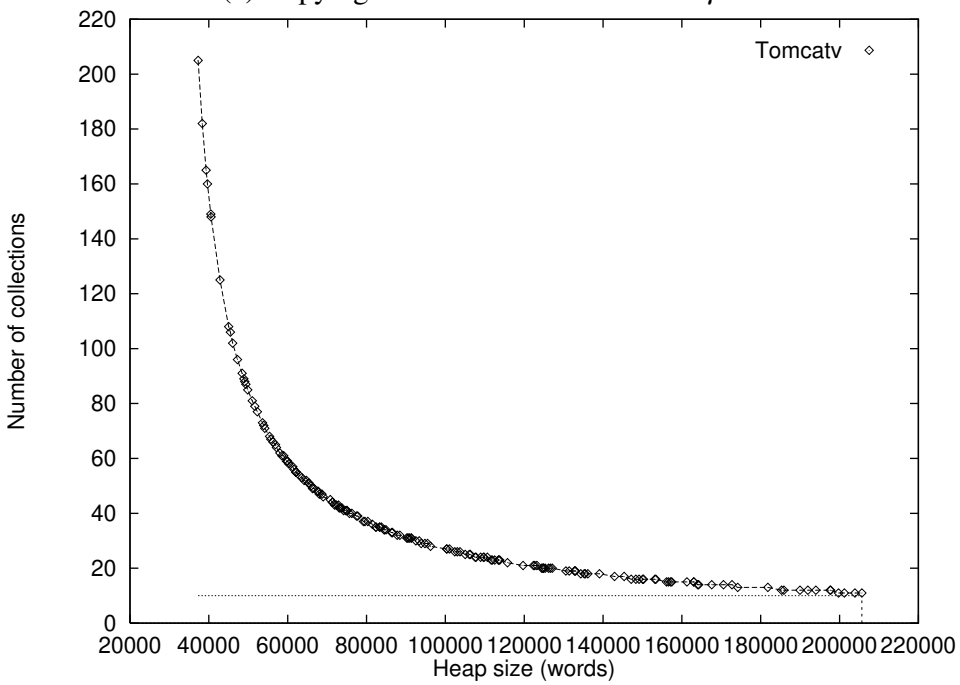


(b) Collector invocation count

Figure 4.7. Non-generational collection performance: Swim.

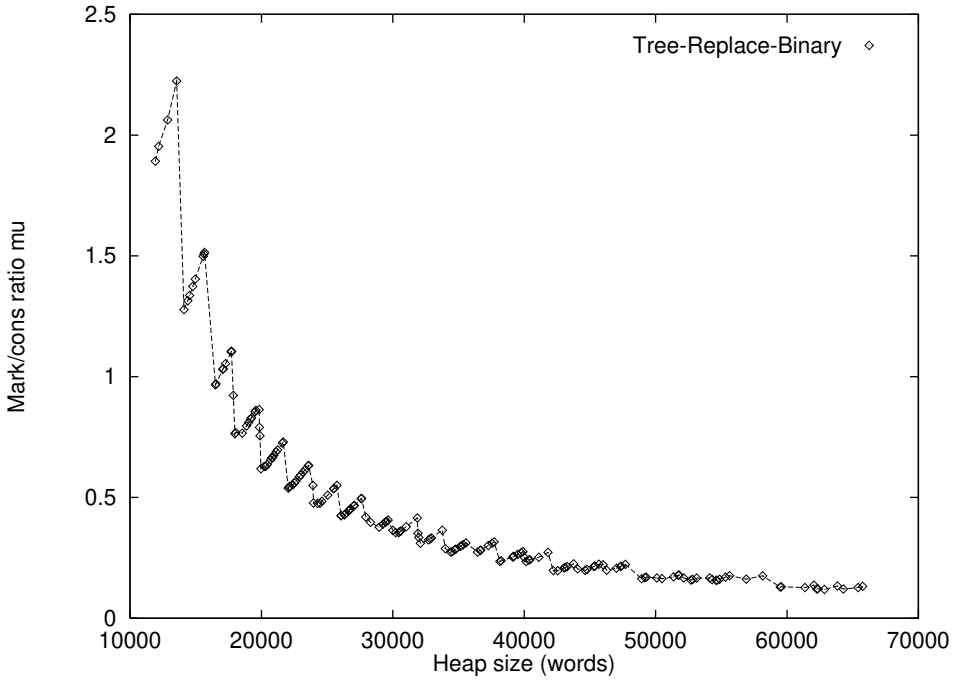


(a) Copying cost as the mark/cons ratio μ

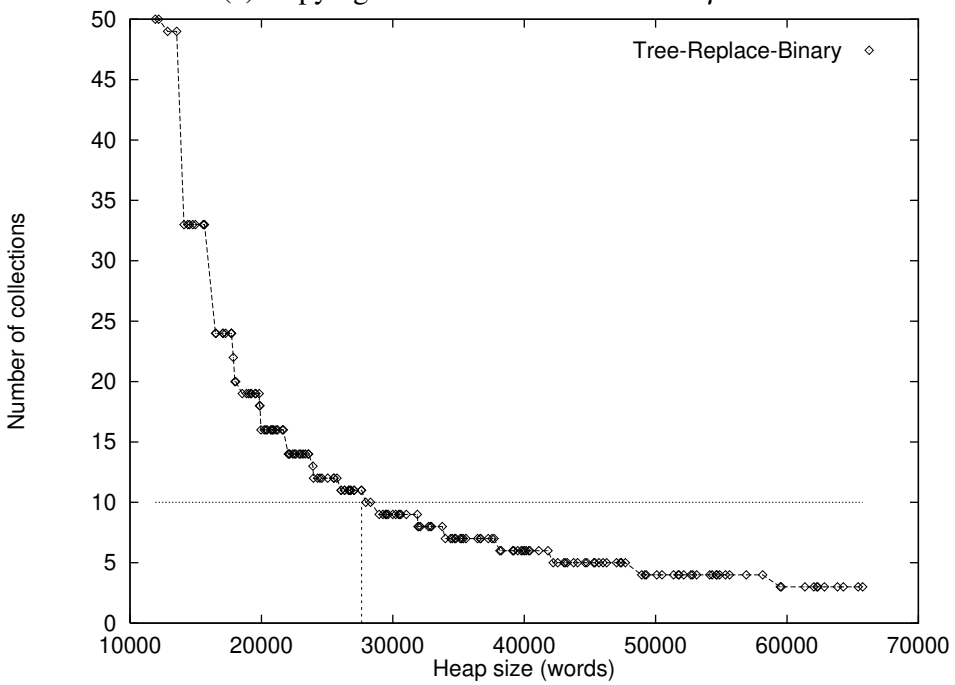


(b) Collector invocation count

Figure 4.8. Non-generational collection performance: Tomcatv.

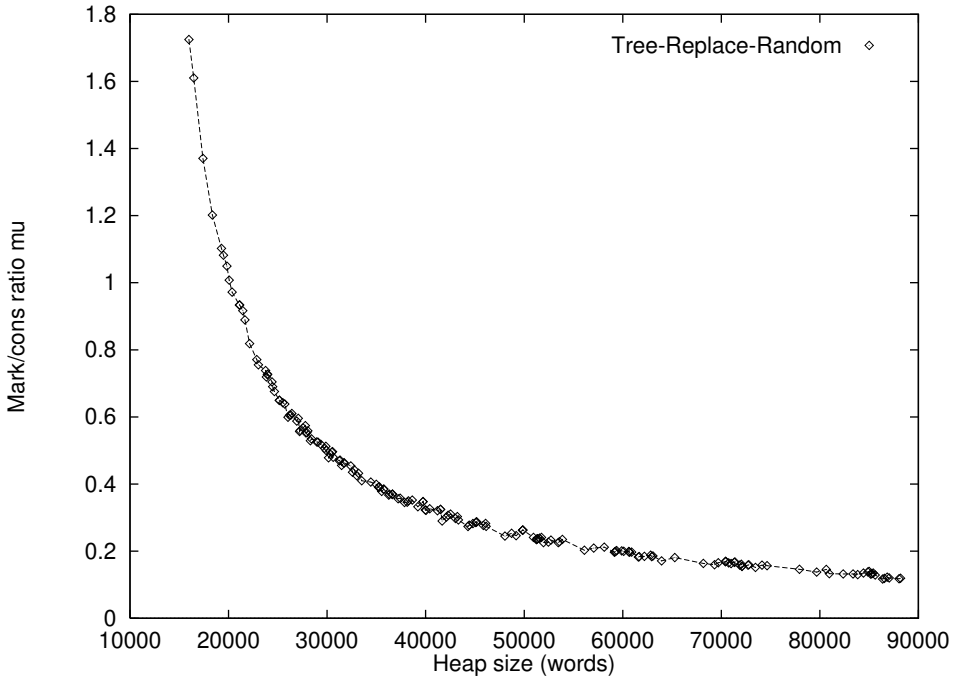


(a) Copying cost as the mark/cons ratio μ

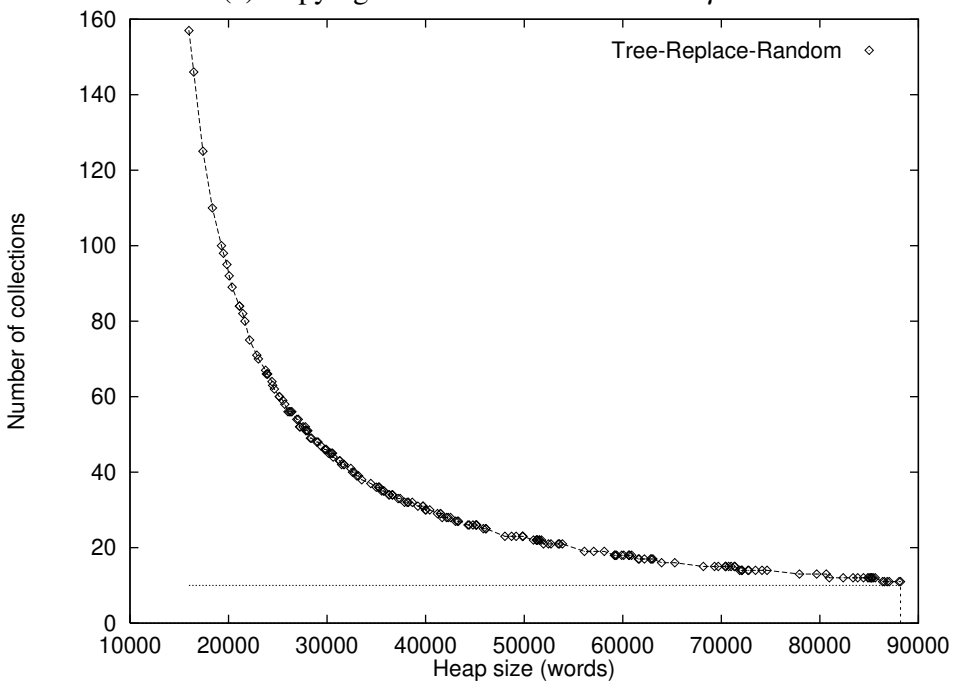


(b) Collector invocation count

Figure 4.9. Non-generational collection performance: Tree-Replace-Binary.

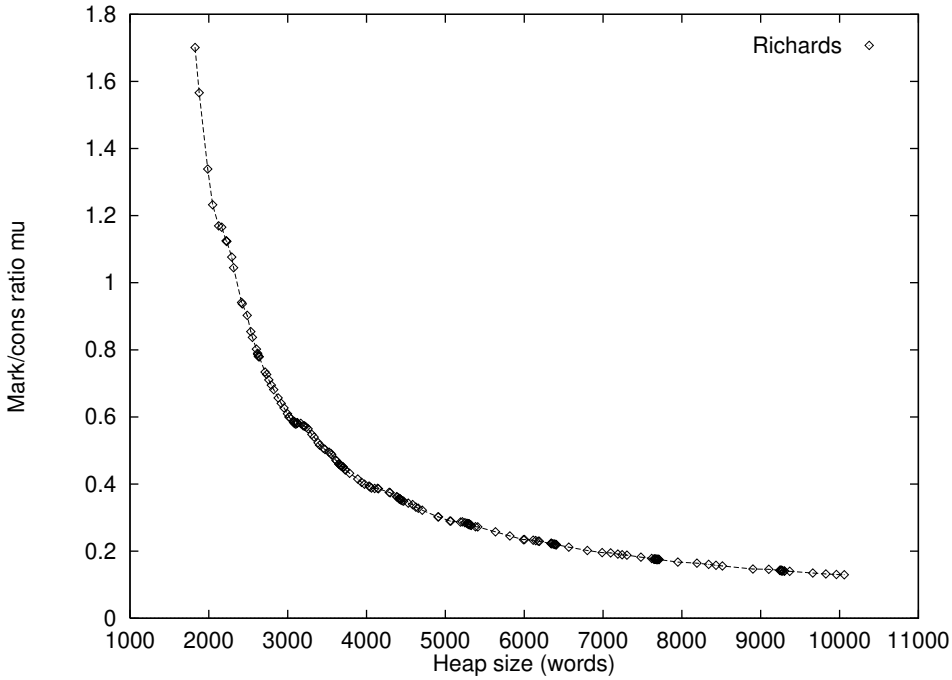


(a) Copying cost as the mark/cons ratio μ

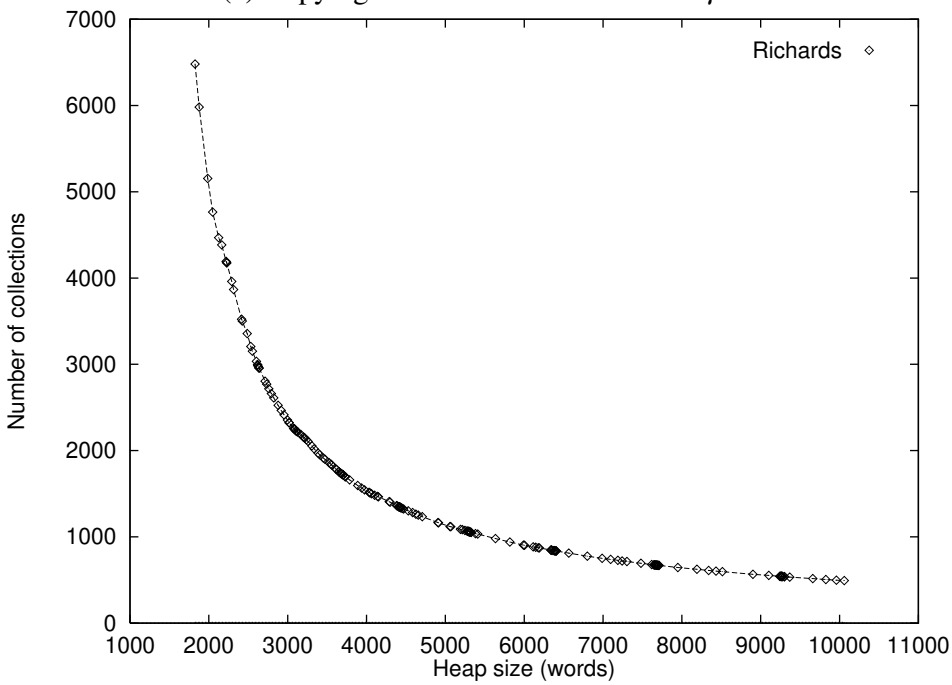


(b) Collector invocation count

Figure 4.10. Non-generational collection performance: Tree-Replace-Random.

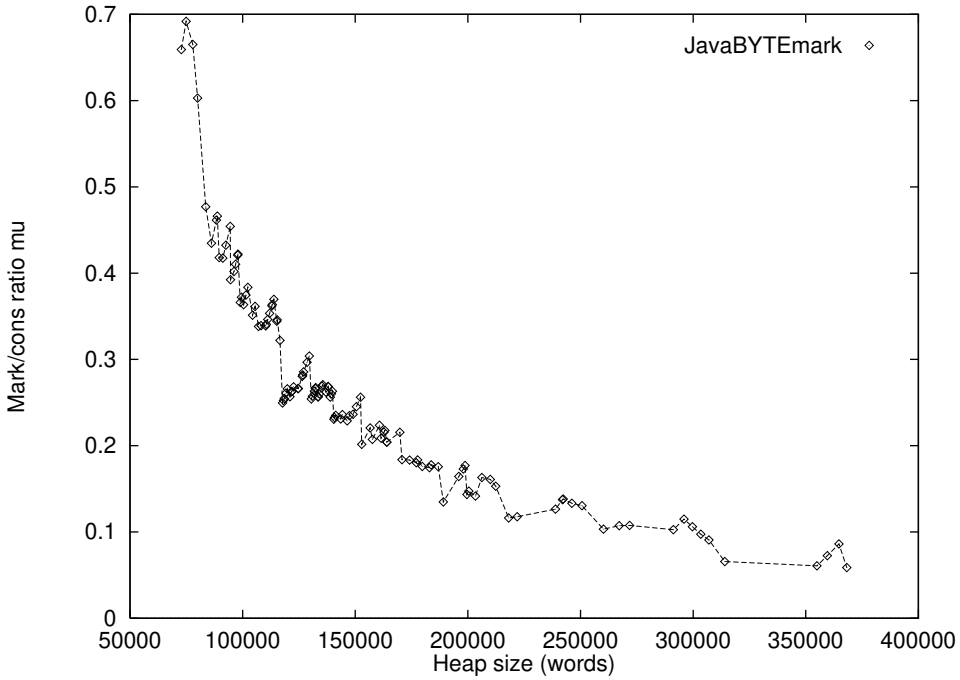


(a) Copying cost as the mark/cons ratio μ

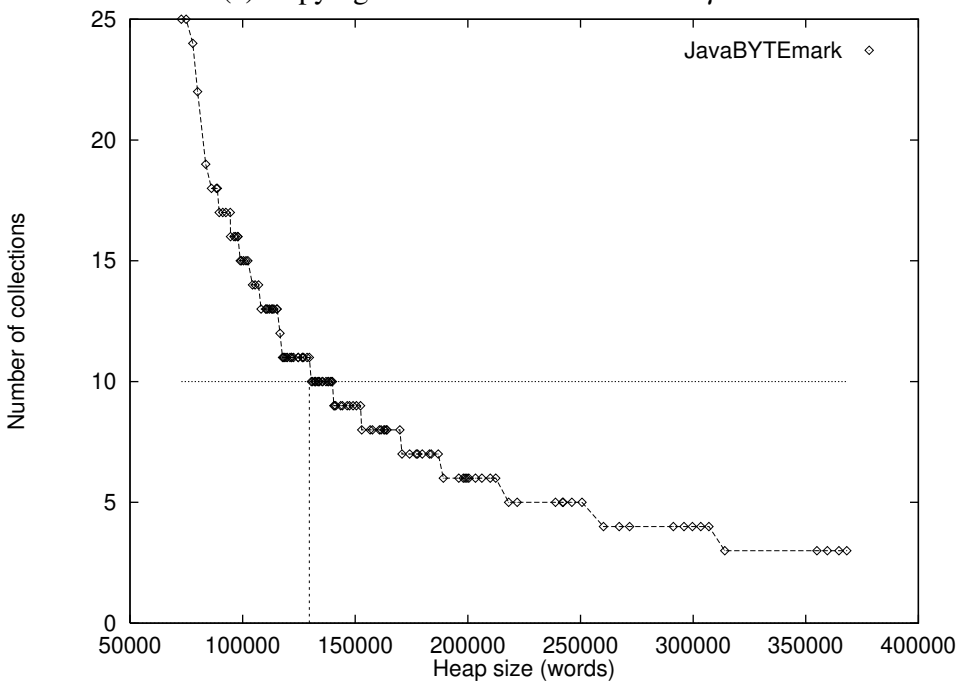


(b) Collector invocation count

Figure 4.11. Non-generational collection performance: Richards.

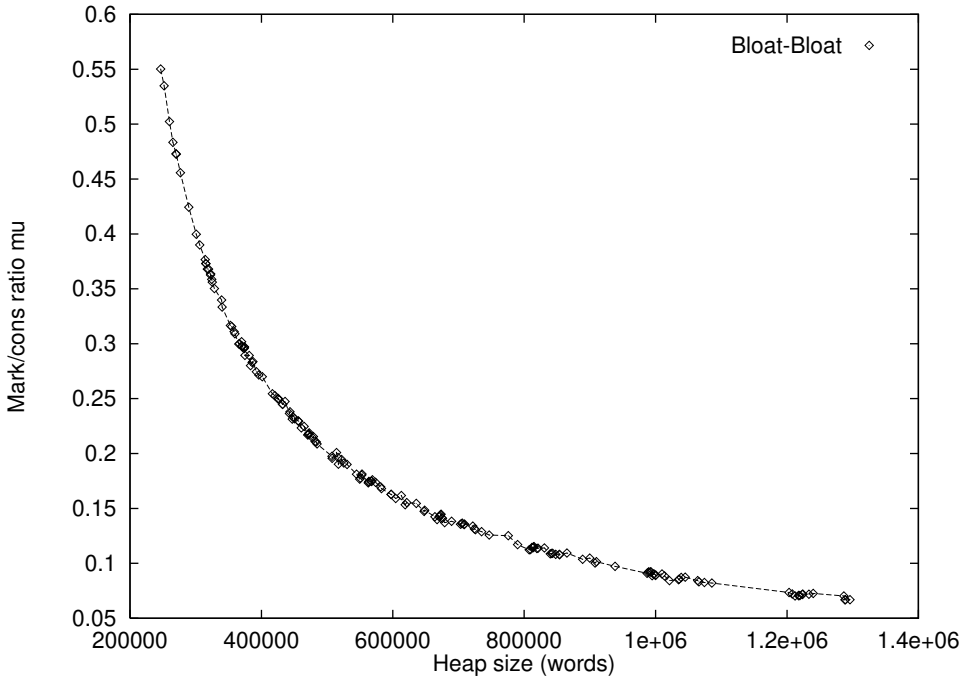


(a) Copying cost as the mark/cons ratio μ

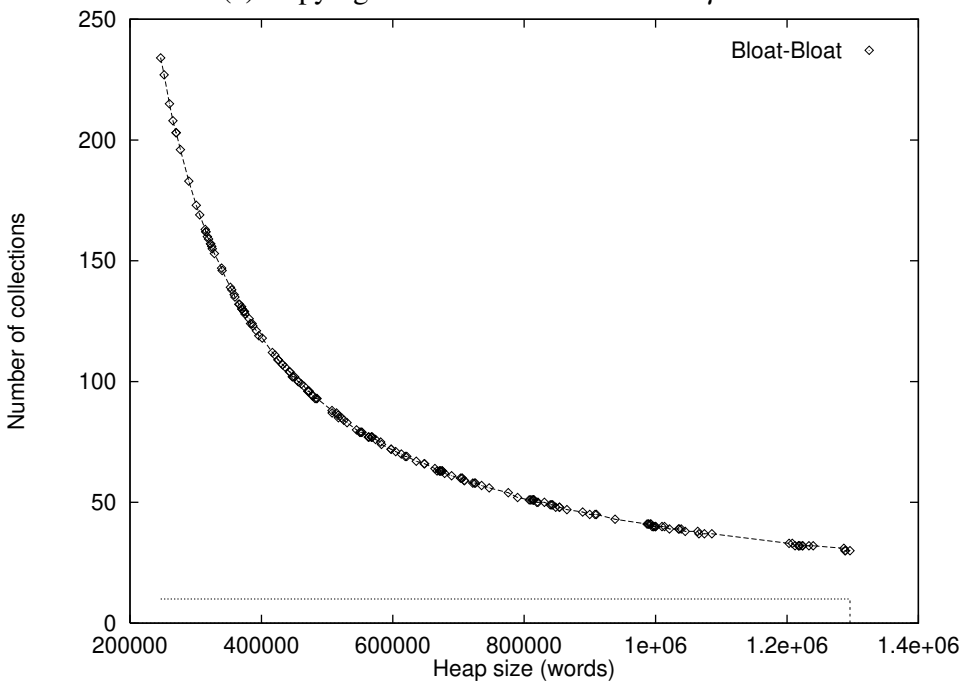


(b) Collector invocation count

Figure 4.12. Non-generational collection performance: JavaBYTEmark.

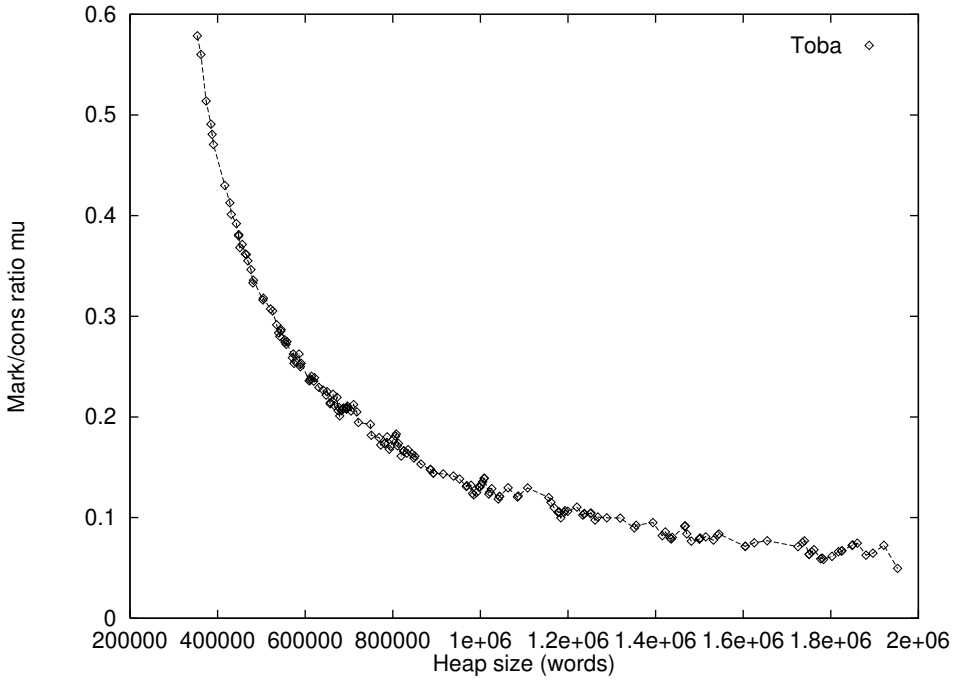


(a) Copying cost as the mark/cons ratio μ

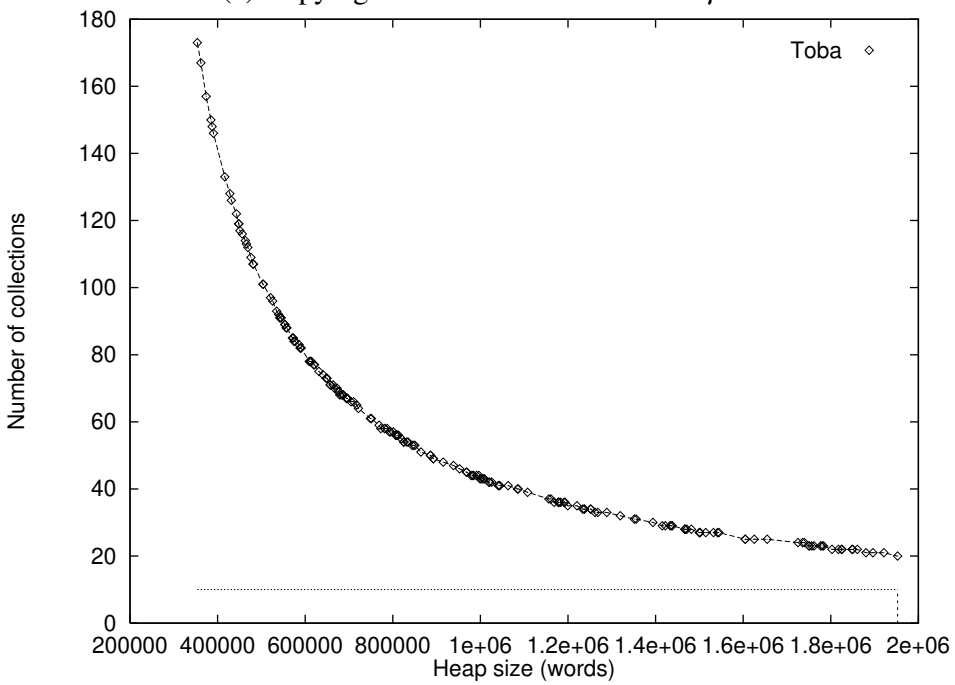


(b) Collector invocation count

Figure 4.13. Non-generational collection performance: Bloat-Bloat.

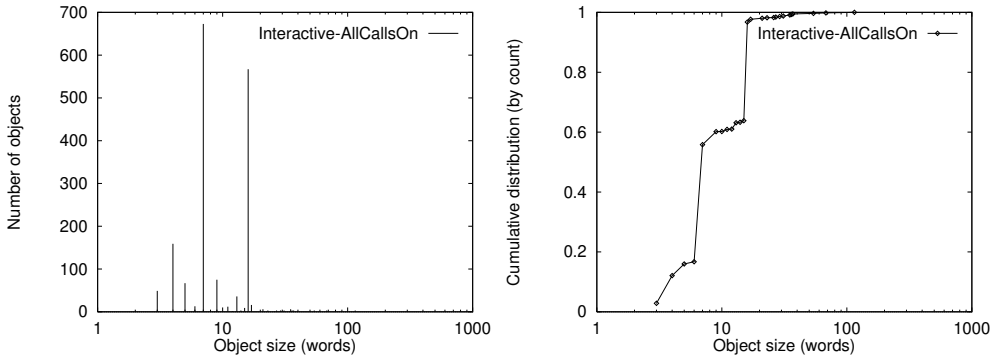


(a) Copying cost as the mark/cons ratio μ

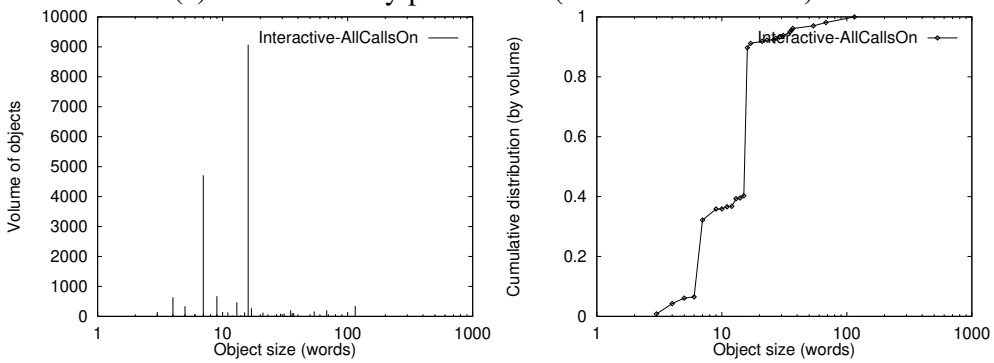


(b) Collector invocation count

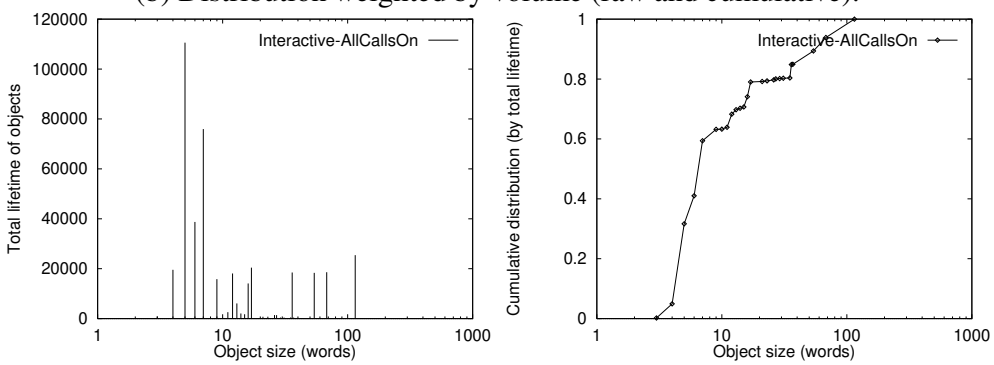
Figure 4.14. Non-generational collection performance: Toba.



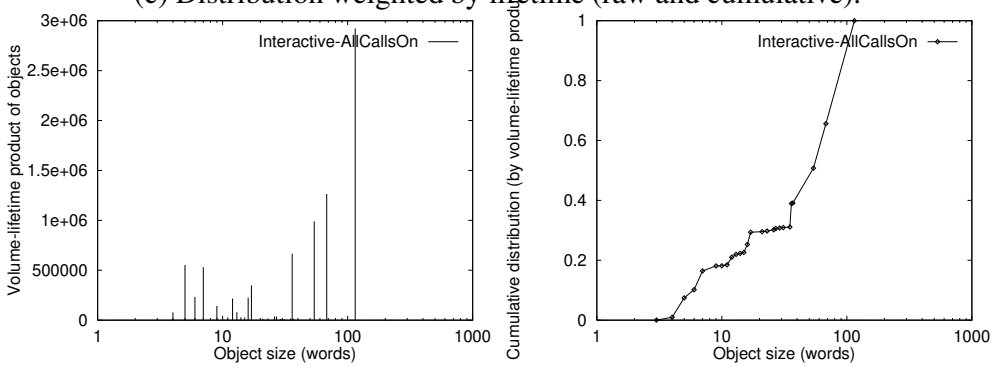
(a) Distribution by plain count (raw and cumulative).



(b) Distribution weighted by volume (raw and cumulative).

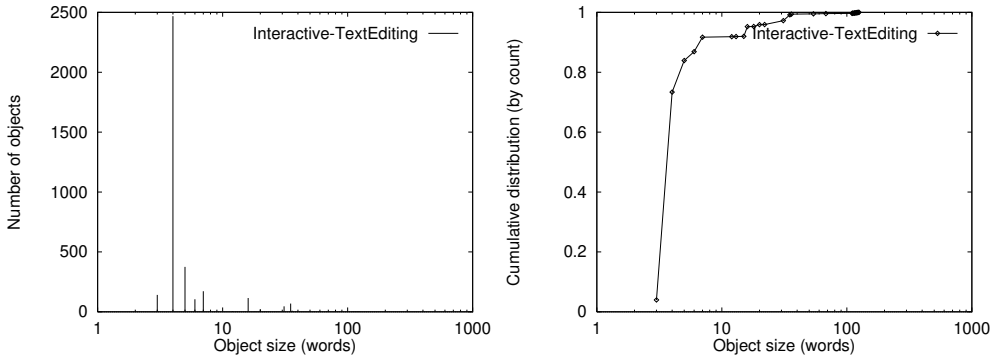


(c) Distribution weighted by lifetime (raw and cumulative).

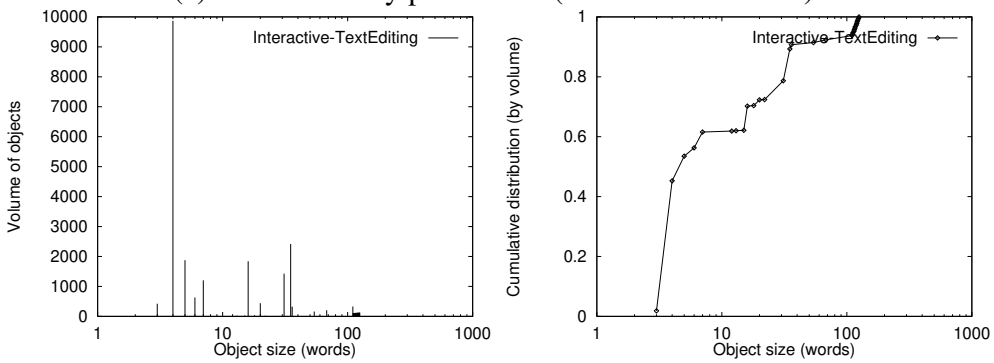


(d) Distribution weighted by lifetime and volume (raw and cumulative).

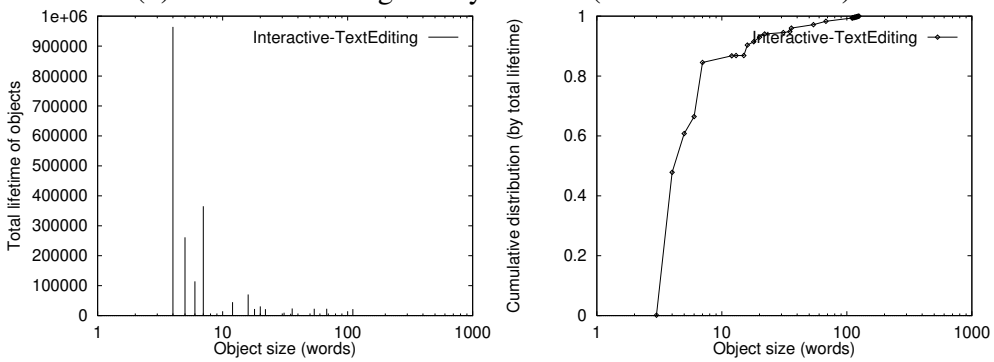
Figure 4.15. Object size distribution: Interactive-AllCallsOn.



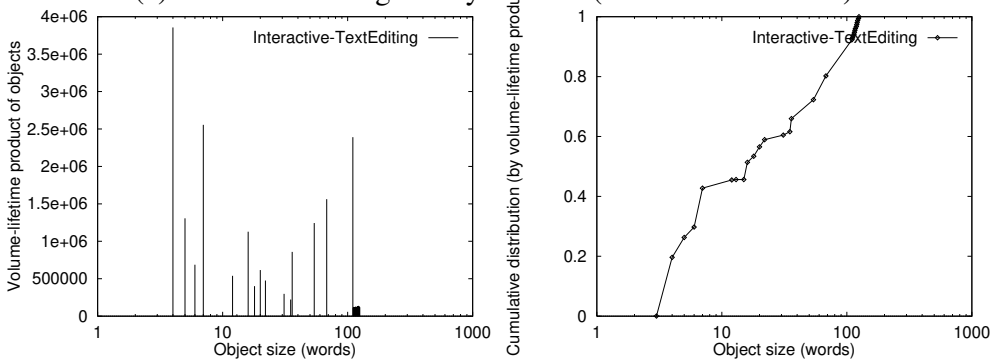
(a) Distribution by plain count (raw and cumulative).



(b) Distribution weighted by volume (raw and cumulative).

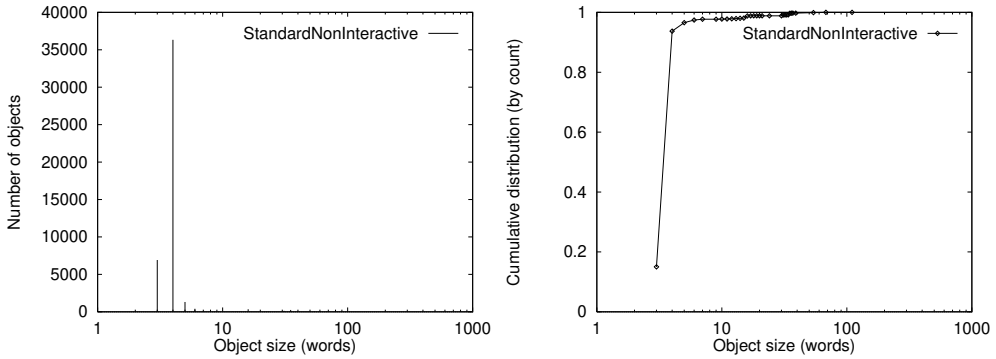


(c) Distribution weighted by lifetime (raw and cumulative).

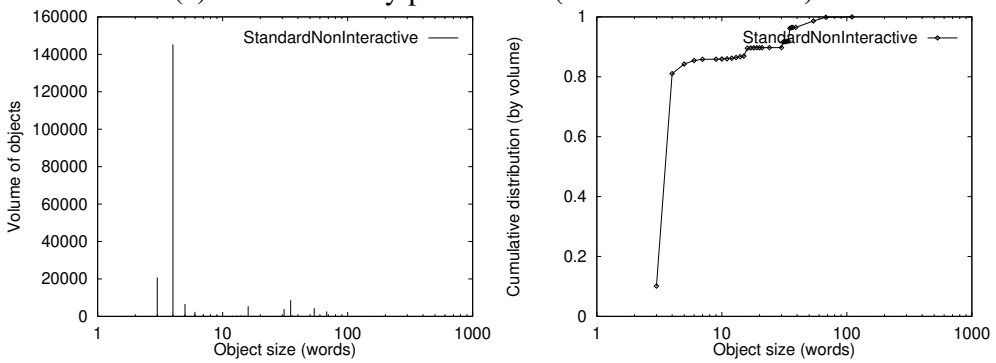


(d) Distribution weighted by lifetime and volume (raw and cumulative).

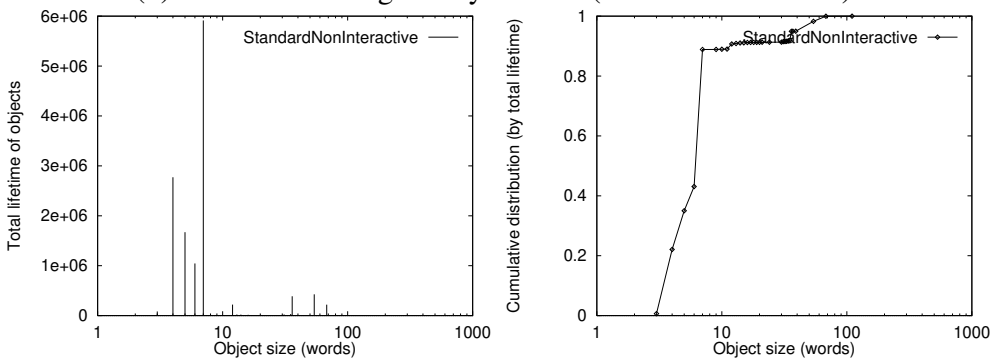
Figure 4.16. Object size distribution: Interactive-TextEditing.



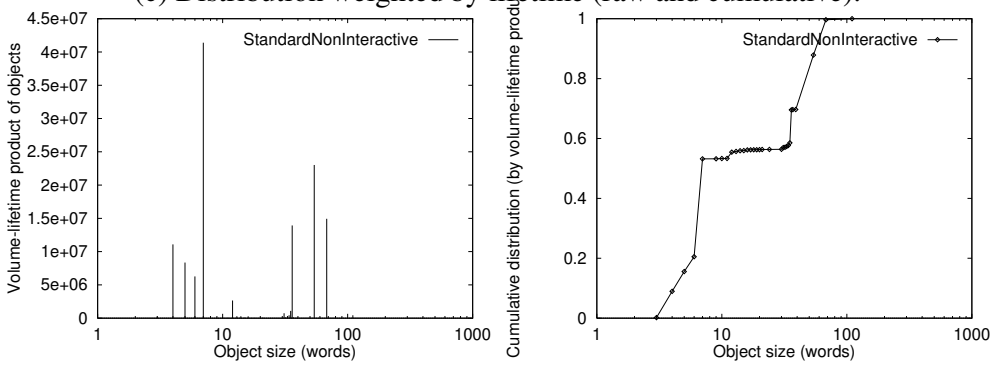
(a) Distribution by plain count (raw and cumulative).



(b) Distribution weighted by volume (raw and cumulative).

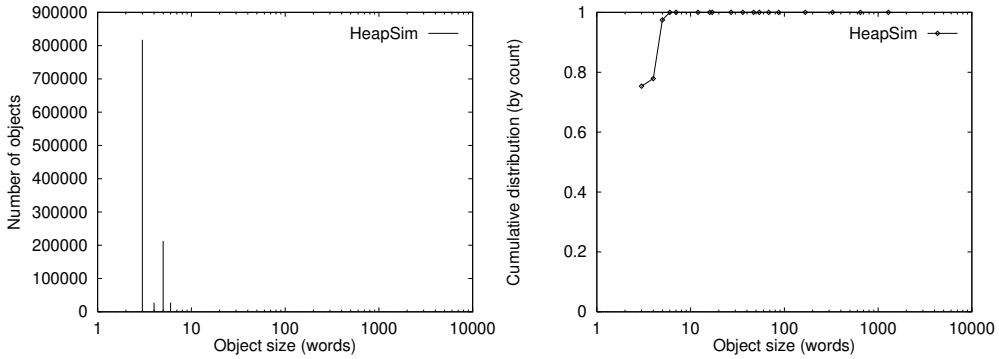


(c) Distribution weighted by lifetime (raw and cumulative).

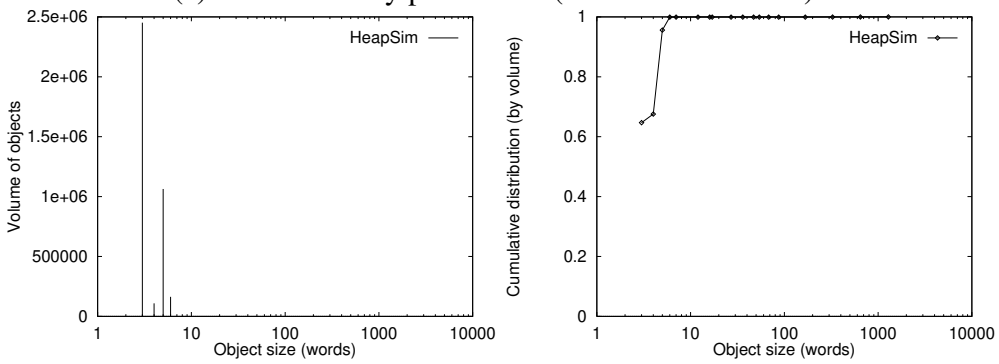


(d) Distribution weighted by lifetime and volume (raw and cumulative).

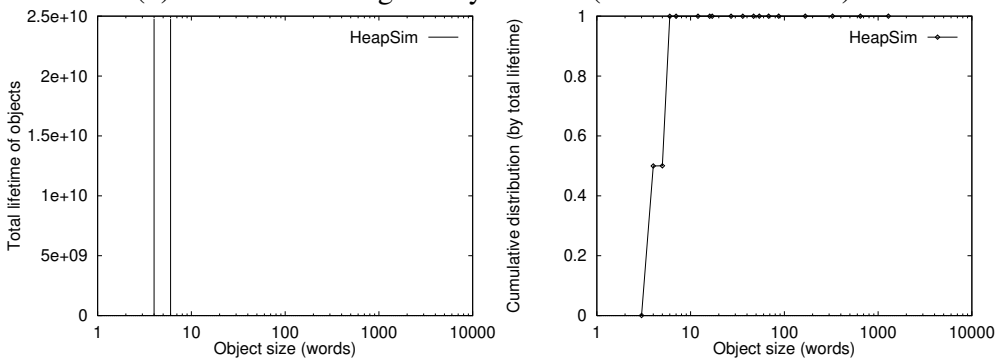
Figure 4.17. Object size distribution: StandardNonInteractive.



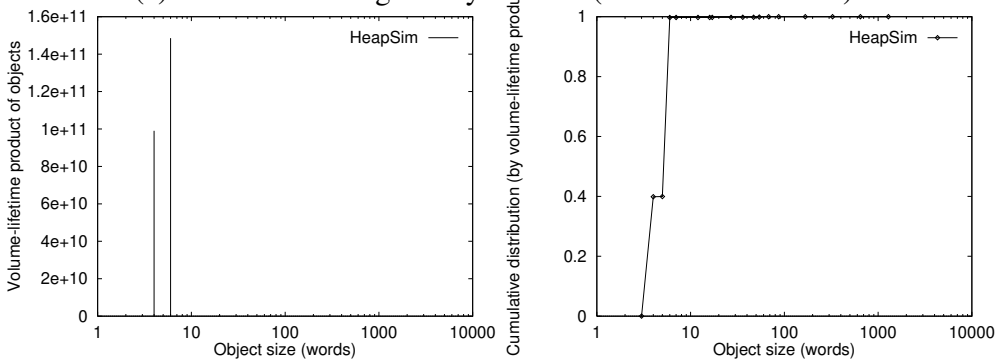
(a) Distribution by plain count (raw and cumulative).



(b) Distribution weighted by volume (raw and cumulative).

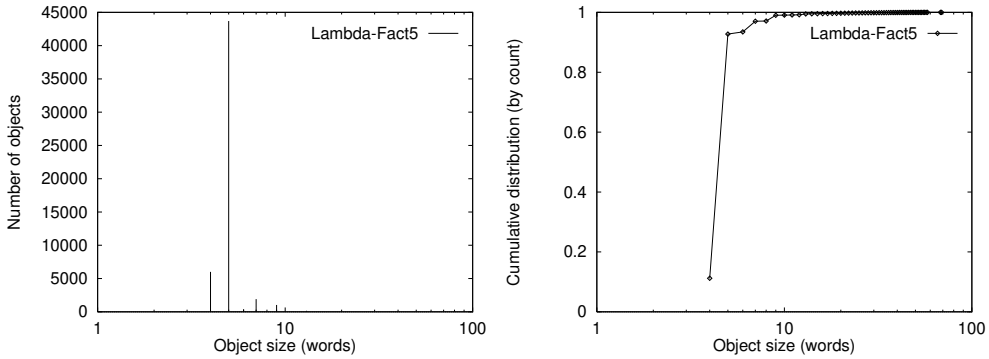


(c) Distribution weighted by lifetime (raw and cumulative).

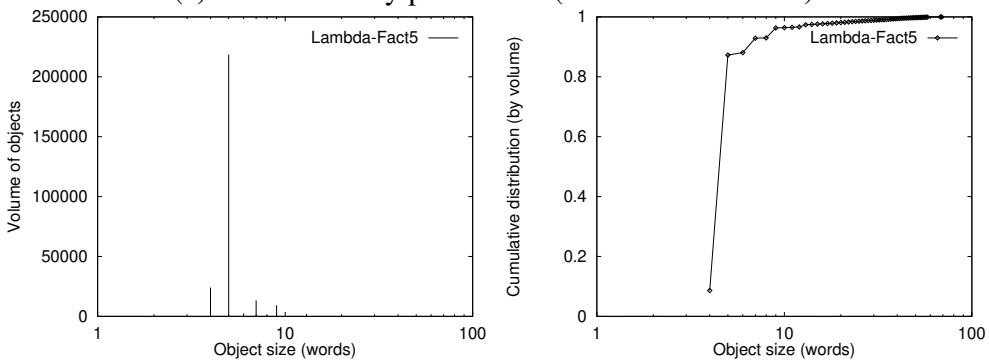


(d) Distribution weighted by lifetime and volume (raw and cumulative).

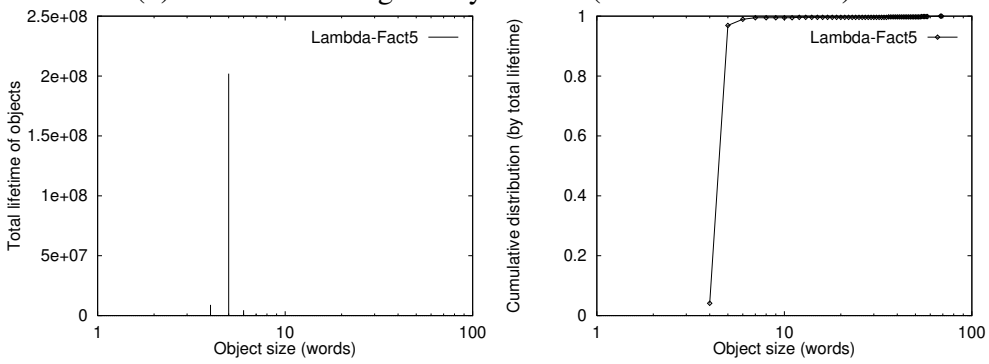
Figure 4.18. Object size distribution: HeapSim.



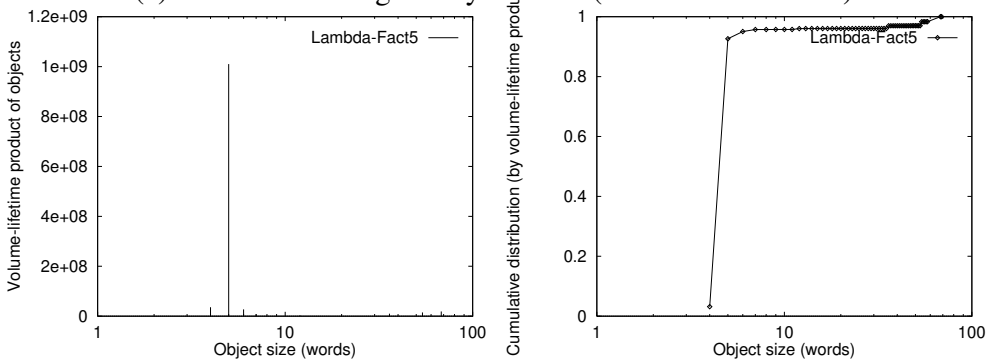
(a) Distribution by plain count (raw and cumulative).



(b) Distribution weighted by volume (raw and cumulative).

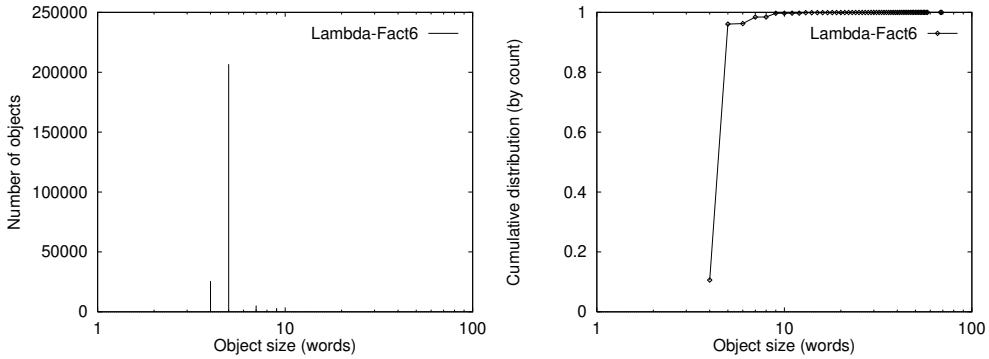


(c) Distribution weighted by lifetime (raw and cumulative).

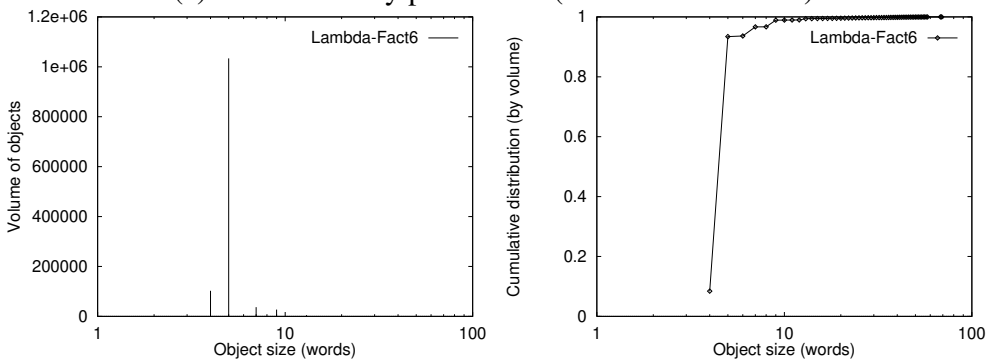


(d) Distribution weighted by lifetime and volume (raw and cumulative).

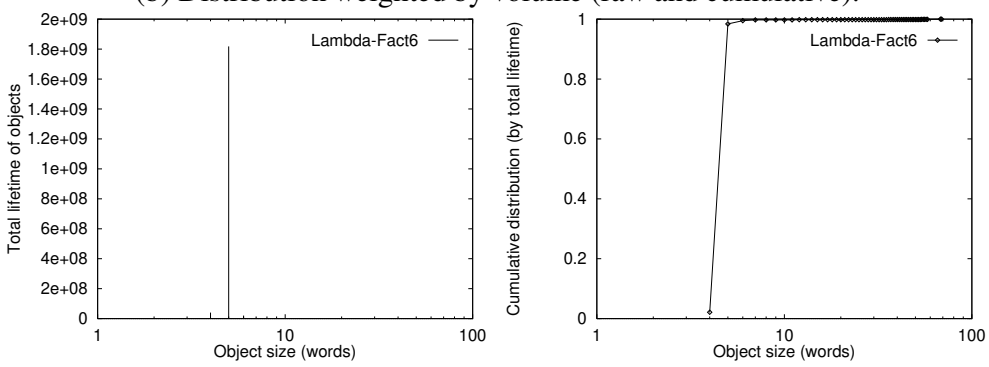
Figure 4.19. Object size distribution: Lambda-Fact5.



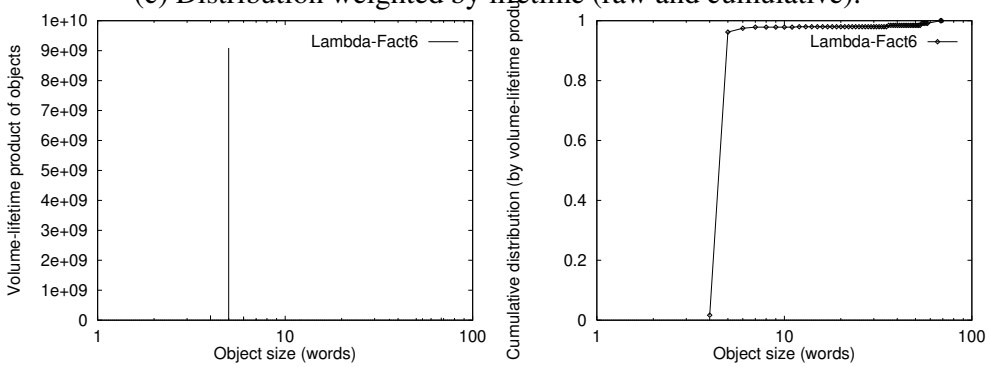
(a) Distribution by plain count (raw and cumulative).



(b) Distribution weighted by volume (raw and cumulative).

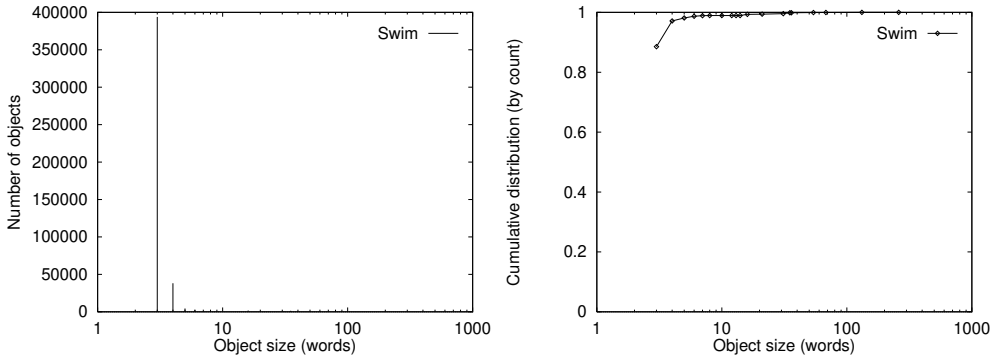


(c) Distribution weighted by lifetime (raw and cumulative).

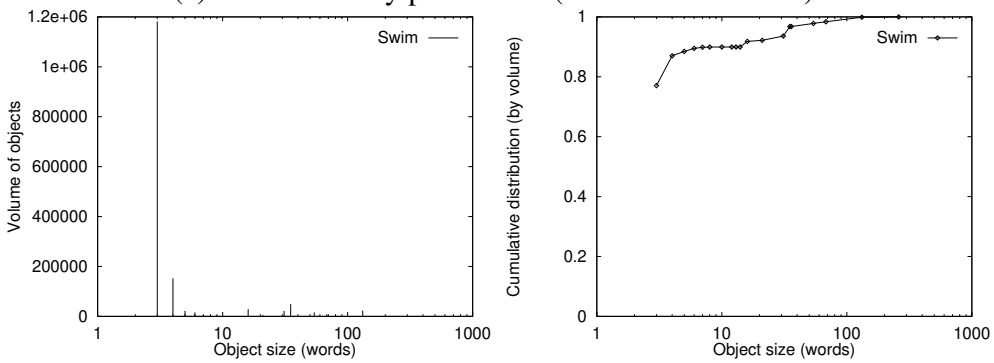


(d) Distribution weighted by lifetime and volume (raw and cumulative).

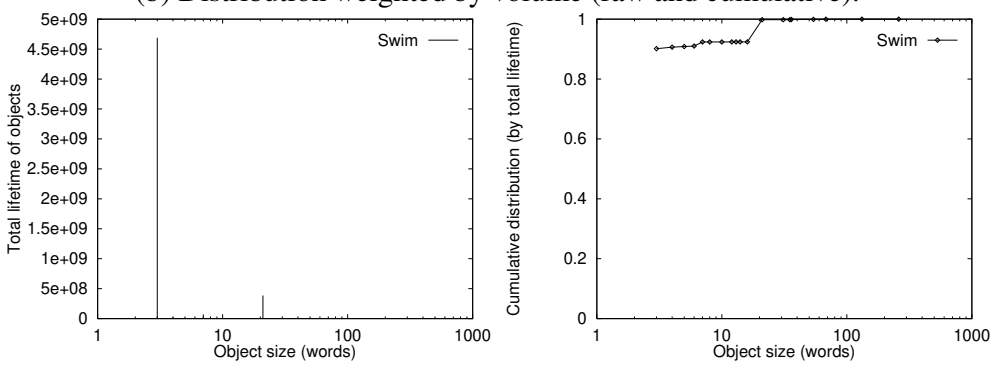
Figure 4.20. Object size distribution: Lambda-Fact6.



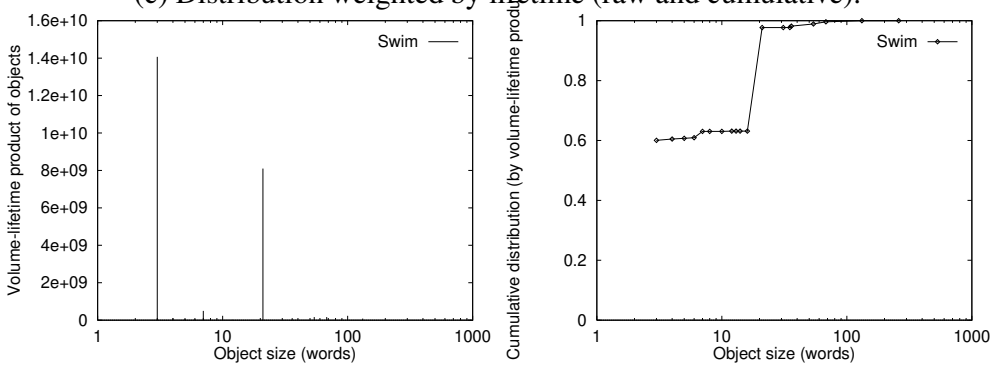
(a) Distribution by plain count (raw and cumulative).



(b) Distribution weighted by volume (raw and cumulative).

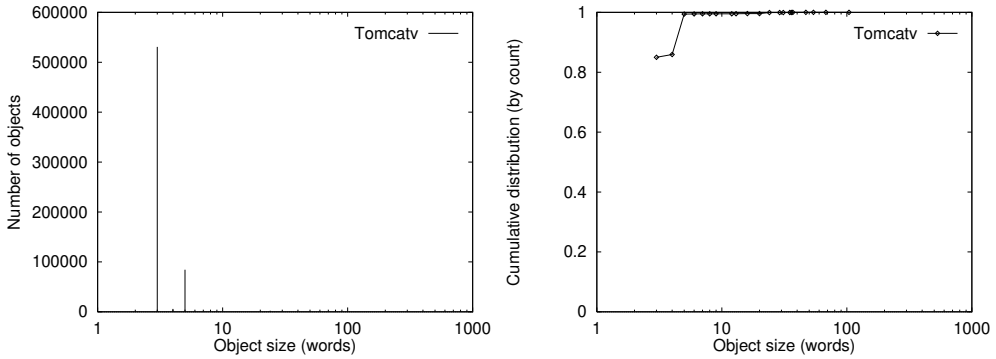


(c) Distribution weighted by lifetime (raw and cumulative).

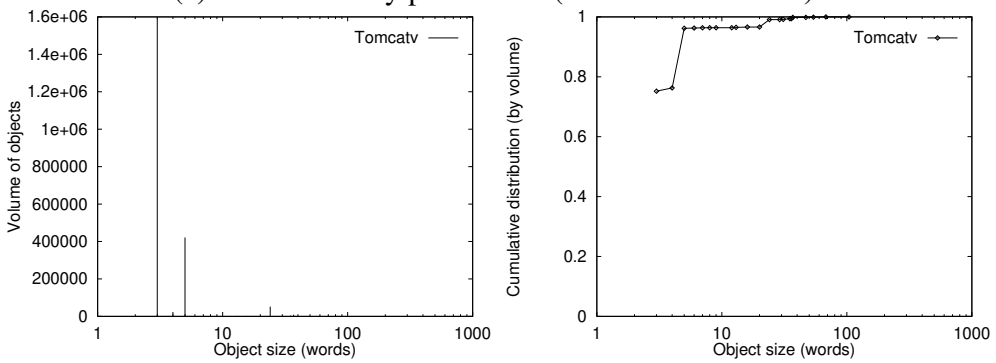


(d) Distribution weighted by lifetime and volume (raw and cumulative).

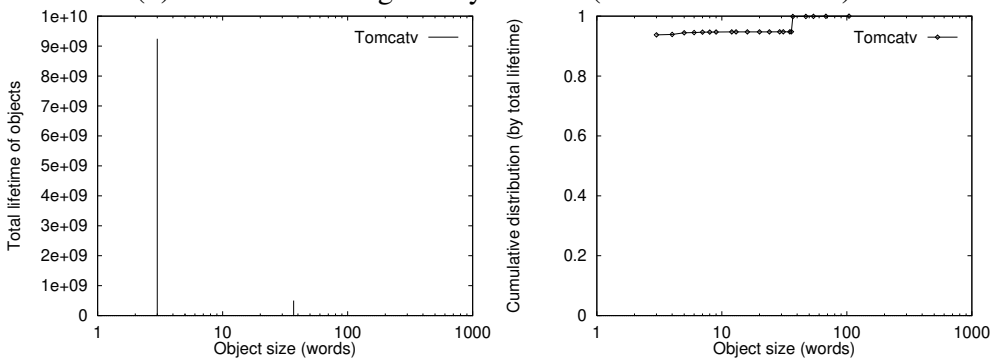
Figure 4.21. Object size distribution: Swim.



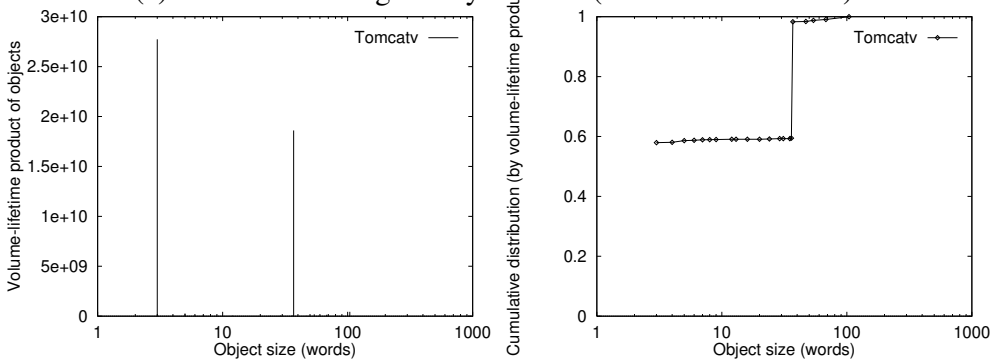
(a) Distribution by plain count (raw and cumulative).



(b) Distribution weighted by volume (raw and cumulative).

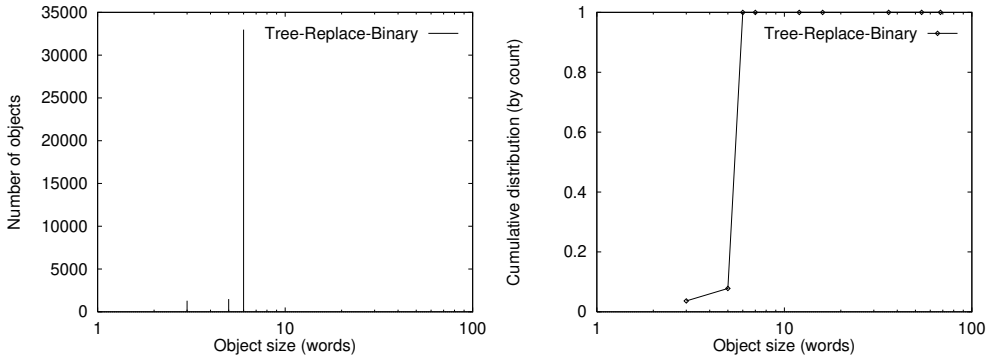


(c) Distribution weighted by lifetime (raw and cumulative).

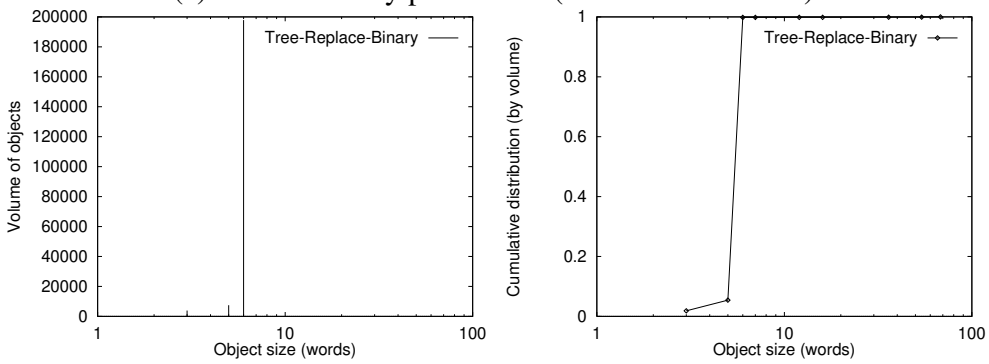


(d) Distribution weighted by lifetime and volume (raw and cumulative).

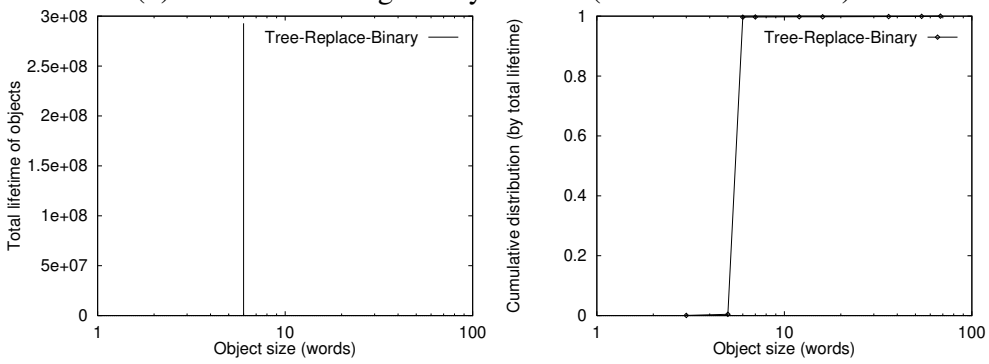
Figure 4.22. Object size distribution: Tomcatv.



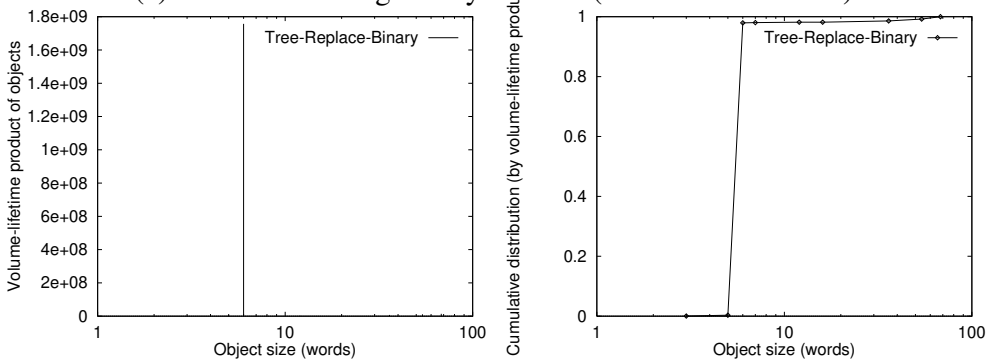
(a) Distribution by plain count (raw and cumulative).



(b) Distribution weighted by volume (raw and cumulative).

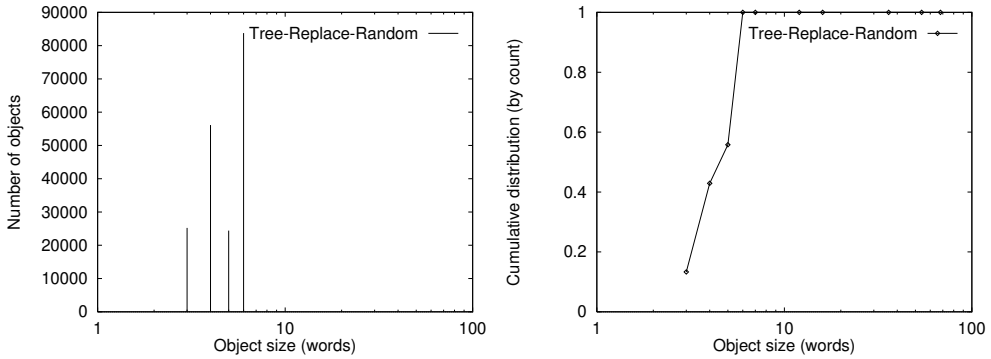


(c) Distribution weighted by lifetime (raw and cumulative).

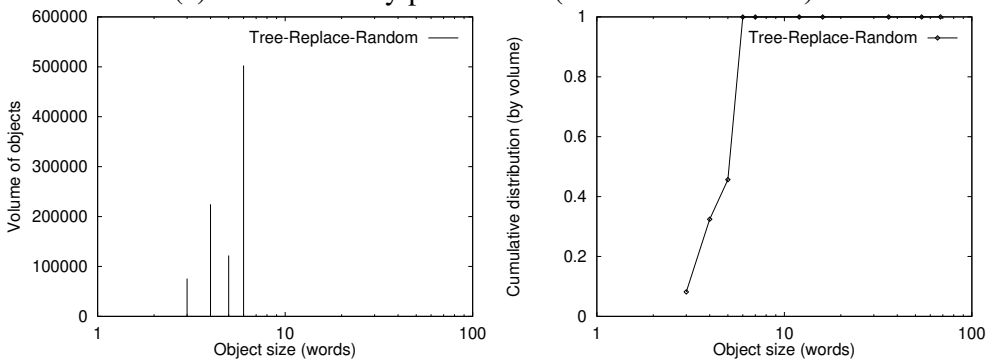


(d) Distribution weighted by lifetime and volume (raw and cumulative).

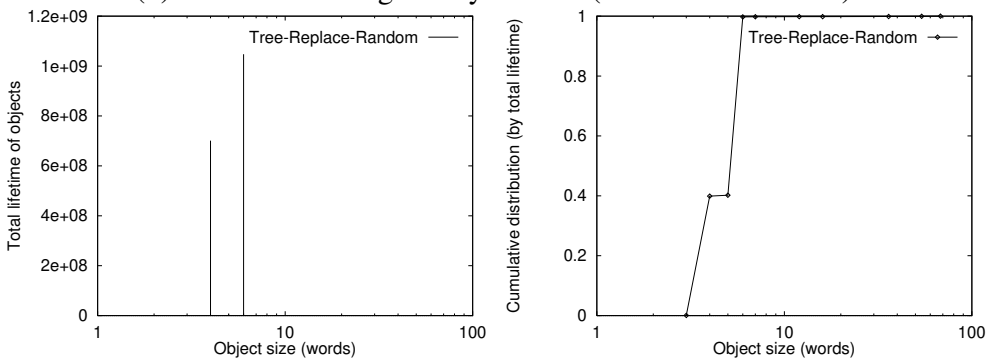
Figure 4.23. Object size distribution: Tree-Replace-Binary.



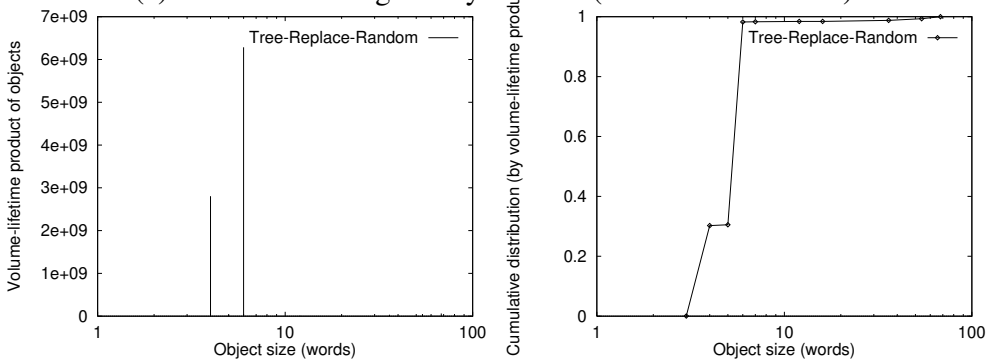
(a) Distribution by plain count (raw and cumulative).



(b) Distribution weighted by volume (raw and cumulative).

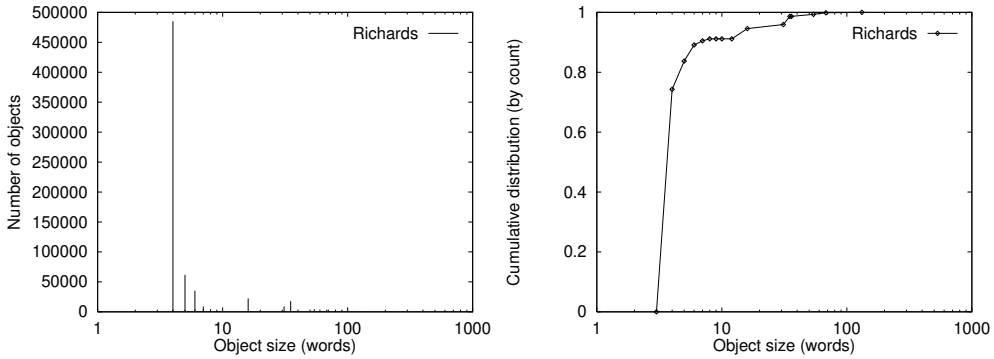


(c) Distribution weighted by lifetime (raw and cumulative).

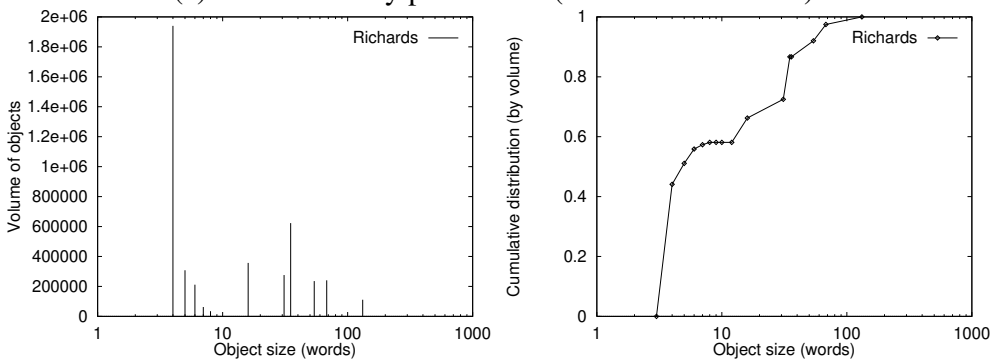


(d) Distribution weighted by lifetime and volume (raw and cumulative).

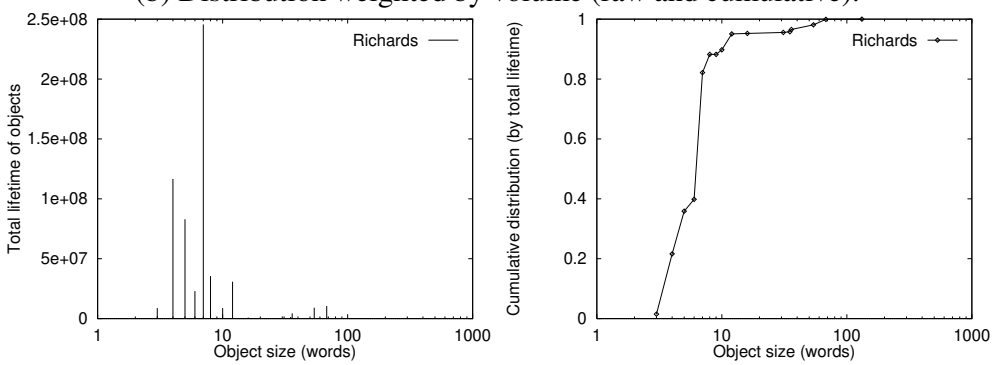
Figure 4.24. Object size distribution: Tree-Replace-Random.



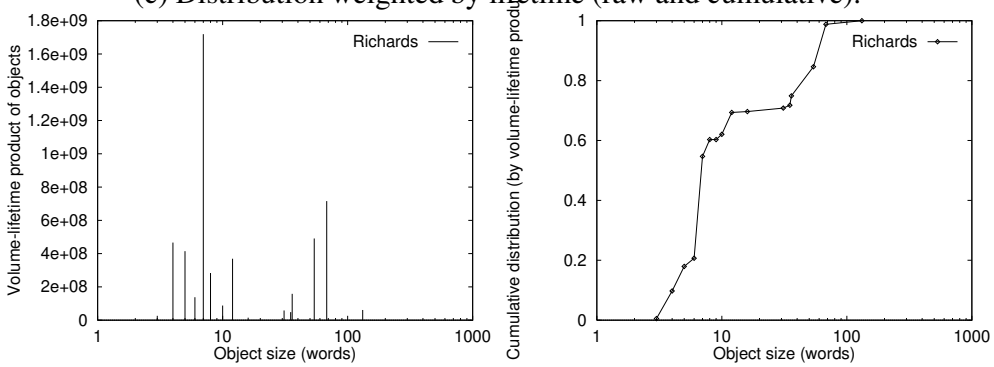
(a) Distribution by plain count (raw and cumulative).



(b) Distribution weighted by volume (raw and cumulative).

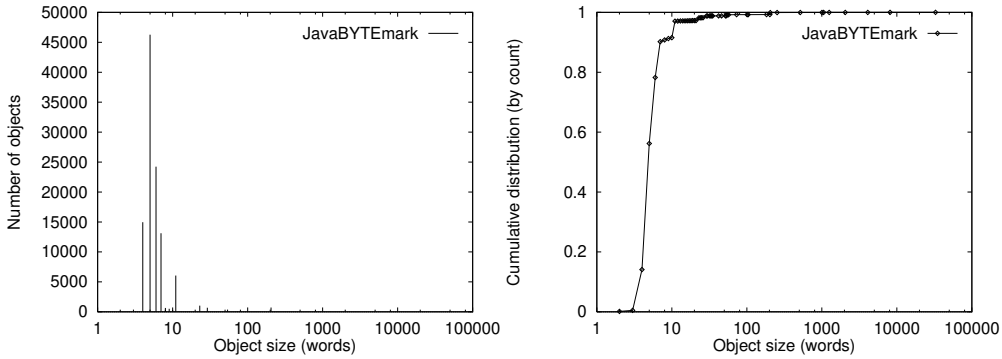


(c) Distribution weighted by lifetime (raw and cumulative).

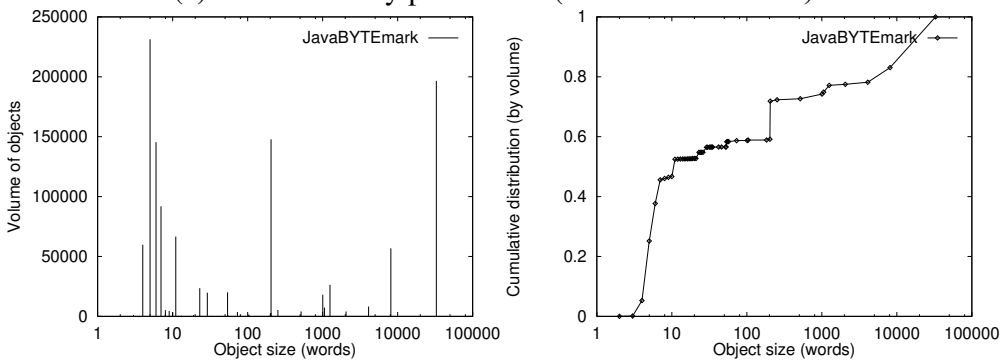


(d) Distribution weighted by lifetime and volume (raw and cumulative).

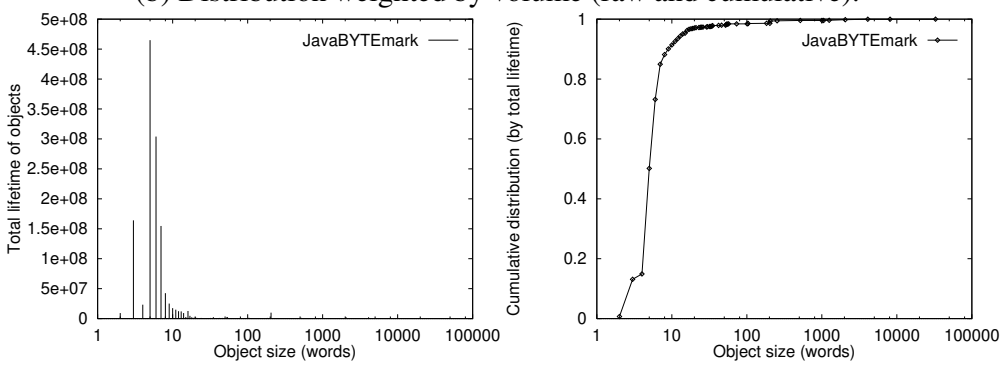
Figure 4.25. Object size distribution: Richards.



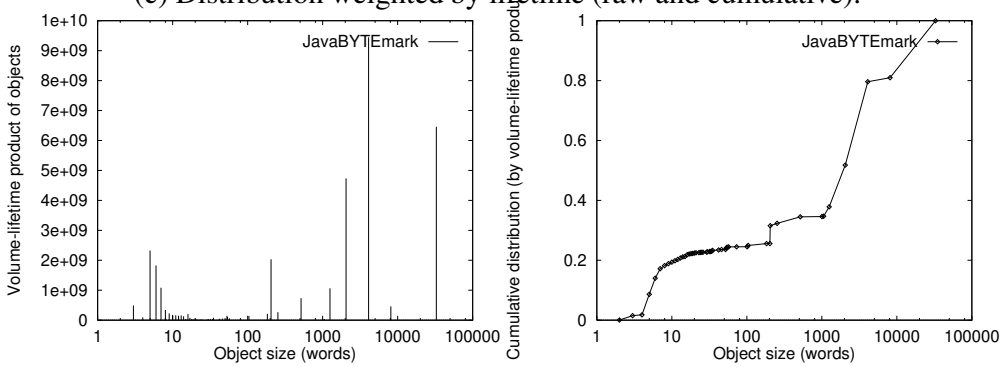
(a) Distribution by plain count (raw and cumulative).



(b) Distribution weighted by volume (raw and cumulative).

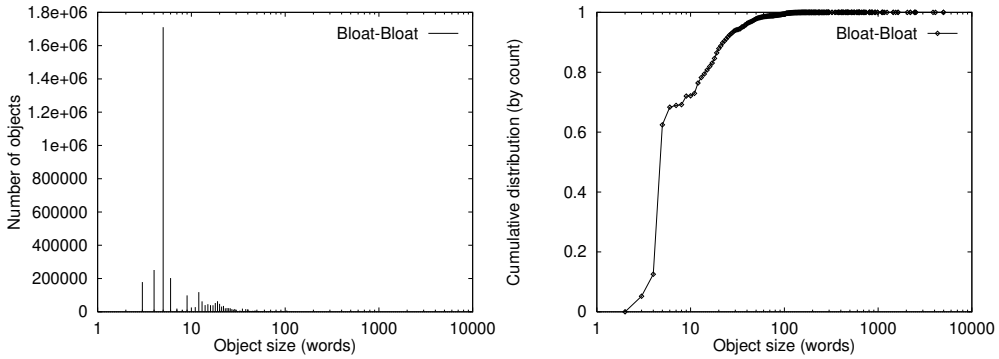


(c) Distribution weighted by lifetime (raw and cumulative).

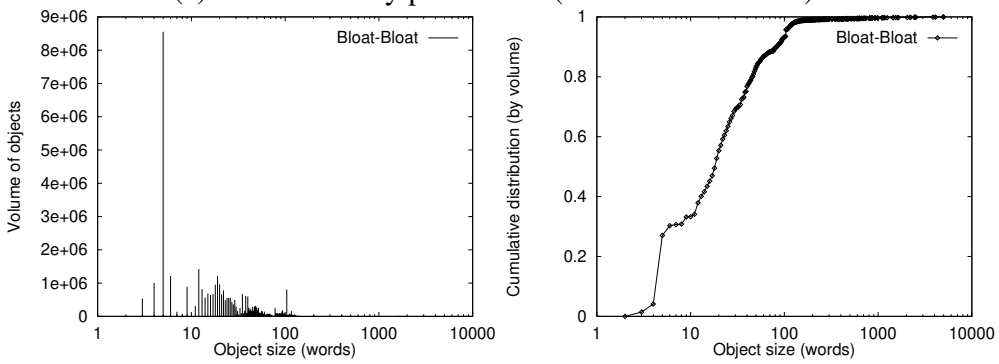


(d) Distribution weighted by lifetime and volume (raw and cumulative).

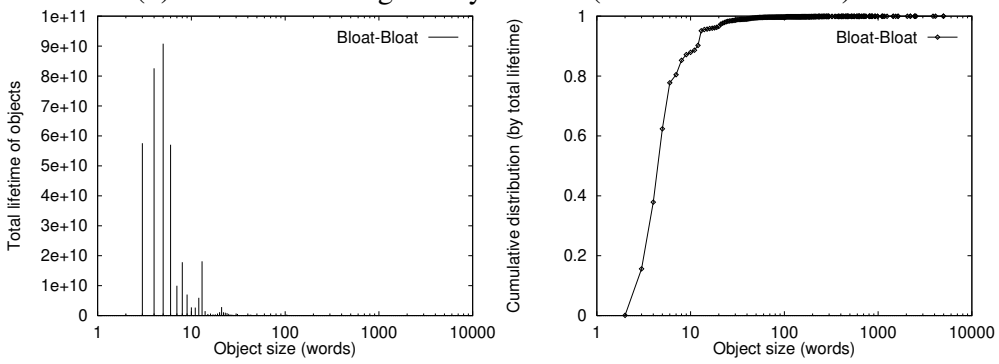
Figure 4.26. Object size distribution: JavaBYTEmark.



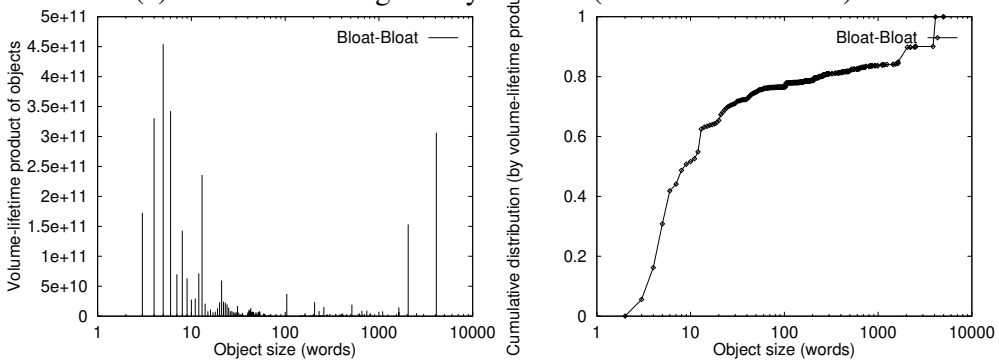
(a) Distribution by plain count (raw and cumulative).



(b) Distribution weighted by volume (raw and cumulative).

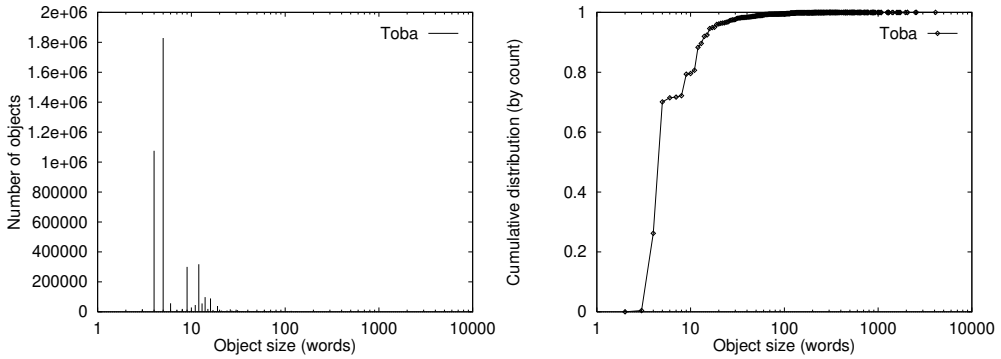


(c) Distribution weighted by lifetime (raw and cumulative).

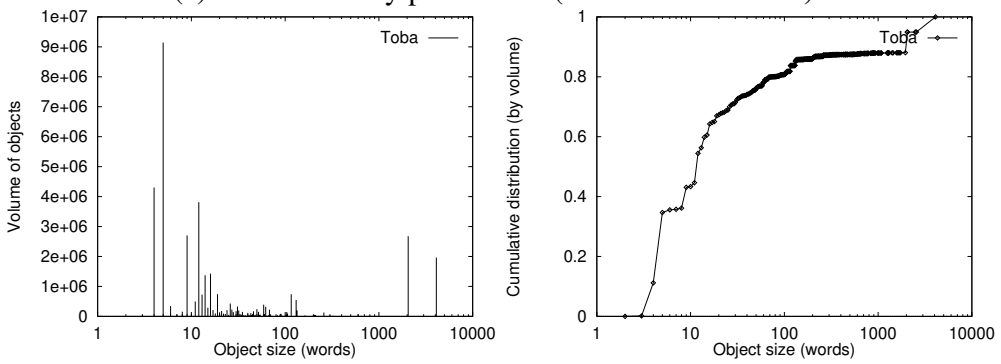


(d) Distribution weighted by lifetime and volume (raw and cumulative).

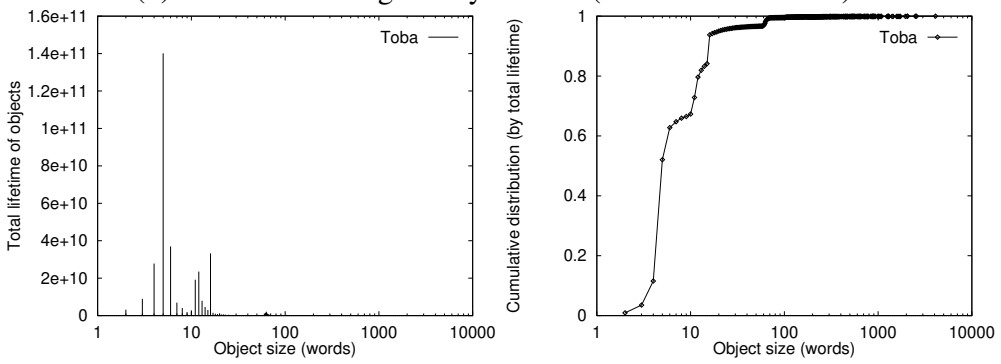
Figure 4.27. Object size distribution: Bloat-Bloat.



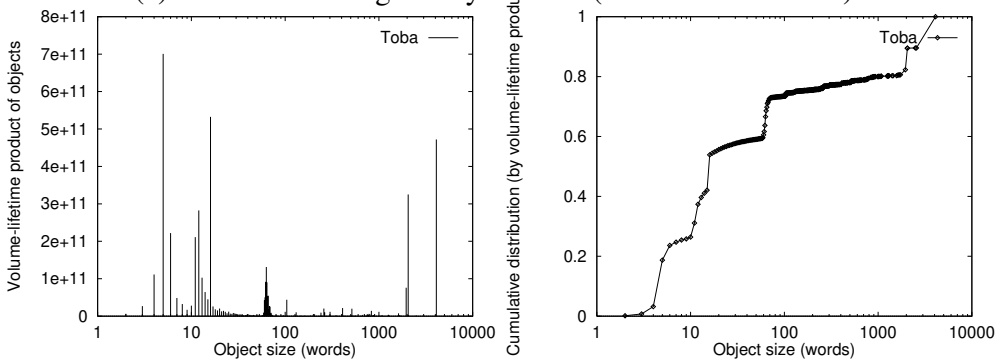
(a) Distribution by plain count (raw and cumulative).



(b) Distribution weighted by volume (raw and cumulative).

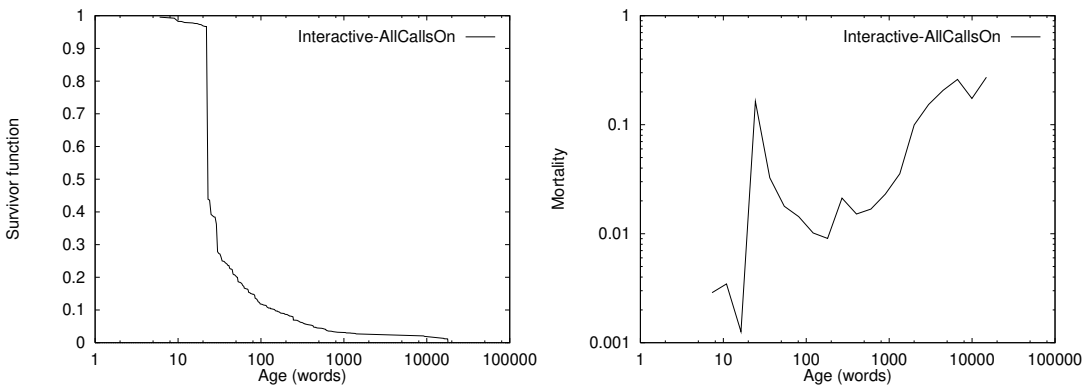


(c) Distribution weighted by lifetime (raw and cumulative).



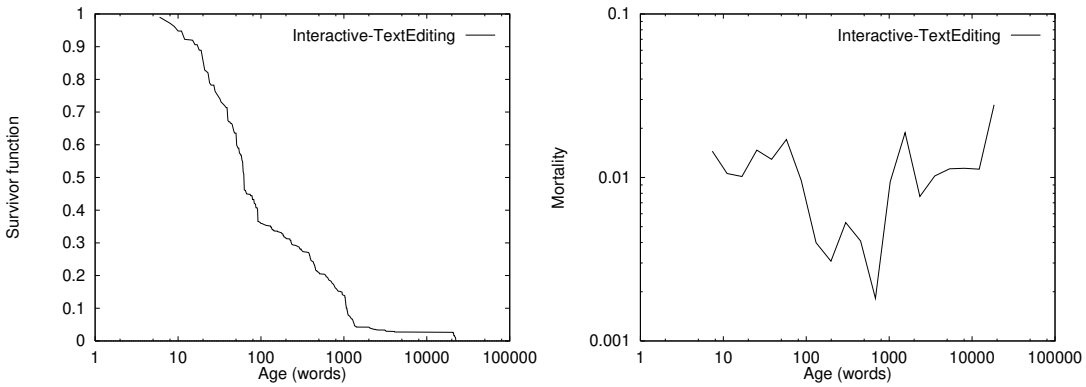
(d) Distribution weighted by lifetime and volume (raw and cumulative).

Figure 4.28. Object size distribution: Toba.



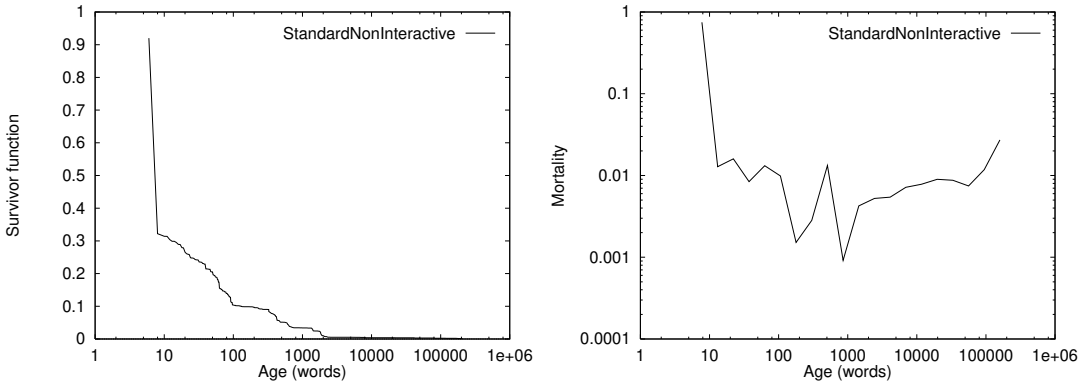
(a) Survivor function (b) Mortality function, smoothed

Figure 4.29. Object lifetime distribution: Interactive-AllCallsOn.



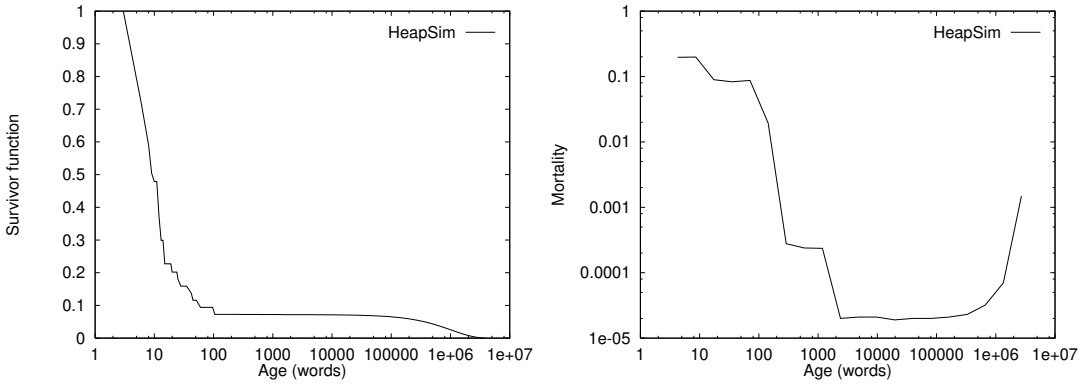
(a) Survivor function (b) Mortality function, smoothed

Figure 4.30. Object lifetime distribution: Interactive-TextEditing.



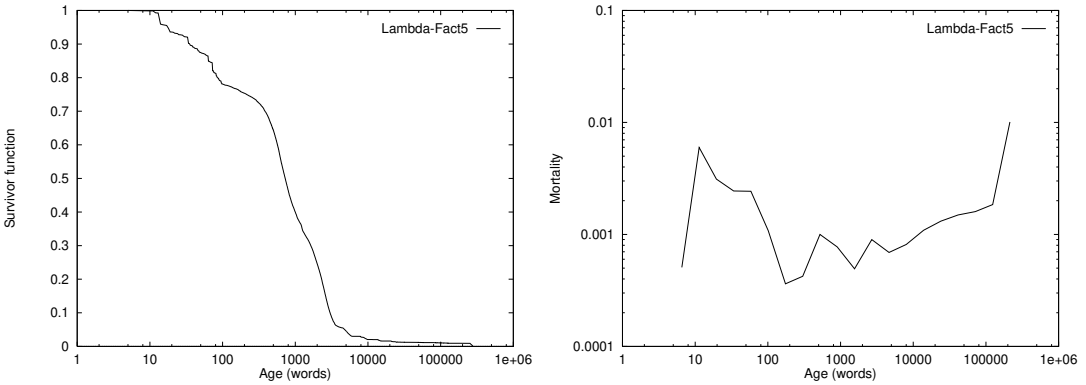
(a) Survivor function (b) Mortality function, smoothed

Figure 4.31. Object lifetime distribution: StandardNonInteractive.



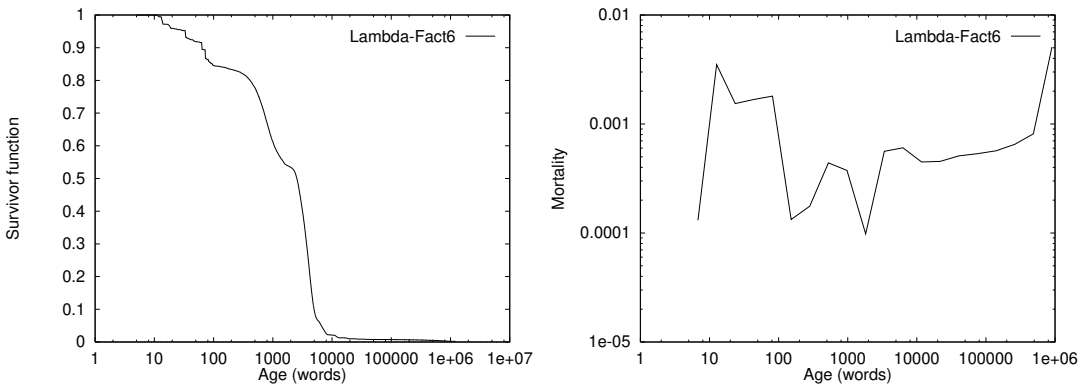
(a) Survivor function (b) Mortality function, smoothed

Figure 4.32. Object lifetime distribution: HeapSim.



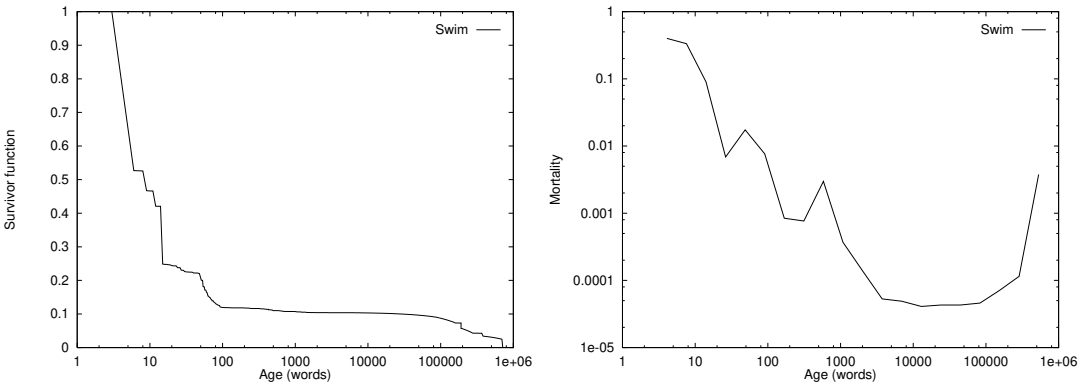
(a) Survivor function (b) Mortality function, smoothed

Figure 4.33. Object lifetime distribution: Lambda-Fact5.



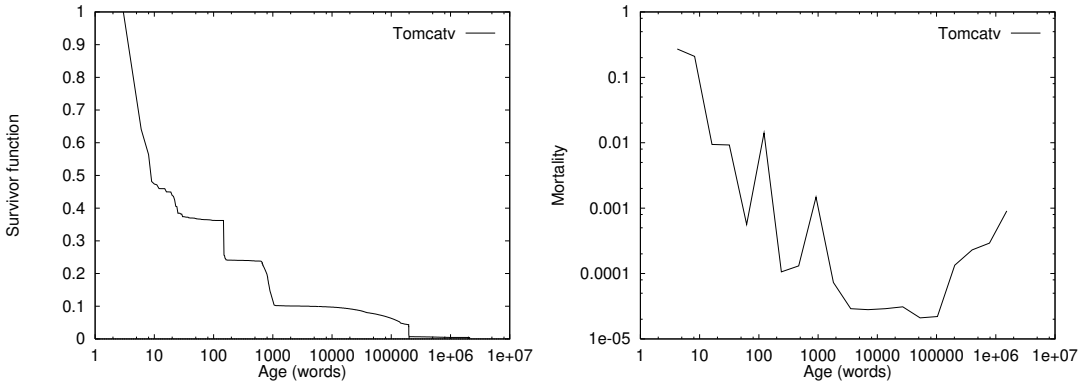
(a) Survivor function (b) Mortality function, smoothed

Figure 4.34. Object lifetime distribution: Lambda-Fact6.



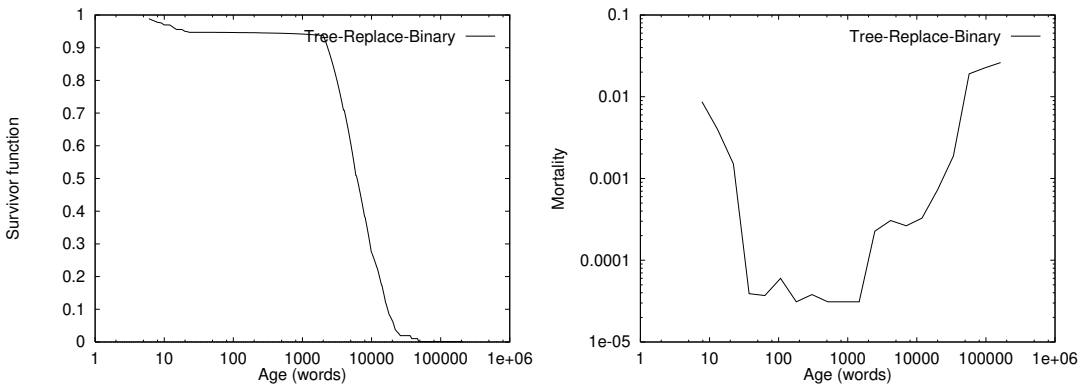
(a) Survivor function (b) Mortality function, smoothed

Figure 4.35. Object lifetime distribution: Swim.



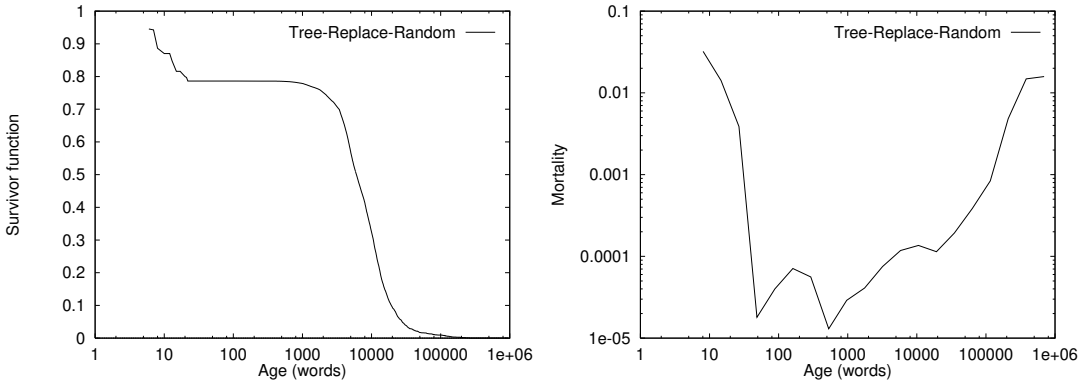
(a) Survivor function (b) Mortality function, smoothed

Figure 4.36. Object lifetime distribution: Tomcatv.



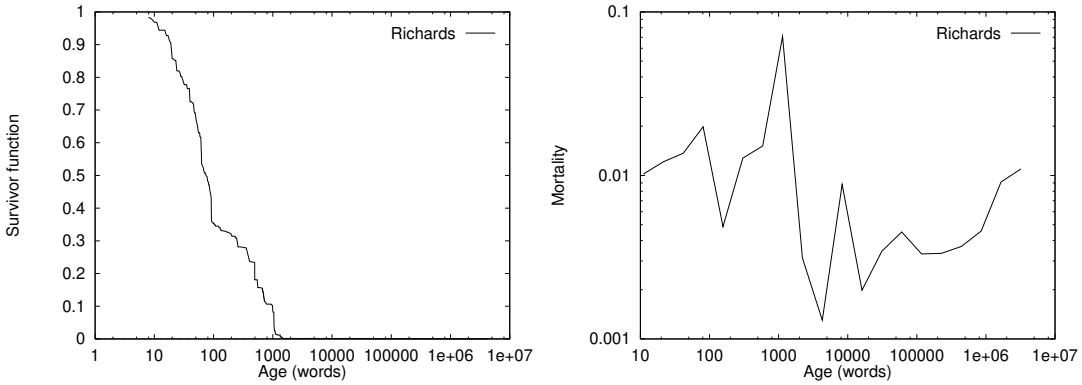
(a) Survivor function (b) Mortality function, smoothed

Figure 4.37. Object lifetime distribution: Tree-Replace-Binary.



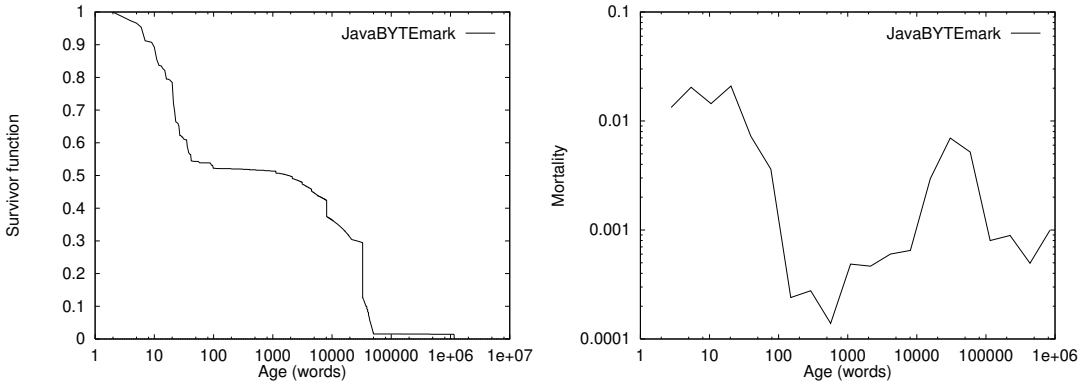
(a) Survivor function (b) Mortality function, smoothed

Figure 4.38. Object lifetime distribution: Tree-Replace-Random.



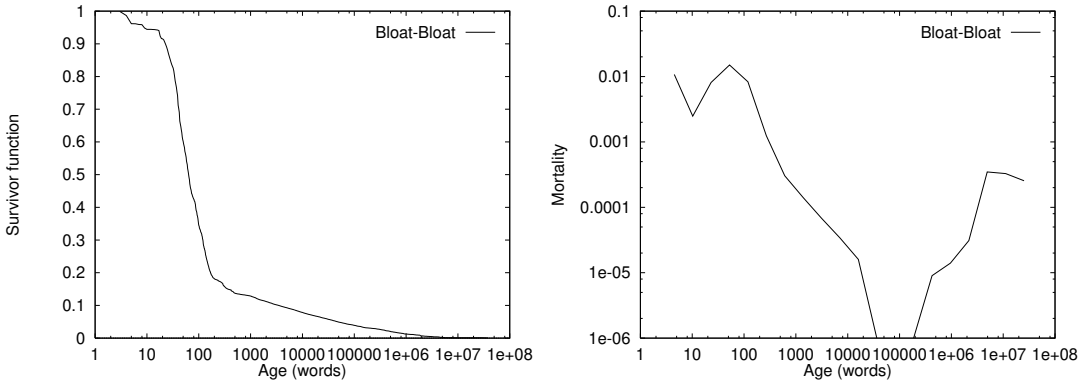
(a) Survivor function (b) Mortality function, smoothed

Figure 4.39. Object lifetime distribution: Richards.



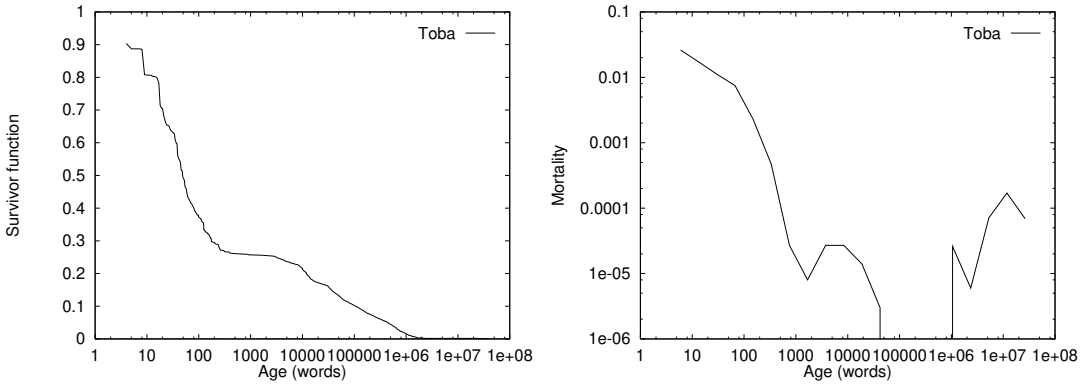
(a) Survivor function (b) Mortality function, smoothed

Figure 4.40. Object lifetime distribution: JavaBYTEmark.



(a) Survivor function (b) Mortality function, smoothed

Figure 4.41. Object lifetime distribution: Bloat-Bloat.



(a) Survivor function (b) Mortality function, smoothed

Figure 4.42. Object lifetime distribution: Toba.

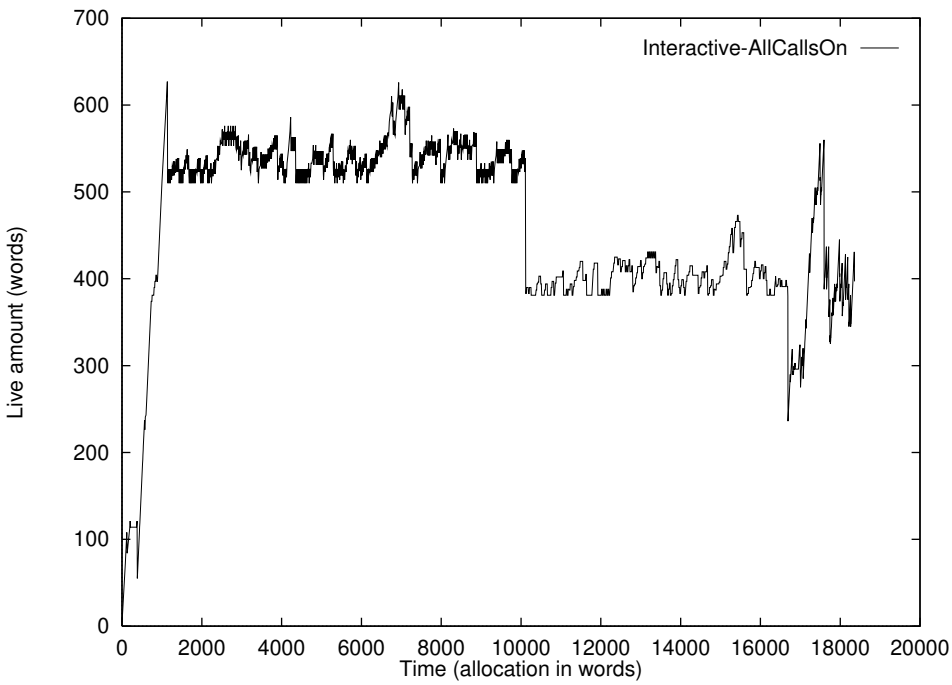


Figure 4.43. Live profile: Interactive-AllCallsOn.

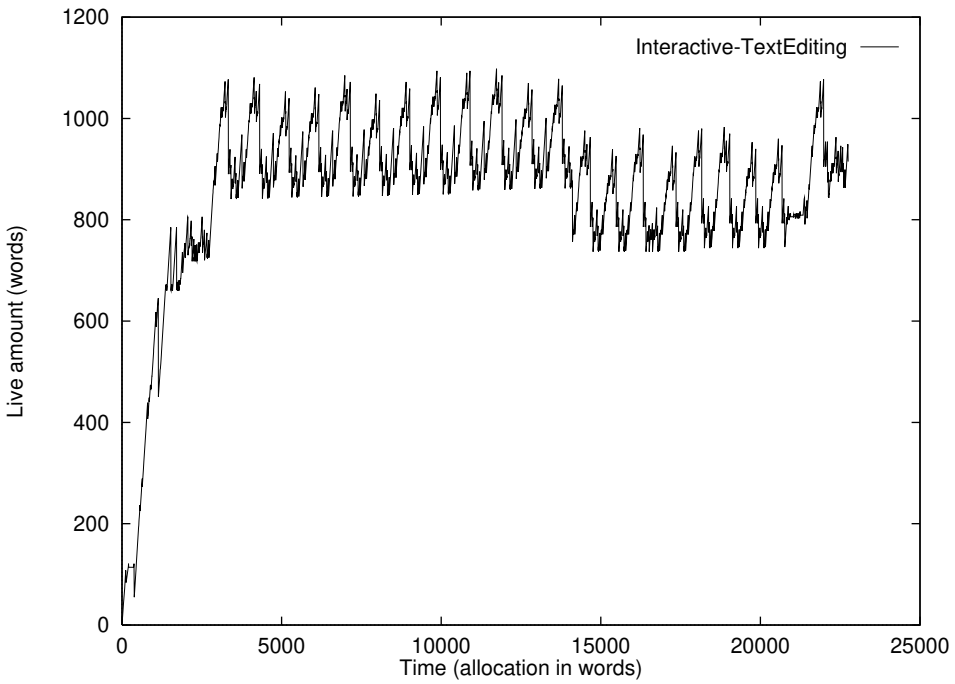


Figure 4.44. Live profile: Interactive-TextEditing.

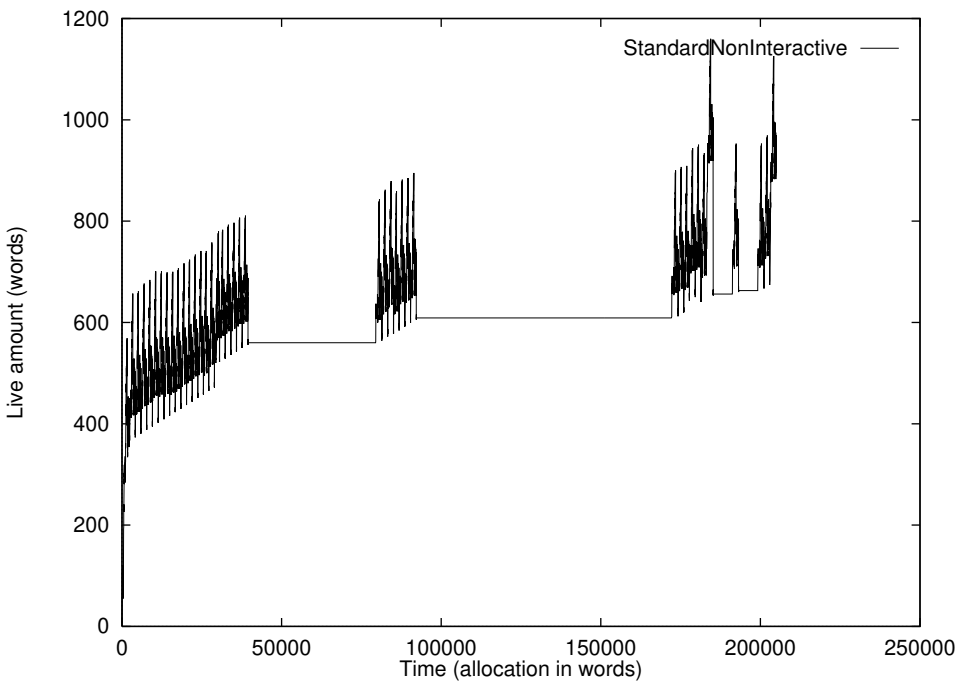


Figure 4.45. Live profile: StandardNonInteractive.

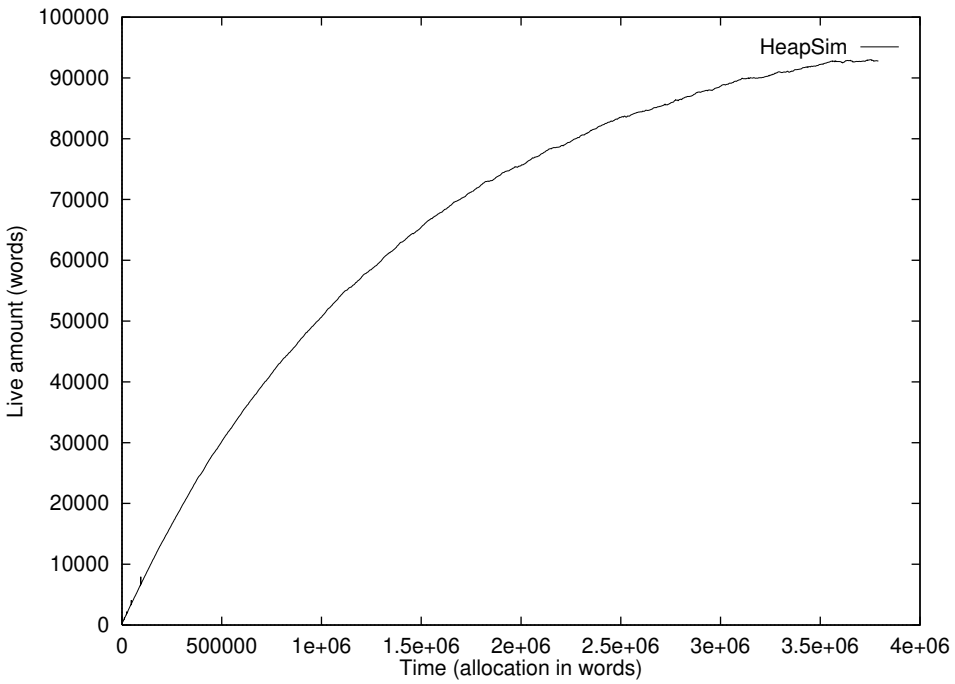


Figure 4.46. Live profile: HeapSim.

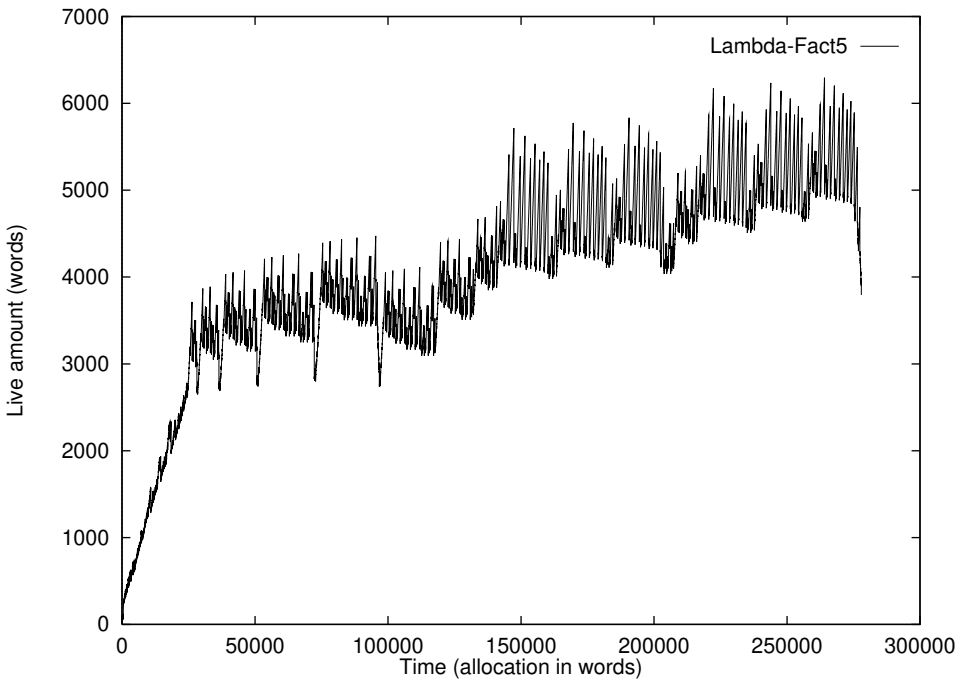


Figure 4.47. Live profile: Lambda-Fact5.

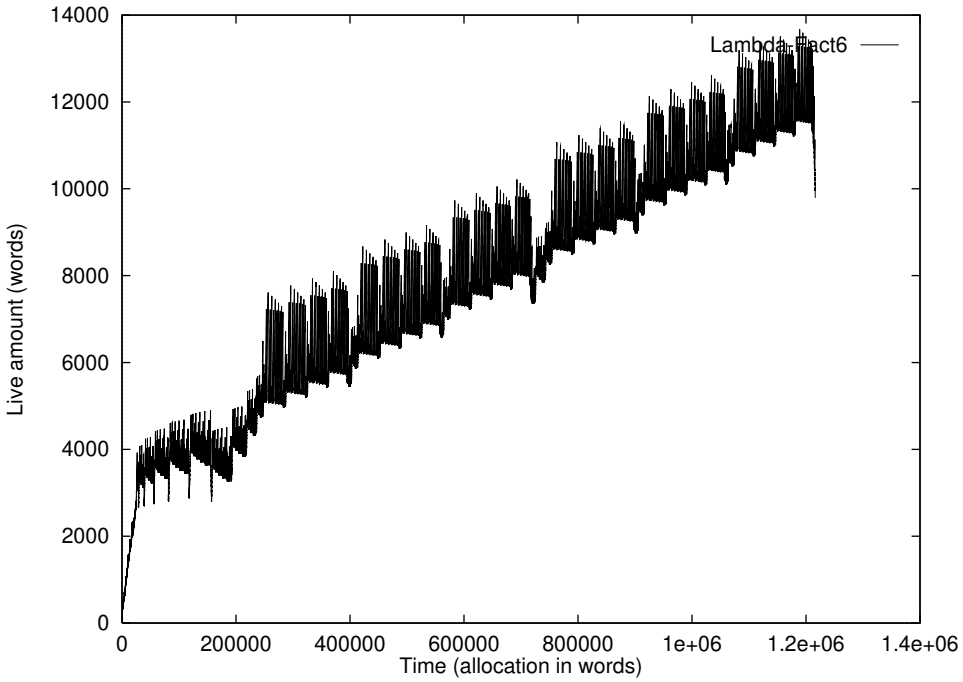


Figure 4.48. Live profile: Lambda-Fact6.

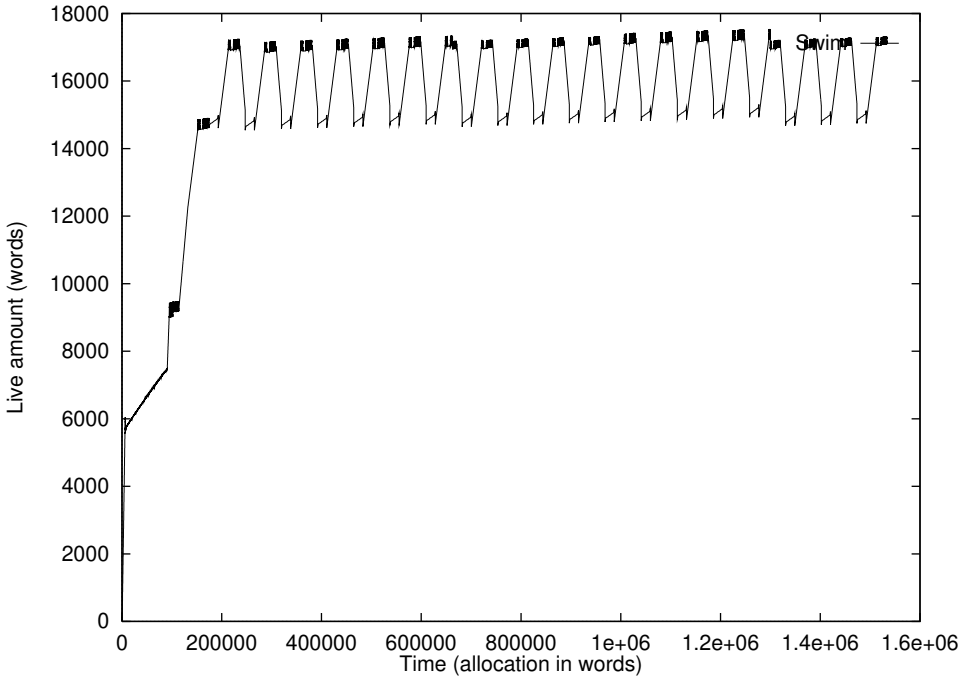


Figure 4.49. Live profile: Swim.

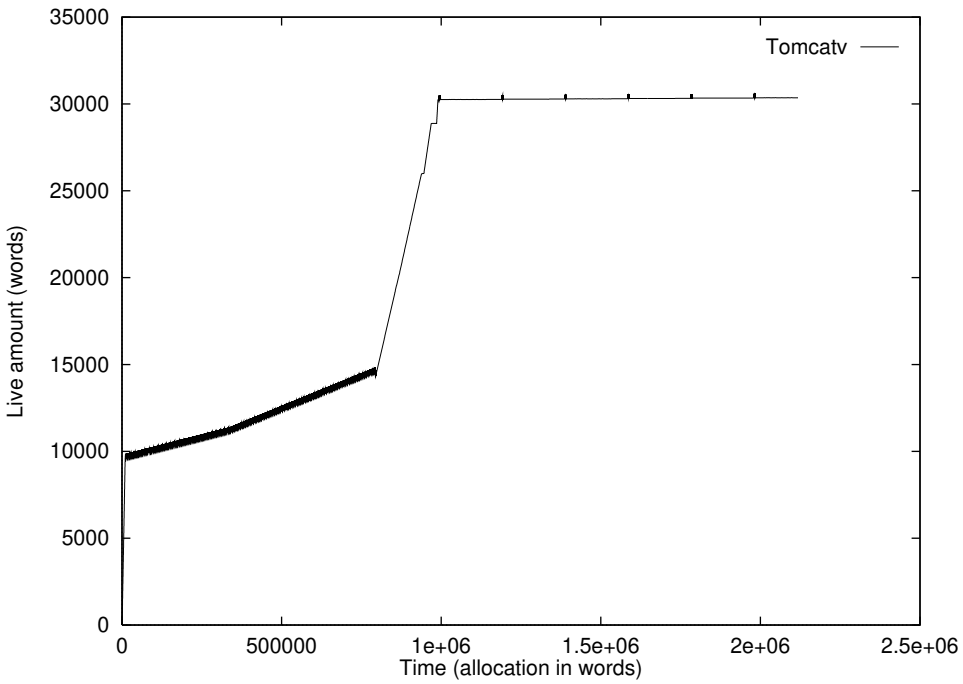


Figure 4.50. Live profile: Tomcatv.

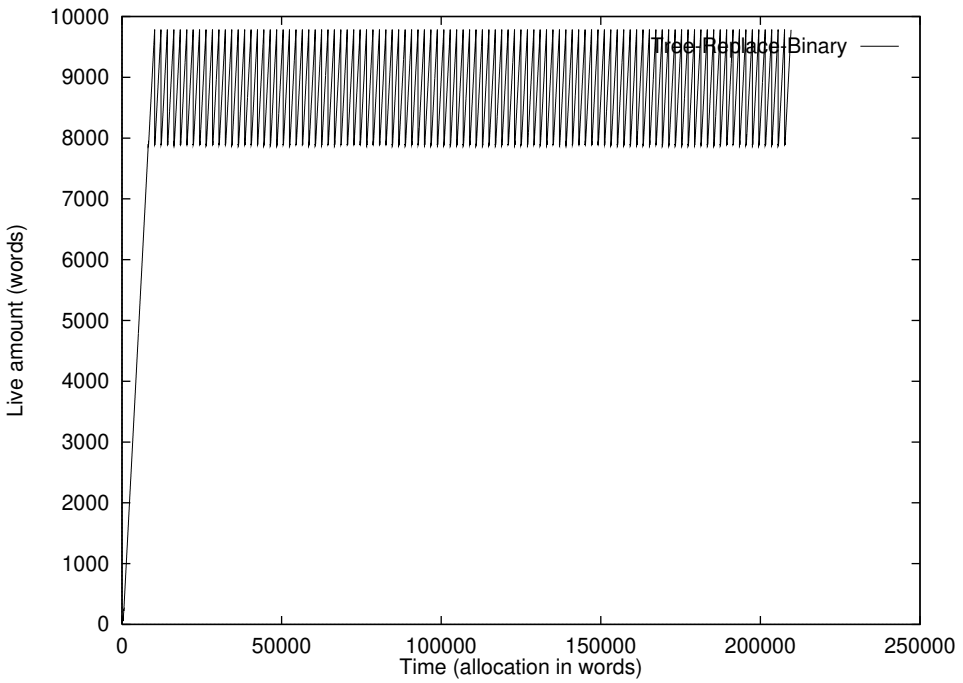


Figure 4.51. Live profile: Tree-Replace-Binary.

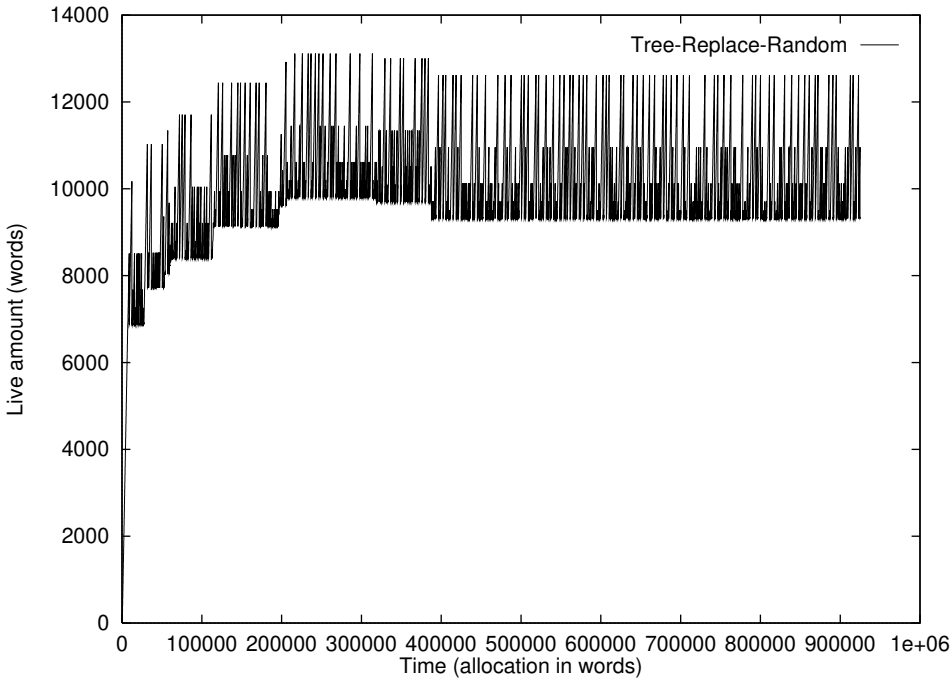


Figure 4.52. Live profile: Tree-Replace-Random.

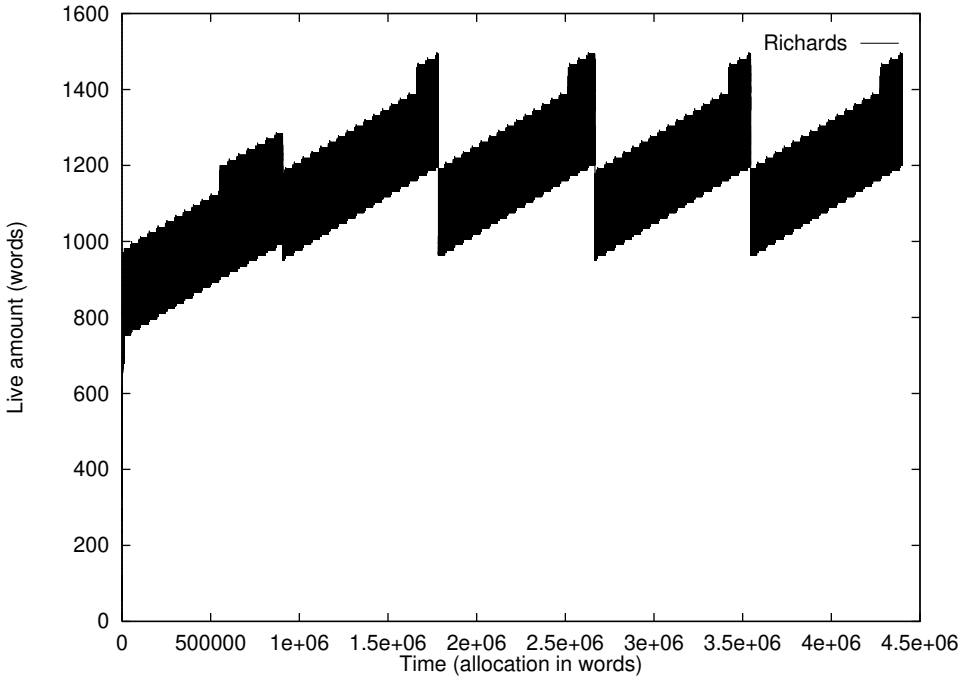


Figure 4.53. Live profile: Richards.

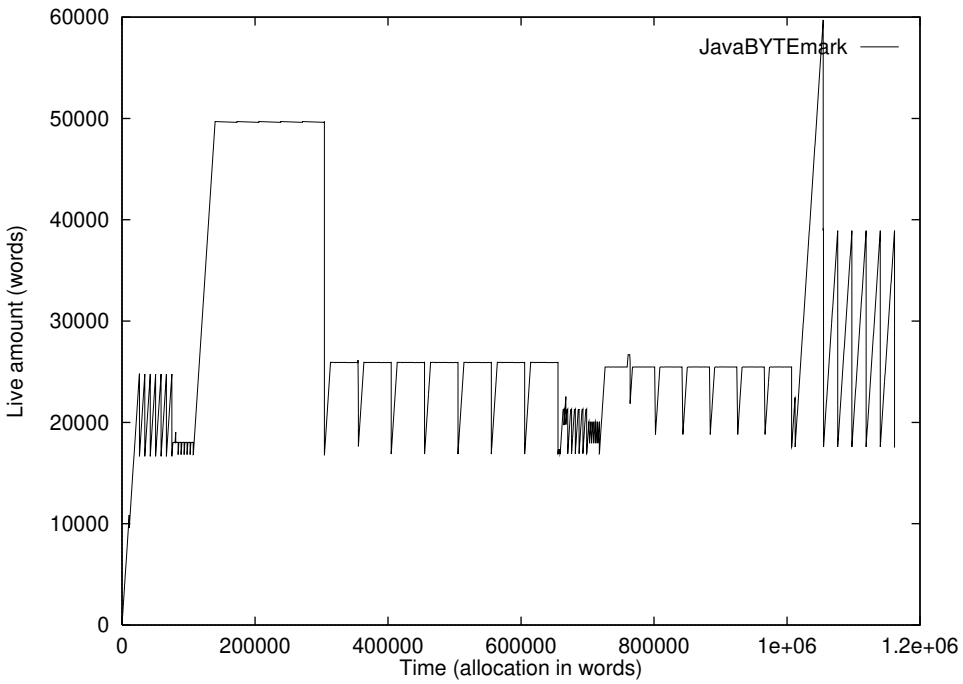


Figure 4.54. Live profile: JavaBYTEmark.

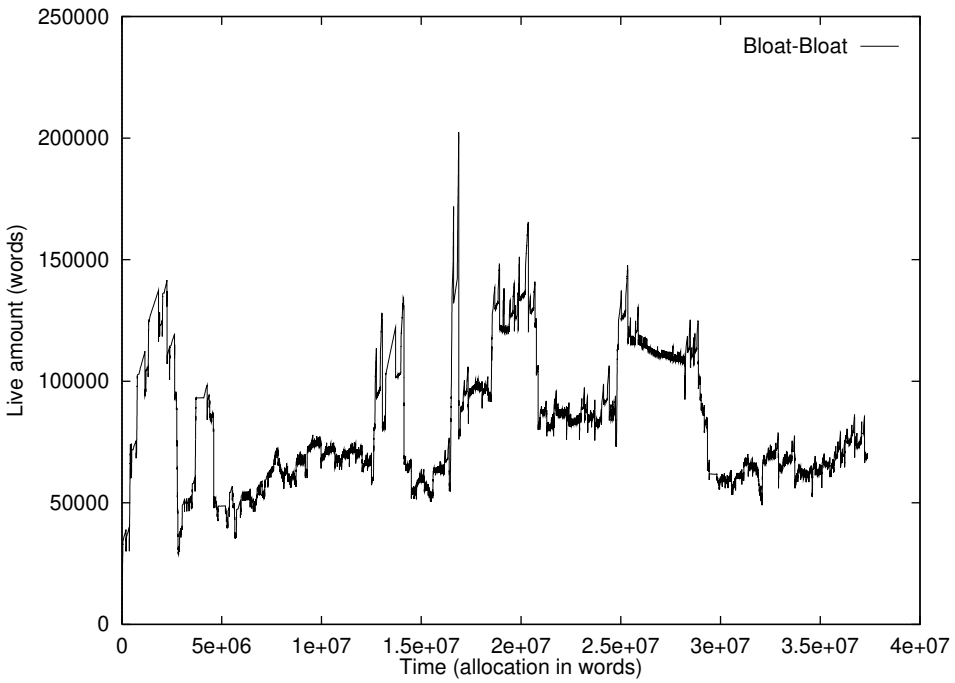


Figure 4.55. Live profile: Bloat-Bloat.

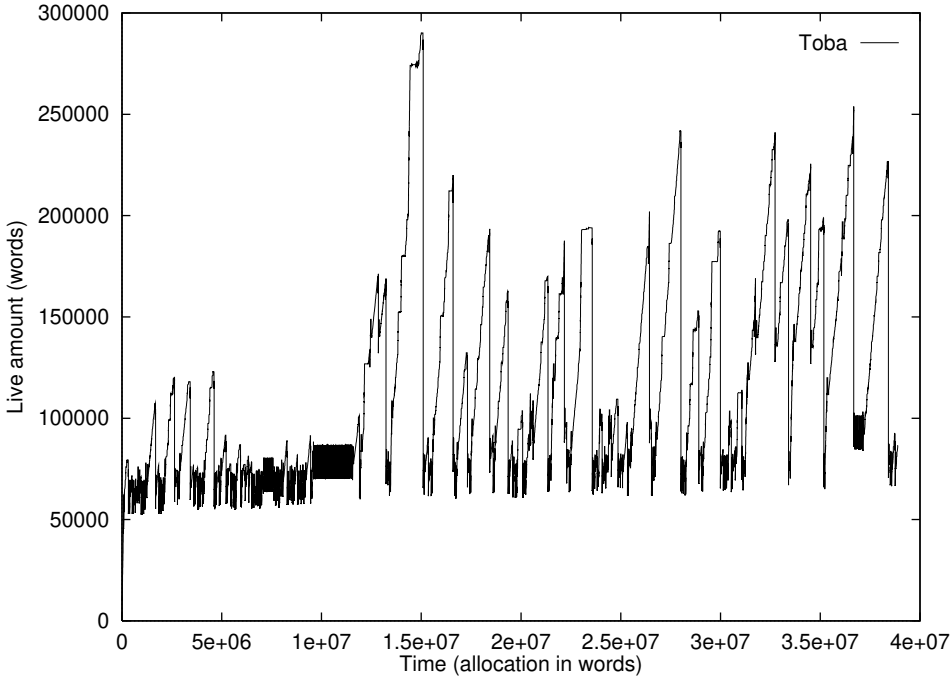


Figure 4.56. Live profile: Toba.

CHAPTER 5

EVALUATION: COPYING COST

In this chapter we focus on the copying cost of collection to the exclusion of other cost factors. We look at the measured copying cost of the various age-based algorithms, including the traditional generational youngest-first collectors and the newly proposed deferred older-first collector. We examine the contribution of excess retention to the copying cost, and search for the lower bounds of copying cost given the constraint of heap size, by looking at a hypothetical optimal collector and at different tentative adaptive collectors.

5.1 Method for evaluating different configurations of a collector

The copying cost of a given garbage collection scheme depends primarily on the size of the available heap space and the settings of configuration parameters. The available heap space is described by a single number, V, since in this study that space is constant. For collection schemes with fixed size of the collected region (FC), that size is the only configuration parameter, and is conveniently expressed as a fraction g of total heap volume V, thus $g = \frac{C}{V}$.

For each benchmark, a suitable set of values is chosen for V. The heap volume must be at least equal to the maximum amount of live data in the program. The performance of any collector, however, can be expected to be extremely poor if V is too close to this minimum. We let V assume up to 9 discrete values between 1.2 and 6 times the minimum, in geometric increments of 20%. Note that fewer than 9 heap sizes are reported for some benchmarks: when the criterion for benchmark relevance discussed above (more than 10 non-generational collections) is not met for the largest tested heap sizes, they are not reported. A suitable set of values is chosen for g as well: we let the fraction assume 20 discrete values between 0 and 1.

For each pair of V and g values, we performed a simulation of the benchmark. For a given heap size V, by varying the configuration parameter g (how *much* data is collected each time), we explore its effects on copying cost, within a fixed strategy of *where* to choose the data to collect.

Similarly, for the generational schemes, specifically 2GYF and 3GYF, the fraction of heap space dedicated to the nursery is the primary configuration parameter. For 2GYF it is the only parameter for a given heap size, and for 3GYF there is an additional parameter in the choice of division of the remainder of the heap between the middle and the oldest generation, but, as we shall see, its influence on performance is less pronounced.

5.1.1 Results

We present the results for each benchmark in turn, and for each benchmark we give a series of plots, ordered by increasing heap size. For each heap size, we show in separate plots (a) the performance of the FC schemes as a function of the fixed fraction collected, and (b) the performance of the GYF schemes as a function of the nursery size.

Each of the figures in this section (Figures 5.7–5.110) consists of two plots, on the left (a) the plot for FC schemes and on the right (b) the plot for GYF schemes. The vertical axis (the ordinate) has the same meaning in both: it gives the relative mark/cons ratio ρ, which is the mark/cons ratio of a particular scheme relative to the mark/cons ratio of the non-generational collector for the same heap size. The scale on the vertical axis is the same for FC and for GYF, and for all heap sizes for the same benchmark. The horizontal axis (the abscissa) in the FC plots gives the size of the collected region expressed as the fraction g of heap size. In the GYF plots, the horizontal axis gives the size of the nursery as a fraction of heap size. It would be misleading to show both in the same graph, since the size of the collected region in GYF is occasionally the whole heap. In plots (a), we have drawn the curves for the following schemes:

FC-TOF, FC-TYF, FC-DOF, FC-DYF, and FC-ROF.[1] In plots (b), we have drawn a curve for the scheme 2GYF, whereas the measurements for the 3GYF scheme are presented as a scatter plot, since the same nursery size can be combined with different divisions of the remaining space into the two older generations. These plots permit us to analyze how the choice of configuration affects the copying cost. Furthermore, for FC schemes, the configuration determines the maximum amount that can be copied on each collection, thus it (approximately) bounds pause times.

5.1.1.1 The FC-TOF collector

We observe that in all benchmarks save the two Tree-Replace programs, FC-TOF performs poorly. Its copying cost is generally higher than that of non-generational collection: $\rho > 1$. When the size of the collected region is close to the whole heap size, FC-TOF performs as well as non-generational, which is to be expected, but when the collected region is reduced, the collection cost rises dramatically, and ultimately, the collector fails. This result is easily explained by the presence of some amount of permanent data in the heap: permanent objects are the oldest objects, hence the collector copies them repeatedly, and if the collected region is so small that it contains nothing but permanent objects, the collector finds no garbage and fails.

For the programs Tree-Replace-Binary and Tree-Replace-Random, on the contrary, FC-TOF is the best scheme, and its copying cost is as low as one half that of non-generational collection. In these (admittedly synthetic) benchmarks, the amount of permanent data is relatively small—only the first several layers of a tree from the root are permanent. Moreover, the subtrees are chosen for replacement *at random*, which means that older subtrees do eventually become garbage. Our intuitive argument in Chapter 1 and Figure 1.1, that the longer the collector waits the more likely it is that an object is garbage, holds in these programs.

[1] If a curve is entirely missing from a plot, it is because no configuration of that scheme successfully completes execution. (See Section 4.3.1.)

5.1.1.2 The FC-ROF collector

The FC-ROF collector behaves just like the FC-TOF collector for large collected region sizes. However, at smaller collected region sizes, the FC-ROF collector diverges from FC-TOF, and is sometimes able to avoid its poor behavior. For instance, in benchmark Swim, with a heap size of 38733 words (Figure 5.53(a), p. 158), whereas the performance of FC-TOF degrades rapidly as g decreases below 0.6, FC-ROF only degrades to $\rho = 1.5$, and then returns to lower levels—but never below 1. An interesting reversal occurs for the two programs Tree-Replace-Binary and Tree-Replace-Random, where FC-TOF works well, and FC-ROF degrades precisely when the size of the collected region is reduced below 0.6 (Figure 5.80(a), p. 169). This phenomenon deserves further investigation.

5.1.1.3 The FC-TYF collector

The FC-TYF collector joins FC-TOF and FC-ROF in the class of collectors that only work well when the collected region is nearly the whole heap. Otherwise, this collector almost always performs worse than the non-generational collector. This is not surprising; if this collector could work in a steady state, it would never examine objects to the left of its collection window. In other words, once a young object survives one nursery collection, it is "tenured". It is surprising that this collector works at all, and this must be due to the fact that programs show phase behavior and do not enter a truly steady state.

5.1.1.4 The FC-DOF collector

The FC-DOF collector is consistently better than other FC schemes, except on the two programs Tree-Replace-Binary and Tree-Replace-Random. It generally works better with *smaller* collected region sizes, except for Toba and Lambda. For many examined programs there exists, for sufficiently large heaps, a characteristic region size below which the FC-DOF collector works particularly well. This passage from acceptable performance to exceptionally good performance is sometimes abrupt, as for StandardNonInteractive at region size $g = 0.9$ (Figure 5.25(a), p. 145), and sometimes gradual, as for Richards, when region size is reduced from

$g = 0.8$ to $g = 0.4$ (Figure 5.85(a), p. 171). The FC-DOF collector is the best among the FC schemes, and we shall investigate its behavior in greater detail in the remainder of this chapter and in Chapter 6.

5.1.1.5 The FC-DYF collector

The FC-DYF collector usually performs better than non-generational collection. However, it is always substantially worse than FC-DOF, from which it differs only in the direction of collection window motion. Its copying cost is remarkably insensitive to the region size parameter for those values where the collector succeeds; these are, however, limited to medium values, whereas for small windows as well as for large windows, this collector is prone to failure. Both the poor performance and the failures (as the ultimate manifestation of poor performance) are caused by the fact that the collection window sweeps the heap too fast. The intuition that survivors of one collection should not be in the collected region of the following collection conspires with the younger-to-older window motion direction in FC-DYF to cause fast window motion, and too many visits to the far ends of the heap where there are many survivors.

5.1.1.6 The GYF collectors

The curve of the 2GYF collector copying cost and the patterns of the 3GYF scatter points reflect the presence of different modes of operation of the generational collector, corresponding to different numbers of younger-generation collections between successive older-generation collections (Figure 5.50(b), p. 157). The resultant 2GYF copying cost is therefore sometimes lowest for medium values of nursery size, as in InteractiveAllCallsOn (Figure 5.12(b), p. 140), sometimes it is lowest for small values of nursery size, as in Swim (Figure 5.50(b), p. 157), and sometimes it is lowest for *large* values of nursery size, as in Lambda-Fact6 (Figure 5.44(b), p. 154).

It will also be observed that the scatter points for 3GYF are clustered closely, and usually not far below or above the line corresponding to 2GYF. Thus, the copying cost of 3GYF is usually comparable to that of 2GYF. Occasionally we see a configuration of 3GYF with only

half the copying cost of 2GYF with the same nursery size, as in Lambda-Fact6 (Figure 5.45(b), p. 154); on the other hand, there are programs where no 3GYF configuration is better than 2GYF, as in Toba (Figure 5.102(b), p. 179), hence the added overhead[2] of 3GYF is only detrimental.

5.2 Method for comparing various collection schemes

We now summarize the findings of the preceding section. In order to compare different collection schemes, we shall restrict each to operate within the same heap size, but we shall let it assume the *best* configuration. In other words, we give each scheme the benefit of the best static choice of the size of the collected region, or the sizes of the various generations. The plots in Figures 5.111–5.124 (pp. 183–190) show the copying cost for these best configurations. Along the horizontal axis is the heap size allotted, and along the vertical axis the relative mark/cons ratio ρ. There are 8 curves in each plot, for the collection schemes FC-TOF, FC-TYF, FC-DOF, FC-DYF, FC-ROF, NGYF,[3] 2GYF, and 3GYF.

In the plots for Interactive-TextEditing (Figure 5.112), StandardNonInteractive (Figure 5.113), HeapSim (Figure 5.114), Richards (Figure 5.121), and Bloat-Bloat (Figure 5.123), there is a clear separation between the schemes FC-TOF, FC-TYF, FC-DYF, and FC-ROF, which exhibit high copying cost, and the schemes FC-DOF, NGYF, 2GYF, and 3GYF, which exhibit low copying cost. In other programs the separation is not as clear, and indeed for Tree-Replace, the FC-TOF and FC-ROF schemes work best.

There is not a uniform dependence of ρ on heap size (for a specific collection scheme): whereas in Richards ρ decreases with heap size for the four better-performing schemes, in Bloat-Bloat and Swim it increases. Note that the *absolute* copying cost μ nevertheless decreases—ρ is relative to the cost of non-generational collection, which itself is decreasing. This fact is

[2] Recall the elision of the write barrier at collection time (Section 3.2.2).

[3] Since NGYF has no configuration parameters, it was not discussed in the preceding section. Heap size uniquely determines the value of ρ reported here for NGYF.

evidenced in Figures 5.125–5.138, pp. 191–198, which display the *absolute* copying cost, expressed as the mark/cons ratio μ, for the best configuration of each scheme.

With respect to copying cost, the newly introduced scheme FC-DOF is competitive with generational collectors for all examined programs, and markedly better for some (Interactive-AllCallsOn, Interactive-TextEditing, StandardNonInteractive, and Richards). For Richards with a heap size of 4914 words, the copying cost of 3GYF is 10.37 times higher than that of FC-DOF.

Observing the performance of all the collectors, we note that most programs permit region-based schemes to cut down the copying cost of collection significantly with respect to non-generational collection: for Interactive-AllCallsOn to the fraction 0.13; for Interactive-TextEditing to 0.18; for StandardNonInteractive to 0.045; for Richards to 0.029; and for HeapSim, Swim, Tomcatv, and Bloat to the vicinity of 0.2. However, the copying cost of Toba is not reduced below 0.6, Tree-Replace-Binary and Tree-Replace-Random are only reduced to about 0.5, and the most intransigent Lambda-Fact5 and Lambda-Fact6 are not reduced below 0.7.

5.3 Method for recognizing and measuring the excess retention cost

Let us examine a garbage collector that considers a collected region C smaller than the entire heap, in order to limit the amount of work done at each collection. The maximum copying cost is bounded by the size of the collected region, since in the worst case all objects within it may be live. The exact computation of the set of live objects, however, requires an examination (pointer tracing) of the entire heap, and not just C, at a cost not much lower than the cost of *collecting* the entire heap. Instead, collectors make a conservative approximation: they assume that all objects in the uncollected region U are live, hence that every pointer that crosses from U to C lies on a path from a root, even if that is not actually the case. *Excess retention* is that part of the collection copying cost which is caused by the inaccuracy of this assumption. On a single collection, the excess retention is manifested as an increase in the

volume of survivors. Since the volume freed is correspondingly reduced, the next collection must take place sooner, and the overall number of collections is increased.

To measure how large the excess retention part of the copying cost is, we devised a variant of collector simulation (deployed in all the simulated schemes) that does not assume liveness for the uncollected region. Instead, the survivor set S among the collected objects C is precisely the set of objects that are not *known-dead*. This manner of simulation is possible because our trace contains accurate liveness information, and can thus act as an oracle guiding the collection. The difference between the actual copying cost, and the copying cost as it would be with this oracle, gives the excess retention cost.

5.3.1 Results

Each of the figures for this section (Figures 5.139–5.242, pp. 199–243) consists of two plots, on the left (a) the plot for FC schemes and on the right (b) the plot for GYF schemes. The vertical axis (the ordinate) shows the excess retention for the configuration, expressed as the ratio of the actual copying cost to the oracular copying cost. Note that the variation of the excess retention ratio is so large that we could not maintain the same scale in the plots, thus visual comparisons between FC and GYF are discouraged. The horizontal axis (the abscissa) in the FC plots gives the size of the collected region expressed as the fraction g of heap size. In the GYF plots, the horizontal axis gives the size of the nursery as a fraction of heap size.

Let us first note that in some rare configurations, both for the FC and for the GYF schemes, the excess retention ratio is below 1, for instance in Interactive-AllCallsOn, Figure 5.141, p. 200. It may seem odd at first glance that the collector without the benefit of an oracle performs better than the one with it, but recall that, if the oracle makes an actual difference in the determination of the survivor set S, then the two collectors will perform their collections at different instants, and, as we discussed in Section 4.2.2.2, the choice of collection instants introduces a slight "random" effect. Having dispatched this anomaly, we note that excess retention displays widely different behavior for the several benchmarks, and for different schemes.

The excess retention of the GYF schemes is low, below 10% (ratio below 1.1) for most programs, with Lambda (Figures 5.166–5.181, pp.211–217), Tree-replace-Random (Figures 5.205–5.212, pp.228–230), and Toba (Figures 5.234–5.242, pp.240–243) as exceptions.

Excess retention does play a critical role in determining the performance of collectors for some programs. For instance, for Lambda-Fact6, we noticed that none of the collectors has a copying cost much lower than non-generational (Figure 5.48, p. 155), and now we can observe in the corresponding plot of excess retention (Figure 5.180, p. 216) that for a nursery size of one-half heap size, the excess ratio of 2GYF is 2, hence with an oracle the copying cost would be halved from $\rho = 1.5$ to $\rho = 0.75$. But the effect on FC-DOF is far stronger: the excess ratio is 50 for collected region size equal to 0.2 of heap size, hence with an oracle the copying cost would be reduced from $\rho = 3$ to $\rho = 0.06$. The previously noted intransigence of Lambda-Fact6 with respect to copying cost thus has its origin in a pointer structure that causes very high excess retention for those schemes (GYF, DOF) that we found to work well on other programs.

On the other hand, consider the benchmark Richards with a heap size of 4032, for which the copying costs are shown in Figure 5.85 on p. 171, and the excess retention in Figure 5.217 on p. 232. The actual copying cost of FC-DOF decreases as the size of the region collected is decreased from 0.8 down to 0.3, *even though* the excess retention ratio increases sharply from 1 to 8. Meanwhile the 2GYF collector enjoys negligible excess retention, yet its copying cost is much higher than that of FC-DOF.

In summary, excess retention sometimes dominantly influences performance, but sometimes its effects are negligible. Whereas it could have been surmised that schemes that collect regions in the older parts of the heap suffer from prohibitively high excess retention because of the supposed predominance of younger-to-older pointers, this case does not occur in our benchmarks.

5.4 Methods for examining the lower limits of copying cost and the feasibility of adaptation

We have seen that the GYF and FC-DOF schemes compete for lowest copying cost, and that FC-DOF is sometimes considerably better. We now examine if copying cost can be reduced further, and how. We remain within the class of FC schemes—thus each collection collects a region of fixed size—but now we allow more latitude in the choice of where the collected region lies within the heap.

5.4.1 Locally-optimal copying cost

We described the locally-optimal schemes in Section 3.1.3. It is straightforward to simulate them in an object-level garbage collection simulator. On each collection, we place the collection window at all possible positions in the heap, and for each position we perform a collection as usual, except for the final step of deleting the garbage found. Thus for each possible position we calculate the corresponding mark/cons ratio, then choose the position with the lowest one, and complete the collection with the window at that position.

The number of possible positions is at most the number of objects in the heap, and somewhat smaller because the window must wholly lie within the heap. However, it can be a large number if the heap consists of many small objects, hence this simulation can only be carried out for smaller benchmarks. In plots in Figures 5.243–5.266, pp. 244–253 (at the end of this chapter), we show the copying costs of the locally optimal schemes together with the age-based schemes considered before, for three representative benchmarks, StandardNonInteractive, Lambda-Fact5, and Tree-Replace-Random.

For StandardNonInteractive, the locally optimal collector only matches the performance of FC-DOF, and is in some configurations not as good as FC-DOF (recall from Section 3.1.3 that FC-OPT is not necessarily optimal globally!). For Lambda-Fact5, on the other hand, FC-OPT is significantly better than any of the realistic FC collectors, and any of the generational collectors, and achieves a copying cost of less than half of the non-generational collector's

cost. The situation is similar for Tree-Replace-Random, where FC-OPT has much lower cost than FC-TOF, the best realistic scheme considered for this benchmark. The explanation is straightforward in this case: the collected region is positioned to include exactly the objects of the subtree that has just been replaced (provided that the size of the collected region is not much larger than the subtree).

5.4.2 Window motion

In order to understand the behavior of FC collectors, we observe how the position of the collected region within the heap changes from one collection to the next. We call this *window motion*. We shall use window motion analysis to guide the design of adaptive collection in Section 5.4.5.2. A window motion plot (such as Figure 5.1, p. 132) shows the location of the collection window in the heap in successive collections. The window is contained between the two marks of the old end and the young end, shown along the vertical axis. The horizontal axis show the progression of collections, and is gauged according to the amount of allocation, so that the window motion graph can be compared against the live profiles (Section 4.2.3) and the detailed heap profiles (next section). The *horizontal displacement* of the window from one collection to the next equals the amount allocated between collections. The *vertical displacement* of the window, for the DOF regime of window motion, equals the amount of survivors of the preceding collection. Therefore, for DOF, the *slope* of the window motion curves equals the instantaneous mark/cons ratio. Hence good performance shows as shallow slope, and ideal performance would show as a flat window motion curve.

5.4.3 Demise point analysis

Our fully accurate trace permits simulation and evaluation of any garbage collection algorithm. The algorithms differ in the choice of the instants when they perform collections, and, at each collection, the choice of the collected region. These differences cause the heaps to contain different amounts of garbage data and in different positions. The sets of live objects, however, are necessarily the same. Objects in the heap are ordered according to their time of

allocation, hence the live objects are not only equal as a set, but also as an ordered list, across all collectors.

When the collection algorithm decides which region it should collect, its goal is to choose that region which has the greatest garbage content. We have observed that garbage collectors differ in the amount and position of garbage that they keep in the heap. However, the component of garbage that just came into being—objects that have most recently died—is equal for all collectors. Indeed, if a collector succeeds in collecting objects immediately after they become garbage (or very soon), that will be the only component of garbage. Therefore, the focus of our method is to identify when and where fresh garbage arises.

To answer the temporal question, we imagine a collector that *immediately* discovers all objects that become garbage, and immediately collects them. Clearly, the information in the trace as described in Section 4.1 suffices to simulate this collector. To answer the spatial question, recall that ordering by age is assumed. Therefore, in a linear list of objects currently in the heap, the collector can mark those it has just discovered are garbage, and report their position before collecting them away. The position is given as the number of words from the young end of the heap, for each word of a garbage object.

Since the purpose of the method is to guide designers of garbage collection algorithms in their analysis of collector performance, we turn now to the task of presenting visually the findings of the described ideal collector.

5.4.3.1 Visualization: demise maps

We observed that the ideal collector reports the demise of each object as soon as it happens. More precisely, it reports the demise of each *word* of the object, giving the time of demise exactly, as the amount of allocation in the program up to that point. It is then natural to construct a two-dimensional grid, with one dimension corresponding to the allocation time, and the other to the position of the word in the heap, and the grid point is set if the word at the corresponding heap position died at the corresponding time. We lay the time along the longer,

horizontal, axis, and heap position along the vertical axis. The heap position is measured from the *young*, most recently allocated end of the heap in age order. A demise point is shown as a black dot. Thus we have constructed a *demise bitmap*.

In practice, however, we must take an additional step to produce visual displays of demise. Note that the dimensions of the demise bitmap are (Total words allocated) $\times$ (Maximum words live), which, even for the relatively small benchmarks in our suite, can be of the order $10^6 \times 10^8$. Such a large bitmap cannot be graphically rendered directly. Instead, we divide both the time and the space axis into clusters, compute the number of set bits of the demise bitmap in each cluster, and assign greyscale levels to each cluster accordingly. Having considered the capability of the rendering device and the size of the final images, and having experimented with different granularities, we settled on a 100×660 greyscale map. Thus obtained demise greyscale maps for our benchmark are presented in Figures 5.267–5.280 on pp. 254–267. Each panel is in three aligned displays: the middle display shows the demise points, while the top and bottom display show heap profiles, which we presently discuss (the top display shows the curves of equal time of allocation; and the bottom display shows the curves of equal age).

5.4.3.2 Visualization: heap profiles

The demise map faithfully presents temporally local information. We know which words have just died in the sense that the map shows *where* they were. It does not tell us the identity of these words—at which point were they allocated. However, the ideal trace can be used to produce a telling visual display of this information, in the form of a *heap profile*. The duration of program execution is divided into a number of equal-sized segments, and the heap profile tracks the motion of each segment in the heap from the time of allocation onwards. Since the heap is age-ordered, the objects allocated during one segment remain contiguous in the heap. Thus it is possible to draw nonintersecting lines of constant allocation time for each of the times at the boundary of two allocation segments. If we again show the heap positions with youngest end at the bottom, then subsequent segments are stacked below previous segments

in a banded plot. The height of a band indicates the currently surviving amount of data from a given segment of allocation. The total height of all bands indicates the total amount of live data in the heap.

The information in this plot is, in principle, equivalent to that in the demise map, but it has different visual properties. For instance, in benchmark Interactive-AllCallsOn, Figure 5.267, it is easy to see a correspondence between the sharp fall of the lines in the heap profile and the stripe of demise points at time 10200. However, it would be difficult to discern in the heap profile a corresponding feature for the stripe of demise points at time 1000, because it happens when the lines in the heap profile are mostly rising. Thus the two forms of display are complementary.

Lastly, the bottom displays of Figures 5.267–5.280 show the heap profiles using lines of constant *age*, allowing us to observe the relative participation of younger and older objects in the heap, as execution progresses.

Heap profiling is not a new concept. It has been used in analyzing heap usage in lazy functional programming languages [Runciman and Wakeling, 1992; Runciman and Röjemo, 1995; Sansom, 1994; Sansom and Peyton Jones, 1994; Röjemo and Runciman, 1996], where, however, the grouping criterion of interest is not the age of objects, but the function-point that allocates them. We earlier proposed a related three-dimensional visualization based on object age [Stefanović, 1993a; Stefanović and Moss, 1994].

5.4.3.3 Visualization: position mortality

The demise maps and heap profiles provide a graphical representation of the time-varying behavior of a program, but we nevertheless desire a time-averaged summary. Even if the demise maps show that garbage will occasionally arise anywhere in the heap, we want to know where most of it should be expected, in the hope that a collector might work well by concentrating its efforts there. A useful summary is obtained by counting the number of demise points that occur at each heap position. Note that if we ascribe demise not to an object, as

we usually do, but instead to the heap position the object occupies, then we can speak of *position* mortality, as the likelihood that the word at certain position (whichever that word is) will become garbage upon the next time increment. In the frequency interpretation of probability, this likelihood is exactly given by the count of demise points. We show the plots of this quantity in Figures 5.281–5.294 on pp. 268–275. The horizontal axis corresponds to heap position, with the young end at 0. The vertical axis, on a logarithmic scale, gives the number of demise points at each heap position.

5.4.4 Case studies

We now look in detail at the spatio-temporal behavior of benchmarks Richards and Lambda-Fact5. In addition to discussing the information directly available in the graphs obtained by the method of the preceding section, we shall relate it to the object lifetime statistics, and the observed performance of different age-based garbage collection schemes. This analysis will serve as the basis for improving the copying cost of collection in the following section.

5.4.4.1 Benchmark Richards

The benchmark Richards is characterized by a small amount of live data in comparison with its long duration. The average lifetime of objects is short, which is reflected in the strong band of demise points (black) in the demise map (Figure 5.277) along the bottom edge of the map, corresponding to the demise of recently allocated objects. A long-term periodic pattern is clear as well, both in the demise map, and in the heap profile, with about four periods (of duration approximately 900000 words) completing. The end of a period is marked in the demise map by a sudden demise of a large number of words in the mid-to-older part of the heap.

The heap profile is revealing: the main feature is the wide band of oldest data, which consists of objects allocated entirely within the first segment or about 60000 words. Once the size of this segment has dropped to about 500 words, it remains constant for the remainder of program execution. In other words, these are the objects that the program uses permanently:

its "static" data. It is worth noting that these objects represent a large fraction of the total live amount, above one third. Indeed, in almost all the benchmarks we examined, there is a significant amount of static data. In addition to the band that survives from the very start of the program, within each period there is an additional accreting set of bands, which die together at the end of their period. For the duration of the period, these bands also act as static data. Underneath we observe a region of objects that survive a short time, enough to reach position 200-300 in the heap, and then die.

The presence of a large quantity of static data as the oldest data in the heap has a strong detrimental effect on the performance of garbage collector algorithms that ever choose to collect the older region of the heap. It is remarkable, however, that the FC-DOF algorithm, which repeatedly visits the oldest part of the heap, once for every rightward sweep (Section 3.1.1), works quite well on the Richards benchmark, and outperforms all examined age-based schemes, including generational collection—in spite of the presence of static data. It does so by sweeping the static data *fast*, and then focusing on the region of high demise. In Section 5.4.5.1, we modify that algorithm to avoid the static data and find that its copying cost improves further.

5.4.4.2 Benchmark Lambda-Fact5

The benchmark Lambda-Fact5 is of particular interest because we found that none of the examined garbage collection algorithms performs well on it. Examination of the heap profile, Figure 5.271 reveals that there is a very large fraction of static data, which hampers older-first algorithms. The static data, allocated during the first 10% of program execution, account for over half of the live amount present in the heap. In addition, there is some accumulation of objects allocated later on, seen as an increasingly wide band of narrowly spaced lines.

On the other hand, the demise map shows significant demise areas in middle heap positions, especially as a periodic pattern during the latter half of program execution. The presence of these demise points precludes the dominance of points in the very youngest part of the heap

and hampers youngest-first algorithms, such as traditional generational collection. The plot of the distribution of demise points by position in Figure 5.285 emphasizes this observation.

Generational collection, unless given a very large heap space, works poorly with small nursery (youngest generation) sizes because too much data is promoted out of the nursery (Figure 5.36(b)); if the nursery size is increased, the above-mentioned demise regions are included, and the performance improves, but even then it only begins to approach non-generational collector performance (when the nursery is so large that the collector behaves almost as a non-generational collector). Similarly, age-based collectors with constant collected region size perform poorly on this benchmark (Figure 5.36(a)), no better than the non-generational collector. However, the hypothetical FC-OPT collector is capable of markedly better performance, with copying cost only 27% of the non-generational for heap size 11402 (Figure 5.254).

5.4.5 Adaptive FC collection

In this section we examine how to improve the performance of collection by modifying the idea of sweeping the heap with the collection window in successive collections. We are guided by the demise map, and by the observed motion of the collection window in the locally-optimal collector. We undertake exploratory case studies of *adaptive* policies within the class of age-based collectors with a fixed collected region size. On the example of the Richards benchmark we shall examine how to remove the permanent data of a program from consideration, and on the example of the Lambda-Fact5 benchmark we shall examine how to adjust the position of the collection region adaptively in response to collection results. In both cases, we shall find it possible to reduce the copying cost of collection over fixed policies.

5.4.5.1 Eliminating the "static" data in Richards

We noted in Section 5.4.4.1 that the presence of a substantial amount of permanent, "static" data in the oldest region of the heap degrades the performance of DOF collection, because collections that examine the oldest region find little garbage, or even fail if they find no garbage. If, however, the collector knew that the oldest region contains such permanent data, it could

skip over them, never trying to collect them. It would thus be able to sweep over only the part of the heap where there is some object demise, and potentially incur a lower copying cost. The danger, of course, is if the "static" region is incorrectly identified as such: if some objects in the region that the collector never considers do in fact become garbage, then two penalties are paid. First, the space occupied by these objects is not reclaimed, and the collector works with a heap effectively smaller than it needs to be; and second, since these objects are considered live, any pointers they might have into the collected region cause excess retention. These dangers are familiar from the parallel situation of *premature tenuring* of objects in generational garbage collection [Ungar, 1984; Ungar, 1986]. Most programs, however, do have a region of permanent data, and the question is only how big it is.

The collector should not immediately exclude the oldest region. The heap profiles (Figures 5.267–5.280) show that a significant amount of allocation occurs before the permanent data consolidate, and during this initial period, there is garbage to be collected from among the oldest data as well. Hence, we introduce an adaptive thresholding policy to recognize when the oldest region begins to contain mostly permanent data. The collector initially operates in the usual FC-DOF manner. On each collection upon a window reset, i.e., a collection of an oldest region, the survivor ratio for the collection is computed. As soon as the ratio is above a threshold value θ, all subsequent window resets skip the oldest region.

We take the benchmark Richards for our case study, with the heap size of 4032 words. We use a window size of 1088 words, or $g = 0.27$, for which the FC-DOF collector achieves a copying cost of $\mu = 0.03181$. With the same heap size, the NONGEN collector has a copying cost $\mu = 0.3937$, the best configuration of the 2GYF collector has $\mu = 0.2198$ (with nursery size 2056 words), the best configuration of the 3GYF collector has $\mu = 0.1221$ (with nursery size 2214 words), and the best configuration of the FC-DOF collector has $\mu = 0.02975$ (with window size 1572 words, or $g = 0.39$).

Our first experiment is to supply to the collector the size of the region of permanent data. This value can be read off the heap profile, and for Richards it is 490 words. We expect that if θ

is chosen too low, then the region of permanent data will be taken out of consideration too soon, whereas if θ is chosen too high, the region will be excluded too late to improve performance. Experimentation with different values of θ shows that results are not exceedingly sensitive to the choice, as shown in Table 5.1.

Table 5.1. Copying cost dependence on the adaptation threshold

θ	μ
0.2	0.02759
0.3	0.02734
0.4	0.02734
0.5	0.02734
0.6	0.02760
0.7	0.02830
0.8	0.03260

The best range of threshold values, $0.3 \leqslant \theta \leqslant 0.5$ includes the value $\frac{490}{1088} = 0.45$, and results in a copying cost of $\mu = 0.02734$. Thus, it was possible to reduce the copying cost by 14% with respect to the ordinary FC-DOF collector with the same window size, and in fact by 8% with respect to the best configuration of the ordinary FC-DOF collector.

In the second experiment, the collector is not told the amount of permanent data, but tries to guess it using the following heuristic: each time the first collection following a window reset results in a survivor ratio above the threshold, multiply the amount of *survivors* by a fixed fraction φ, and add the result to the current estimate of the amount of permanent data. The collector guesses that the high survivor ratio for this collection arose because there were permanent data in the collected region, and that they were the oldest data in the region. It is willing to subject the remaining fraction $1 - \varphi$ of survivors to examination at the following reset. Thus the heuristic has two parameters: the threshold θ and the fraction φ. Note that this heuristic is only able to *increase* the estimate, starting from an initial zero, but never to revise the estimate downward. Therefore the danger is that if θ is too low, or if φ is too high, the oldest region may prematurely be considered permanent. We found that the best values are 0.8 for the threshold θ and 0.5 for the fraction φ, in which case the copying cost is $\mu = 0.029$. This

result leads us to believe that it is possible to devise adaptive schemes to sense the presence and gauge the size of permanent data, although our simplistic heuristic is too sensitive to the choice of the two parameters to be reliably used.

5.4.5.2 Choosing the collected region in Lambda-Fact5

Instead of restricting the window motion to the region recognized as non-permanent data, we can modify the behavior of the DOF collector more subtly. The window motion of DOF is based on the intuitive premise that survivors of a collection should not be subject to the immediately succeeding collection. Therefore the window moved rightward over the survivors (Figure 3.9). This policy has excellent performance when the collection survivor ratio is zero or extremely low (as illustrated in Figure 3.10), which is the case in practice for several benchmarks, including Richards and StandardNonInteractive (Section 5.1.1.4). This policy suffers if the survivor ratio is not very low, however, as in Lambda-Fact5. In spite of the many demise points in the middle range of the heap for an ideal collector (demise points map in Figure 5.271), the survivor ratios in actual runs of FC-DOF are high. In Figure 5.2 we show the measurements of the survivor ratios for each collection of the FC-DOF collector on the Lambda-Fact5 benchmark, with a heap size of 11402 words and a window size of 4446 words, for which a copying cost of $\mu = 1.27$ is achieved. The FC-DOF collector performs a total of 150 collections, and each shows up as a point in the scatter plot. The horizontal axis gives the position of the window (indicated by the amount of data between the old end of the window from the old end of the heap), and the vertical axis gives the survivor ratio of the collection (the ratio of the amount of survivors to the window size). The corresponding window motion diagram is in Figure 5.1. Thus, FC-DOF suffers because its window moves too fast across the middle region where it might encounter acceptable survivor ratios, and is too often reset to the old end, window position 0, where survivor ratios are high. Contrast this with the run of the FC-OPT collector with the same heap size and window size, which achieves $\mu = 0.198$. Its window motion graph is in Figure 5.3, and the survivor ratio scatter plot in Figure 5.4. Ob-

serving that FC-OPT moves its window much more slowly, we shall try to emulate its behavior using the following heuristic window motion policy, called the *floating window*.

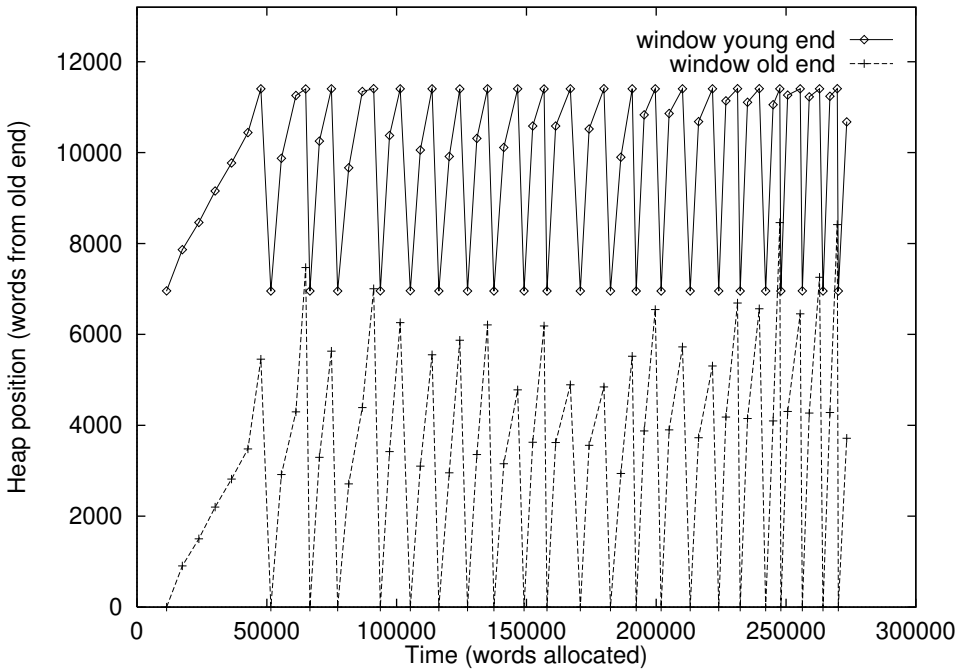


Figure 5.1. Window motion in the FC-DOF scheme.

The position of the window of collection is defined as the amount of data in the age-ordered heap between the old end of the heap and the old end of the window of collection. Let X be the current position of the window of collection. Let W be the size of the window of collection. Upon collection, we find that an amount S has survived out of W. We decide the position for the next collection as $X' = X + \Delta X$ (but resetting X' to zero after the young end of the heap is reached). The increment ΔX is determined as follows. If $S = 0$, then $\Delta X = 0$. Otherwise, note the position p_i of each surviving word in the collection window relative to the old end of the window. The average survivor position is $\bar{p} = \frac{1}{S}\sum_S p_i$. The relative bias is the relative difference between $\bar{p}$ and the value $\frac{W}{2}$ that would be obtained if survivors were evenly spread in the collection window: $r = \frac{\frac{W}{2} - \bar{p}}{\frac{W}{2}}$. Thus $-1 \leqslant r \leqslant 1$. Given an additional fixed external bias e, we set $\Delta X = (r + e)S$. The window is initially placed at X equal to one-half heap size.

In other words, the motion of the window is governed by two forces. An external bias pushes the window in a fixed direction. In particular, the value 1 for the external bias cor-

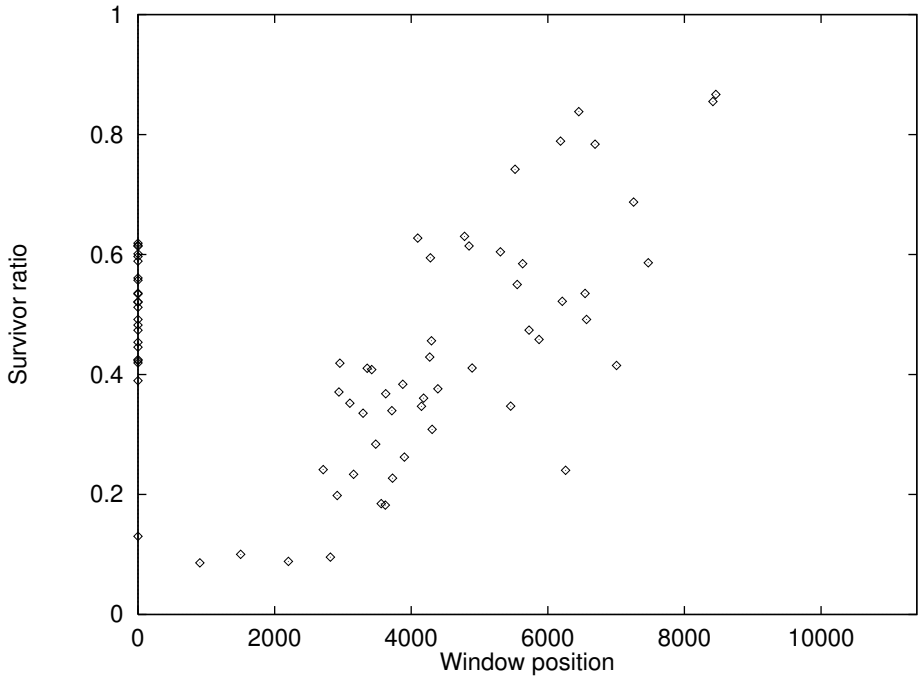


Figure 5.2. Window survivor ratios in the FC-DOF scheme.

responds to the ordinary FC-DOF collector, pushing the window in the direction of younger objects. The second force uses feedback information from the completed collection. Intuitively, if most of the survivors of the collection were found in one part of the window, then the window should be pushed in the opposite direction. For example, if the window straddles the region of permanent data, then most of the survivors will have come from the older part of the window, and it should be pushed towards younger data. We calculate a simple heuristic measure of the desired push as the average position of the survivors within the window. If the average survivor position is left of center, the force is rightwards, and vice versa.

Recall that our test case is the Lambda-Fact5 benchmark, with a heap size of 11402 words and a window size of 4446 words, or $g = 0.39$, for which the FC-DOF collector achieves a copying cost of $\mu = 1.27$. With that heap size, the NONGEN collector has a copying cost $\mu = 0.59$, the best configuration of the 2GYF collector has $\mu = 0.55$, the best configuration of the 3GYF collector has $\mu = 0.56$, and the best configuration of the FC-DOF collector has $\mu = 0.52$. Experiments with different values of the external bias e produce the following results, shown in Table 5.2.

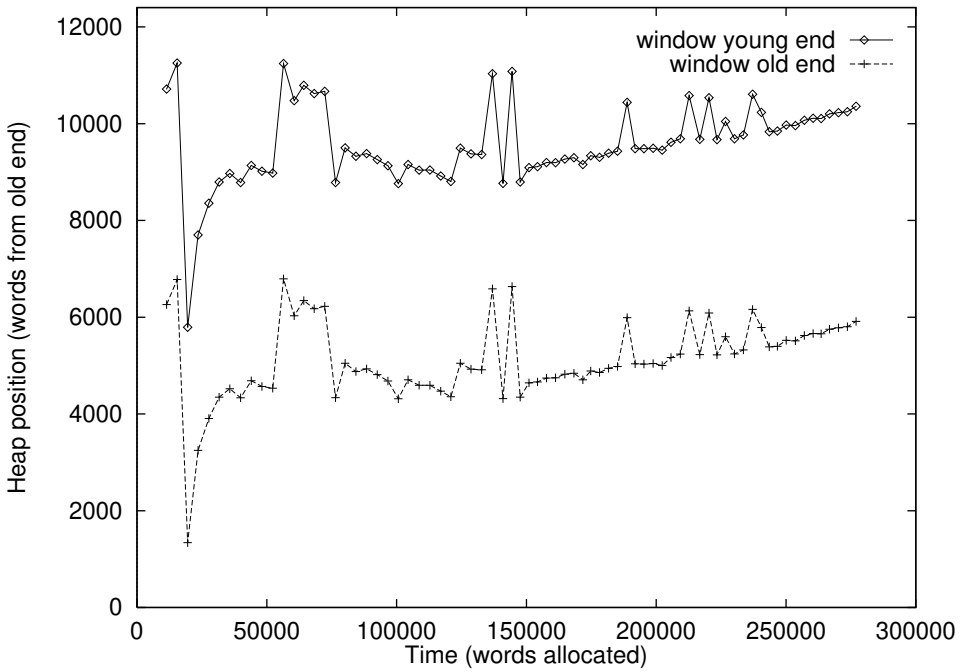


Figure 5.3. Window motion in the FC-OPT scheme.

The lowest achieved copying cost with the floating window policy is $\mu = 0.37$. This is 28% below the best of the fixed schemes for any configuration at the same heap size, namely $\mu = 0.52$ for FC-DOF with a window size of 10135 ($g = 0.88$). We observe the window motion of the adaptive scheme in Figure 5.5, and its scatter plot of window position vs. survivor ratio in Figure 5.6.

Table 5.2. Copying cost dependence on external bias

e	μ
-0.2	0.64
-0.1	0.43
0.0	0.52
0.1	0.46
0.2	0.44
0.3	0.37
0.4	0.40
0.5	0.48

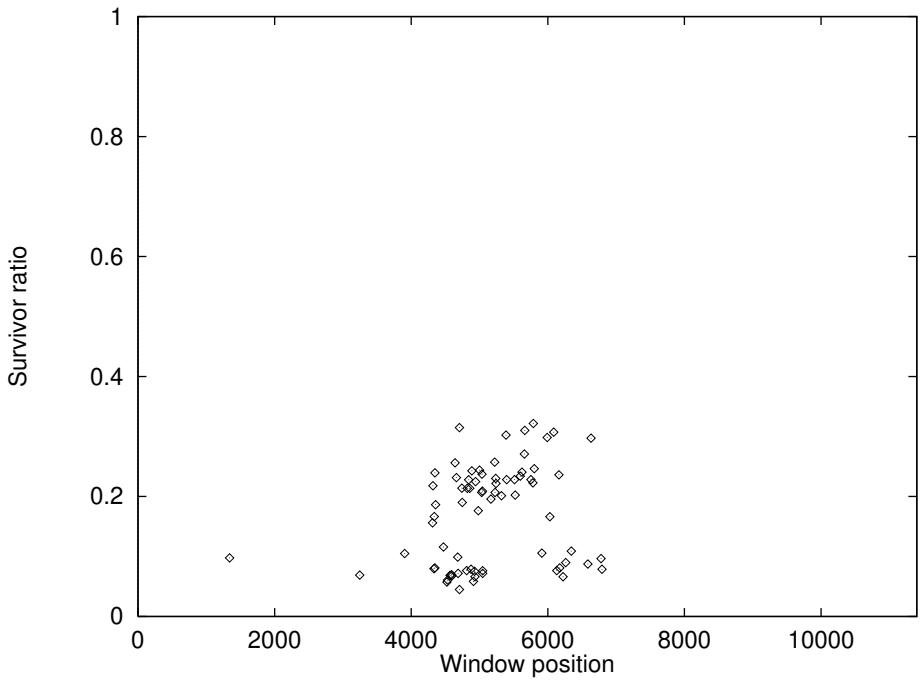


Figure 5.4. Window survivor ratios in the FC-OPT scheme.

The floating window policy is relatively robust with respect to the choice of the external bias, since for the entire region $-0.1 \leqslant e \leqslant 0.5$, the scheme is better than any of the fixed schemes. On the other hand, the FC-OPT copying cost of $\mu = 0.198$, for the same heap size and the same window size, is still another 47% lower.

A final word of caution is in order. We have seen that adaptive variants of FC schemes, and of FC-DOF in particular, can lower the copying cost of collection, and that there exists, for some programs, still more room for improvement, as indicated by the locally-optimal schemes. However, to achieve this further reduction of copying cost requires window motion policies more general than, for instance, the predictable linear sweep of FC-DOF. Recall from Section 3.2.2 that effective pointer filtering at the write barrier exploited the predictability of window motion. Hence, the ultimate reduction of copying cost may well be offset by an increase of pointer management costs.

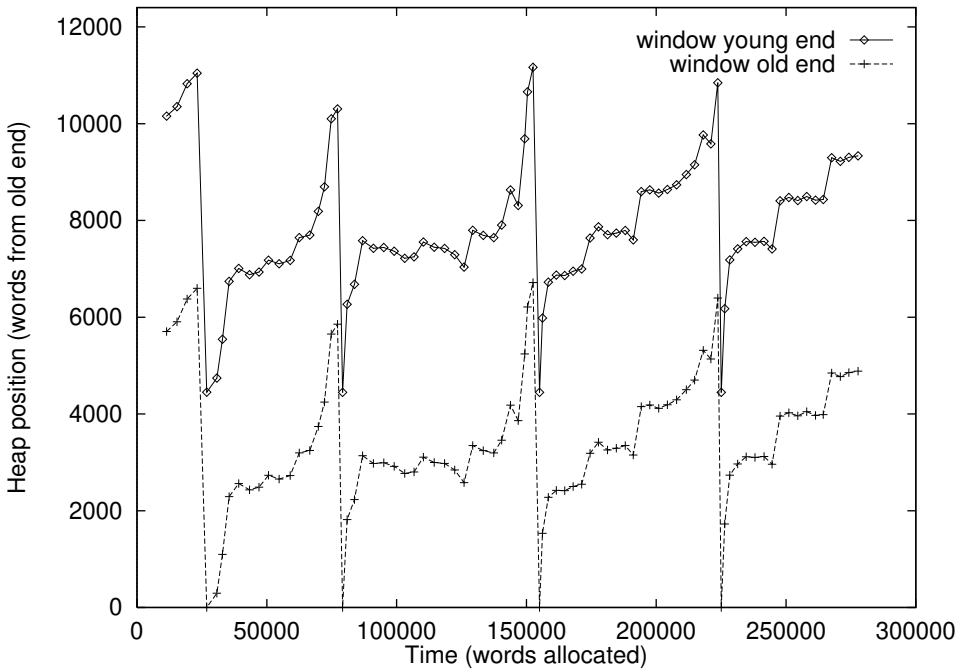


Figure 5.5. Window motion in the adaptive scheme.

5.5 Summary

This chapter examined the copying cost of garbage collection for several age-based algorithms. The main finding is that the promise of low copying cost of our proposed FC-DOF collector is delivered on many actual allocation loads. This result could not be expected given the prior understanding of generational collector performance. Particularly revealing is the fact that excess retention effects, even when very pronounced, do not by themselves rule out using non-youngest-first collectors for our test programs.

The measurements of the performance of generational collectors are interesting in their own right, as they shed light on collector behavior under constraints on total heap size. It is not surprising that small nursery sizes lead to excessive promotion into the older generation, but it is surprising that in many cases the optimal configuration assigns more than half the heap to the nursery. It is possible that more sophisticated generational schemes could perform better—for instance, requiring an object to survive more than one collection in the younger generation before promotion, or, equivalently, dividing the younger generation into steps, possibly combined with adaptive policies, such as Ungar and Jackson's feedback mediation [Ungar and

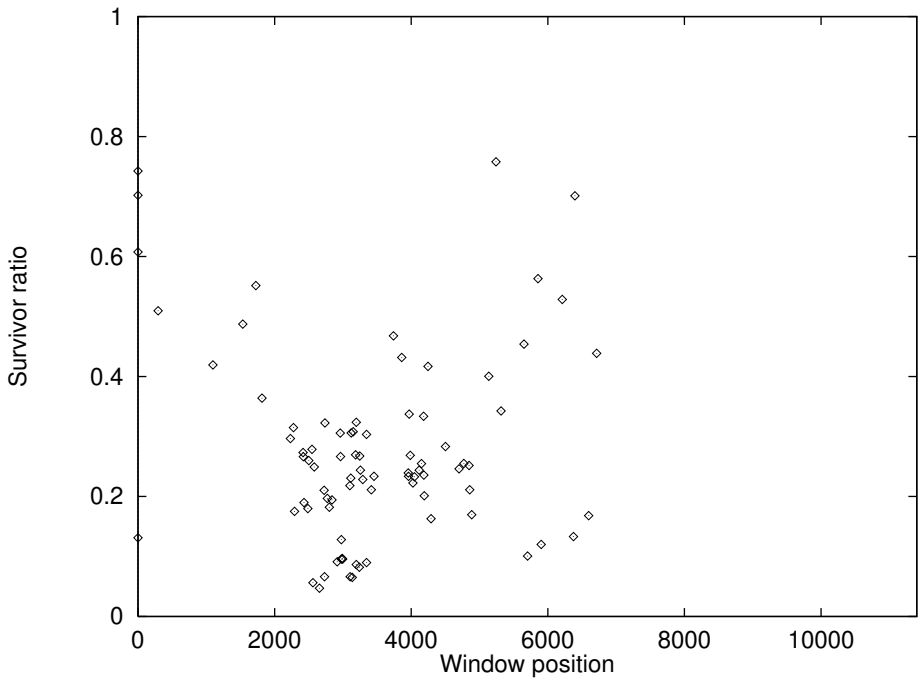
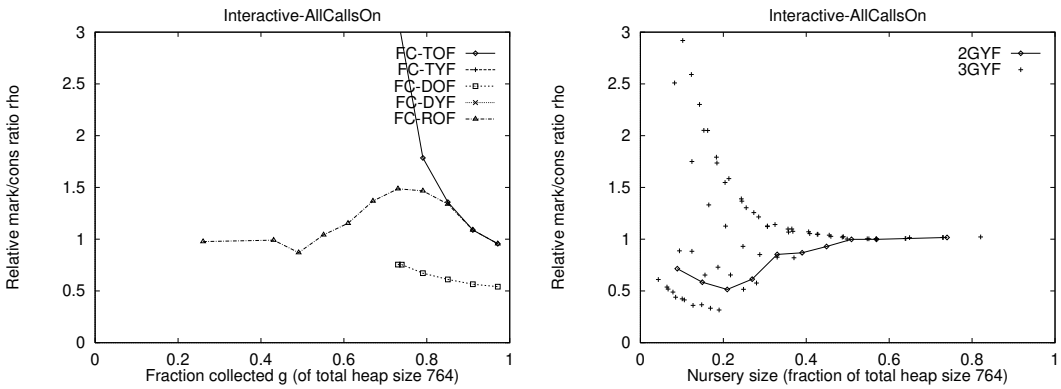


Figure 5.6. Window survivor ratios in the adaptive scheme.

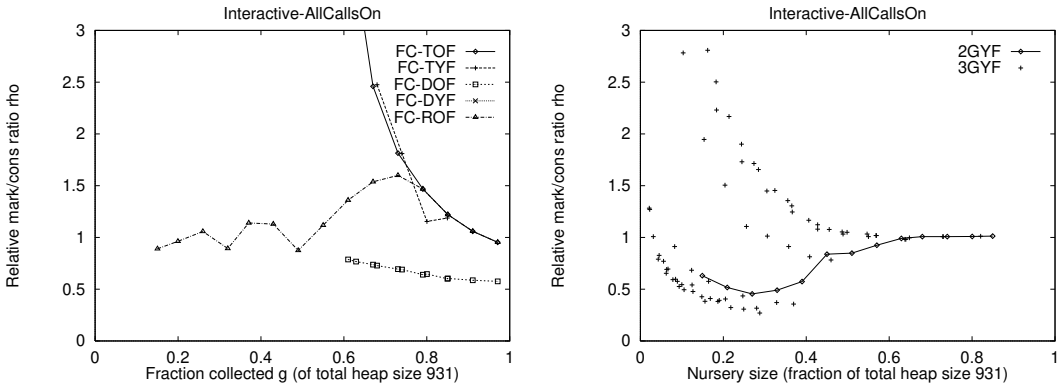
Jackson, 1992]. However, each collection adds to the copying cost, and the final balance cannot be predicted. Unfortunately, the vast dimensions of the space of possible configurations makes it difficult to these collection schemes. (Thus, the selection of their many configurable parameters [Hudson *et al.*, 1991] has largely been a matter of judgement by experience.)

The proposed non-youngest-first schemes can also benefit from more sophisticated configurations. We observed that adaptive positioning of a window of fixed size can reduce copying cost; it is likely that it would be more effective in conjunction with adaptive selection of window size. Most important, however, could be the heuristic discovery of permanent (static) data, and their elimination from consideration by the collector, a task implicitly performed by a multi-generational collector.



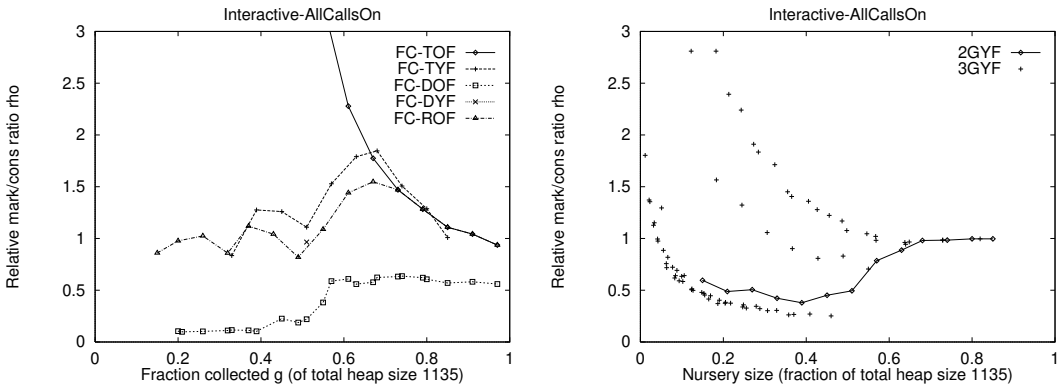
(a) FC schemes (b) GYF schemes

Figure 5.7. Copying cost comparison: Interactive-AllCallsOn, $V = 764$.



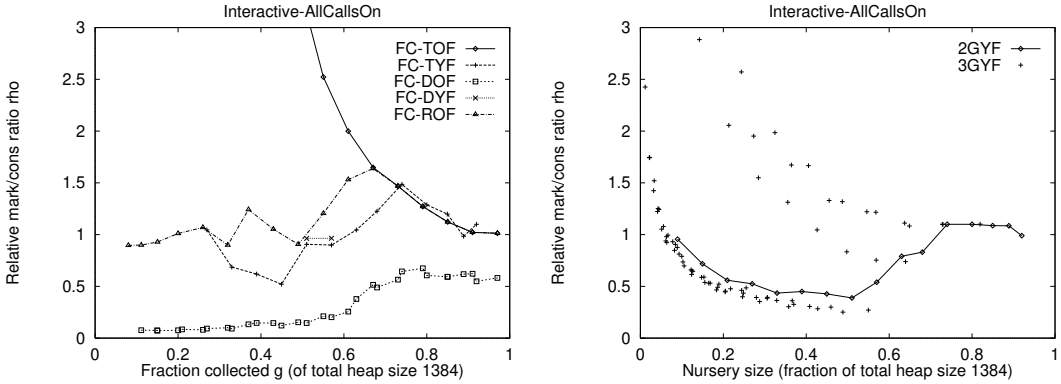
(a) FC schemes (b) GYF schemes

Figure 5.8. Copying cost comparison: Interactive-AllCallsOn, $V = 931$.



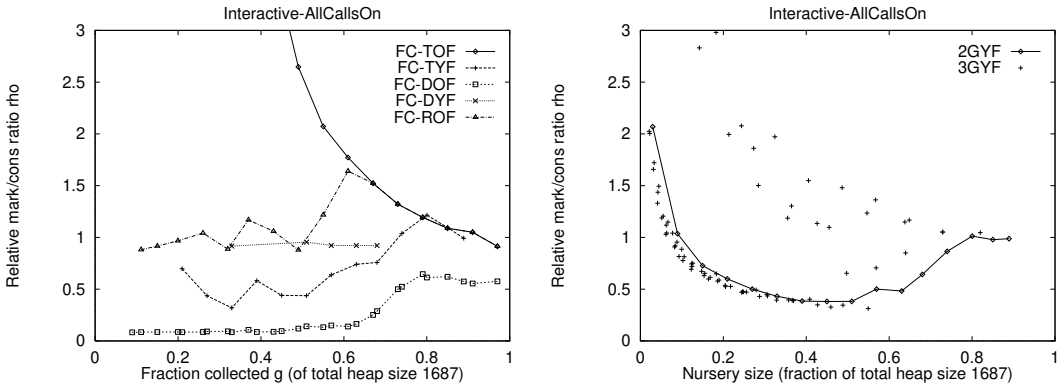
(a) FC schemes (b) GYF schemes

Figure 5.9. Copying cost comparison: Interactive-AllCallsOn, $V = 1135$.



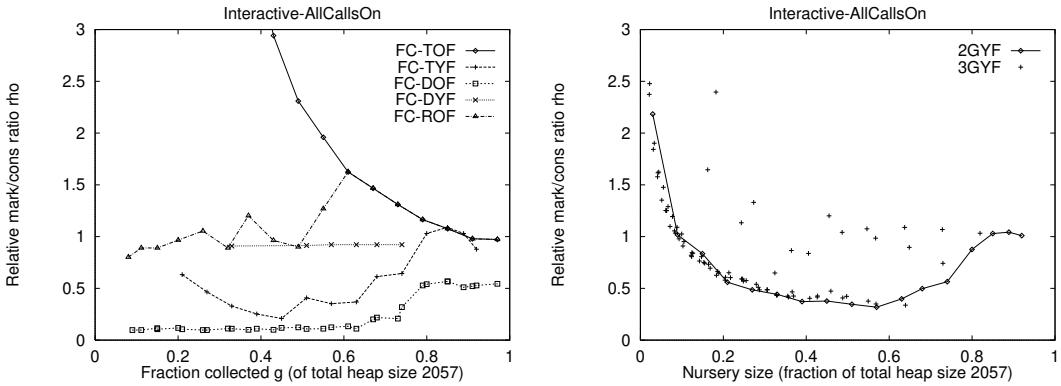
(a) FC schemes (b) GYF schemes

Figure 5.10. Copying cost comparison: Interactive-AllCallsOn, $V = 1384$.



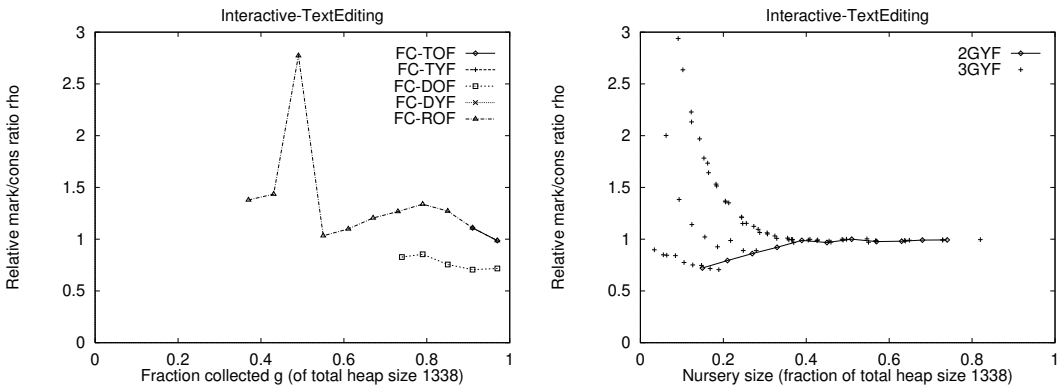
(a) FC schemes (b) GYF schemes

Figure 5.11. Copying cost comparison: Interactive-AllCallsOn, $V = 1687$.



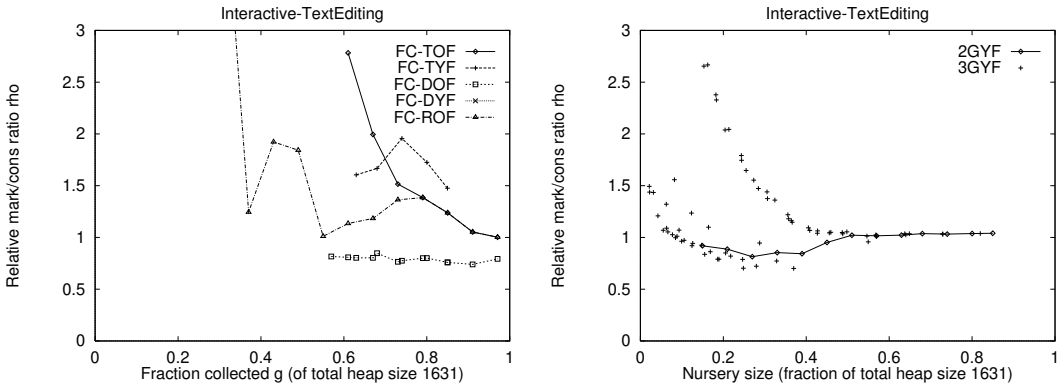
(a) FC schemes (b) GYF schemes

Figure 5.12. Copying cost comparison: Interactive-AllCallsOn, $V = 2057$.



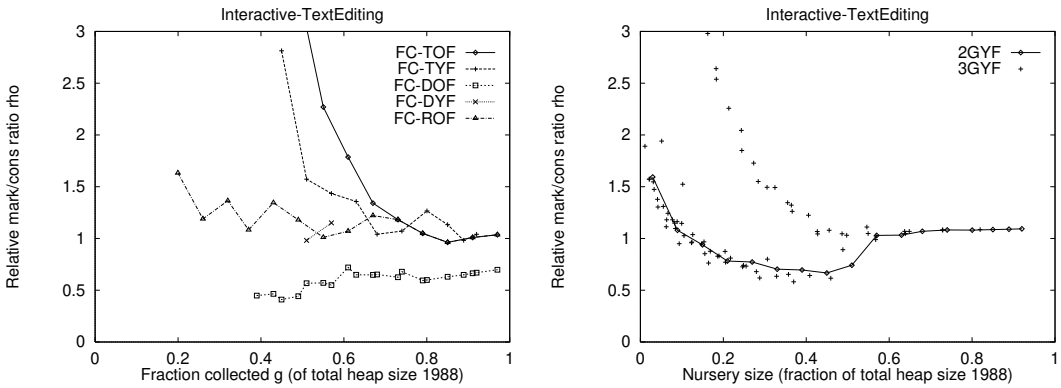
(a) FC schemes (b) GYF schemes

Figure 5.13. Copying cost comparison: Interactive-TextEditing, $V = 1338$.



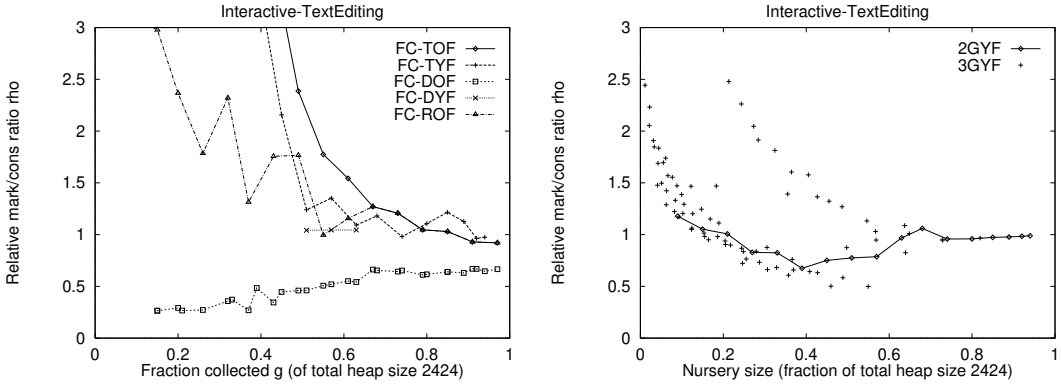
(a) FC schemes (b) GYF schemes

Figure 5.14. Copying cost comparison: Interactive-TextEditing, $V = 1631$.



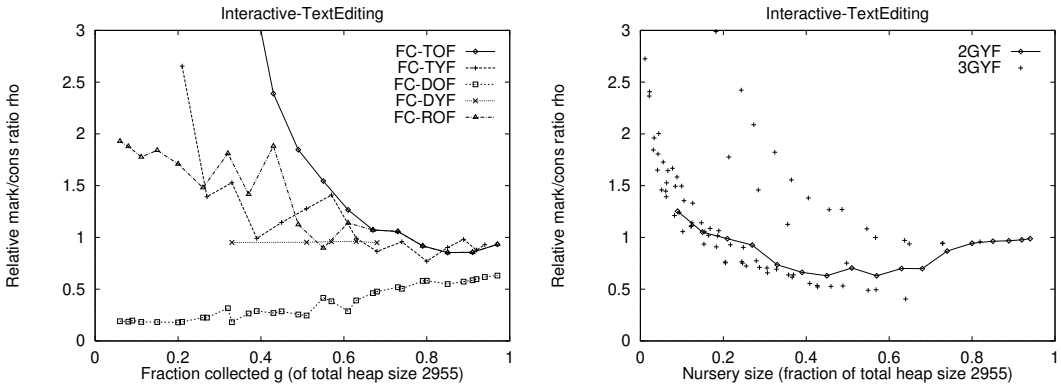
(a) FC schemes (b) GYF schemes

Figure 5.15. Copying cost comparison: Interactive-TextEditing, $V = 1988$.



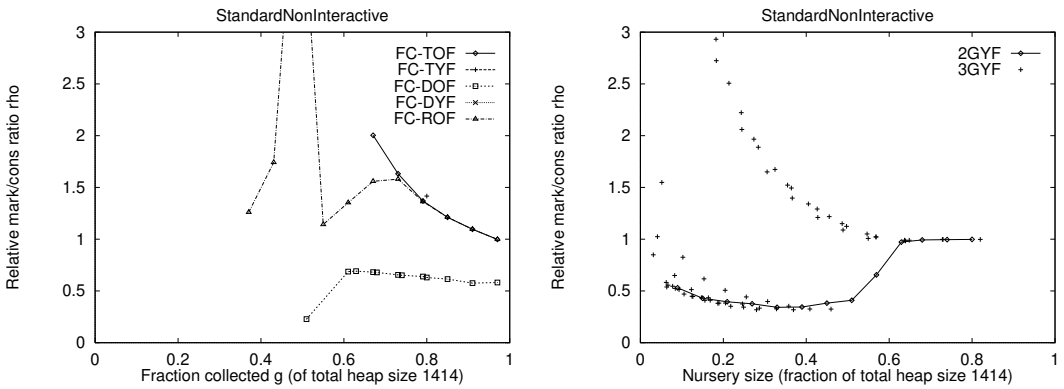
(a) FC schemes (b) GYF schemes

Figure 5.16. Copying cost comparison: Interactive-TextEditing, $V = 2424$.



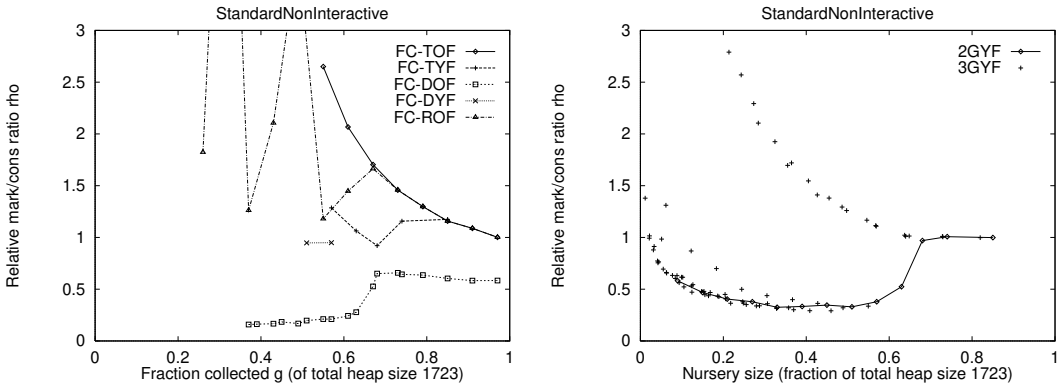
(a) FC schemes (b) GYF schemes

Figure 5.17. Copying cost comparison: Interactive-TextEditing, $V = 2955$.



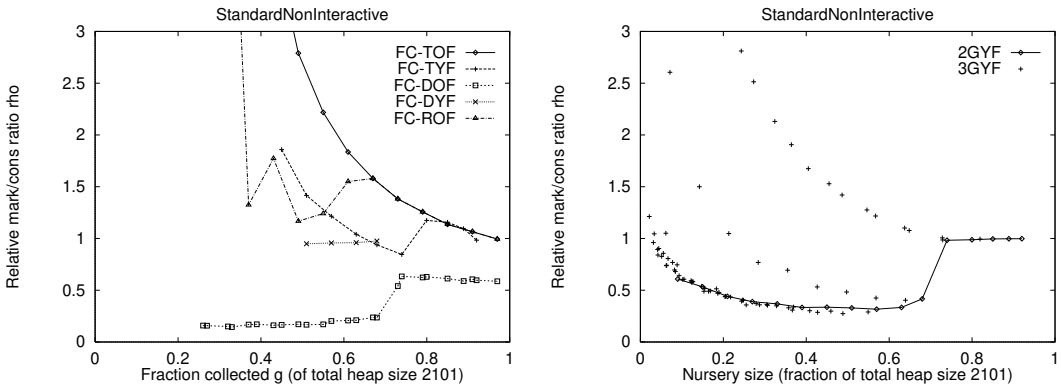
(a) FC schemes (b) GYF schemes

Figure 5.18. Copying cost comparison: StandardNonInteractive, $V = 1414$.



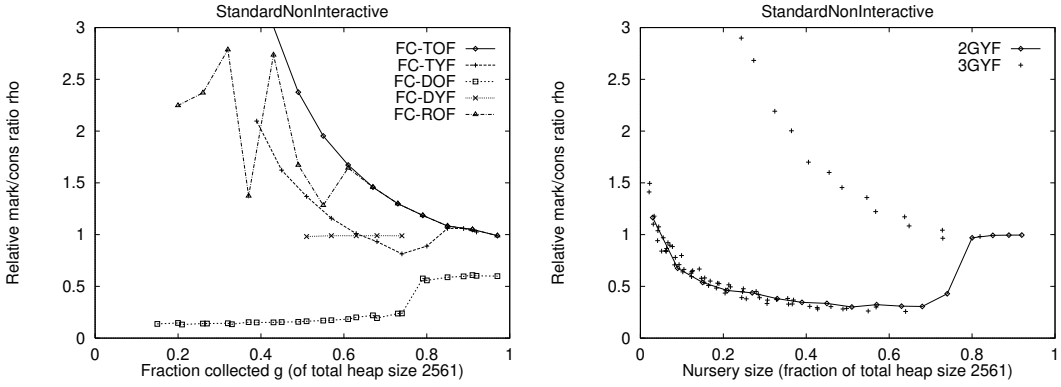
(a) FC schemes (b) GYF schemes

Figure 5.19. Copying cost comparison: StandardNonInteractive, $V = 1723$.



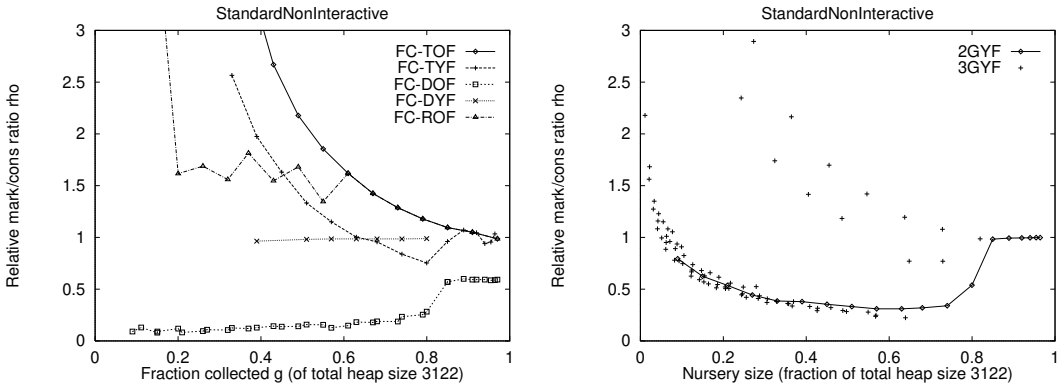
(a) FC schemes (b) GYF schemes

Figure 5.20. Copying cost comparison: StandardNonInteractive, $V = 2101$.



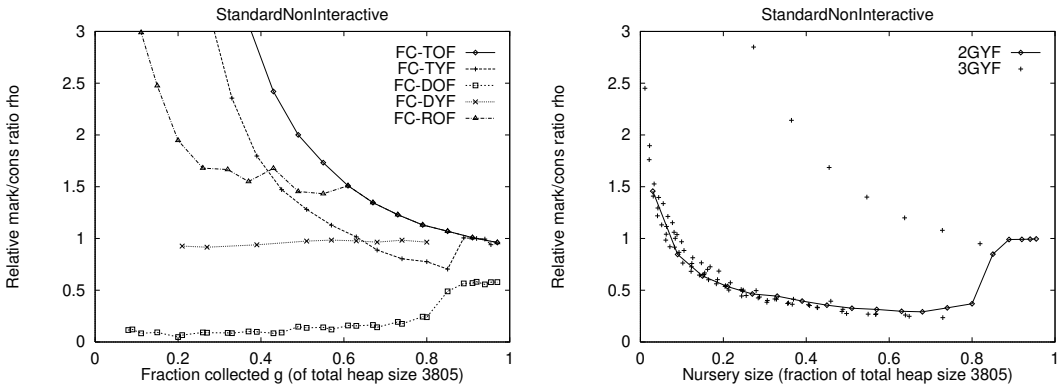
(a) FC schemes (b) GYF schemes

Figure 5.21. Copying cost comparison: StandardNonInteractive, $V = 2561$.



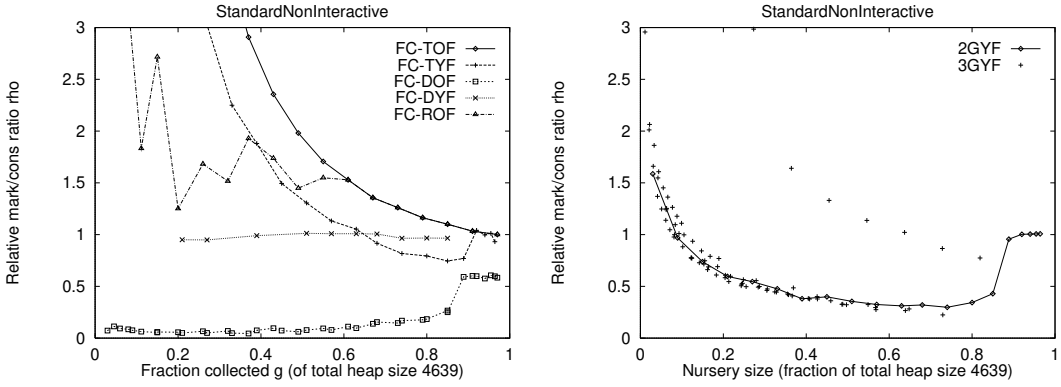
(a) FC schemes (b) GYF schemes

Figure 5.22. Copying cost comparison: StandardNonInteractive, $V = 3122$.



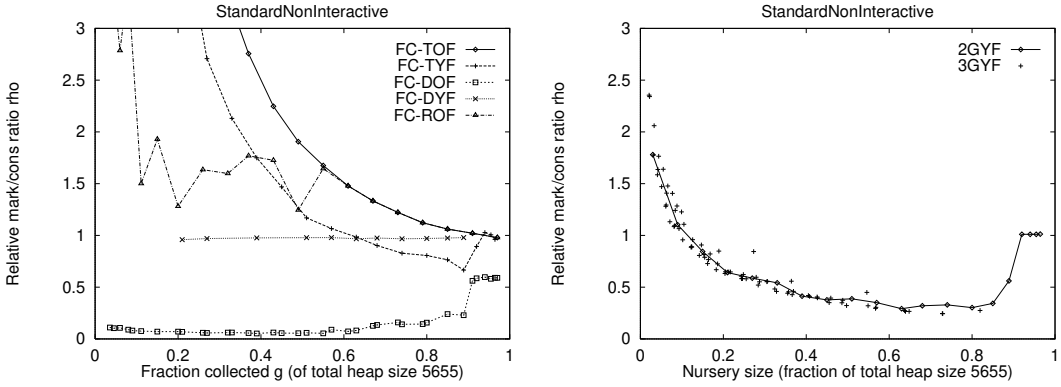
(a) FC schemes (b) GYF schemes

Figure 5.23. Copying cost comparison: StandardNonInteractive, $V = 3805$.



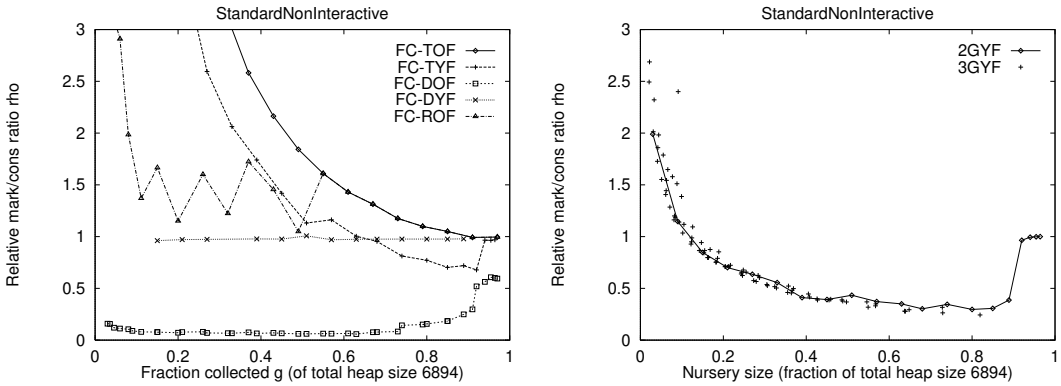
(a) FC schemes (b) GYF schemes

Figure 5.24. Copying cost comparison: StandardNonInteractive, $V = 4639$.



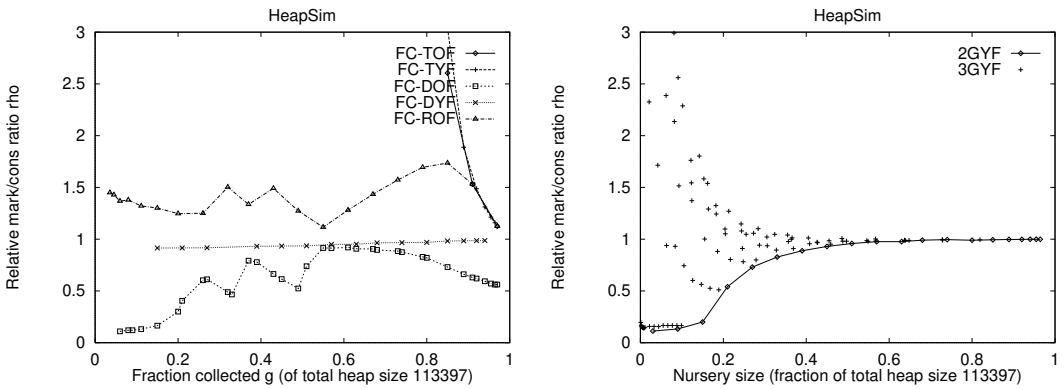
(a) FC schemes (b) GYF schemes

Figure 5.25. Copying cost comparison: StandardNonInteractive, $V = 5655$.



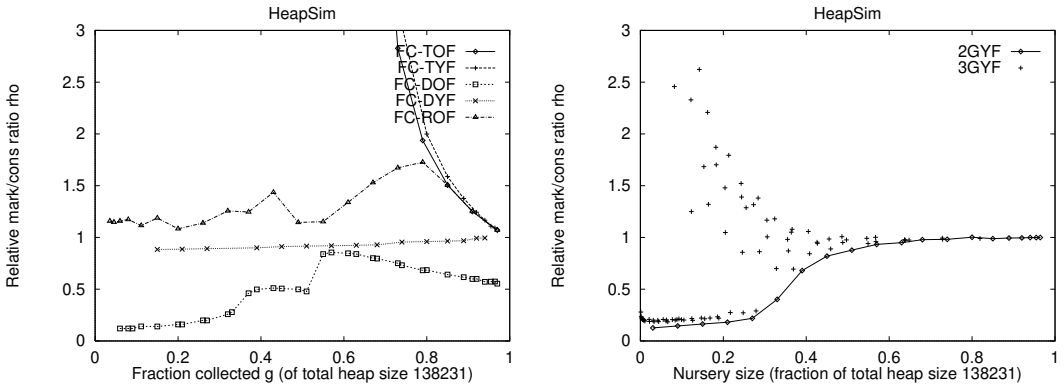
(a) FC schemes (b) GYF schemes

Figure 5.26. Copying cost comparison: StandardNonInteractive, $V = 6894$.



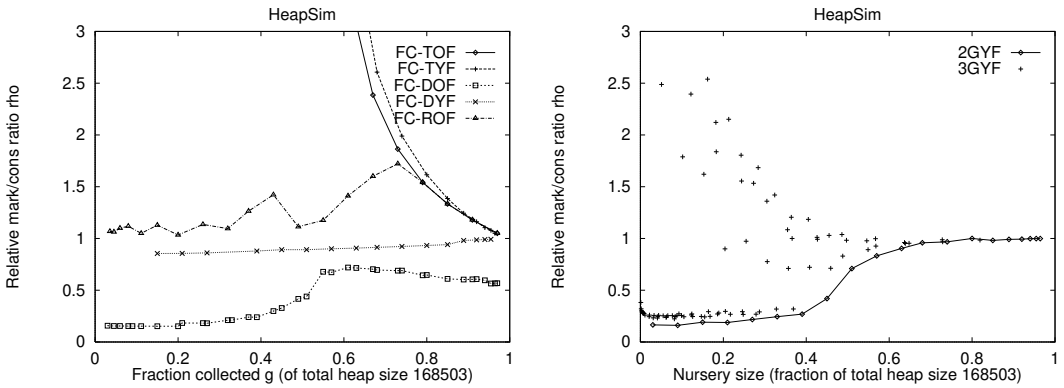
(a) FC schemes (b) GYF schemes

Figure 5.27. Copying cost comparison: HeapSim, $V = 113397$.



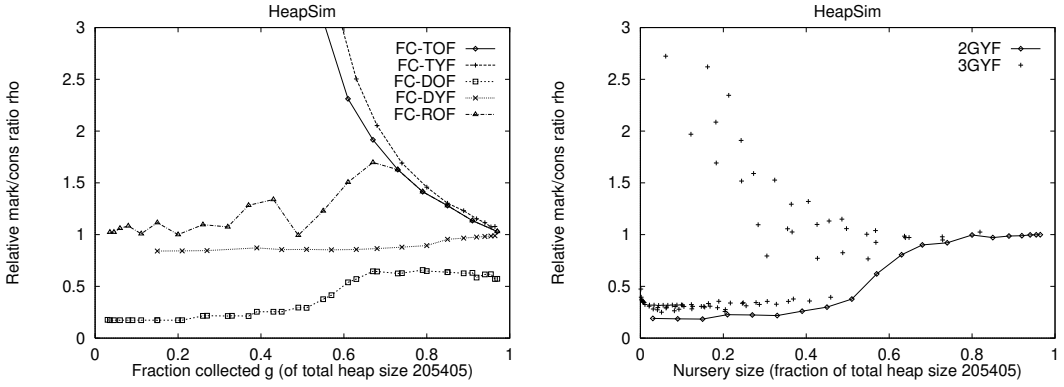
(a) FC schemes (b) GYF schemes

Figure 5.28. Copying cost comparison: HeapSim, $V = 138231$.



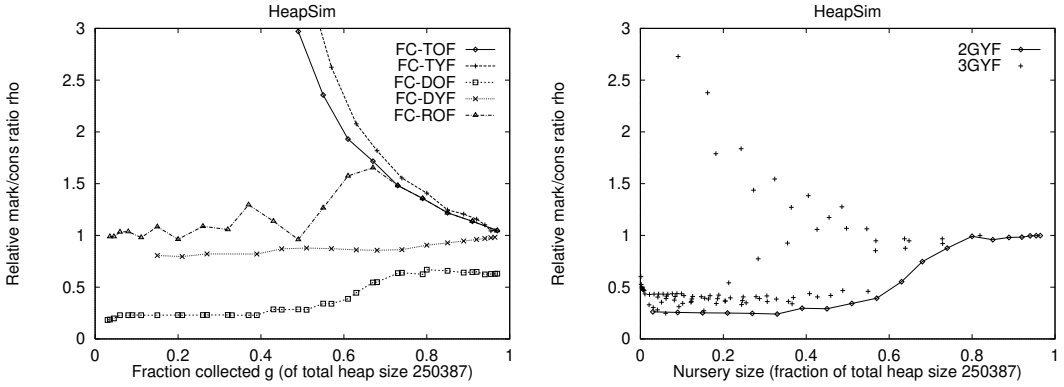
(a) FC schemes (b) GYF schemes

Figure 5.29. Copying cost comparison: HeapSim, $V = 168503$.



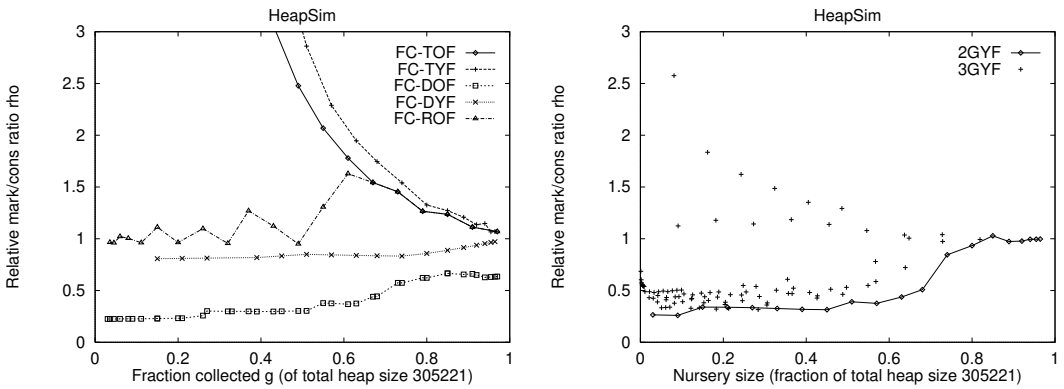
(a) FC schemes (b) GYF schemes

Figure 5.30. Copying cost comparison: HeapSim, $V = 205405$.



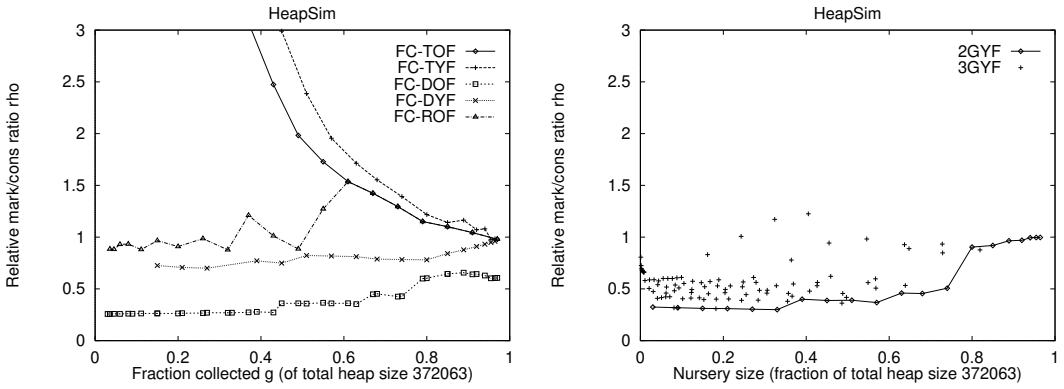
(a) FC schemes (b) GYF schemes

Figure 5.31. Copying cost comparison: HeapSim, $V = 250387$.



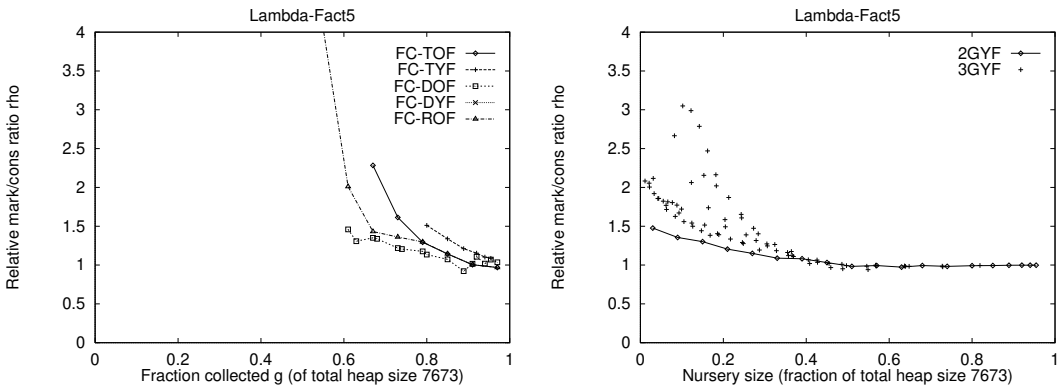
(a) FC schemes (b) GYF schemes

Figure 5.32. Copying cost comparison: HeapSim, $V = 305221$.



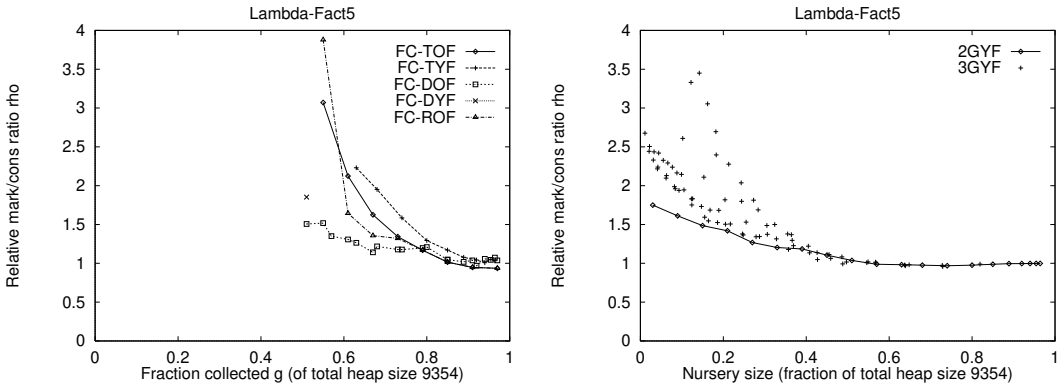
(a) FC schemes (b) GYF schemes

Figure 5.33. Copying cost comparison: HeapSim, $V = 372063$.



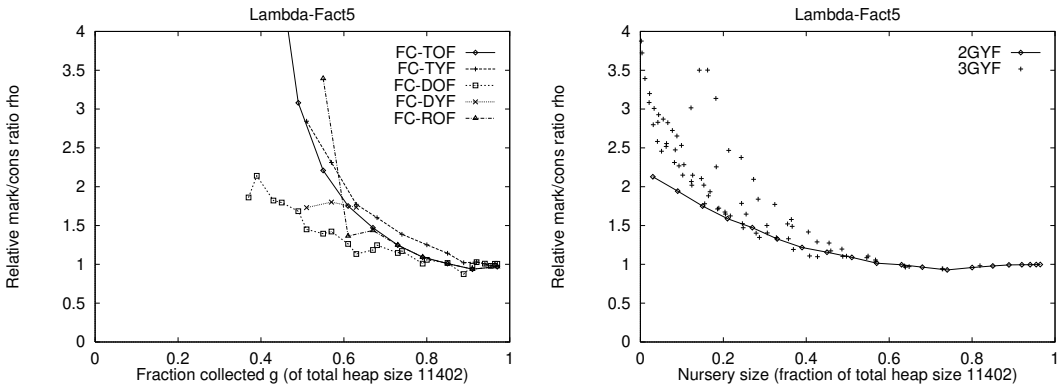
(a) FC schemes (b) GYF schemes

Figure 5.34. Copying cost comparison: Lambda-Fact5, $V = 7673$.



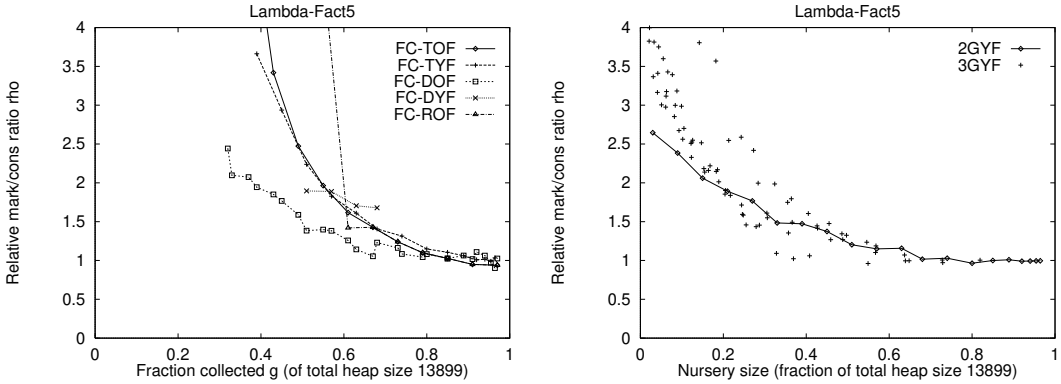
(a) FC schemes (b) GYF schemes

Figure 5.35. Copying cost comparison: Lambda-Fact5, $V = 9354$.



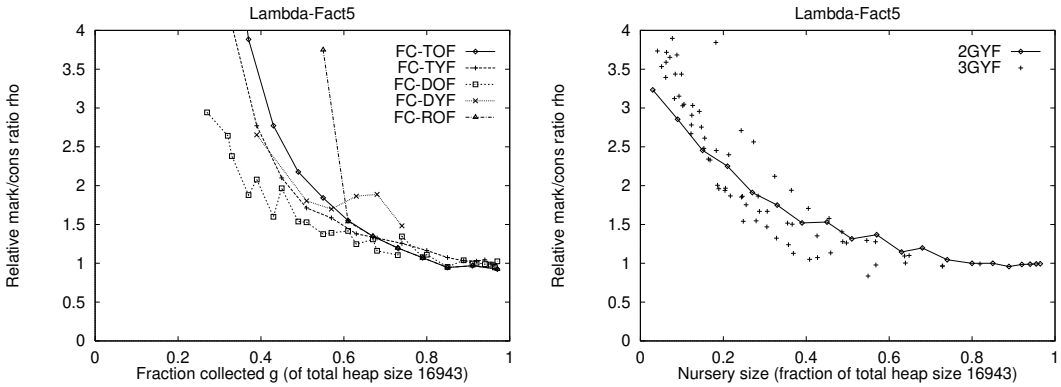
(a) FC schemes (b) GYF schemes

Figure 5.36. Copying cost comparison: Lambda-Fact5, $V = 11402$.



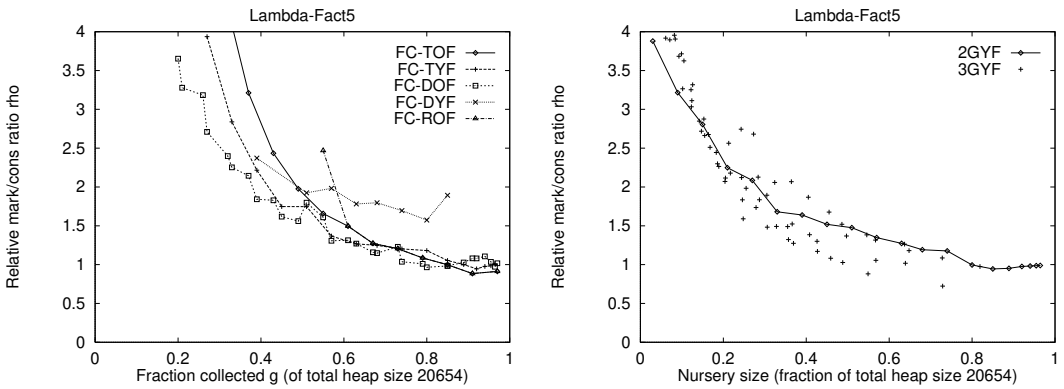
(a) FC schemes (b) GYF schemes

Figure 5.37. Copying cost comparison: Lambda-Fact5, $V = 13899$.



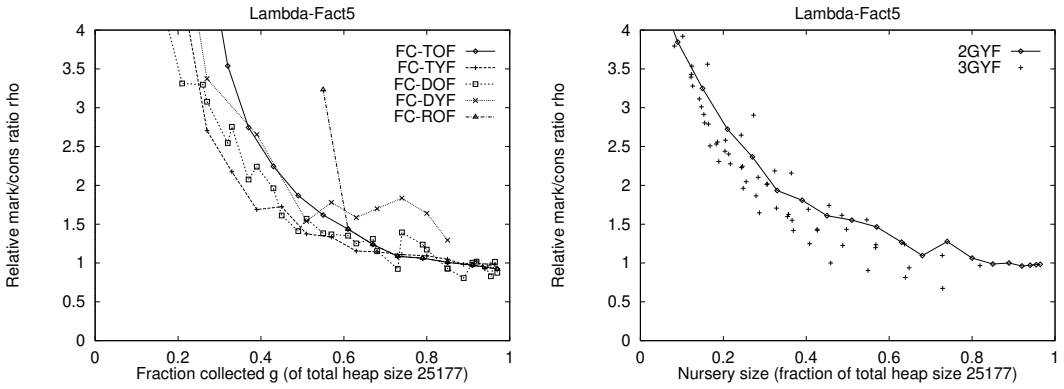
(a) FC schemes (b) GYF schemes

Figure 5.38. Copying cost comparison: Lambda-Fact5, $V = 16943$.



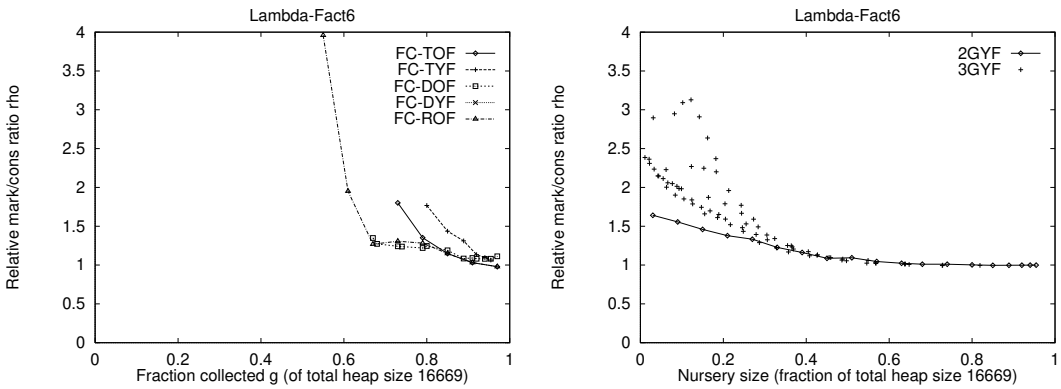
(a) FC schemes (b) GYF schemes

Figure 5.39. Copying cost comparison: Lambda-Fact5, $V = 20654$.



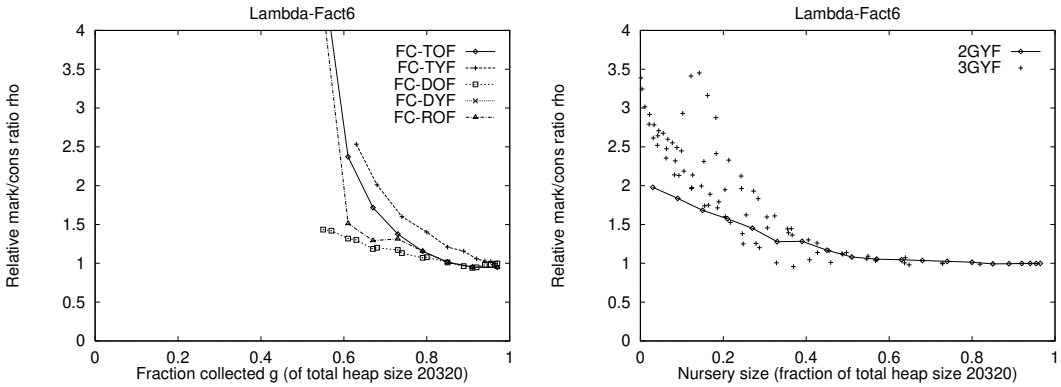
(a) FC schemes (b) GYF schemes

Figure 5.40. Copying cost comparison: Lambda-Fact5, $V = 25177$.



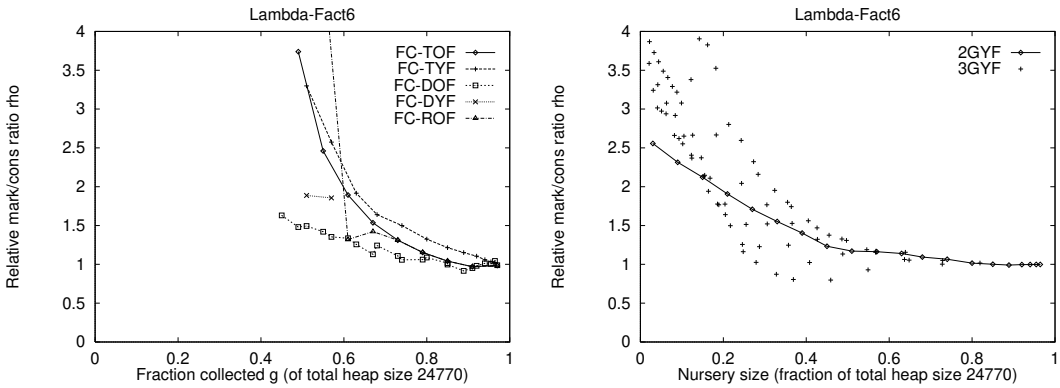
(a) FC schemes (b) GYF schemes

Figure 5.41. Copying cost comparison: Lambda-Fact6, $V = 16669$.



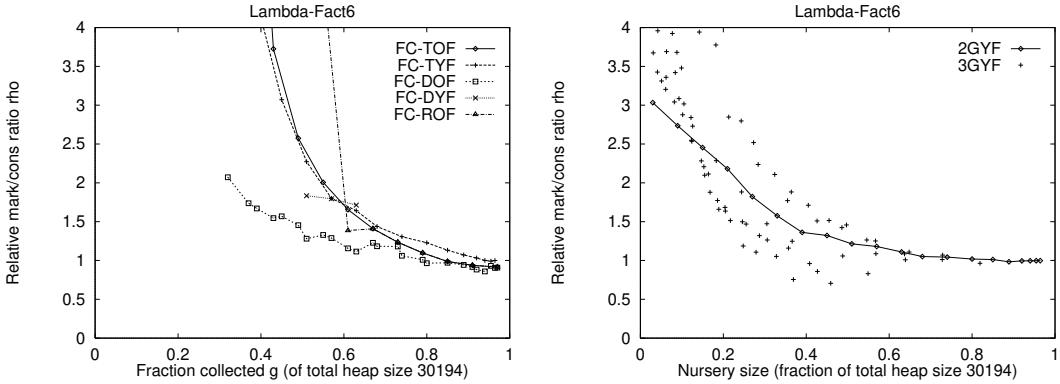
(a) FC schemes (b) GYF schemes

Figure 5.42. Copying cost comparison: Lambda-Fact6, $V = 20320$.



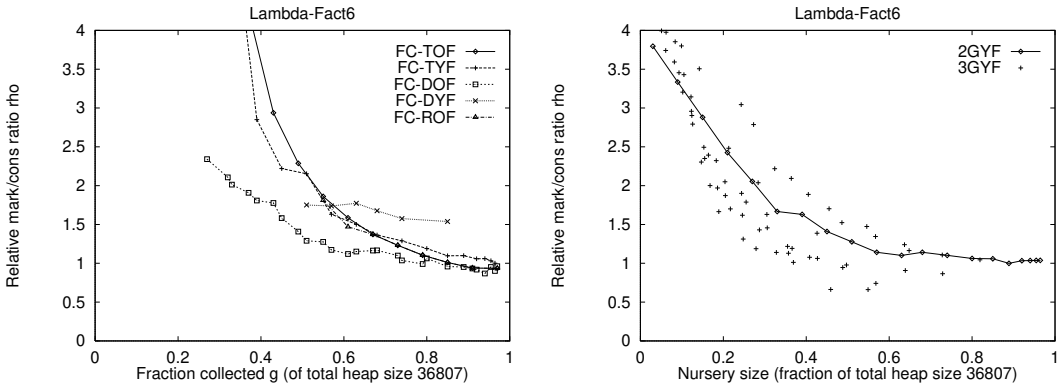
(a) FC schemes (b) GYF schemes

Figure 5.43. Copying cost comparison: Lambda-Fact6, $V = 24770$.



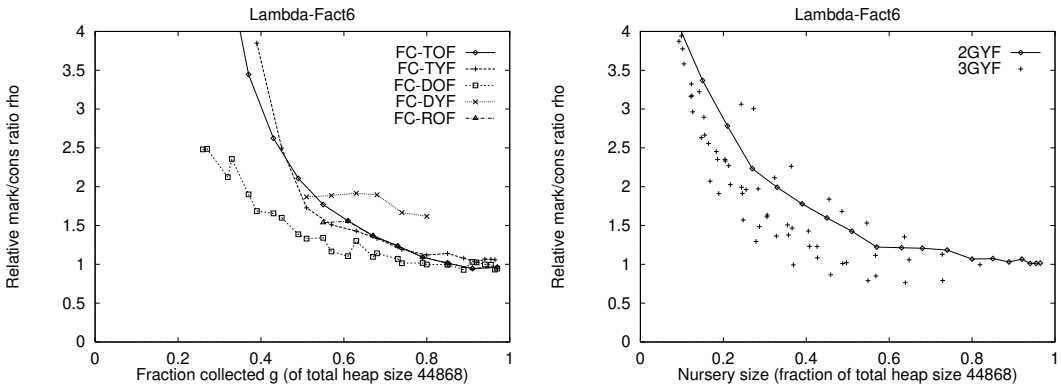
(a) FC schemes (b) GYF schemes

Figure 5.44. Copying cost comparison: Lambda-Fact6, $V = 30194$.



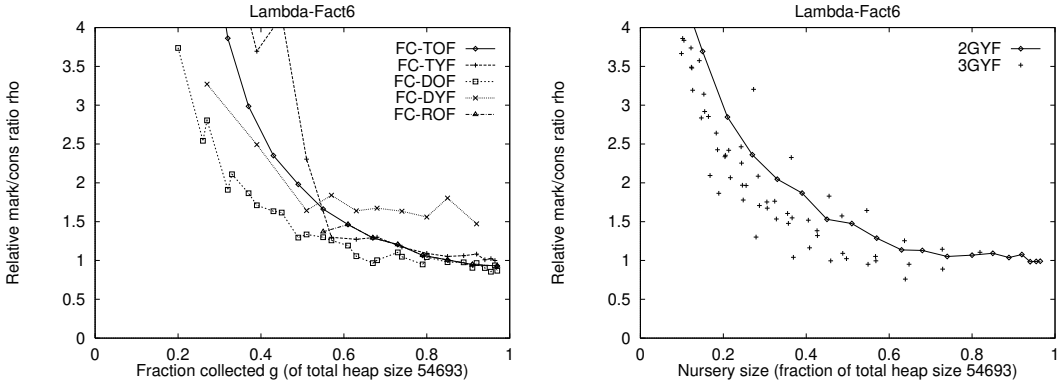
(a) FC schemes (b) GYF schemes

Figure 5.45. Copying cost comparison: Lambda-Fact6, $V = 36807$.



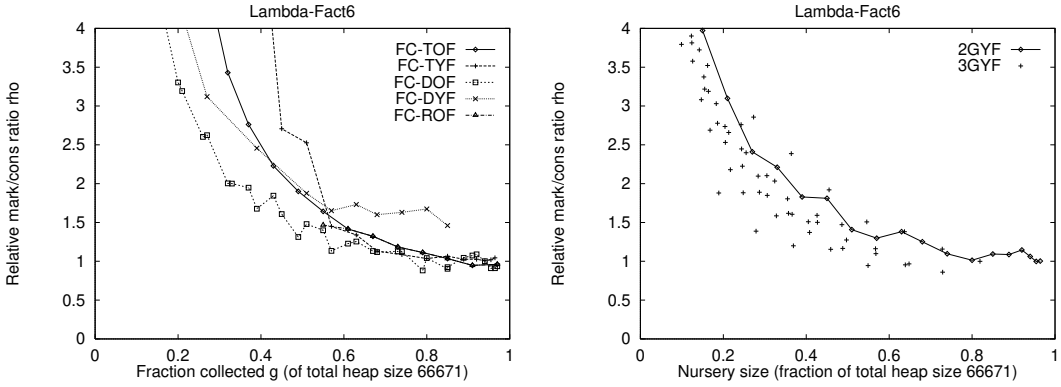
(a) FC schemes (b) GYF schemes

Figure 5.46. Copying cost comparison: Lambda-Fact6, $V = 44868$.



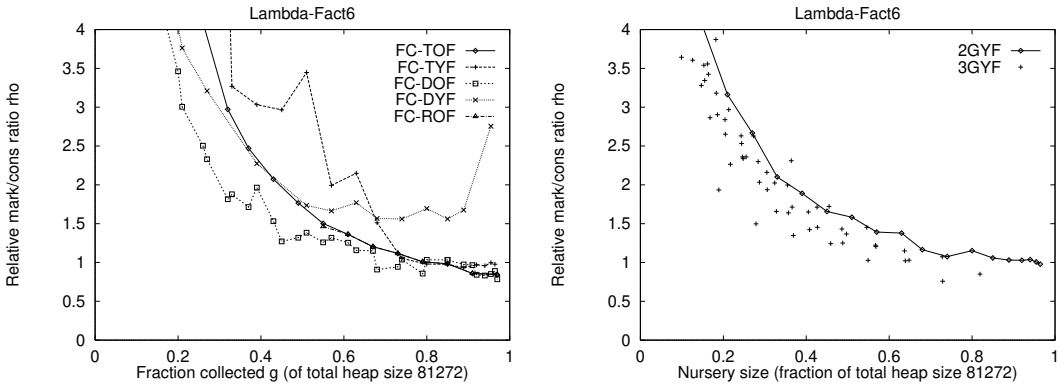
(a) FC schemes (b) GYF schemes

Figure 5.47. Copying cost comparison: Lambda-Fact6, $V = 54693$.



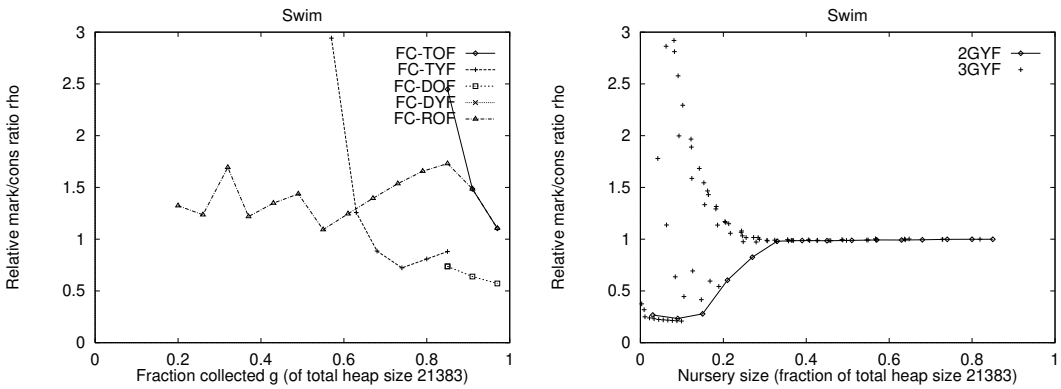
(a) FC schemes (b) GYF schemes

Figure 5.48. Copying cost comparison: Lambda-Fact6, $V = 66671$.



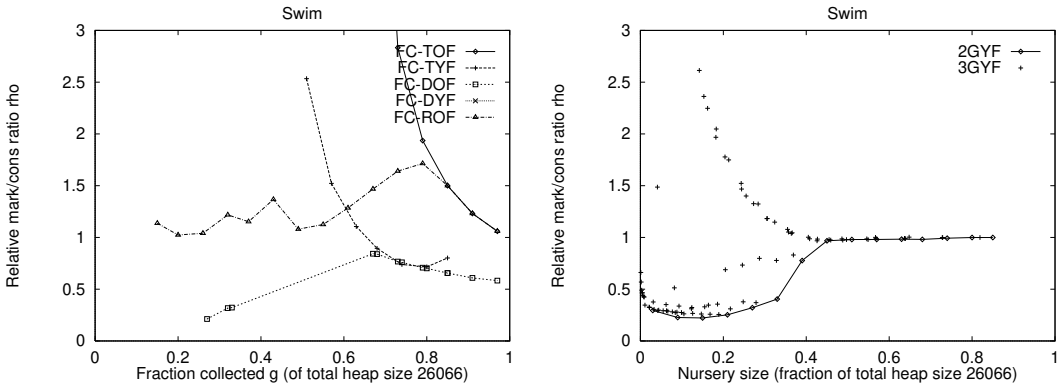
(a) FC schemes (b) GYF schemes

Figure 5.49. Copying cost comparison: Lambda-Fact6, $V = 81272$.



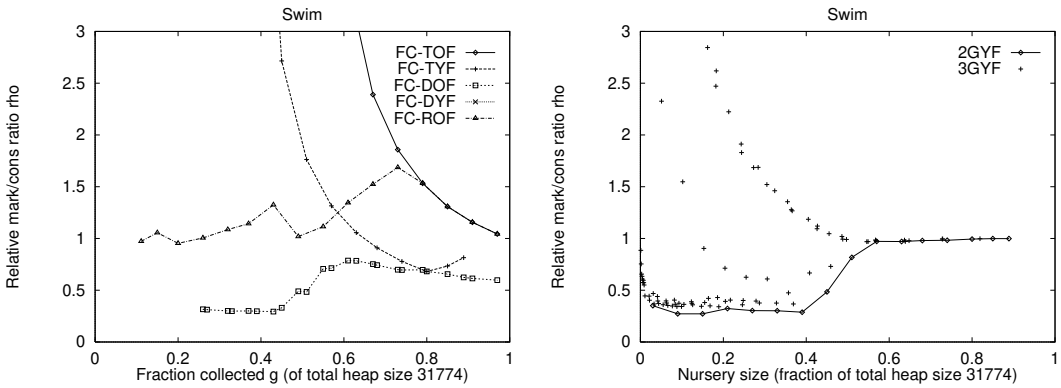
(a) FC schemes (b) GYF schemes

Figure 5.50. Copying cost comparison: Swim, $V = 21383$.



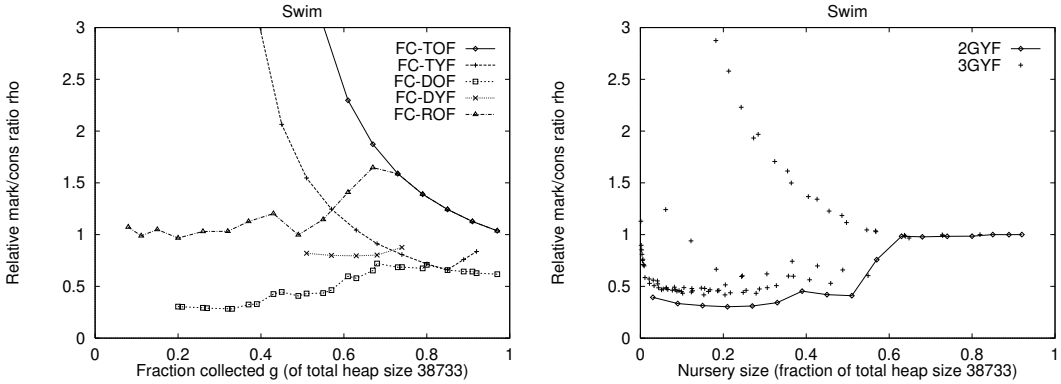
(a) FC schemes (b) GYF schemes

Figure 5.51. Copying cost comparison: Swim, $V = 26066$.



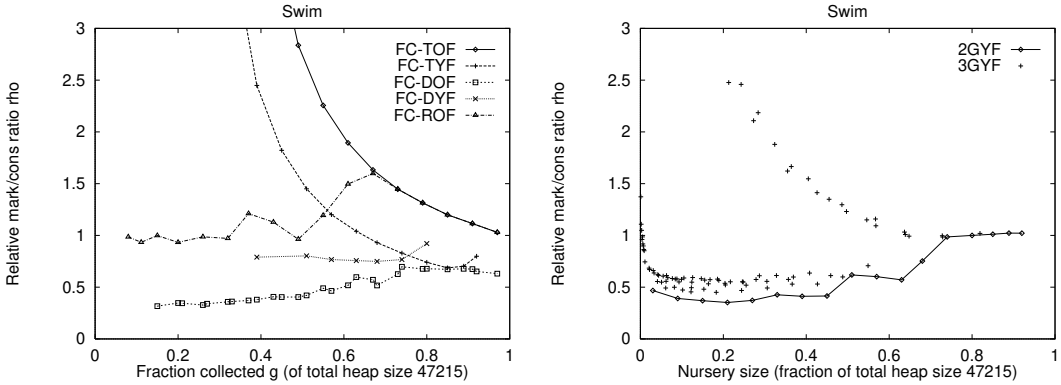
(a) FC schemes (b) GYF schemes

Figure 5.52. Copying cost comparison: Swim, $V = 31774$.



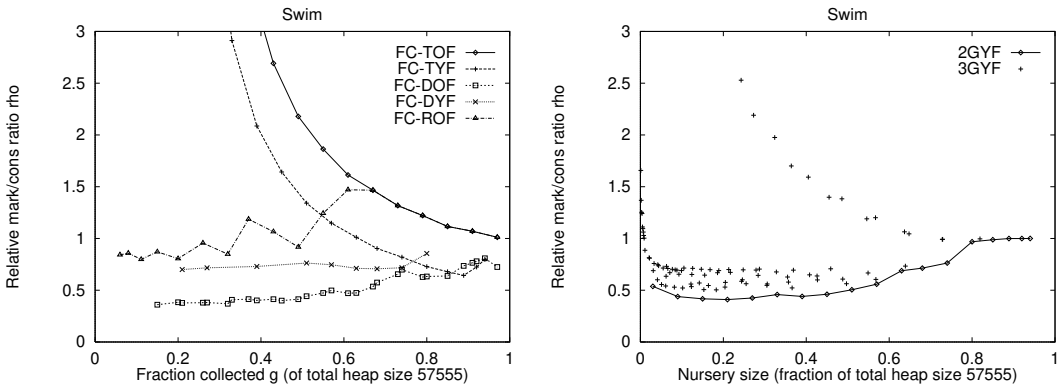
(a) FC schemes (b) GYF schemes

Figure 5.53. Copying cost comparison: Swim, $V = 38733$.



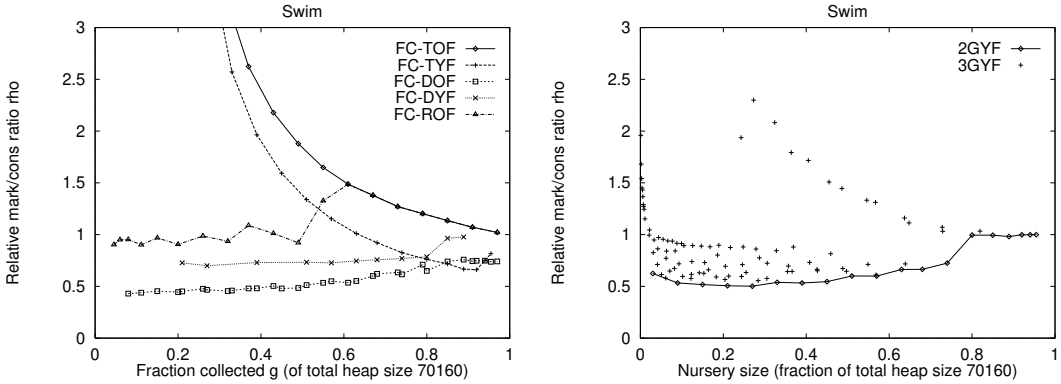
(a) FC schemes (b) GYF schemes

Figure 5.54. Copying cost comparison: Swim, $V = 47215$.



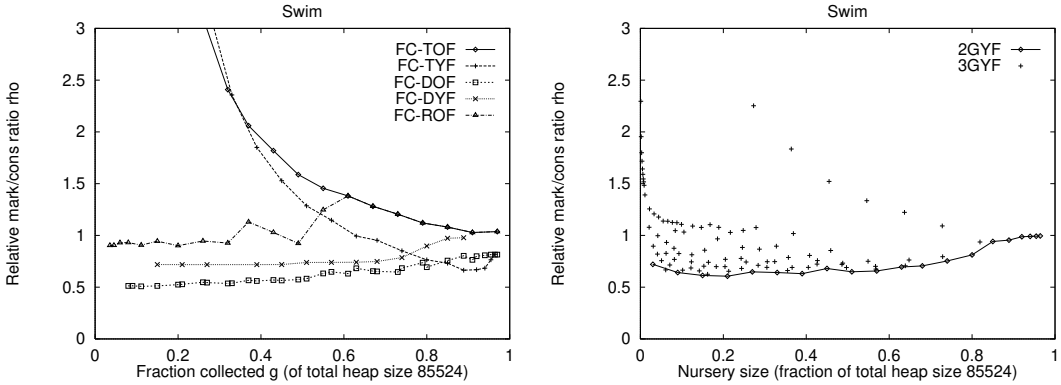
(a) FC schemes (b) GYF schemes

Figure 5.55. Copying cost comparison: Swim, $V = 57555$.



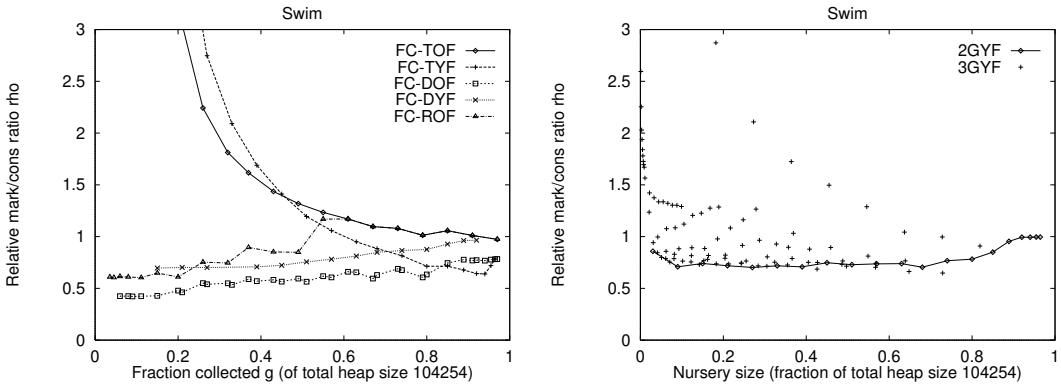
(a) FC schemes (b) GYF schemes

Figure 5.56. Copying cost comparison: Swim, $V = 70160$.



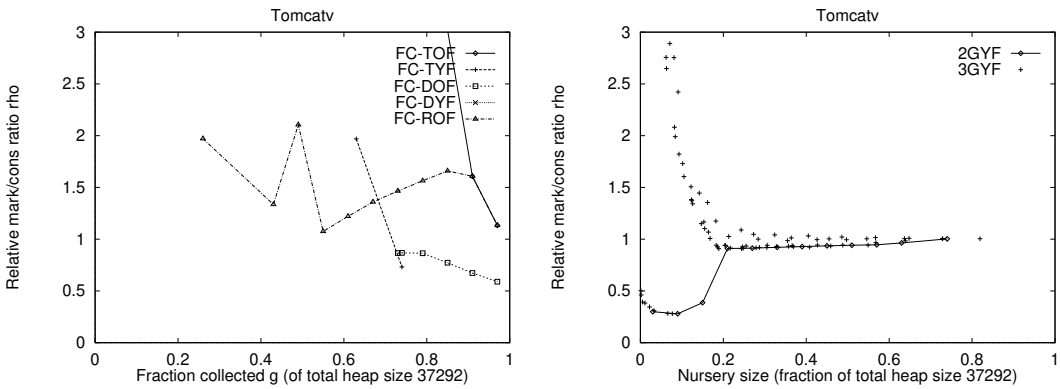
(a) FC schemes (b) GYF schemes

Figure 5.57. Copying cost comparison: Swim, $V = 85524$.



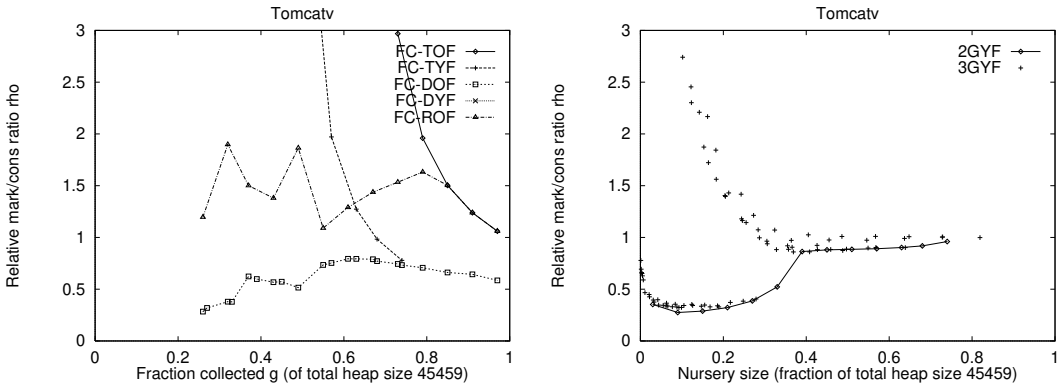
(a) FC schemes (b) GYF schemes

Figure 5.58. Copying cost comparison: Swim, $V = 104254$.



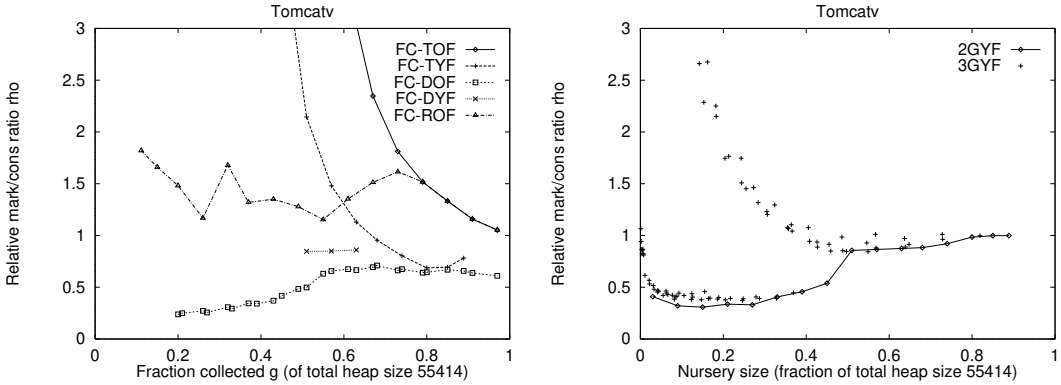
(a) FC schemes (b) GYF schemes

Figure 5.59. Copying cost comparison: Tomcatv, $V = 37292$.



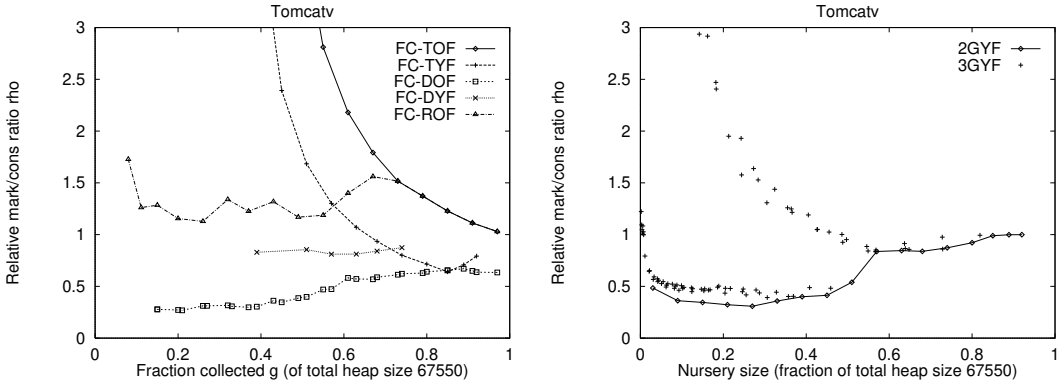
(a) FC schemes (b) GYF schemes

Figure 5.60. Copying cost comparison: Tomcatv, $V = 45459$.



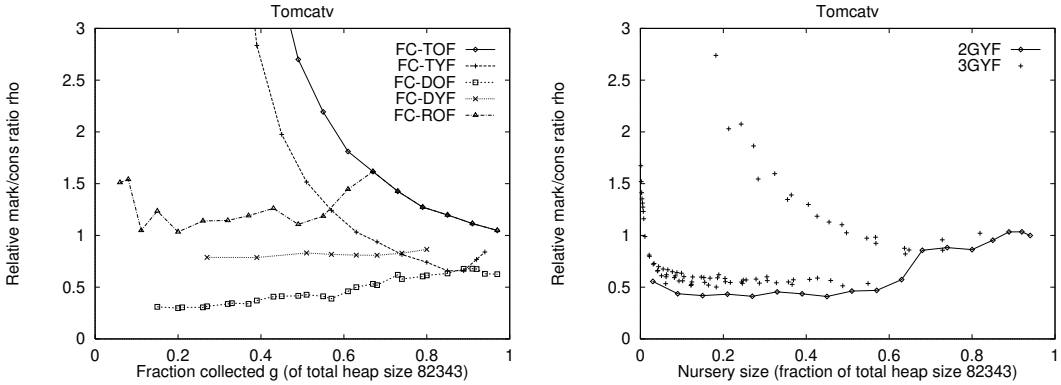
(a) FC schemes (b) GYF schemes

Figure 5.61. Copying cost comparison: Tomcatv, $V = 55414$.



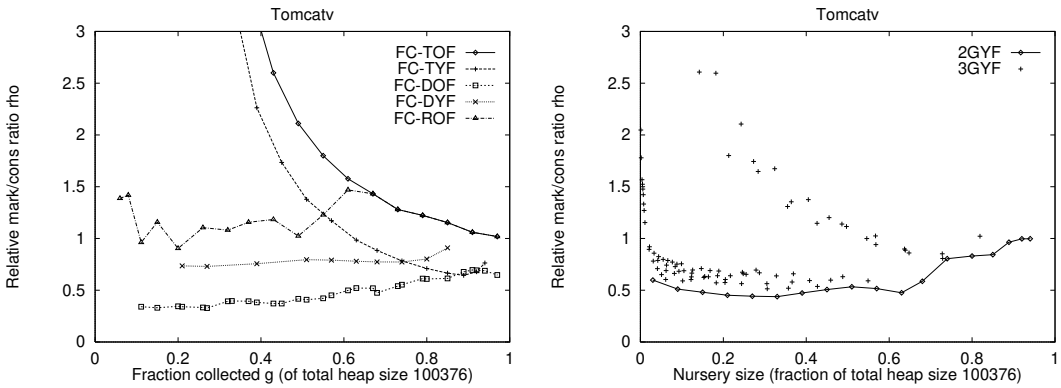
(a) FC schemes (b) GYF schemes

Figure 5.62. Copying cost comparison: Tomcatv, $V = 67550$.



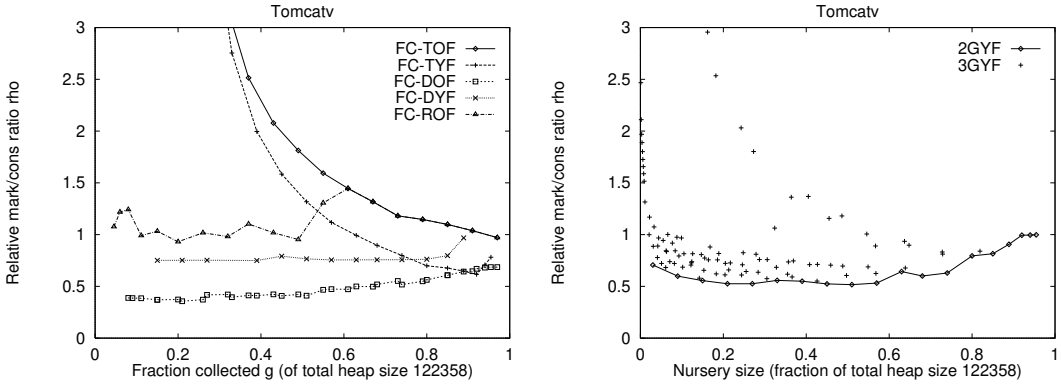
(a) FC schemes (b) GYF schemes

Figure 5.63. Copying cost comparison: Tomcatv, $V = 82343$.



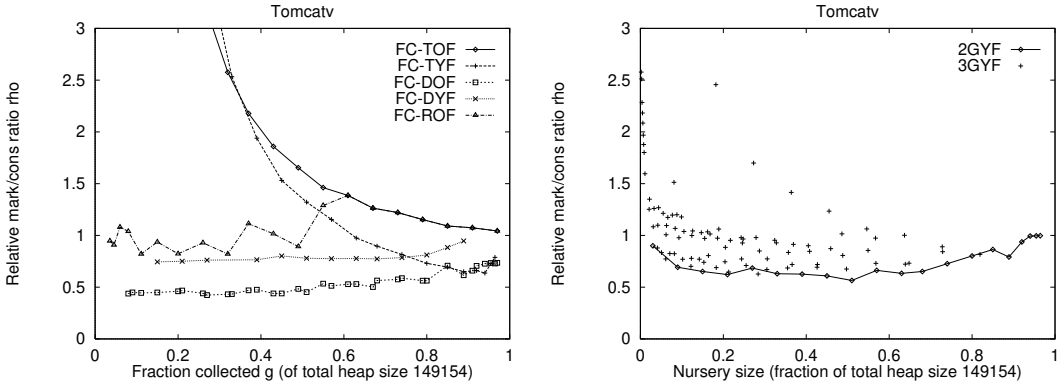
(a) FC schemes (b) GYF schemes

Figure 5.64. Copying cost comparison: Tomcatv, $V = 100376$.



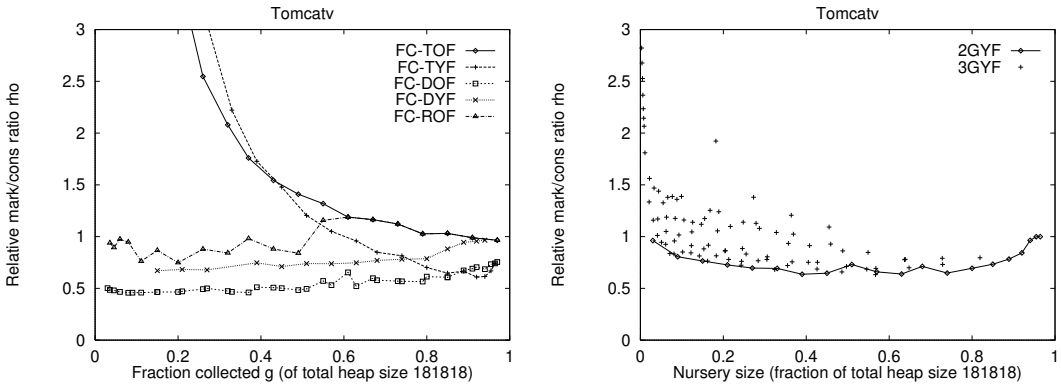
(a) FC schemes (b) GYF schemes

Figure 5.65. Copying cost comparison: Tomcatv, $V = 122358$.



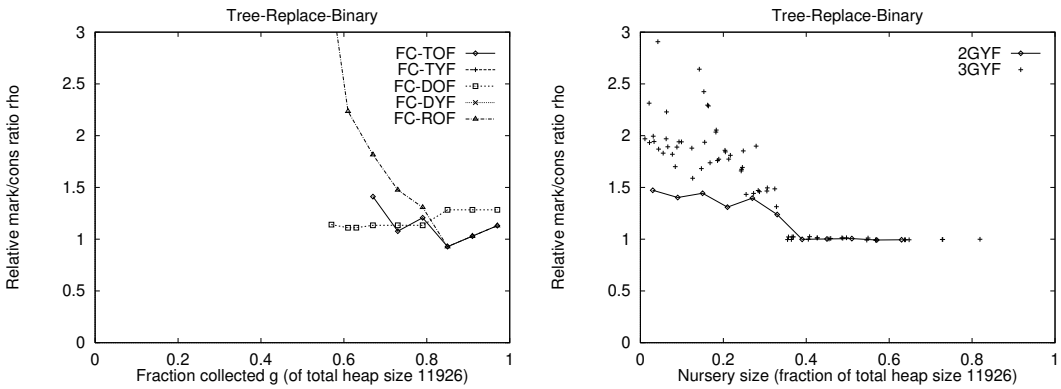
(a) FC schemes (b) GYF schemes

Figure 5.66. Copying cost comparison: Tomcatv, $V = 149154$.



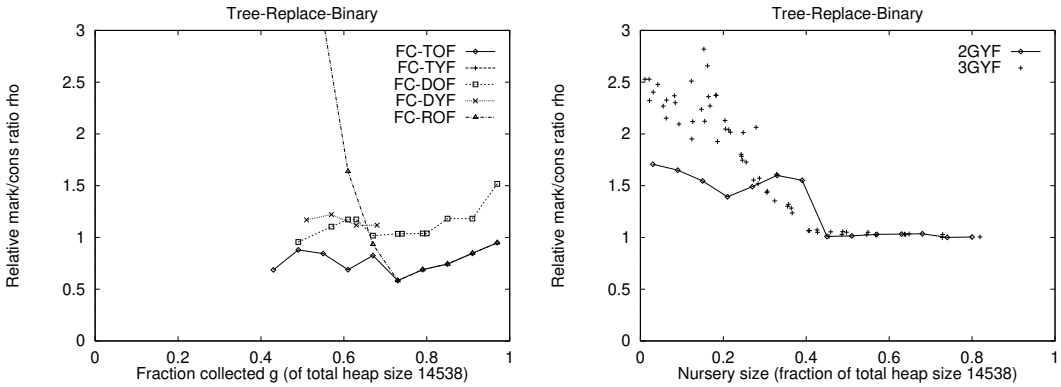
(a) FC schemes (b) GYF schemes

Figure 5.67. Copying cost comparison: Tomcatv, $V = 181818$.



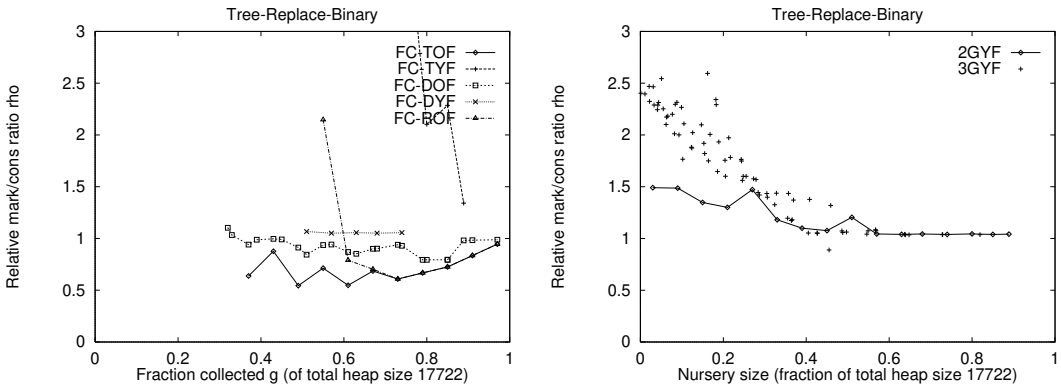
(a) FC schemes (b) GYF schemes

Figure 5.68. Copying cost comparison: Tree-Replace-Binary, $V = 11926$.



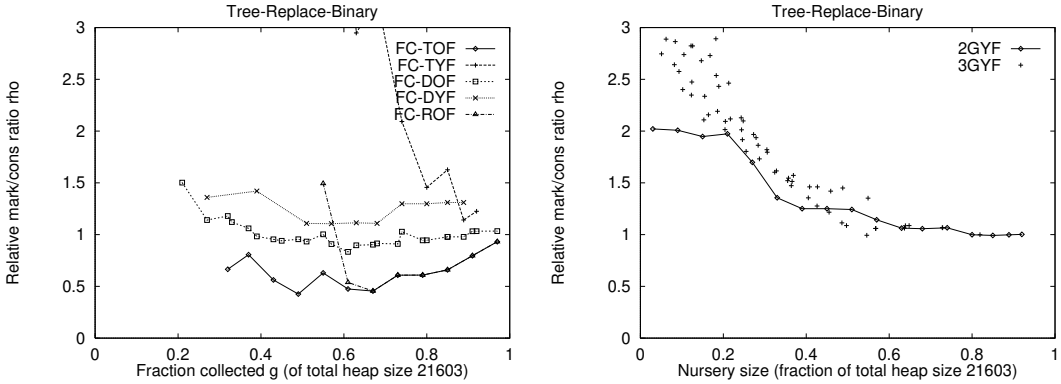
(a) FC schemes (b) GYF schemes

Figure 5.69. Copying cost comparison: Tree-Replace-Binary, $V = 14538$.



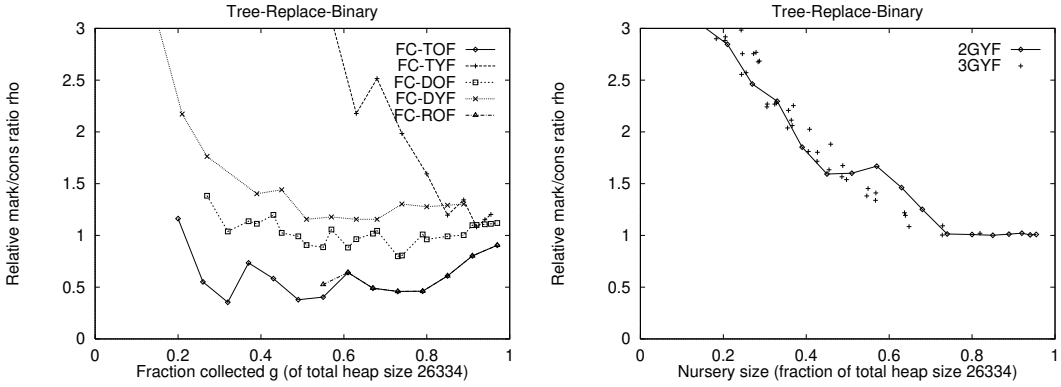
(a) FC schemes (b) GYF schemes

Figure 5.70. Copying cost comparison: Tree-Replace-Binary, $V = 17722$.



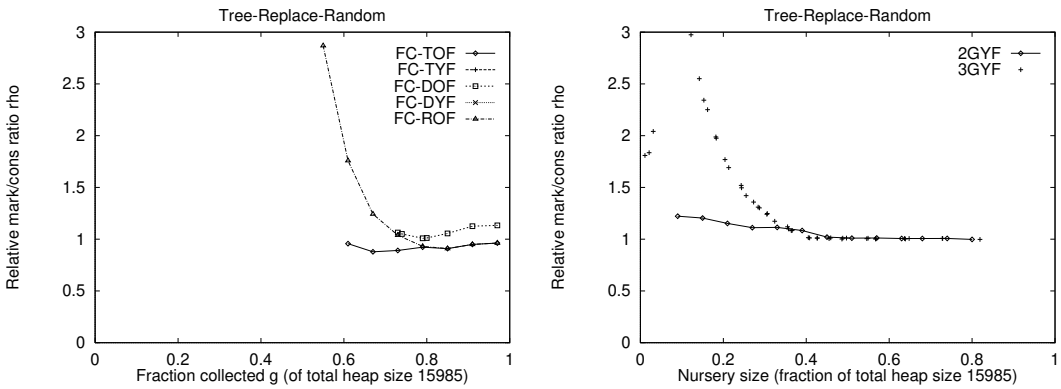
(a) FC schemes (b) GYF schemes

Figure 5.71. Copying cost comparison: Tree-Replace-Binary, $V = 21603$.



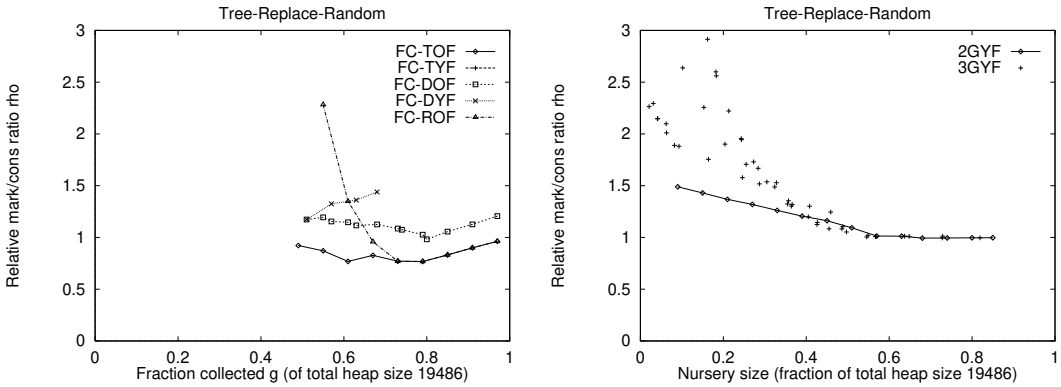
(a) FC schemes (b) GYF schemes

Figure 5.72. Copying cost comparison: Tree-Replace-Binary, $V = 26334$.



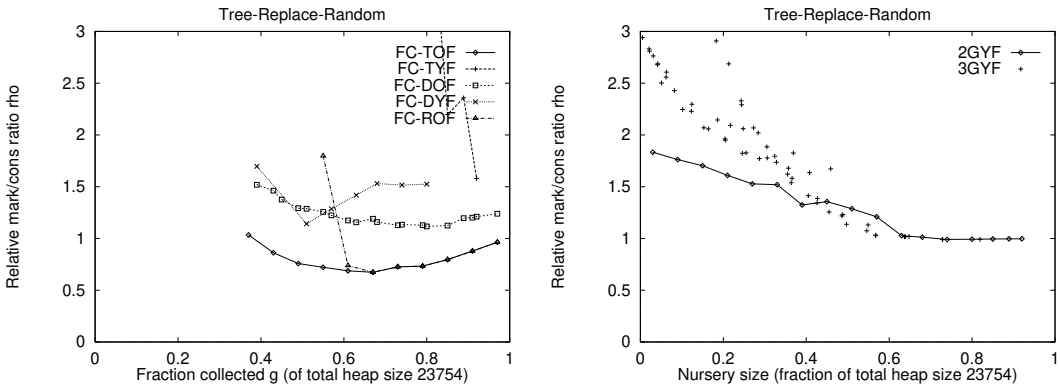
(a) FC schemes (b) GYF schemes

Figure 5.73. Copying cost comparison: Tree-Replace-Random, $V = 15985$.



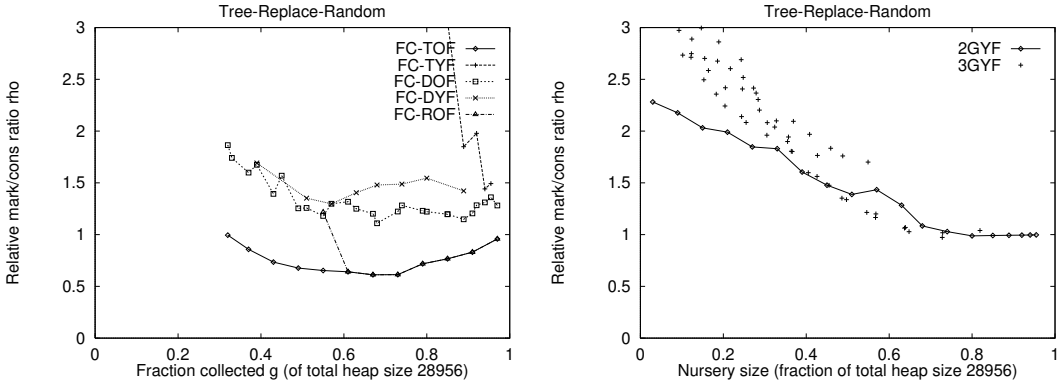
(a) FC schemes (b) GYF schemes

Figure 5.74. Copying cost comparison: Tree-Replace-Random, $V = 19486$.



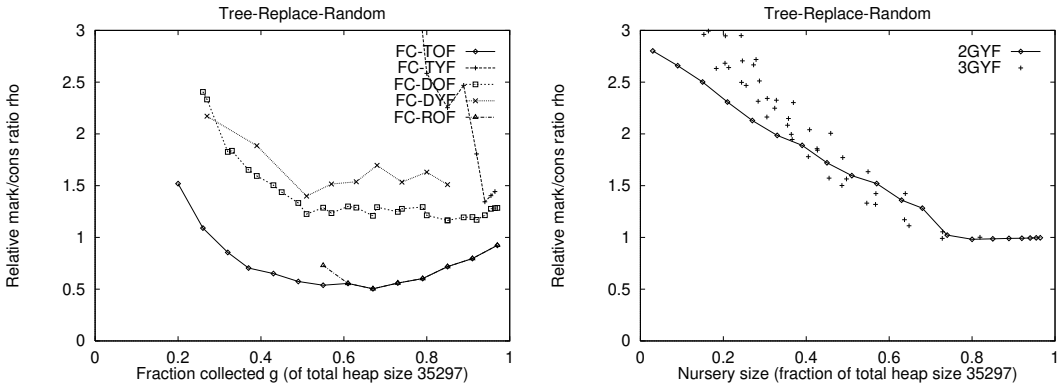
(a) FC schemes (b) GYF schemes

Figure 5.75. Copying cost comparison: Tree-Replace-Random, $V = 23754$.



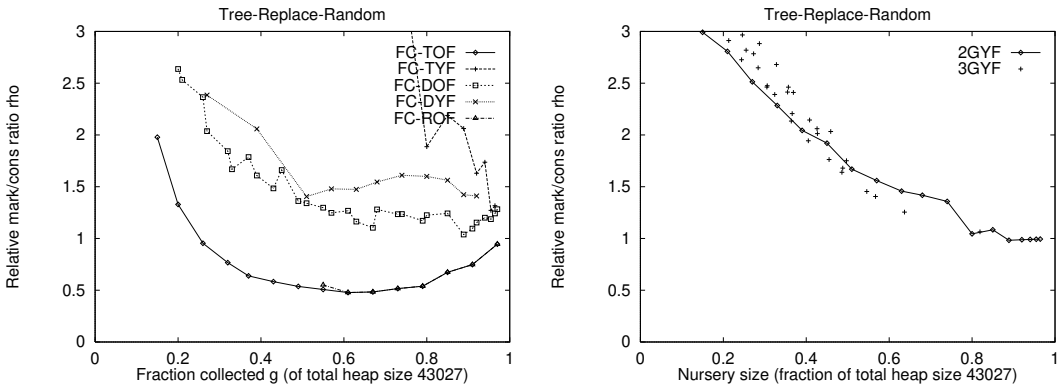
(a) FC schemes (b) GYF schemes

Figure 5.76. Copying cost comparison: Tree-Replace-Random, $V = 28956$.



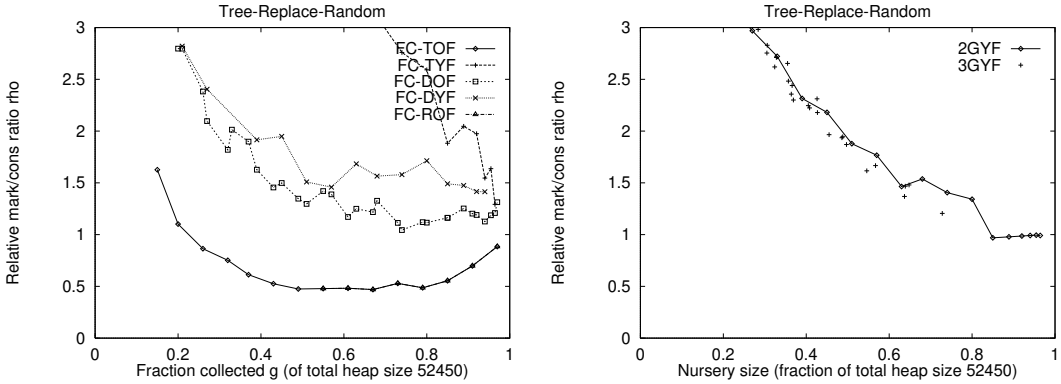
(a) FC schemes (b) GYF schemes

Figure 5.77. Copying cost comparison: Tree-Replace-Random, $V = 35297$.



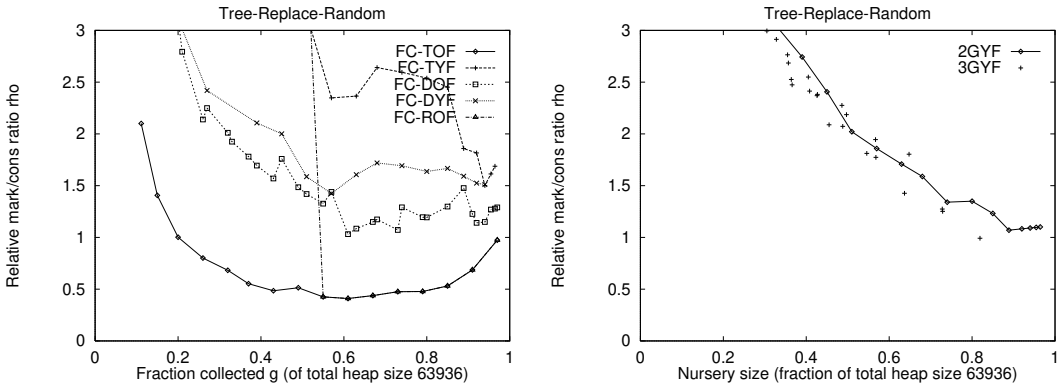
(a) FC schemes (b) GYF schemes

Figure 5.78. Copying cost comparison: Tree-Replace-Random, $V = 43027$.



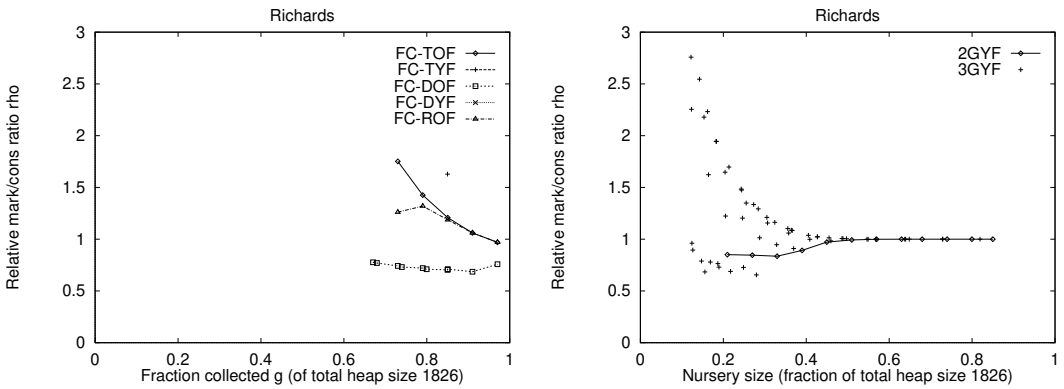
(a) FC schemes (b) GYF schemes

Figure 5.79. Copying cost comparison: Tree-Replace-Random, $V = 52450$.



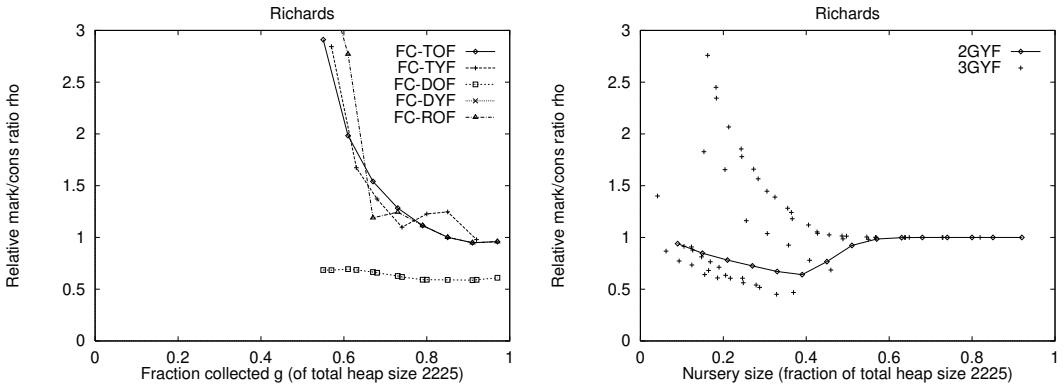
(a) FC schemes (b) GYF schemes

Figure 5.80. Copying cost comparison: Tree-Replace-Random, $V = 63936$.



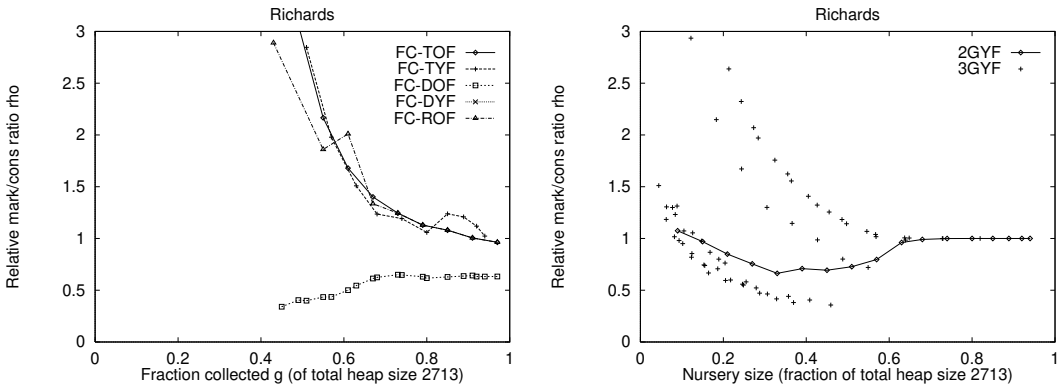
(a) FC schemes (b) GYF schemes

Figure 5.81. Copying cost comparison: Richards, $V = 1826$.



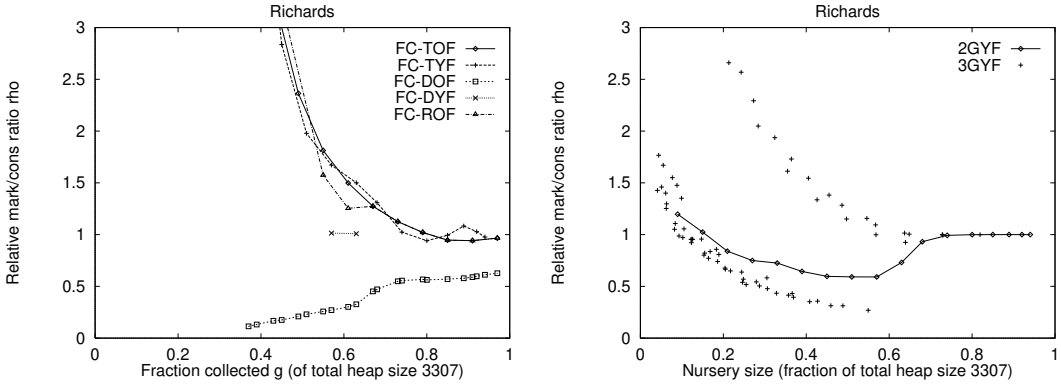
(a) FC schemes (b) GYF schemes

Figure 5.82. Copying cost comparison: Richards, $V = 2225$.



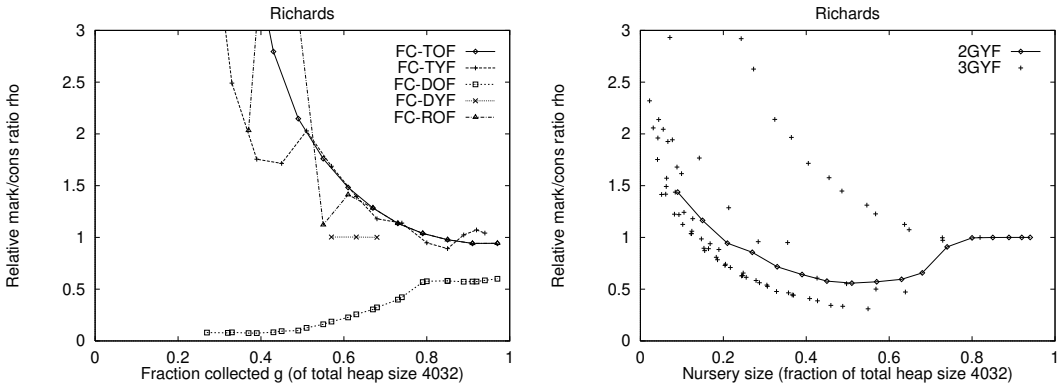
(a) FC schemes (b) GYF schemes

Figure 5.83. Copying cost comparison: Richards, $V = 2713$.



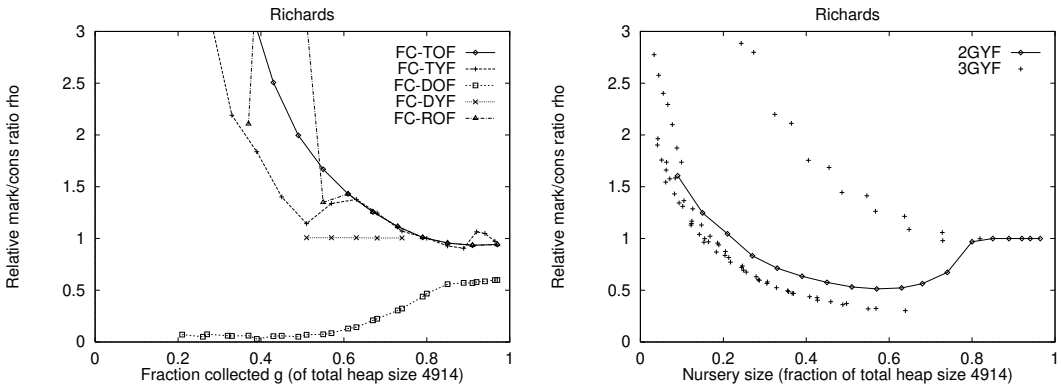
(a) FC schemes (b) GYF schemes

Figure 5.84. Copying cost comparison: Richards, $V = 3307$.



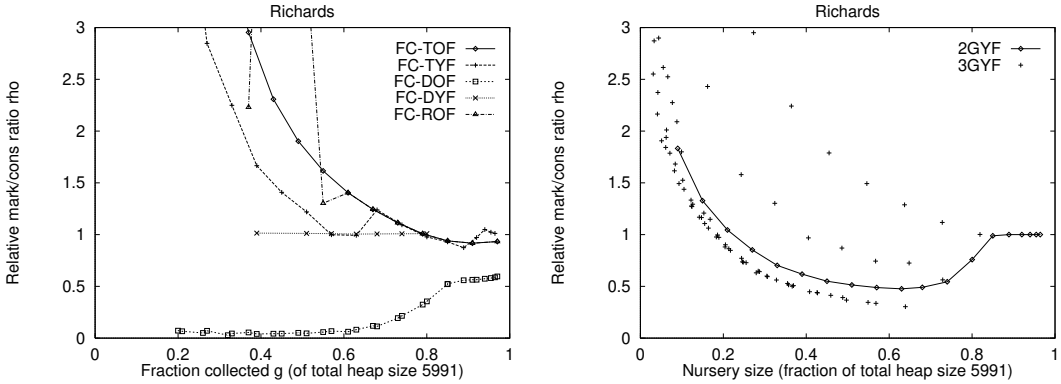
(a) FC schemes (b) GYF schemes

Figure 5.85. Copying cost comparison: Richards, $V = 4032$.



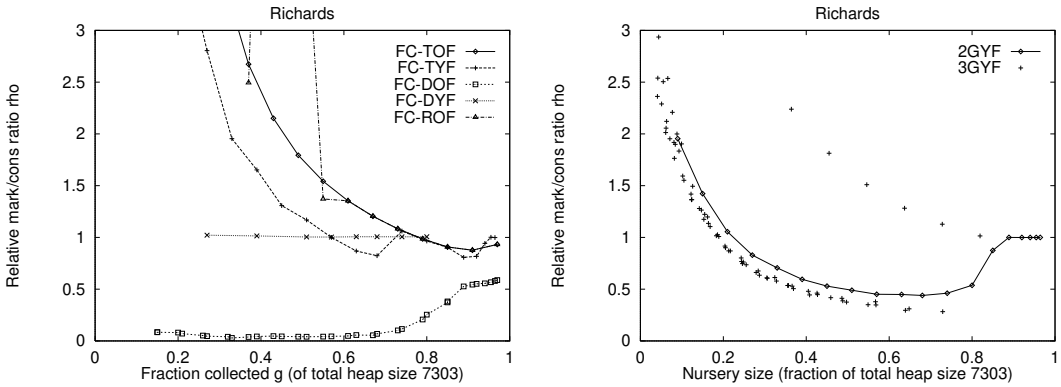
(a) FC schemes (b) GYF schemes

Figure 5.86. Copying cost comparison: Richards, $V = 4914$.



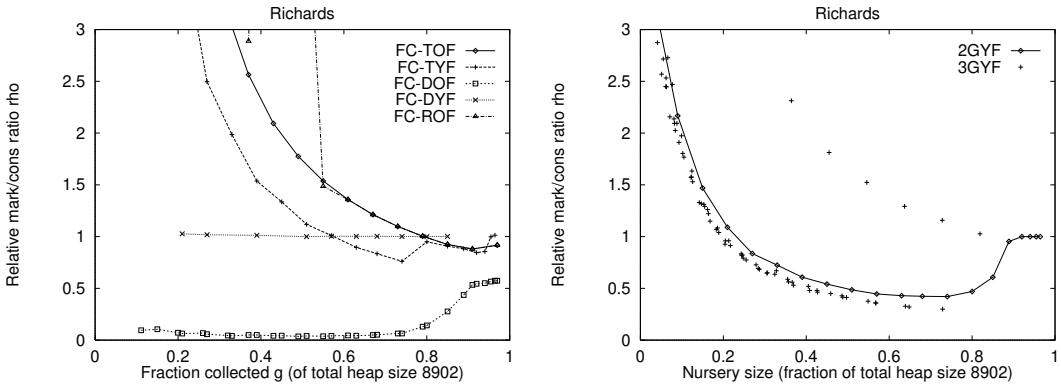
(a) FC schemes (b) GYF schemes

Figure 5.87. Copying cost comparison: Richards, $V = 5991$.



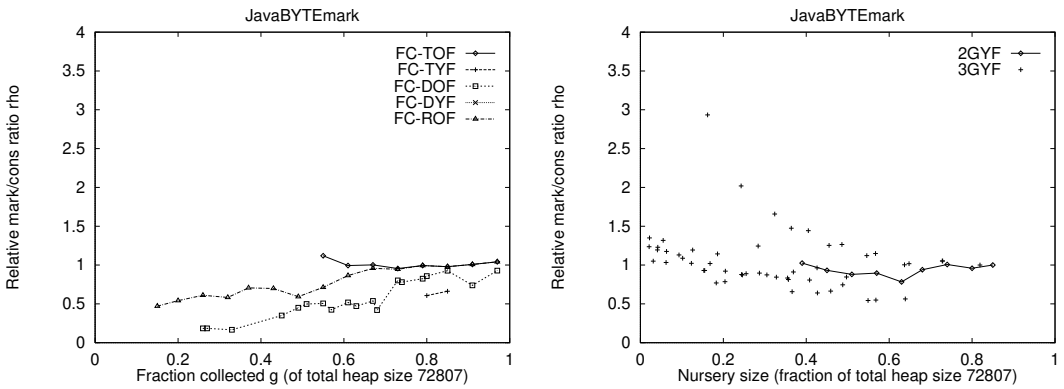
(a) FC schemes (b) GYF schemes

Figure 5.88. Copying cost comparison: Richards, $V = 7303$.



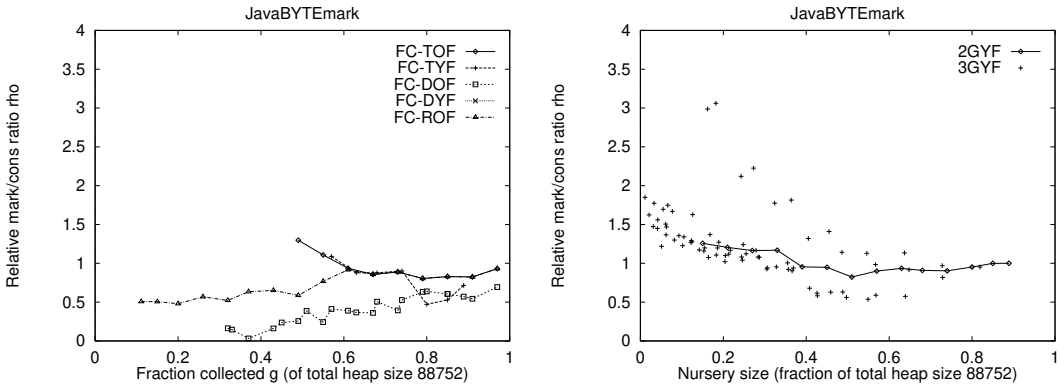
(a) FC schemes (b) GYF schemes

Figure 5.89. Copying cost comparison: Richards, $V = 8902$.



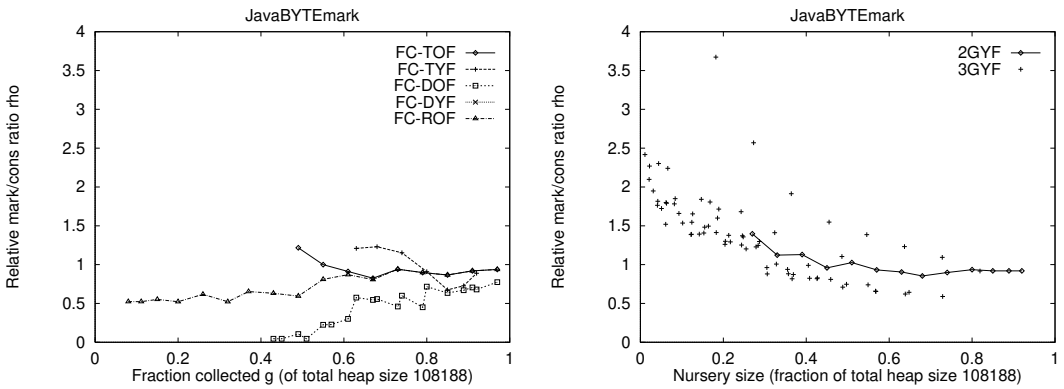
(a) FC schemes (b) GYF schemes

Figure 5.90. Copying cost comparison: JavaBYTEmark, $V = 72807$.



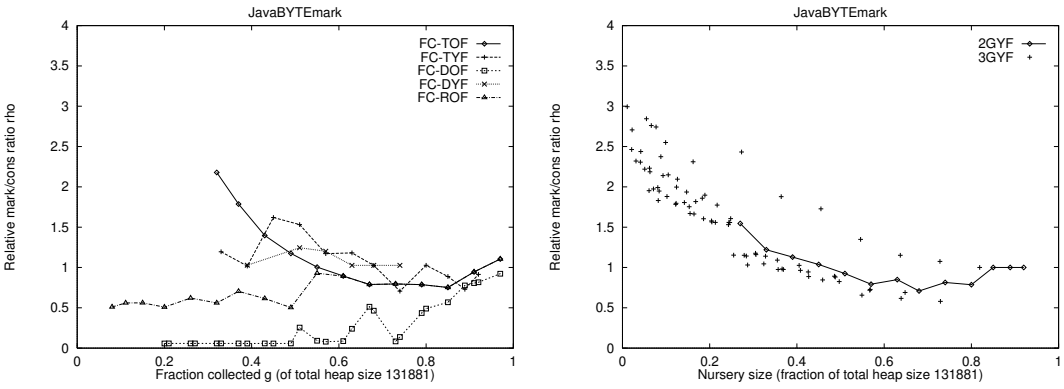
(a) FC schemes (b) GYF schemes

Figure 5.91. Copying cost comparison: JavaBYTEmark, $V = 88752$.



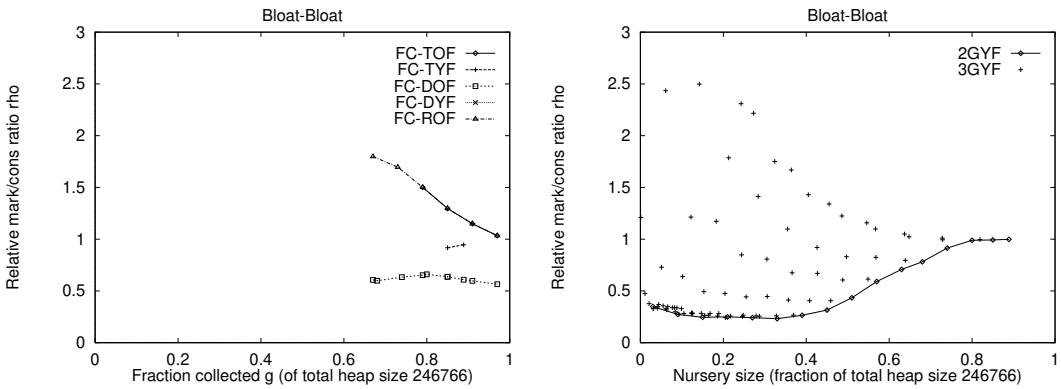
(a) FC schemes (b) GYF schemes

Figure 5.92. Copying cost comparison: JavaBYTEmark, $V = 108188$.



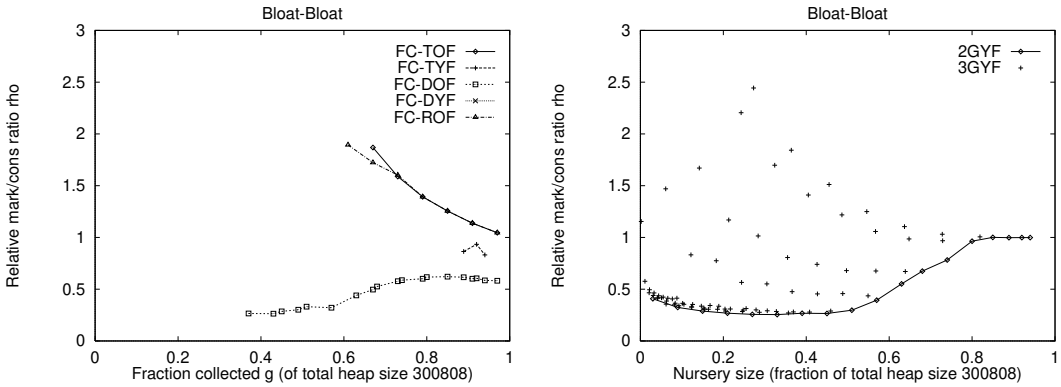
(a) FC schemes (b) GYF schemes

Figure 5.93. Copying cost comparison: JavaBYTEmark, $V = 131881$.



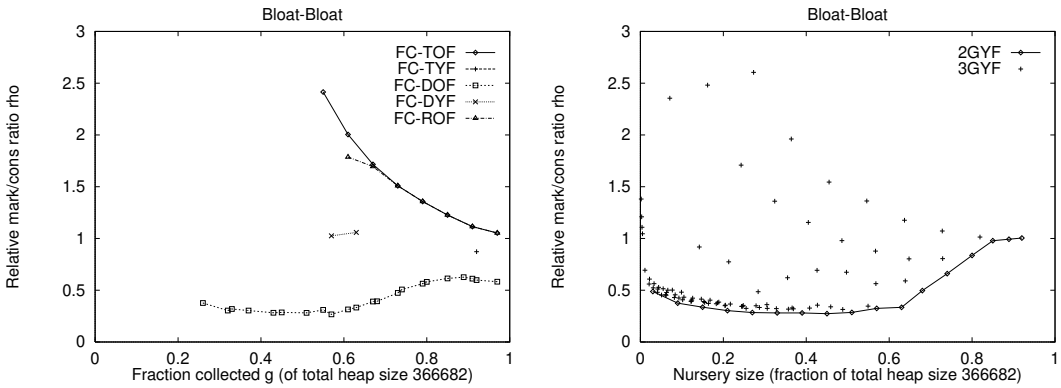
(a) FC schemes (b) GYF schemes

Figure 5.94. Copying cost comparison: Bloat-Bloat, $V = 246766$.



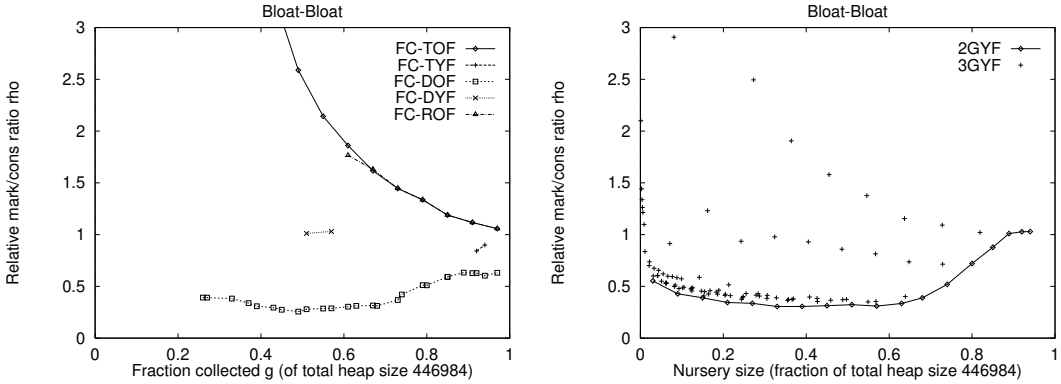
(a) FC schemes (b) GYF schemes

Figure 5.95. Copying cost comparison: Bloat-Bloat, $V = 300808$.



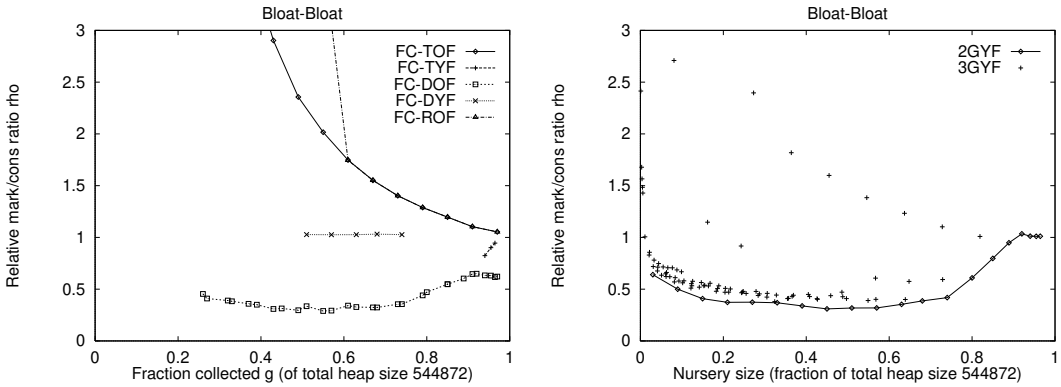
(a) FC schemes (b) GYF schemes

Figure 5.96. Copying cost comparison: Bloat-Bloat, $V = 366682$.



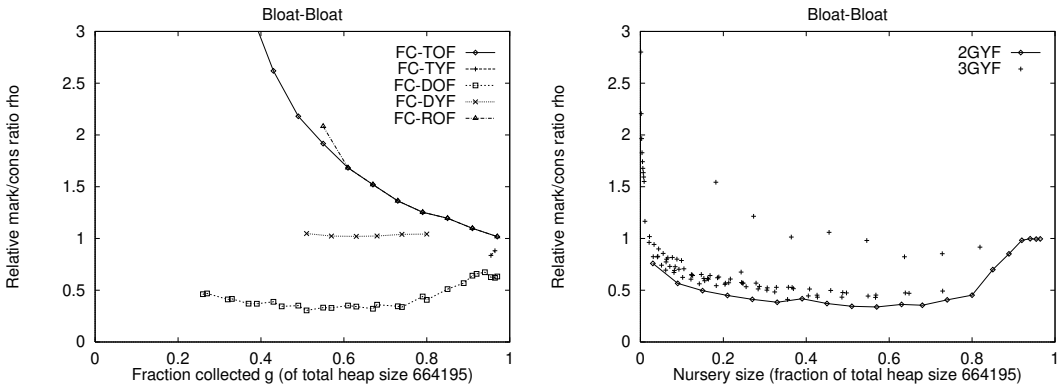
(a) FC schemes (b) GYF schemes

Figure 5.97. Copying cost comparison: Bloat-Bloat, $V = 446984$.



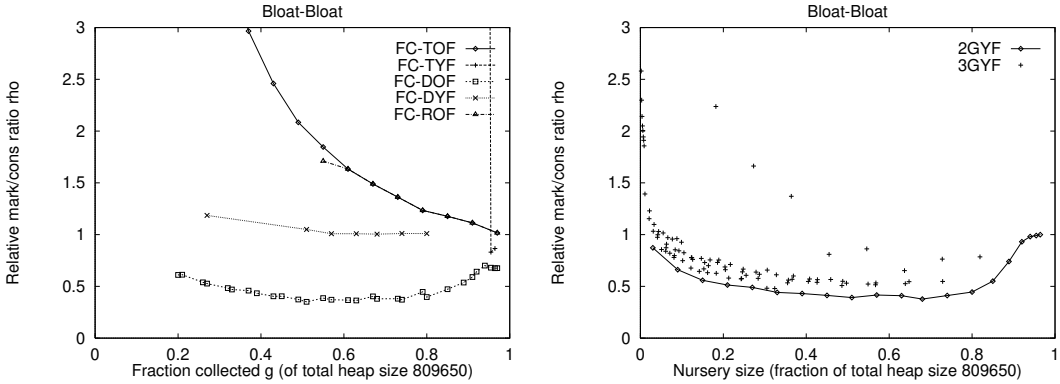
(a) FC schemes (b) GYF schemes

Figure 5.98. Copying cost comparison: Bloat-Bloat, $V = 544872$.



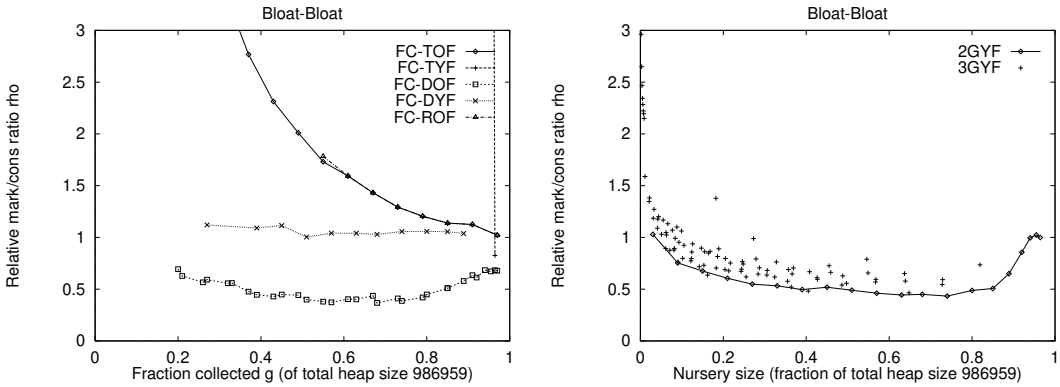
(a) FC schemes (b) GYF schemes

Figure 5.99. Copying cost comparison: Bloat-Bloat, $V = 664195$.



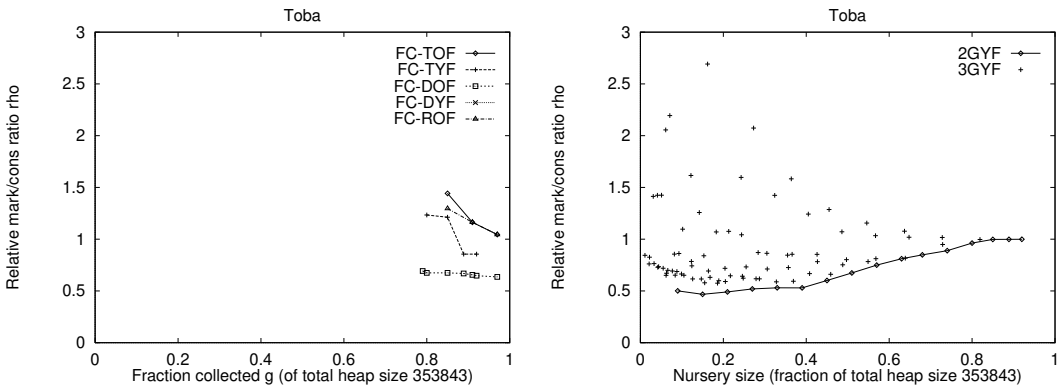
(a) FC schemes (b) GYF schemes

Figure 5.100. Copying cost comparison: Bloat-Bloat, $V = 809650$.



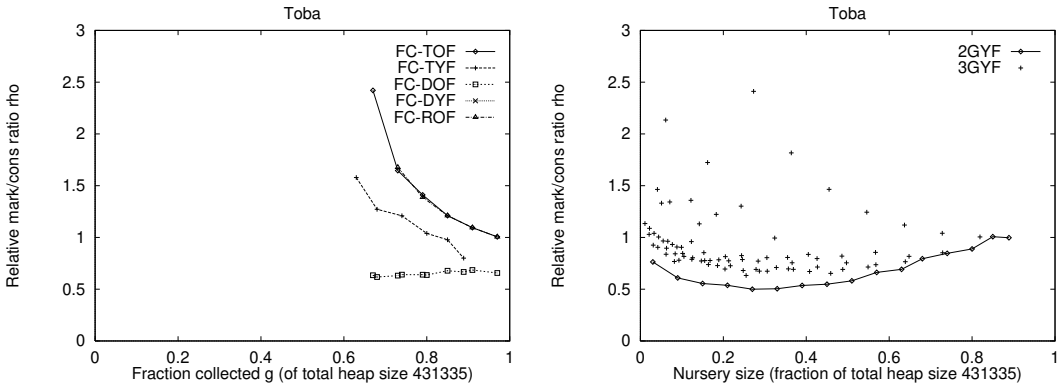
(a) FC schemes (b) GYF schemes

Figure 5.101. Copying cost comparison: Bloat-Bloat, $V = 986959$.



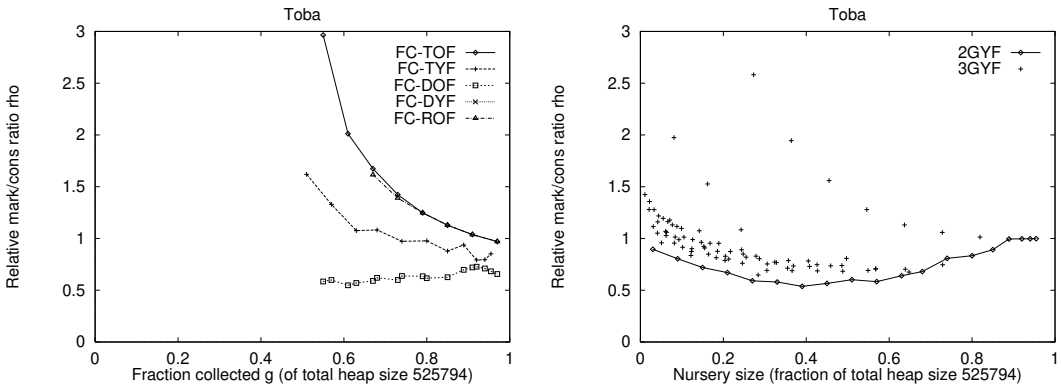
(a) FC schemes (b) GYF schemes

Figure 5.102. Copying cost comparison: Toba, $V = 353843$.



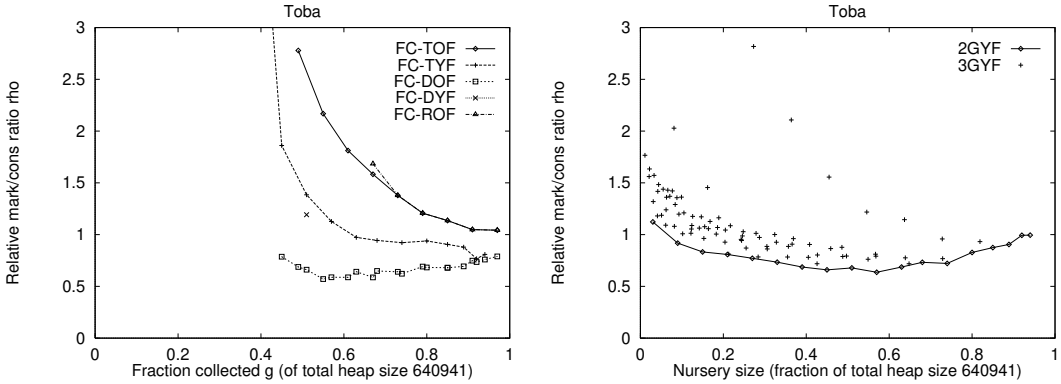
(a) FC schemes (b) GYF schemes

Figure 5.103. Copying cost comparison: Toba, $V = 431335$.



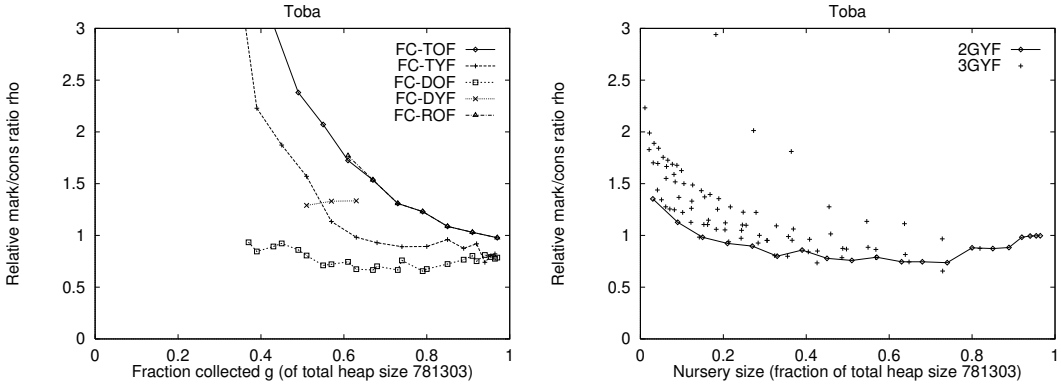
(a) FC schemes (b) GYF schemes

Figure 5.104. Copying cost comparison: Toba, $V = 525794$.



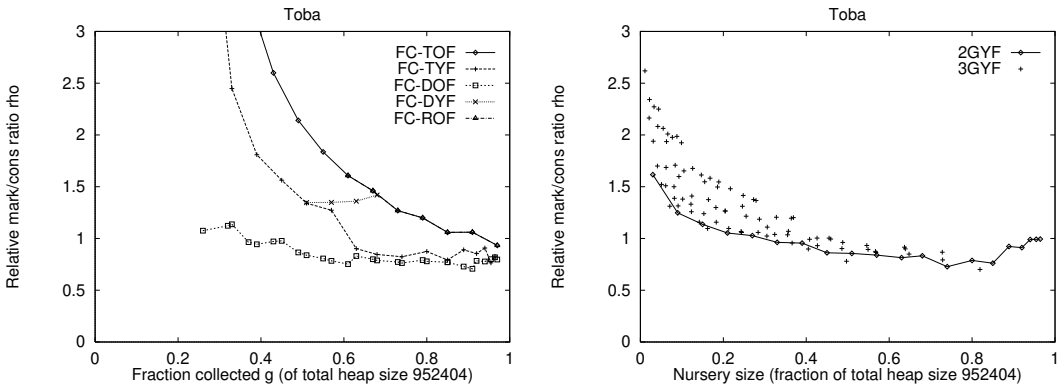
(a) FC schemes (b) GYF schemes

Figure 5.105. Copying cost comparison: Toba, $V = 640941$.



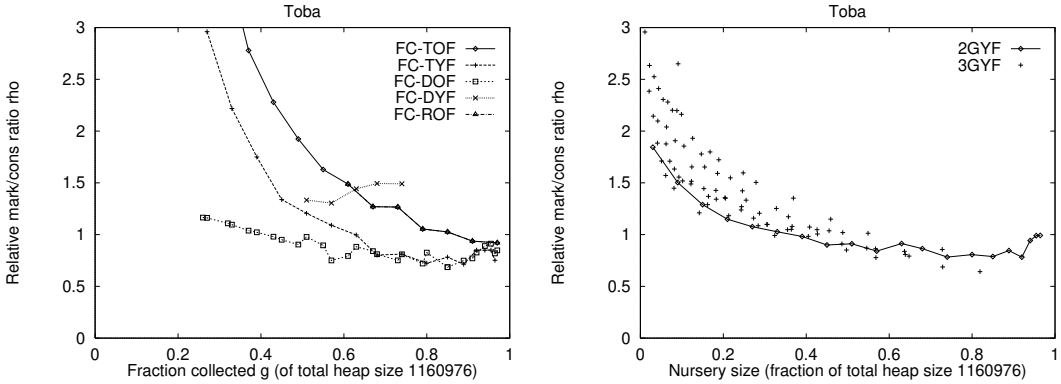
(a) FC schemes (b) GYF schemes

Figure 5.106. Copying cost comparison: Toba, $V = 781303$.



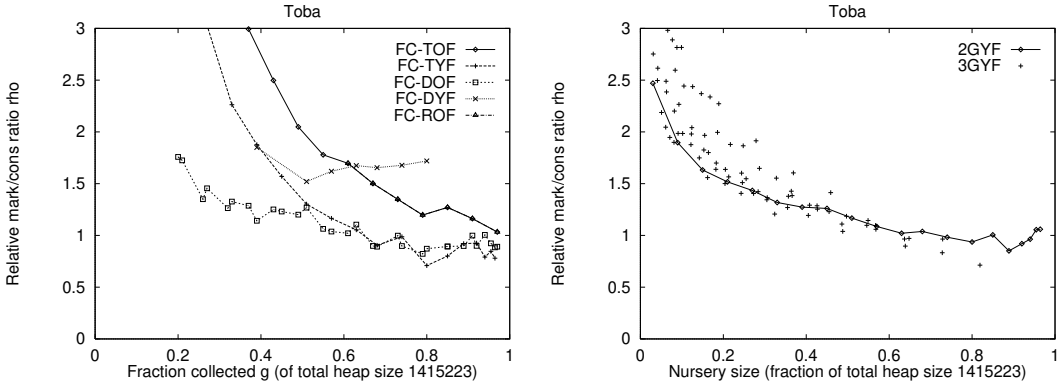
(a) FC schemes (b) GYF schemes

Figure 5.107. Copying cost comparison: Toba, $V = 952404$.



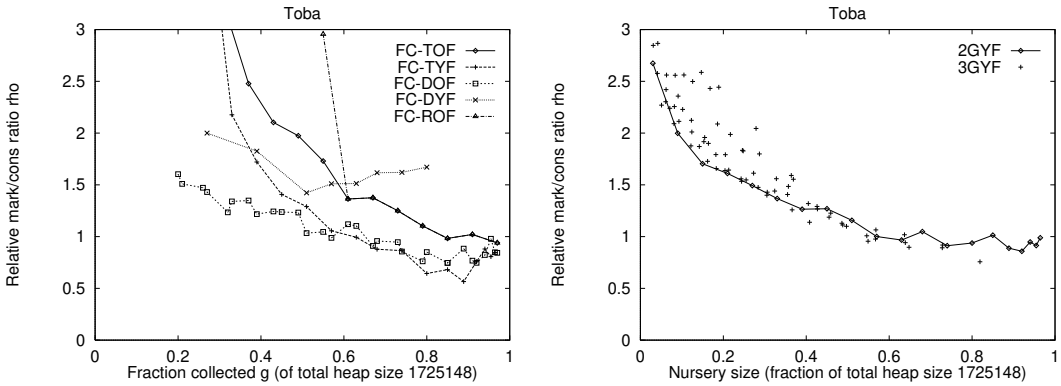
(a) FC schemes (b) GYF schemes

Figure 5.108. Copying cost comparison: Toba, $V = 1160976$.



(a) FC schemes (b) GYF schemes

Figure 5.109. Copying cost comparison: Toba, $V = 1415223$.



(a) FC schemes (b) GYF schemes

Figure 5.110. Copying cost comparison: Toba, $V = 1725148$.

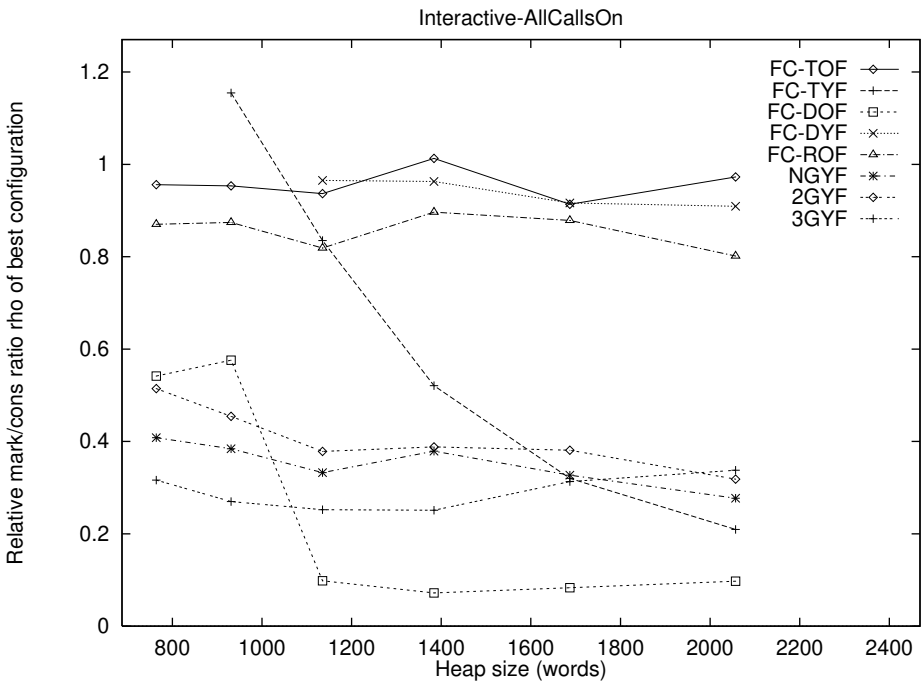


Figure 5.111. Best configuration comparison: Interactive-AllCallsOn.

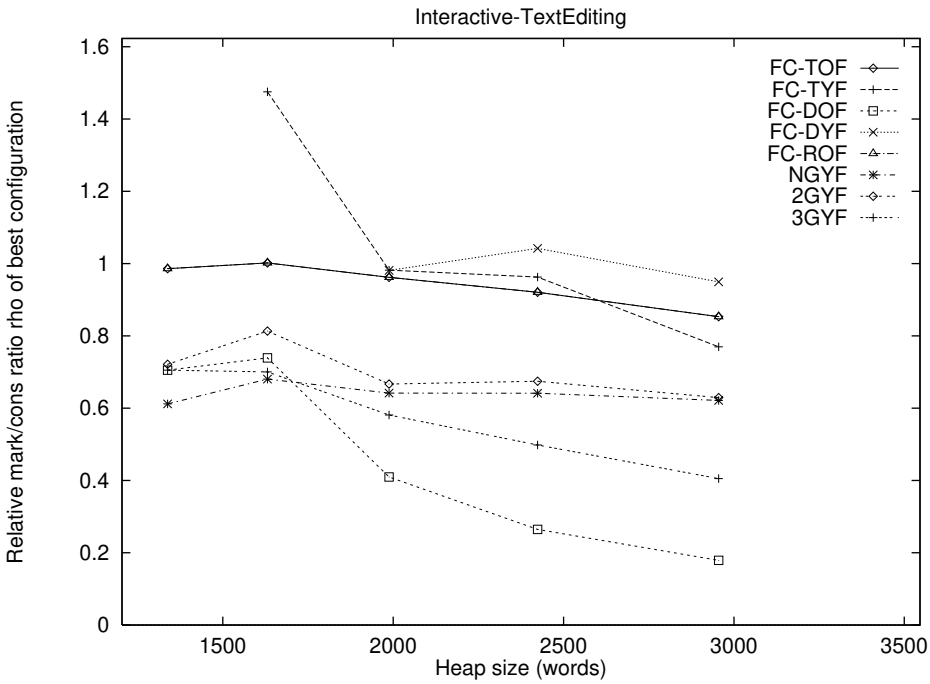


Figure 5.112. Best configuration comparison: Interactive-TextEditing.

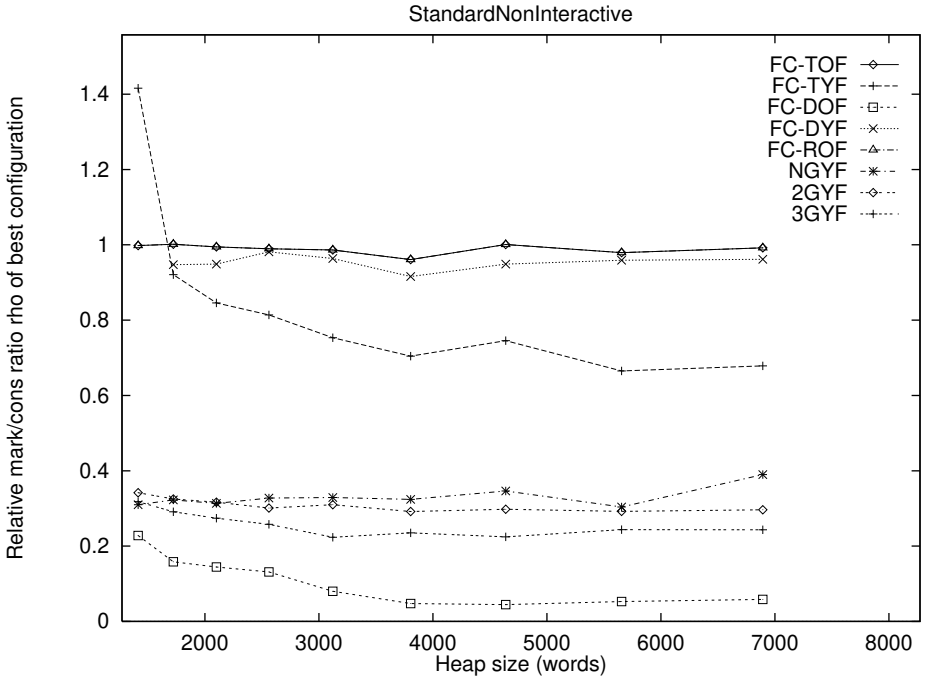


Figure 5.113. Best configuration comparison: StandardNonInteractive.

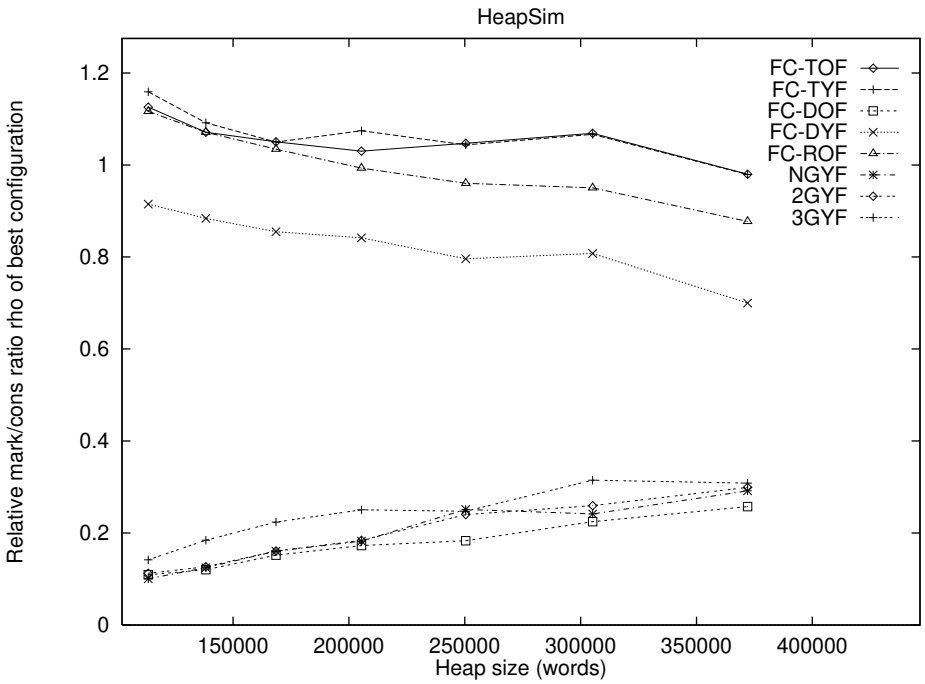


Figure 5.114. Best configuration comparison: HeapSim.

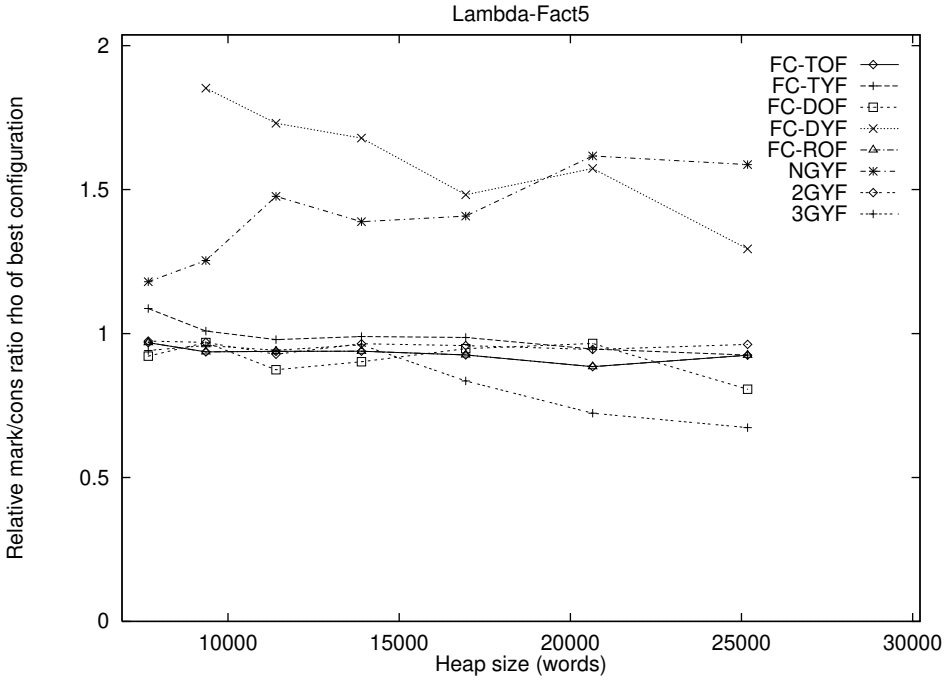


Figure 5.115. Best configuration comparison: Lambda-Fact5.

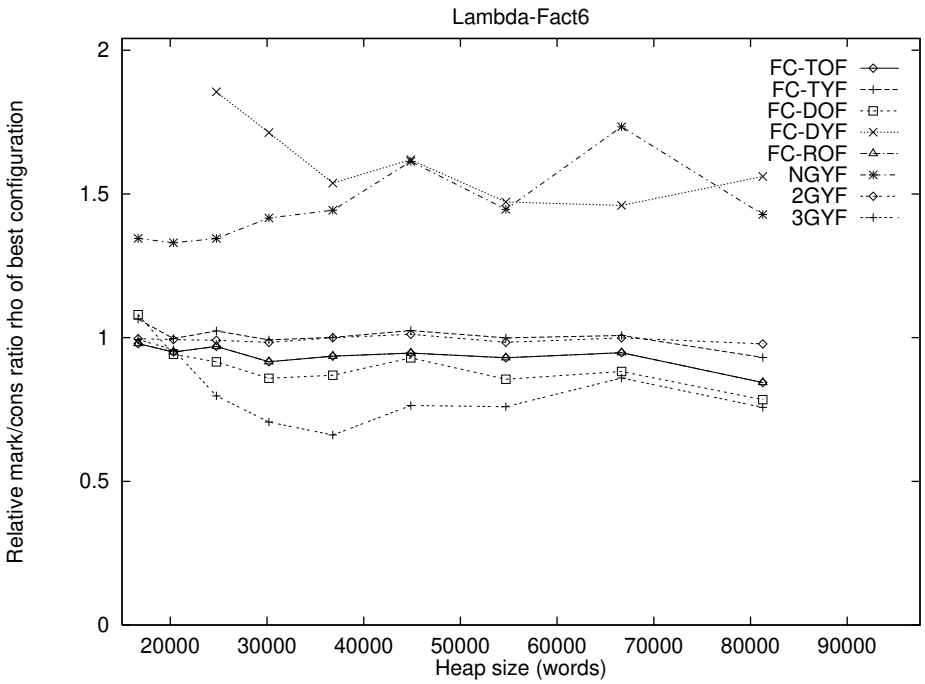


Figure 5.116. Best configuration comparison: Lambda-Fact6.

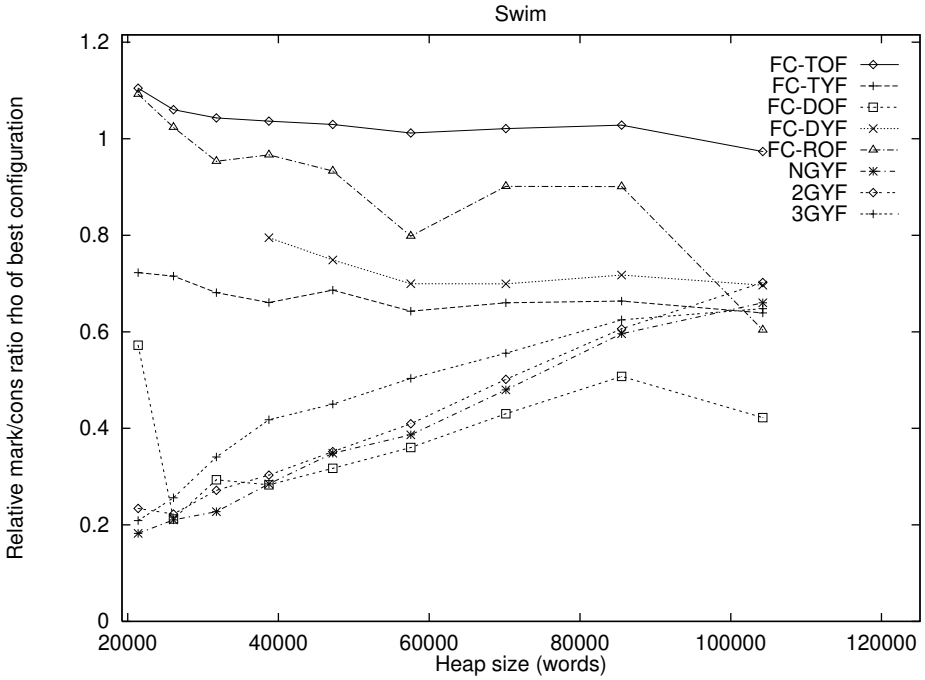


Figure 5.117. Best configuration comparison: Swim.

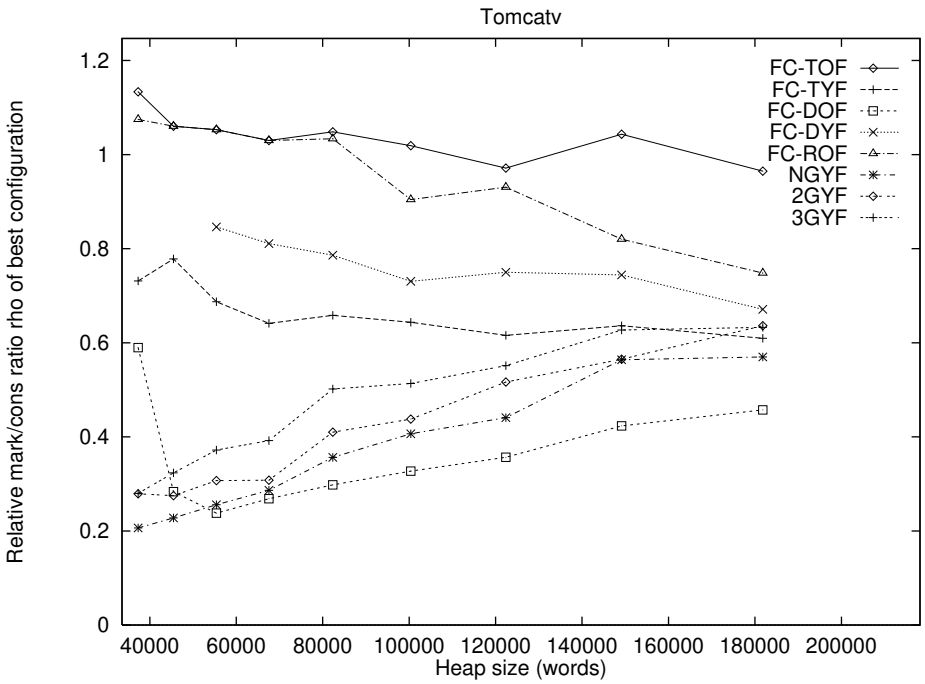


Figure 5.118. Best configuration comparison: Tomcatv.

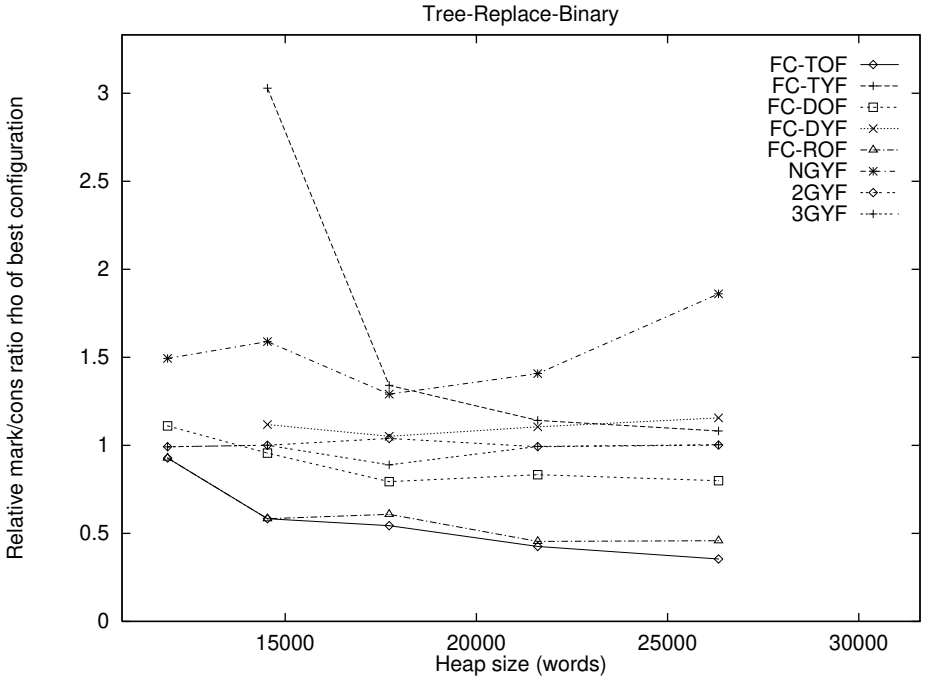


Figure 5.119. Best configuration comparison: Tree-Replace-Binary.

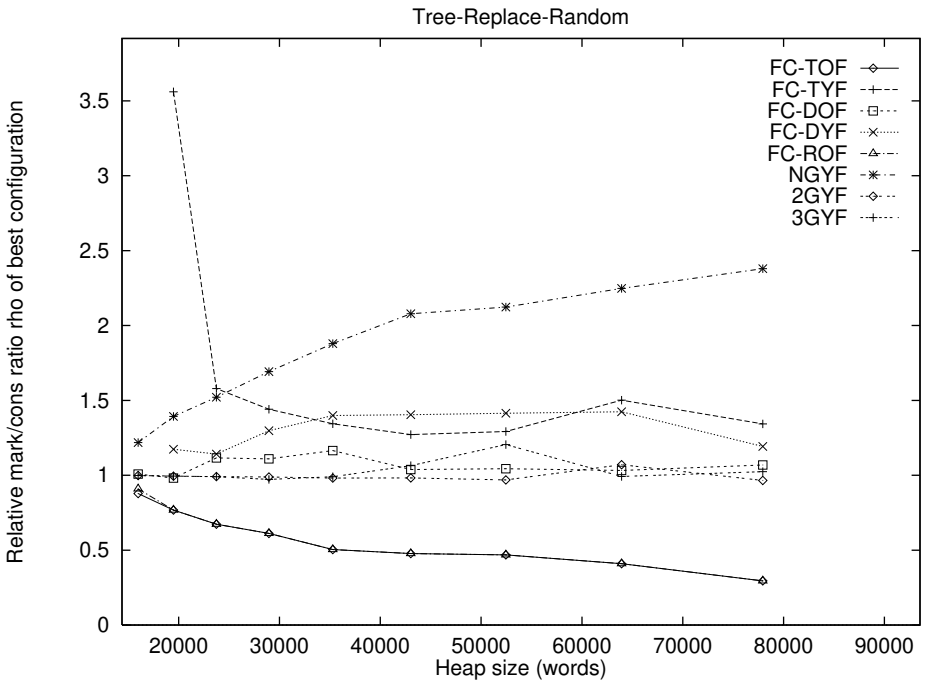


Figure 5.120. Best configuration comparison: Tree-Replace-Random.

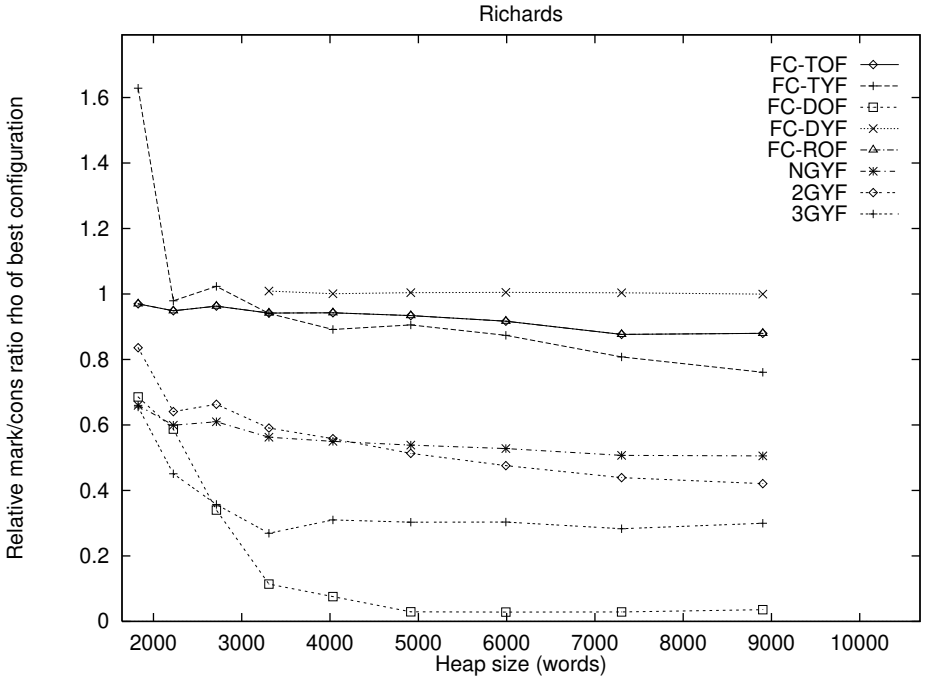


Figure 5.121. Best configuration comparison: Richards.

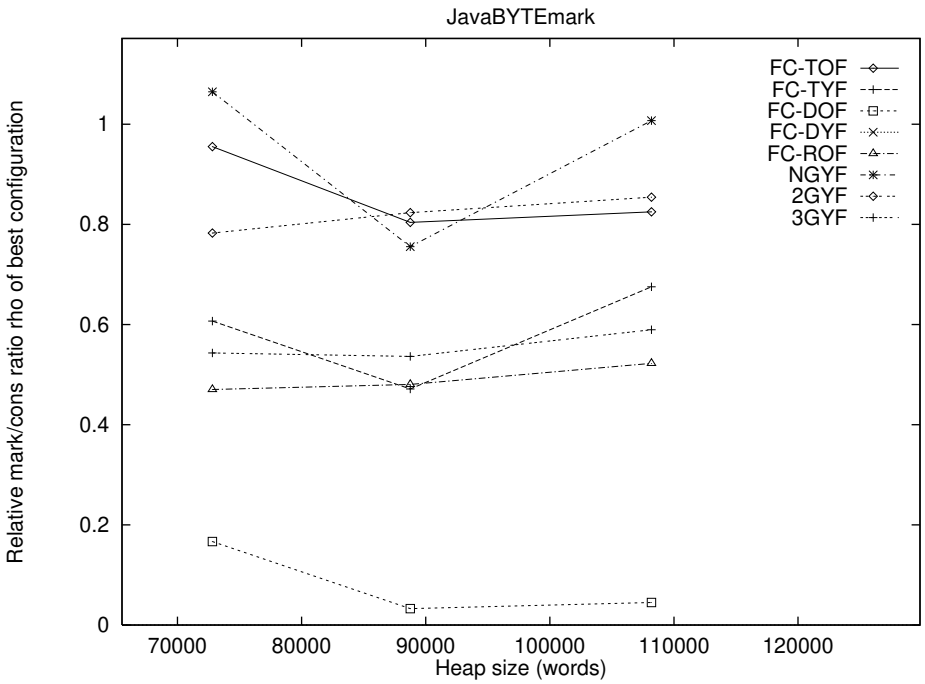


Figure 5.122. Best configuration comparison: JavaBYTEmark.

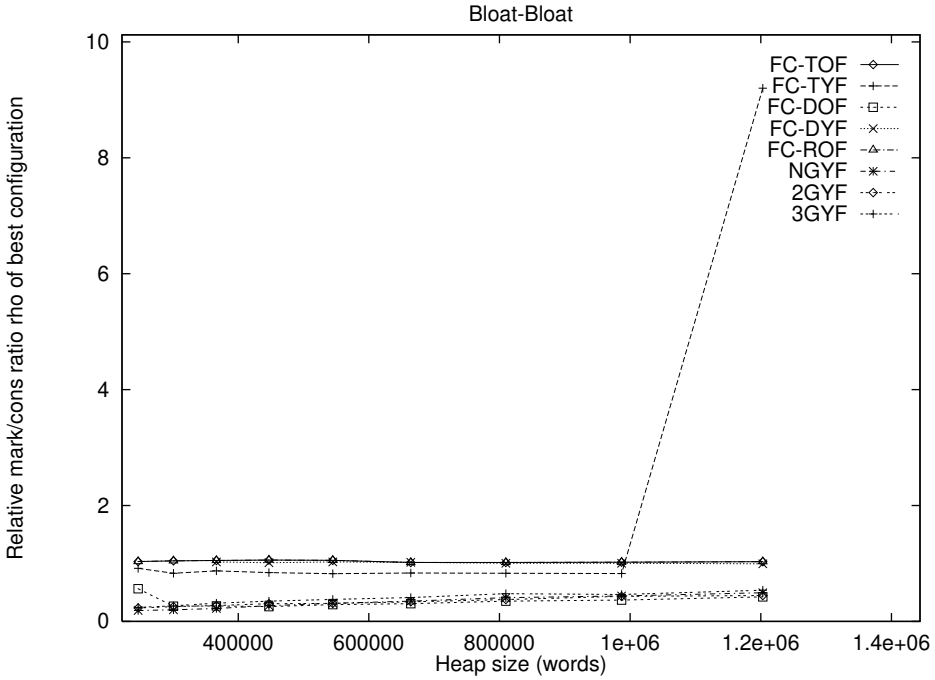


Figure 5.123. Best configuration comparison: Bloat-Bloat.

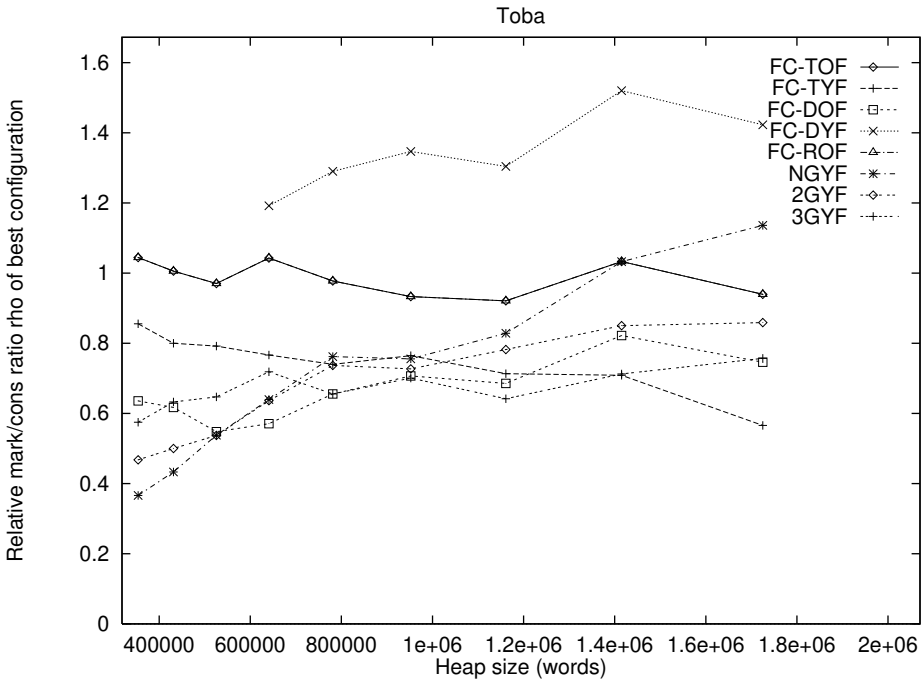


Figure 5.124. Best configuration comparison: Toba.

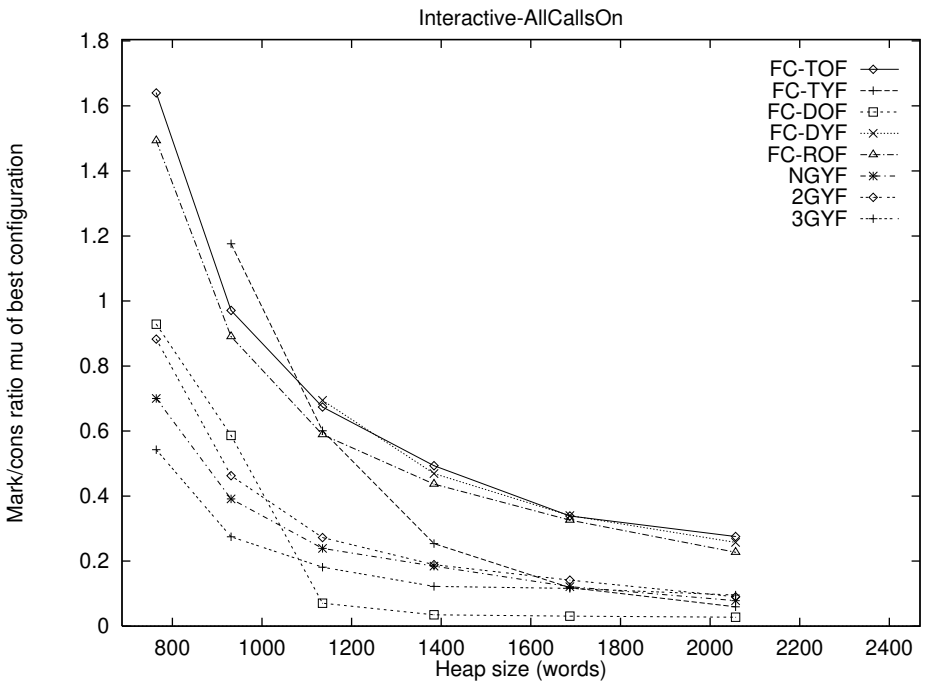


Figure 5.125. Best configuration comparison: Interactive-AllCallsOn.

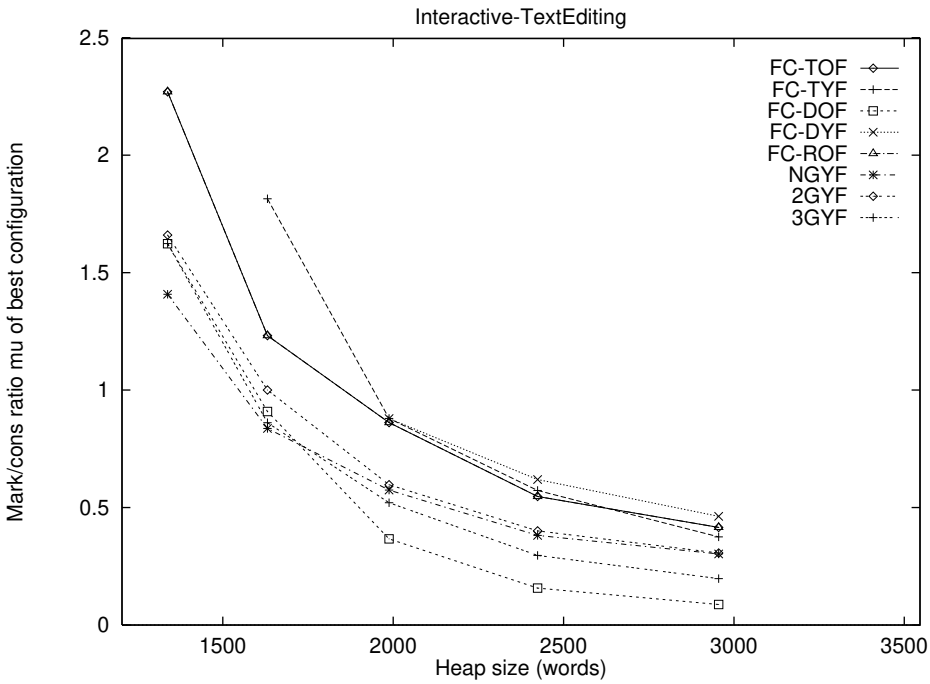


Figure 5.126. Best configuration comparison: Interactive-TextEditing.

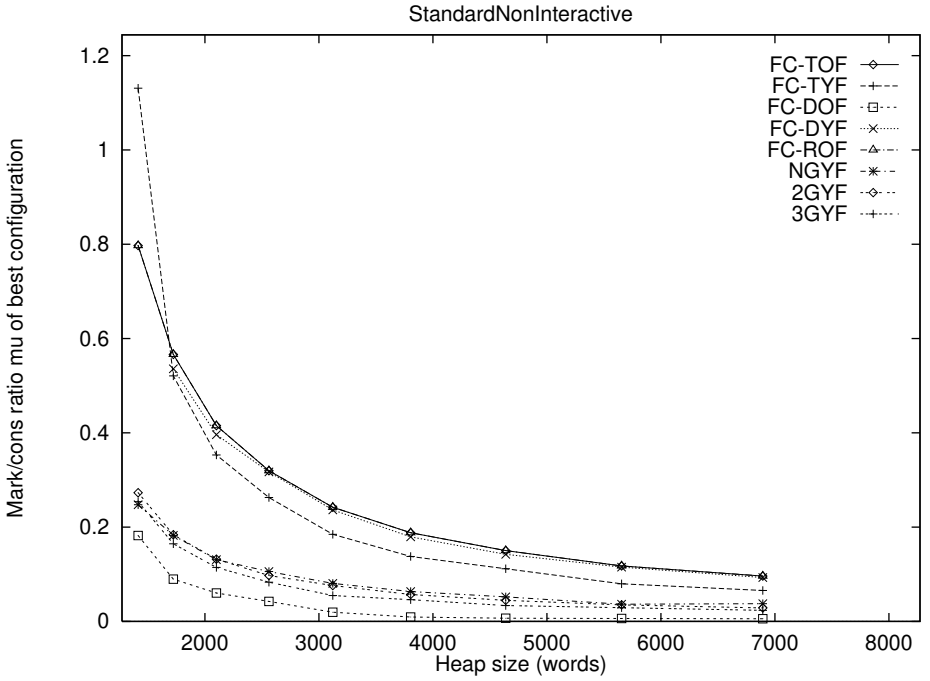


Figure 5.127. Best configuration comparison: StandardNonInteractive.

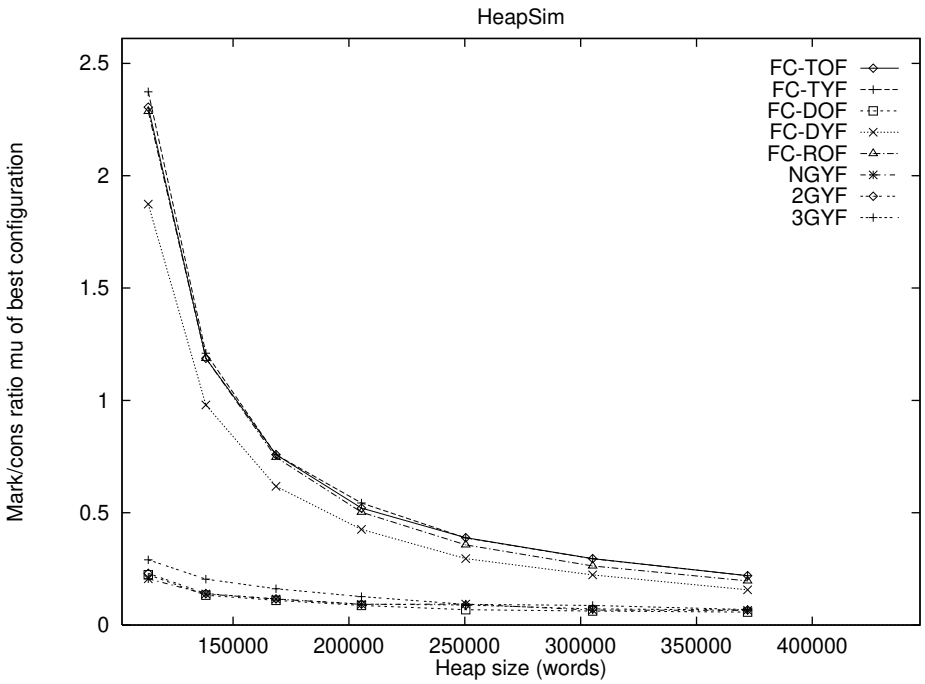


Figure 5.128. Best configuration comparison: HeapSim.

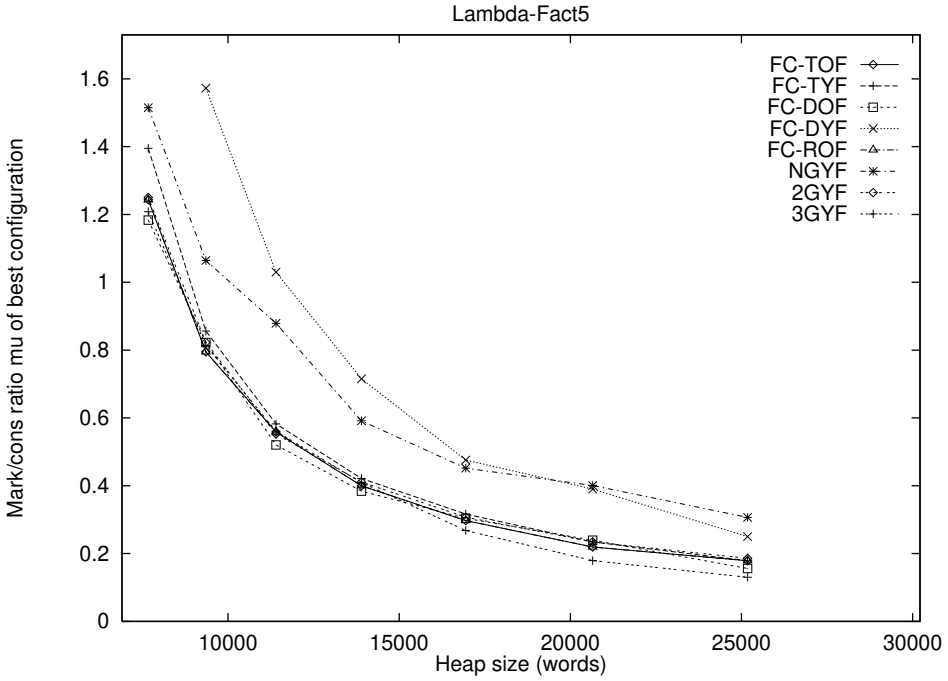


Figure 5.129. Best configuration comparison: Lambda-Fact5.

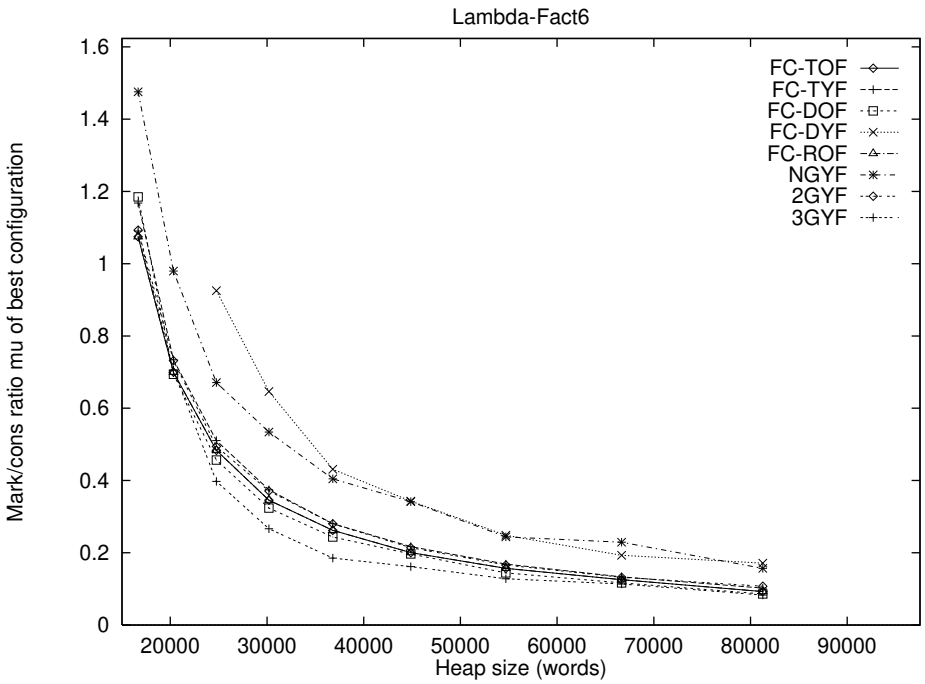


Figure 5.130. Best configuration comparison: Lambda-Fact6.

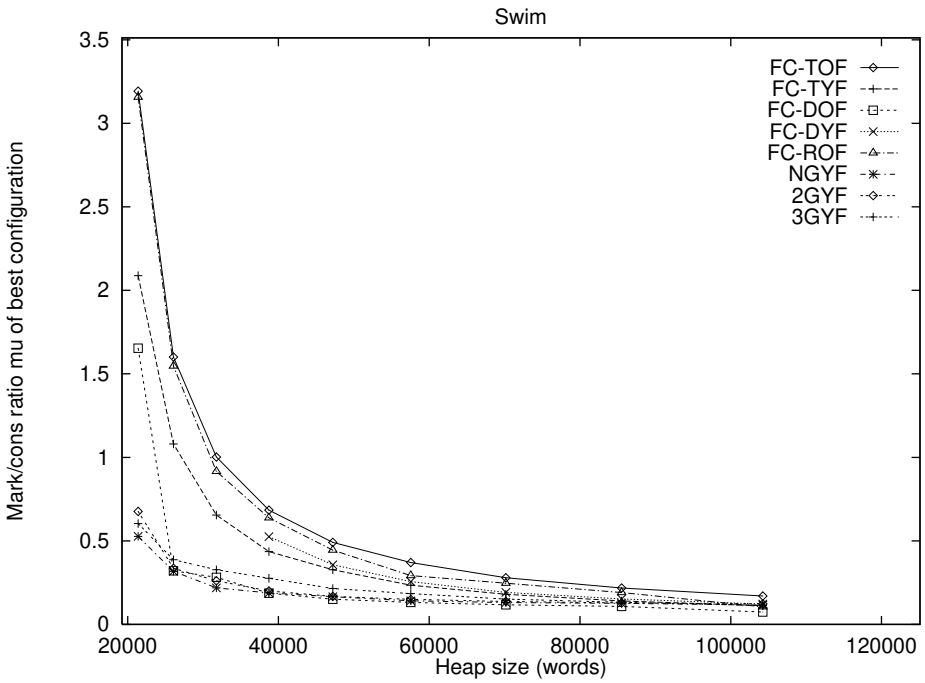


Figure 5.131. Best configuration comparison: Swim.

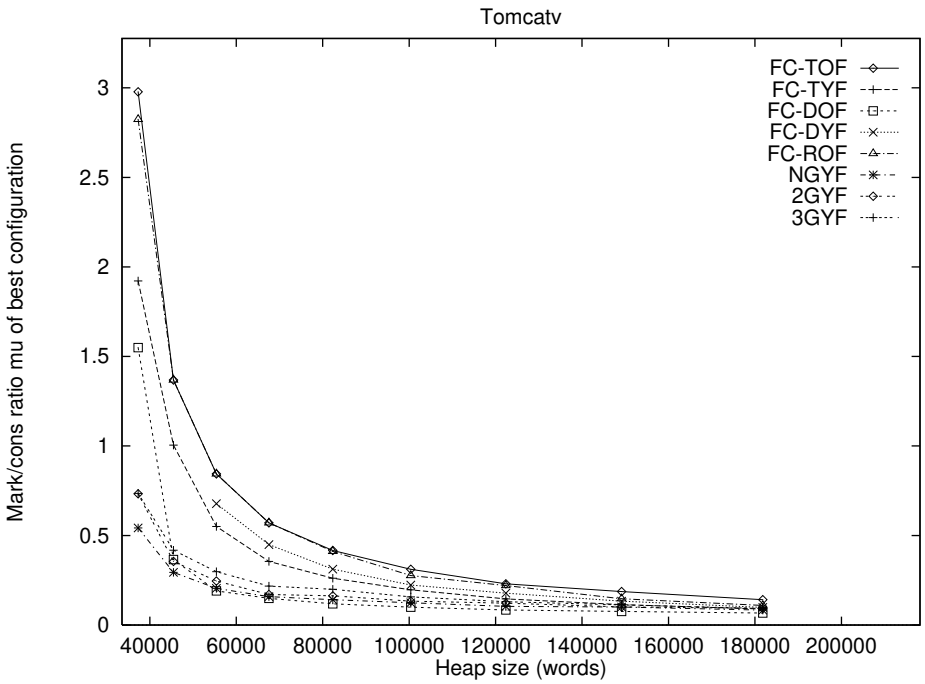


Figure 5.132. Best configuration comparison: Tomcatv.

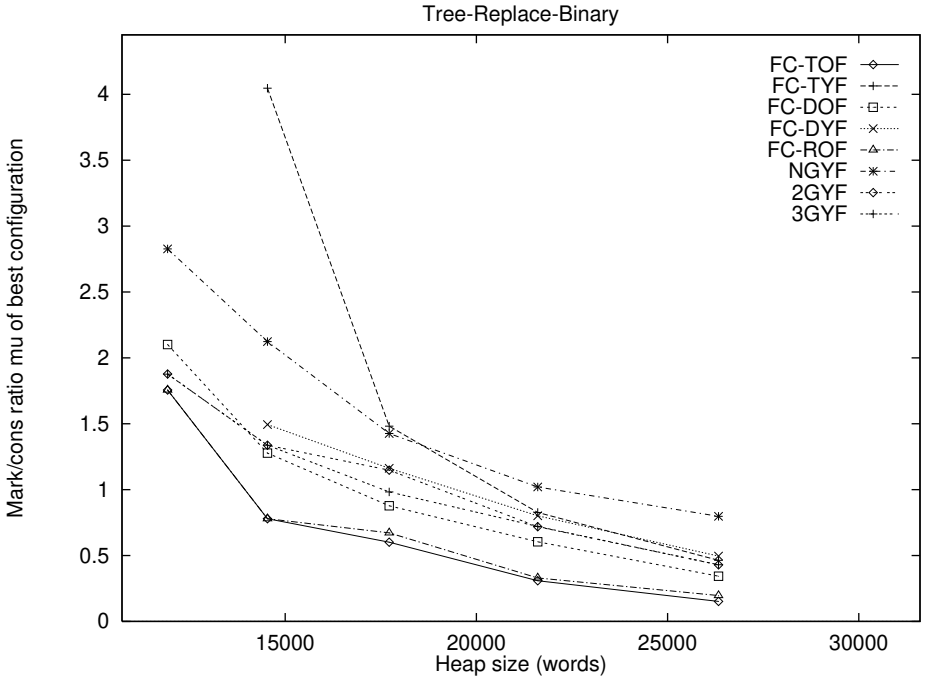


Figure 5.133. Best configuration comparison: Tree-Replace-Binary.

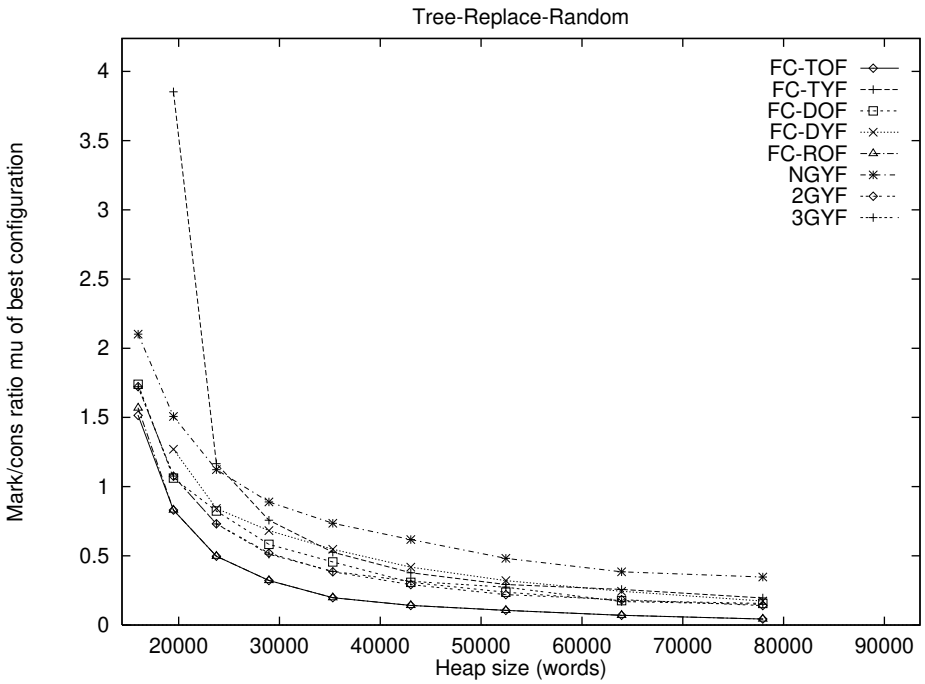


Figure 5.134. Best configuration comparison: Tree-Replace-Random.

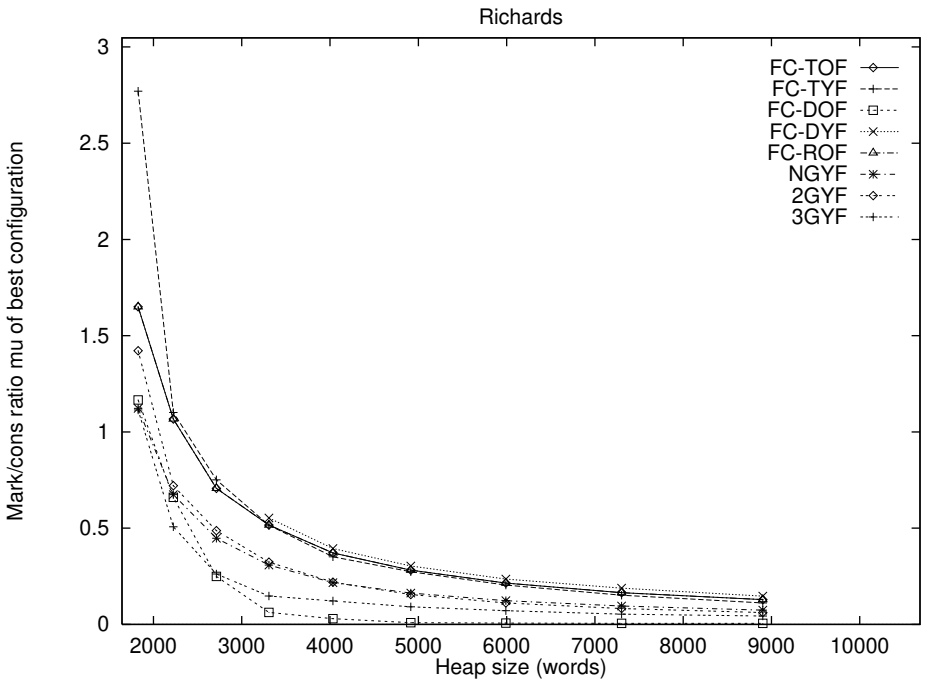


Figure 5.135. Best configuration comparison: Richards.

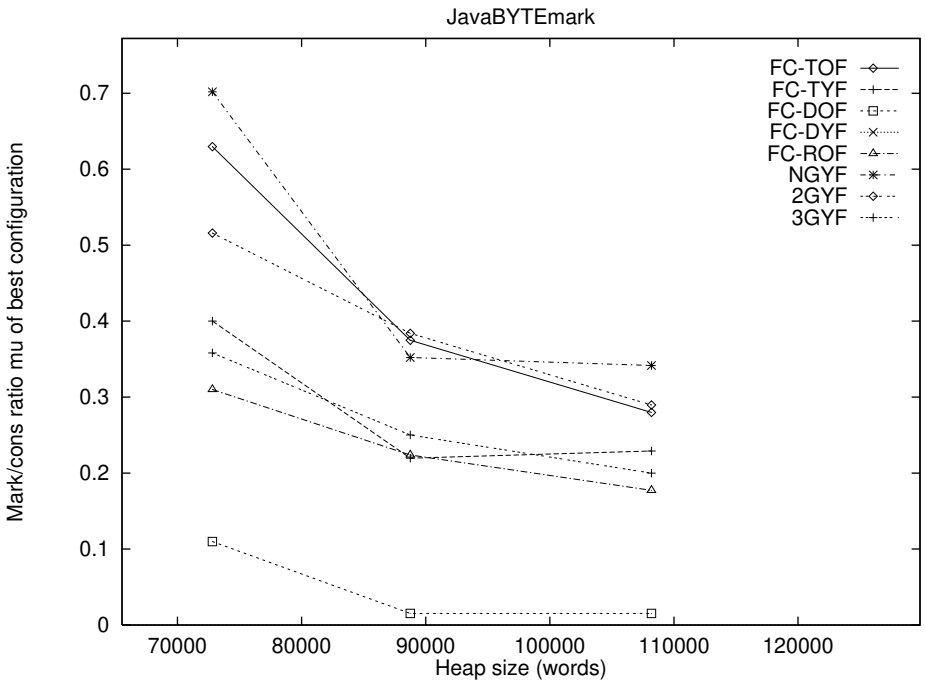


Figure 5.136. Best configuration comparison: JavaBYTEmark.

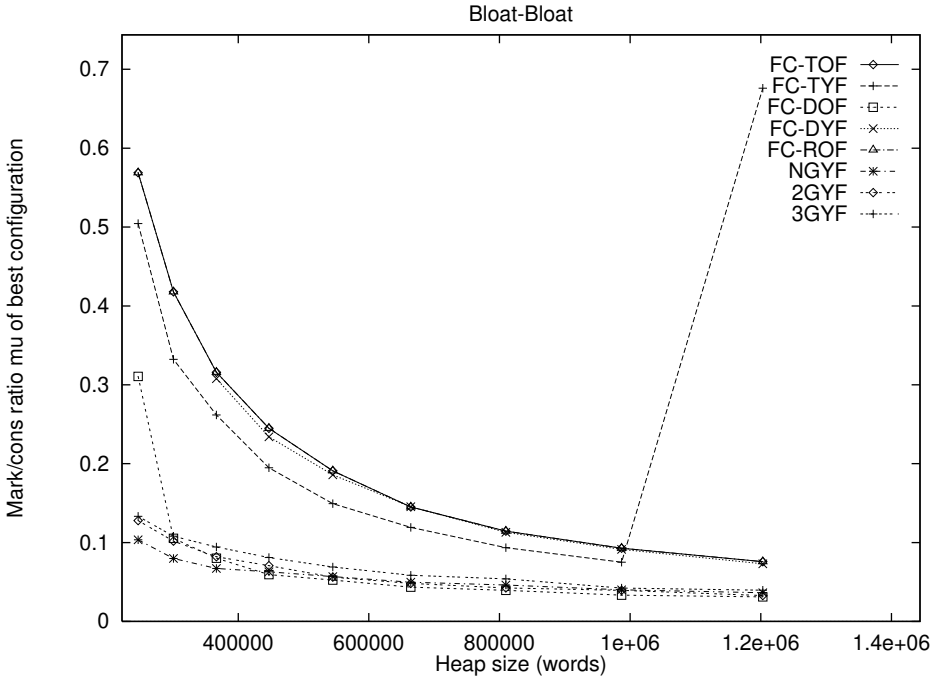


Figure 5.137. Best configuration comparison: Bloat-Bloat.

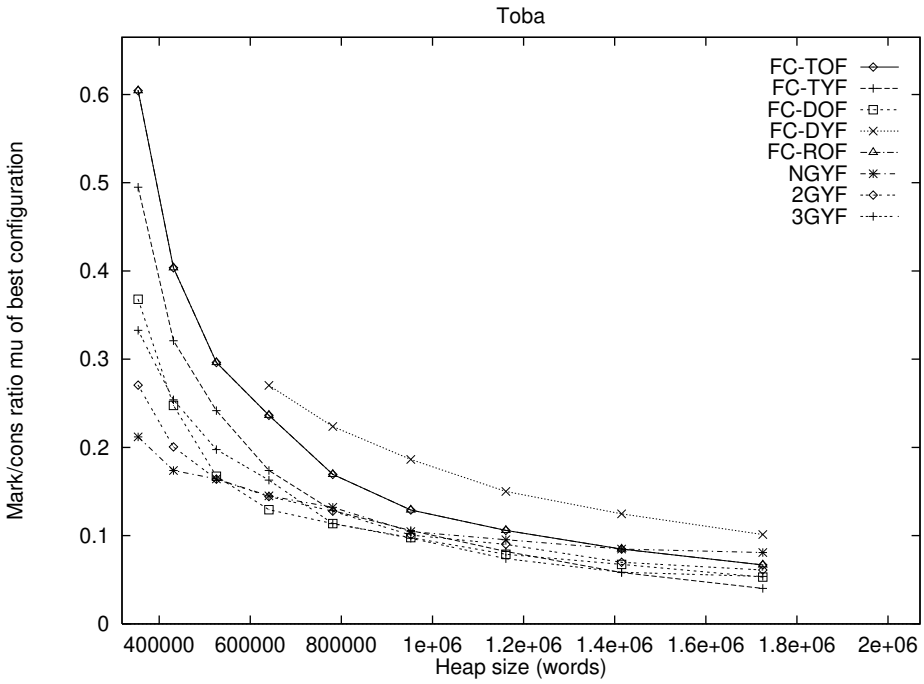
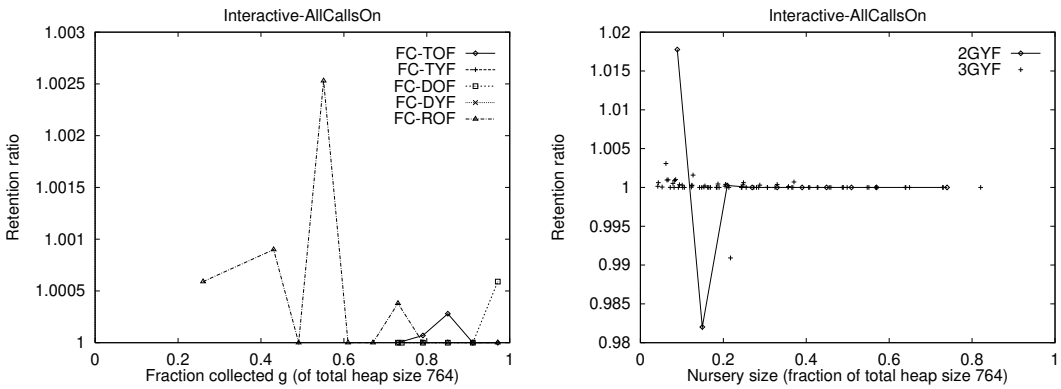
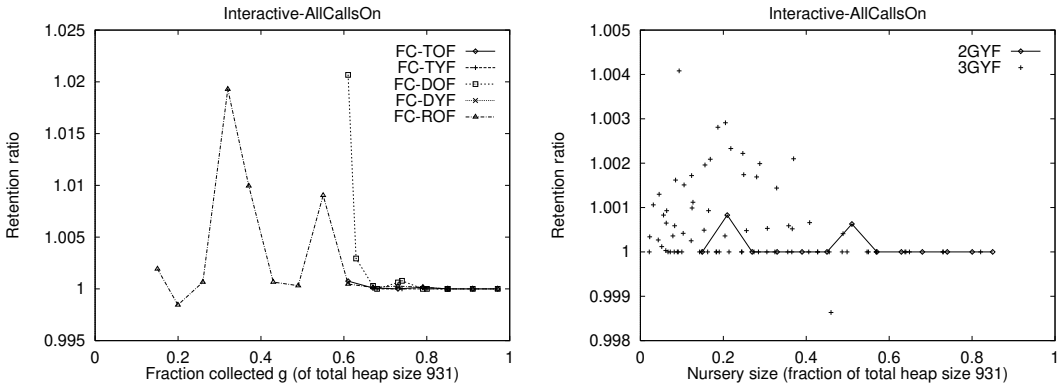


Figure 5.138. Best configuration comparison: Toba.



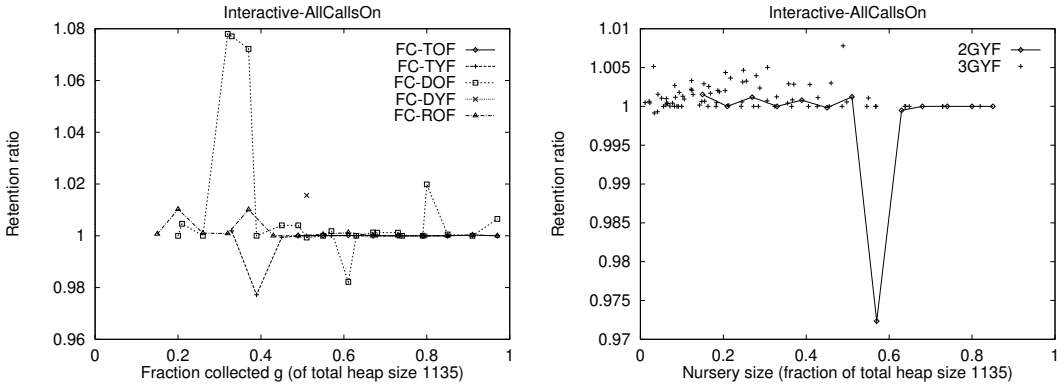
(a) FC schemes (b) GYF schemes

Figure 5.139. Excess retention ratios: Interactive-AllCallsOn, $V = 764$.



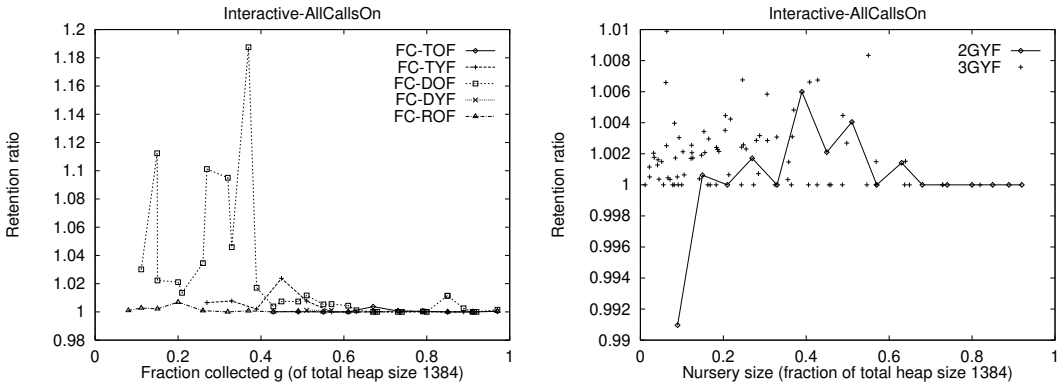
(a) FC schemes (b) GYF schemes

Figure 5.140. Excess retention ratios: Interactive-AllCallsOn, $V = 931$.



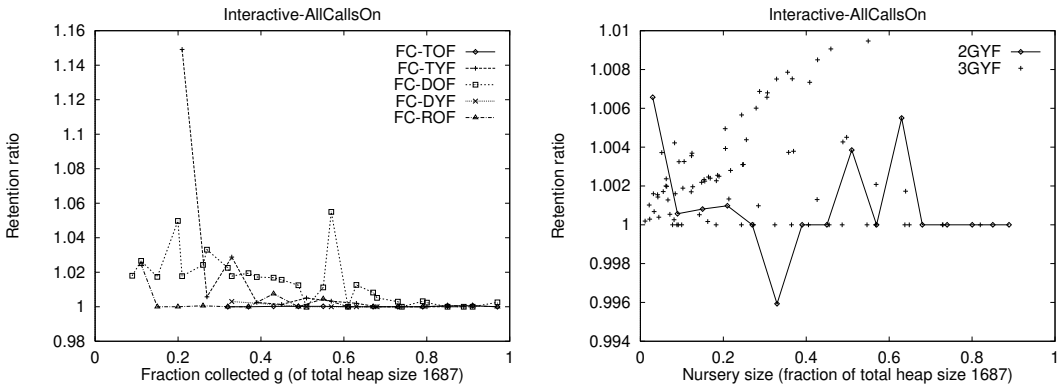
(a) FC schemes (b) GYF schemes

Figure 5.141. Excess retention ratios: Interactive-AllCallsOn, $V = 1135$.



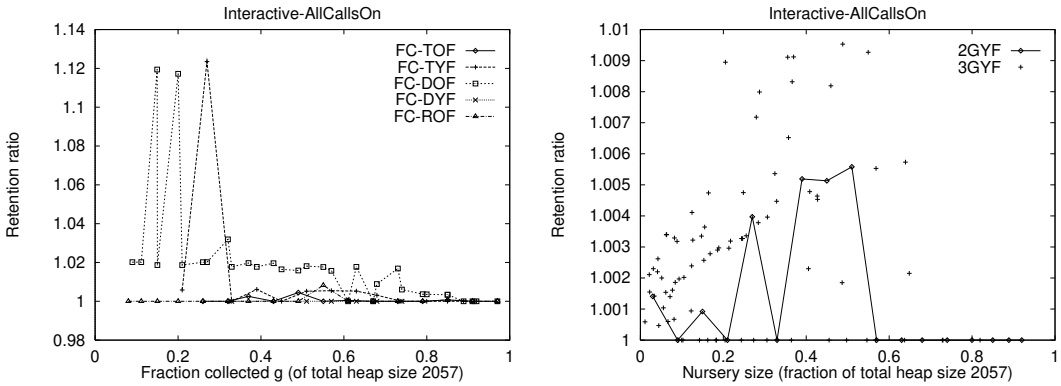
(a) FC schemes (b) GYF schemes

Figure 5.142. Excess retention ratios: Interactive-AllCallsOn, $V = 1384$.



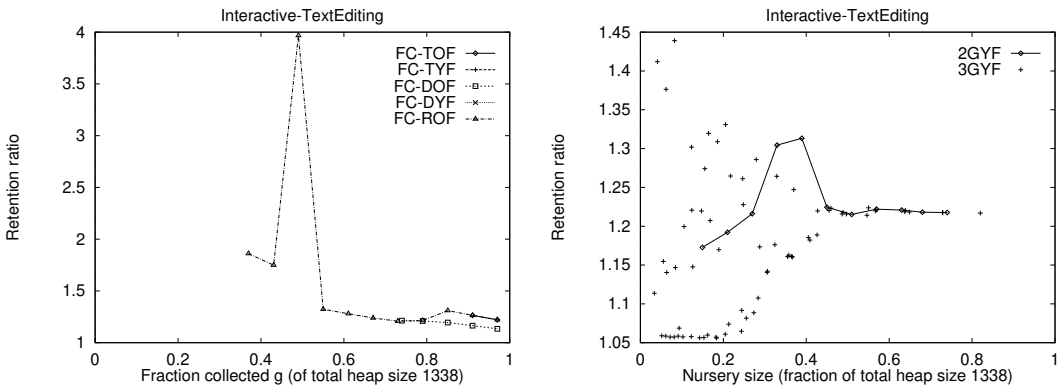
(a) FC schemes (b) GYF schemes

Figure 5.143. Excess retention ratios: Interactive-AllCallsOn, $V = 1687$.



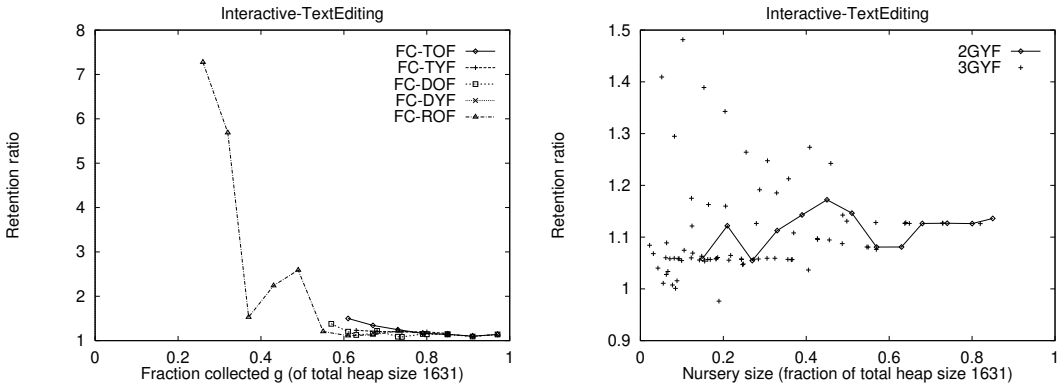
(a) FC schemes (b) GYF schemes

Figure 5.144. Excess retention ratios: Interactive-AllCallsOn, $V = 2057$.



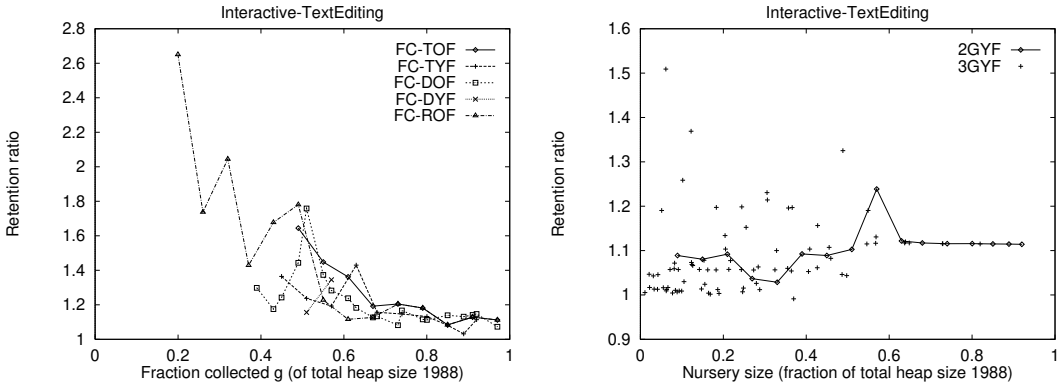
(a) FC schemes (b) GYF schemes

Figure 5.145. Excess retention ratios: Interactive-TextEditing, $V = 1338$.



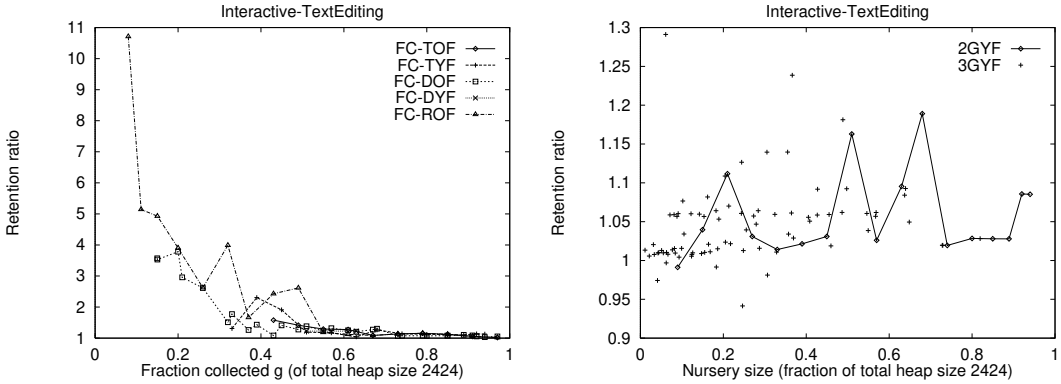
(a) FC schemes (b) GYF schemes

Figure 5.146. Excess retention ratios: Interactive-TextEditing, $V = 1631$.



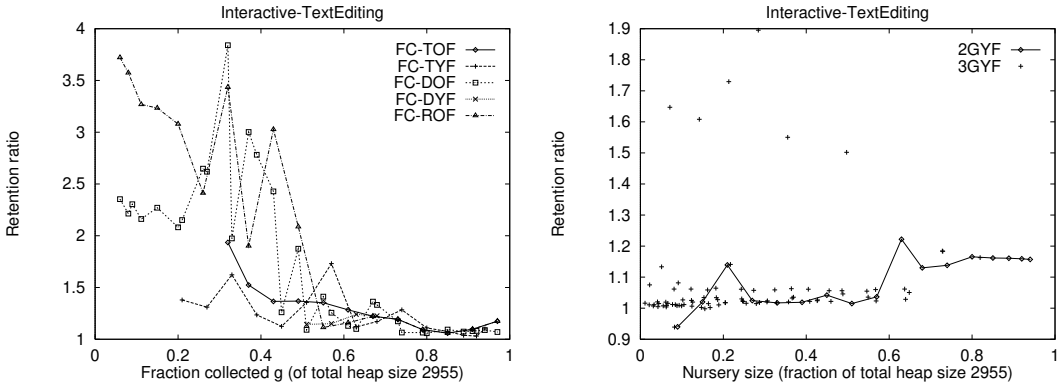
(a) FC schemes (b) GYF schemes

Figure 5.147. Excess retention ratios: Interactive-TextEditing, $V = 1988$.



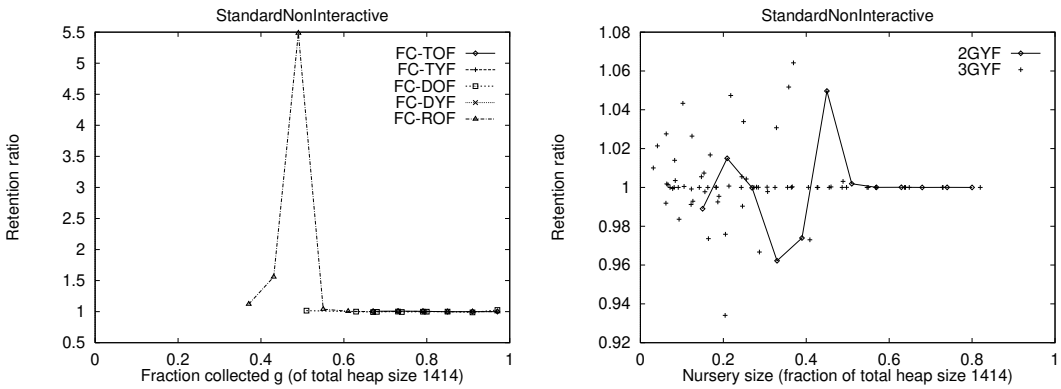
(a) FC schemes (b) GYF schemes

Figure 5.148. Excess retention ratios: Interactive-TextEditing, $V = 2424$.



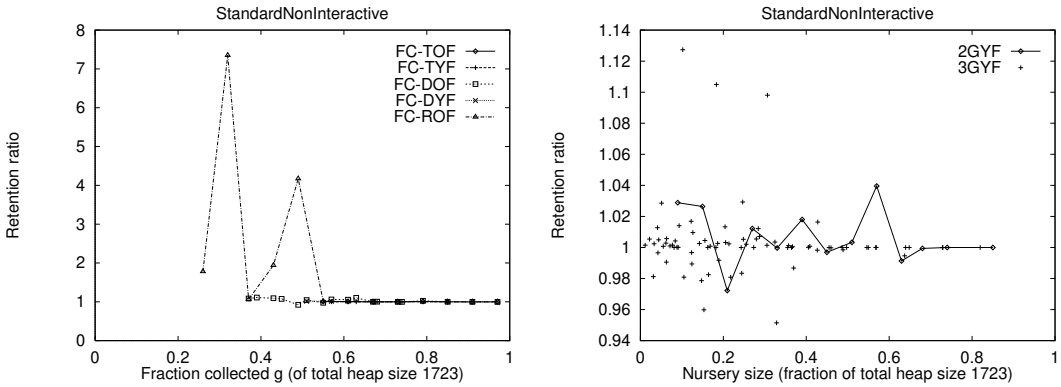
(a) FC schemes (b) GYF schemes

Figure 5.149. Excess retention ratios: Interactive-TextEditing, $V = 2955$.



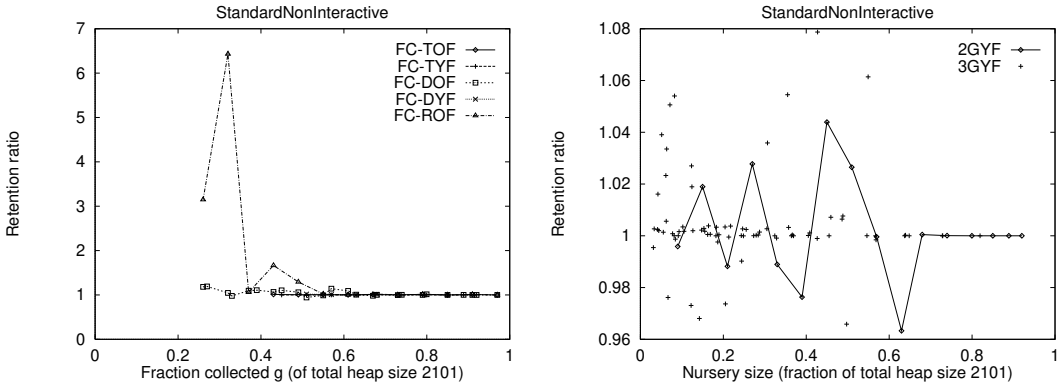
(a) FC schemes (b) GYF schemes

Figure 5.150. Excess retention ratios: StandardNonInteractive, $V = 1414$.



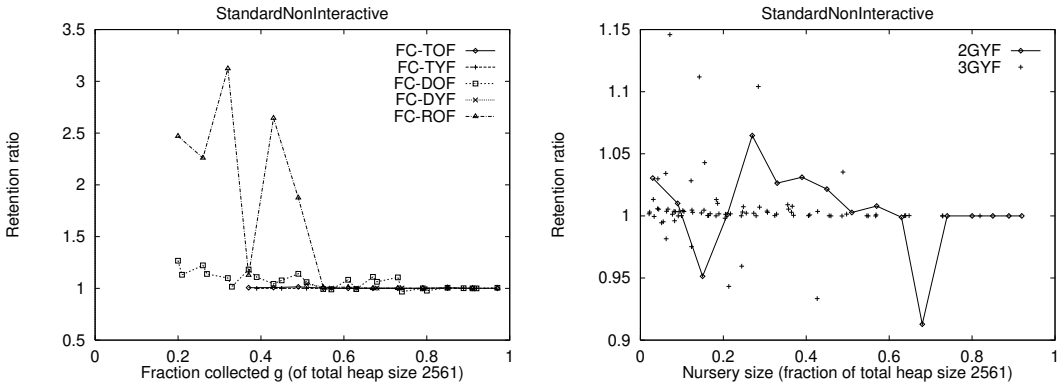
(a) FC schemes (b) GYF schemes

Figure 5.151. Excess retention ratios: StandardNonInteractive, $V = 1723$.



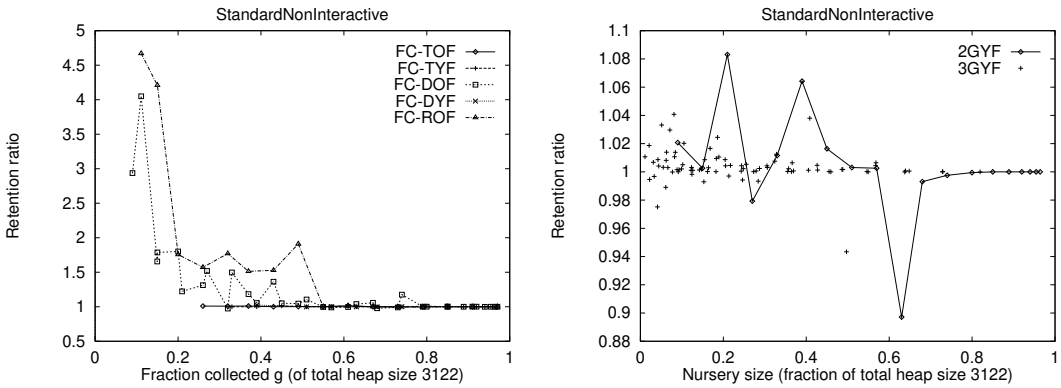
(a) FC schemes (b) GYF schemes

Figure 5.152. Excess retention ratios: StandardNonInteractive, $V = 2101$.



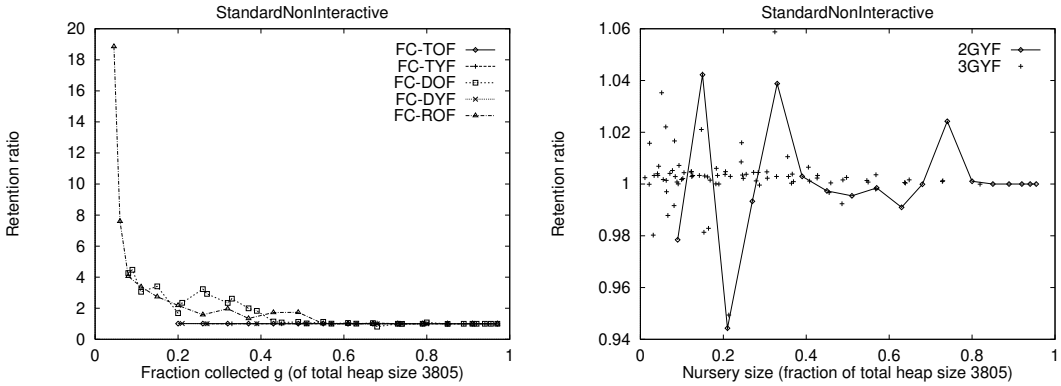
(a) FC schemes (b) GYF schemes

Figure 5.153. Excess retention ratios: StandardNonInteractive, $V = 2561$.



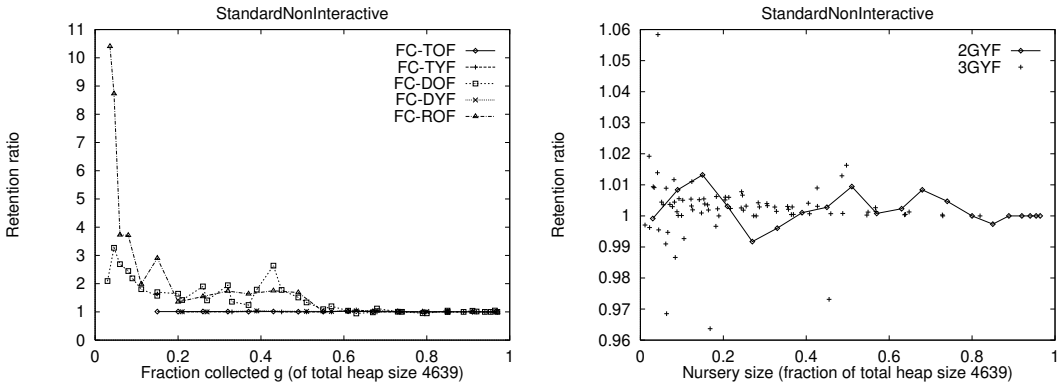
(a) FC schemes (b) GYF schemes

Figure 5.154. Excess retention ratios: StandardNonInteractive, $V = 3122$.



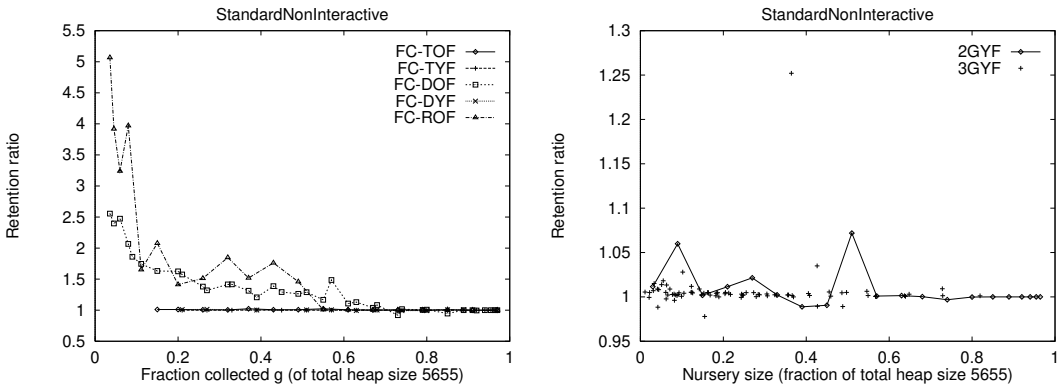
(a) FC schemes (b) GYF schemes

Figure 5.155. Excess retention ratios: StandardNonInteractive, $V = 3805$.



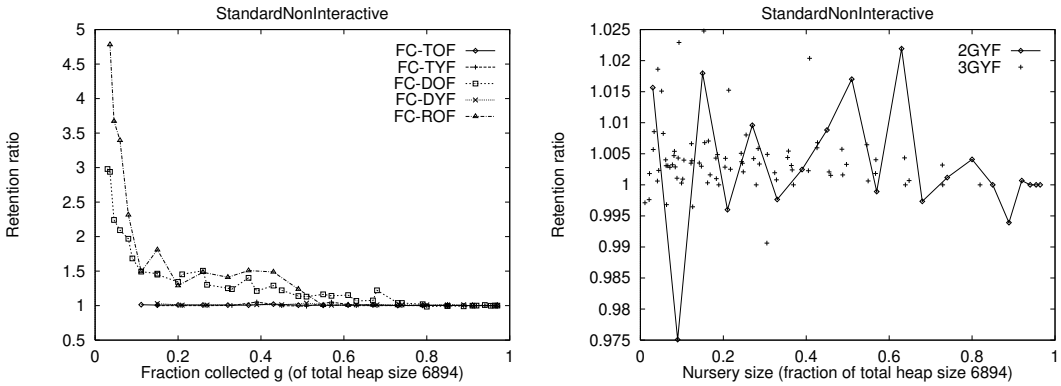
(a) FC schemes (b) GYF schemes

Figure 5.156. Excess retention ratios: StandardNonInteractive, $V = 4639$.



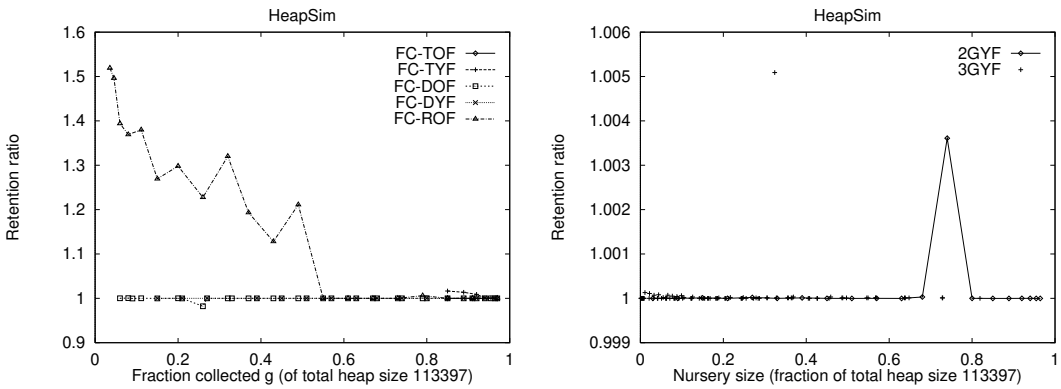
(a) FC schemes (b) GYF schemes

Figure 5.157. Excess retention ratios: StandardNonInteractive, $V = 5655$.



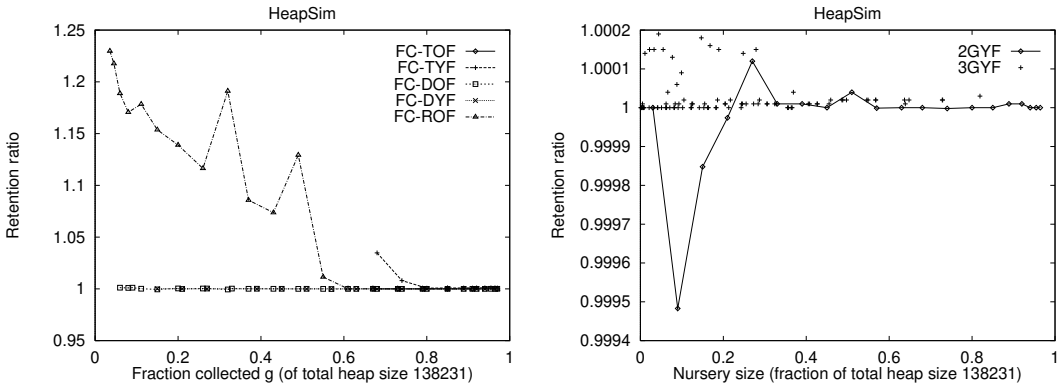
(a) FC schemes (b) GYF schemes

Figure 5.158. Excess retention ratios: StandardNonInteractive, $V = 6894$.



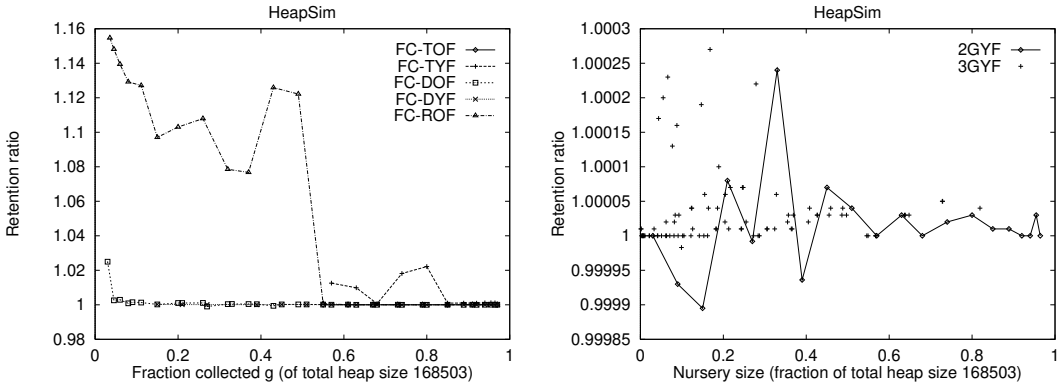
(a) FC schemes (b) GYF schemes

Figure 5.159. Excess retention ratios: HeapSim, $V = 113397$.



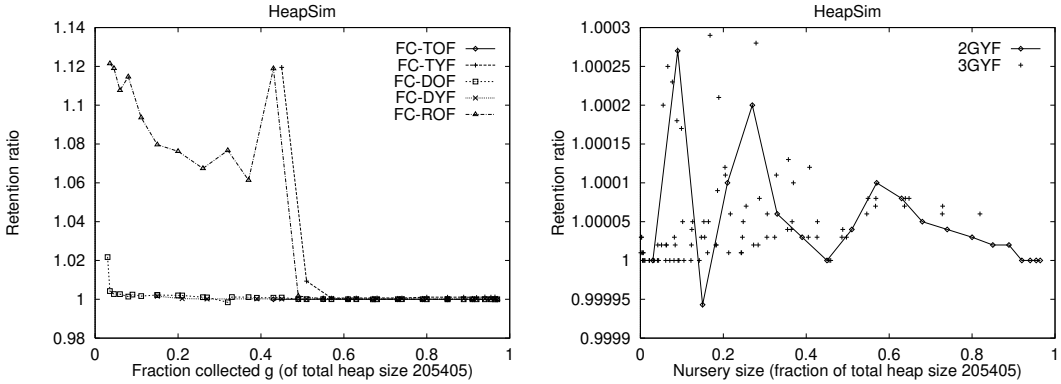
(a) FC schemes (b) GYF schemes

Figure 5.160. Excess retention ratios: HeapSim, $V = 138231$.



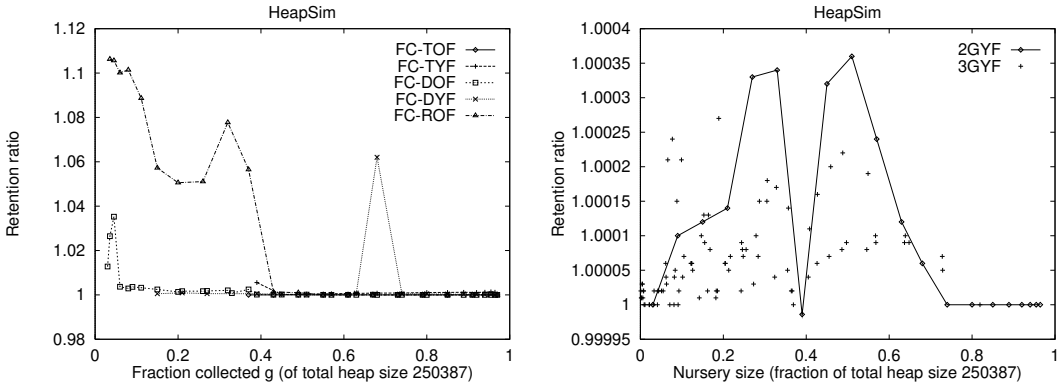
(a) FC schemes (b) GYF schemes

Figure 5.161. Excess retention ratios: HeapSim, $V = 168503$.



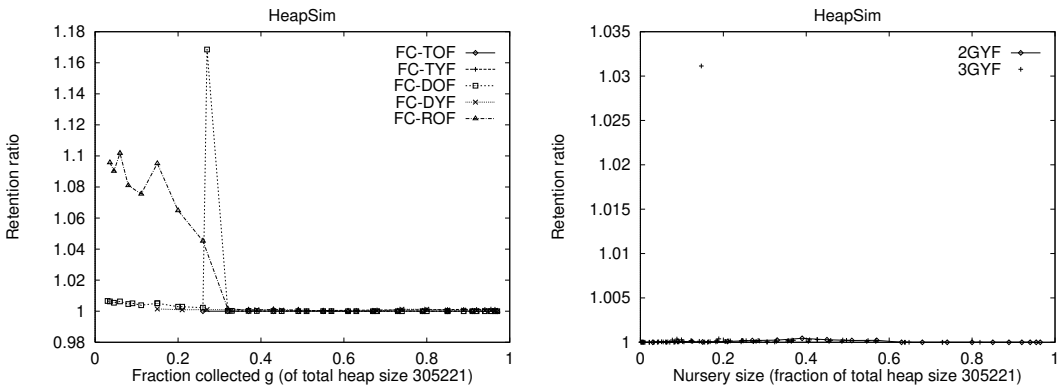
(a) FC schemes (b) GYF schemes

Figure 5.162. Excess retention ratios: HeapSim, $V = 205405$.



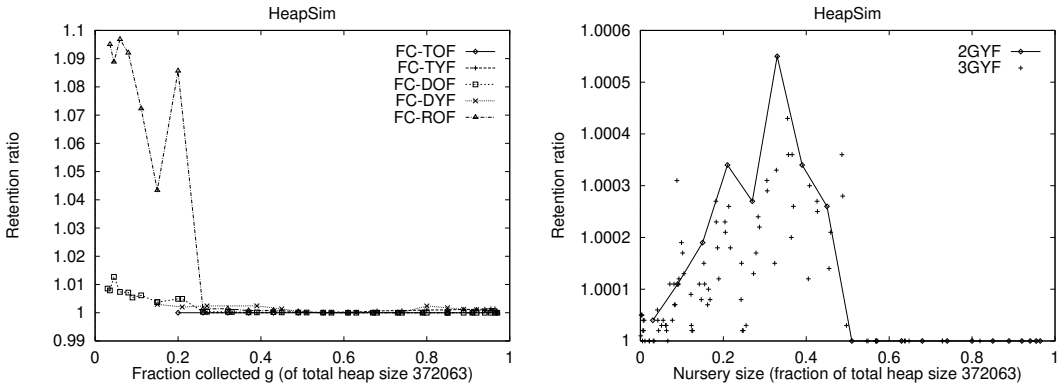
(a) FC schemes (b) GYF schemes

Figure 5.163. Excess retention ratios: HeapSim, $V = 250387$.



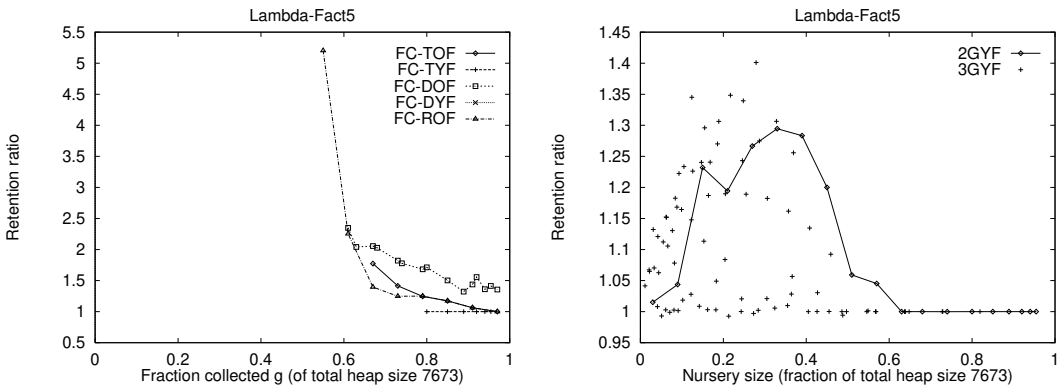
(a) FC schemes (b) GYF schemes

Figure 5.164. Excess retention ratios: HeapSim, $V = 305221$.



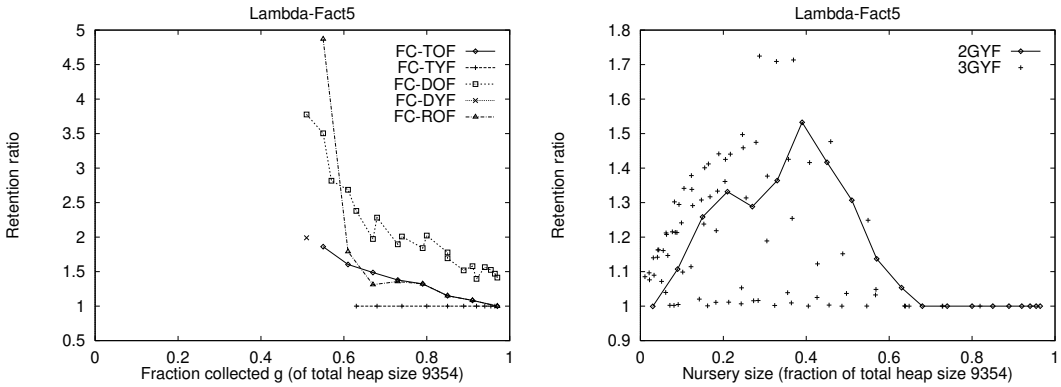
(a) FC schemes (b) GYF schemes

Figure 5.165. Excess retention ratios: HeapSim, $V = 372063$.



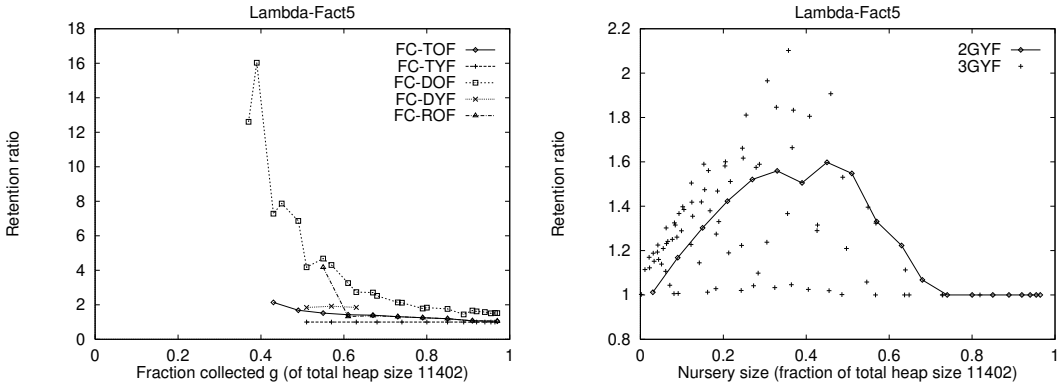
(a) FC schemes (b) GYF schemes

Figure 5.166. Excess retention ratios: Lambda-Fact5, $V = 7673$.



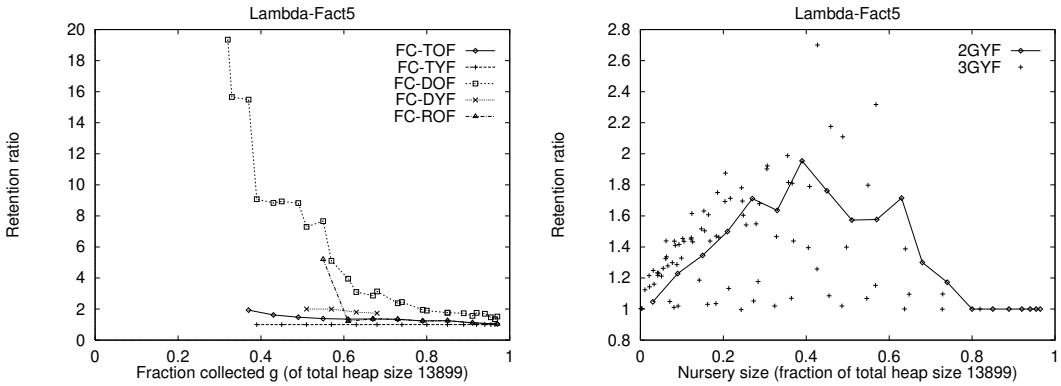
(a) FC schemes (b) GYF schemes

Figure 5.167. Excess retention ratios: Lambda-Fact5, $V = 9354$.



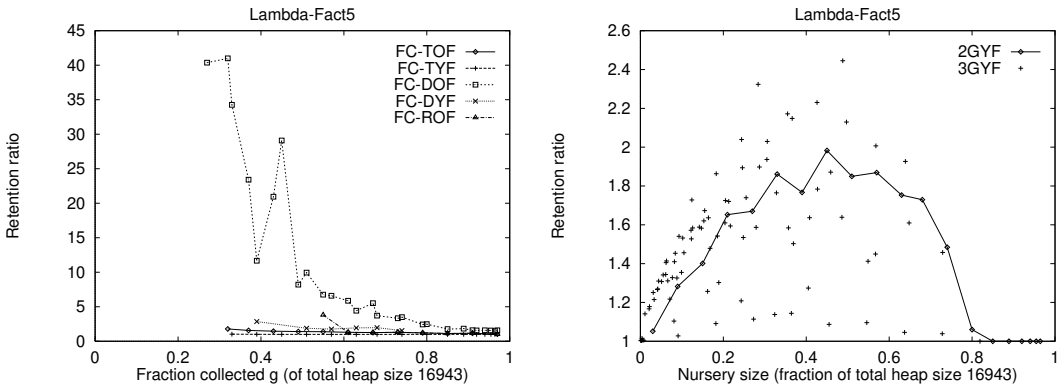
(a) FC schemes (b) GYF schemes

Figure 5.168. Excess retention ratios: Lambda-Fact5, $V = 11402$.



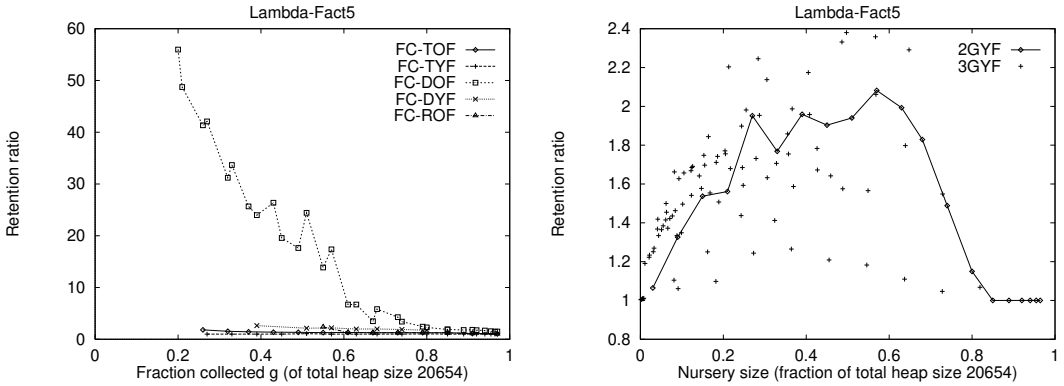
(a) FC schemes (b) GYF schemes

Figure 5.169. Excess retention ratios: Lambda-Fact5, $V = 13899$.



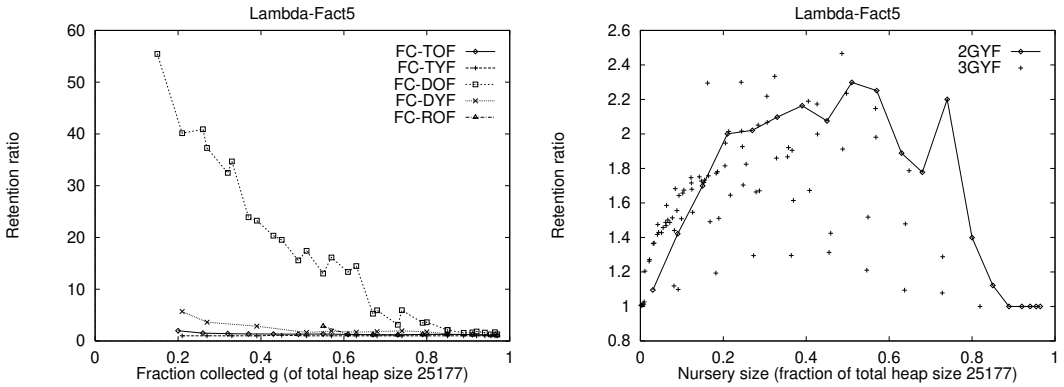
(a) FC schemes (b) GYF schemes

Figure 5.170. Excess retention ratios: Lambda-Fact5, $V = 16943$.



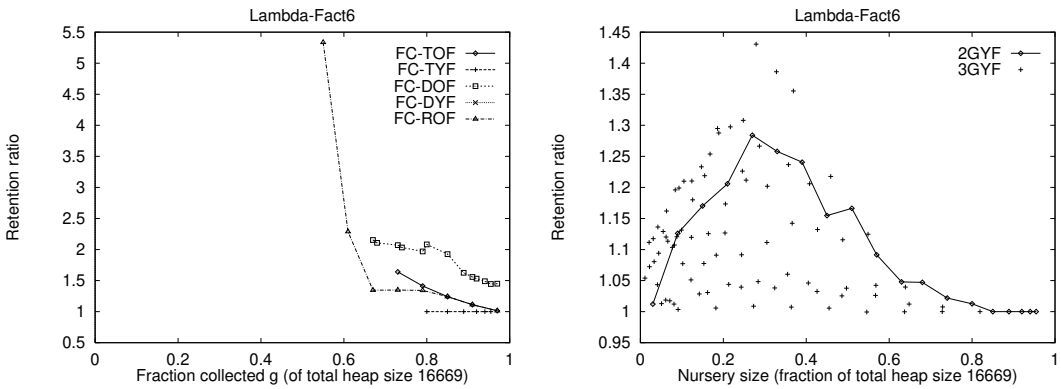
(a) FC schemes (b) GYF schemes

Figure 5.171. Excess retention ratios: Lambda-Fact5, $V = 20654$.



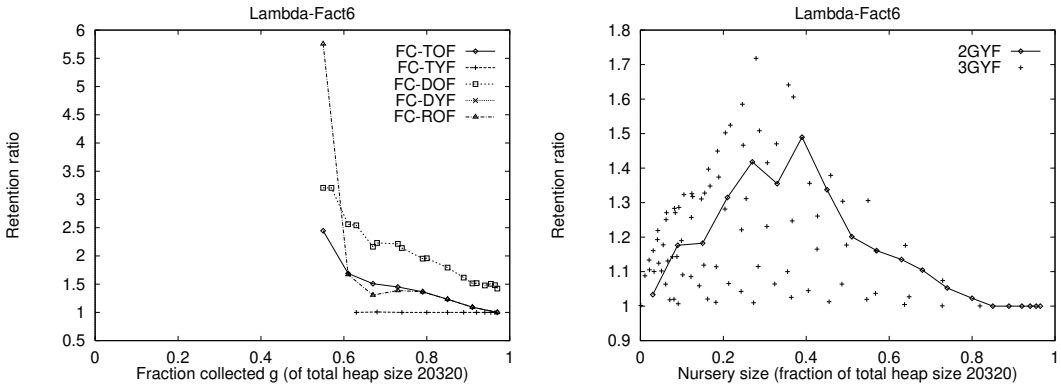
(a) FC schemes (b) GYF schemes

Figure 5.172. Excess retention ratios: Lambda-Fact5, $V = 25177$.



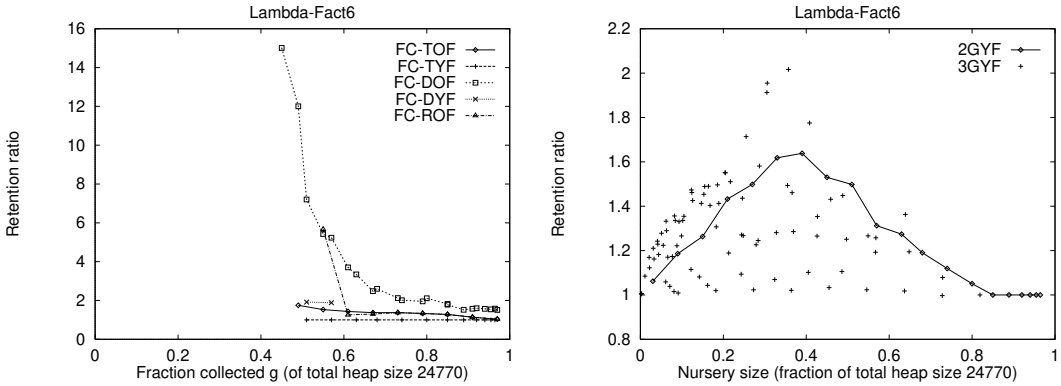
(a) FC schemes (b) GYF schemes

Figure 5.173. Excess retention ratios: Lambda-Fact6, $V = 16669$.



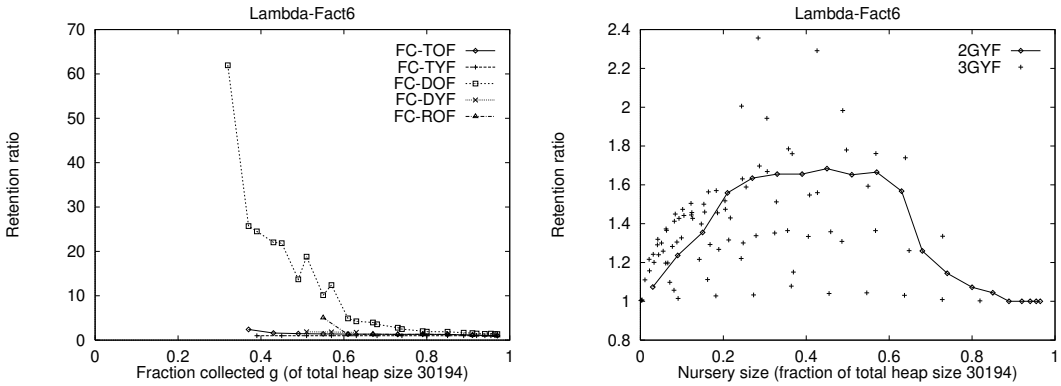
(a) FC schemes (b) GYF schemes

Figure 5.174. Excess retention ratios: Lambda-Fact6, $V = 20320$.



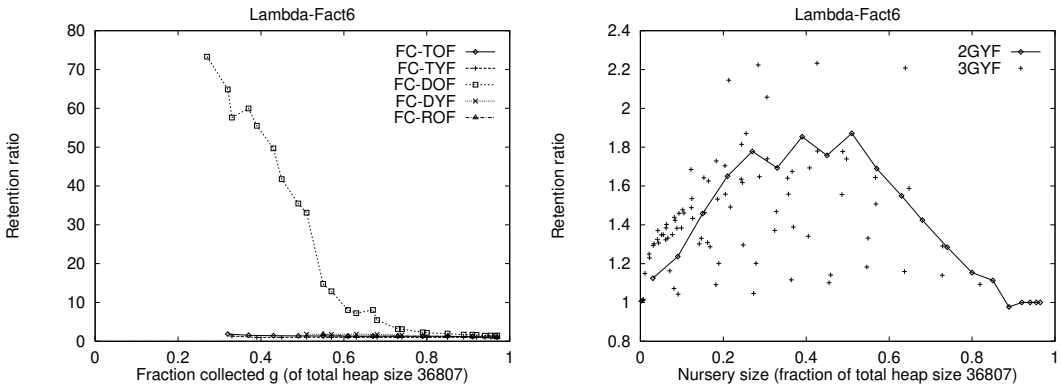
(a) FC schemes (b) GYF schemes

Figure 5.175. Excess retention ratios: Lambda-Fact6, $V = 24770$.



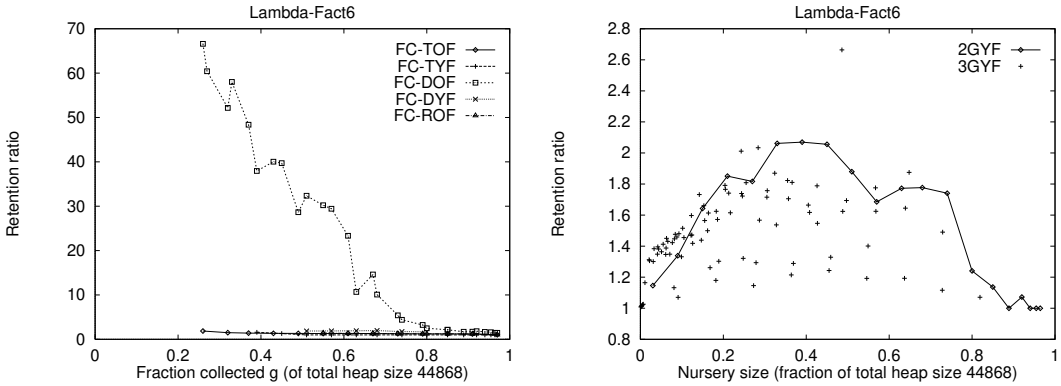
(a) FC schemes (b) GYF schemes

Figure 5.176. Excess retention ratios: Lambda-Fact6, $V = 30194$.



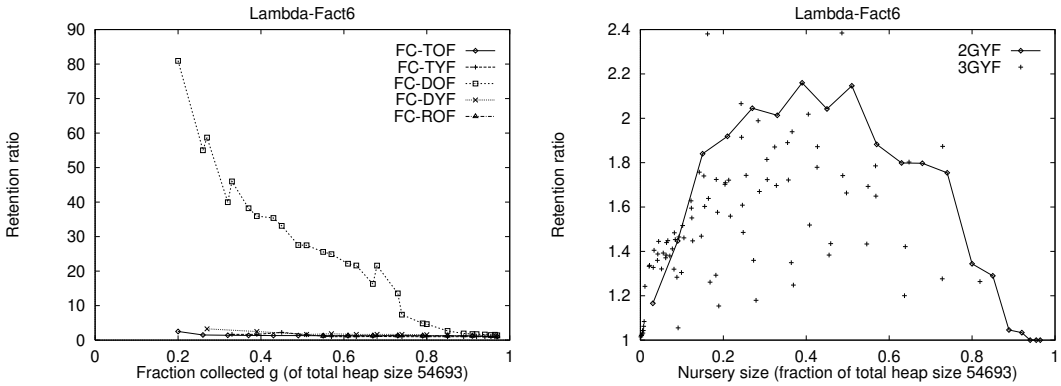
(a) FC schemes (b) GYF schemes

Figure 5.177. Excess retention ratios: Lambda-Fact6, $V = 36807$.



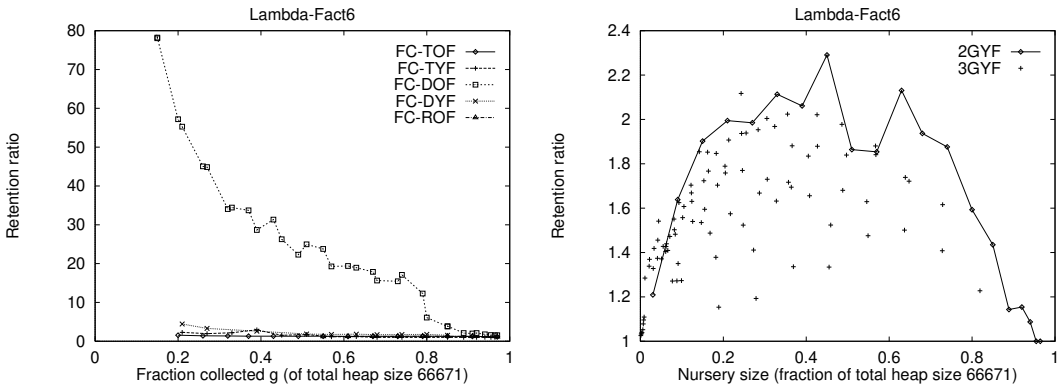
(a) FC schemes (b) GYF schemes

Figure 5.178. Excess retention ratios: Lambda-Fact6, $V = 44868$.



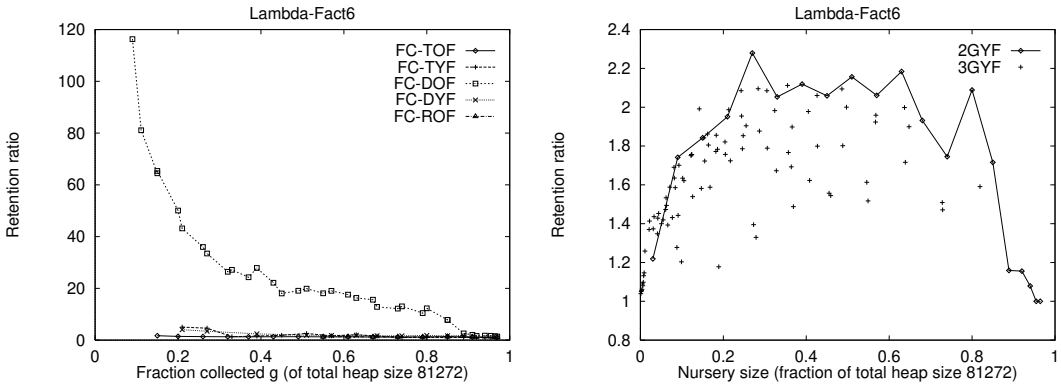
(a) FC schemes (b) GYF schemes

Figure 5.179. Excess retention ratios: Lambda-Fact6, $V = 54693$.



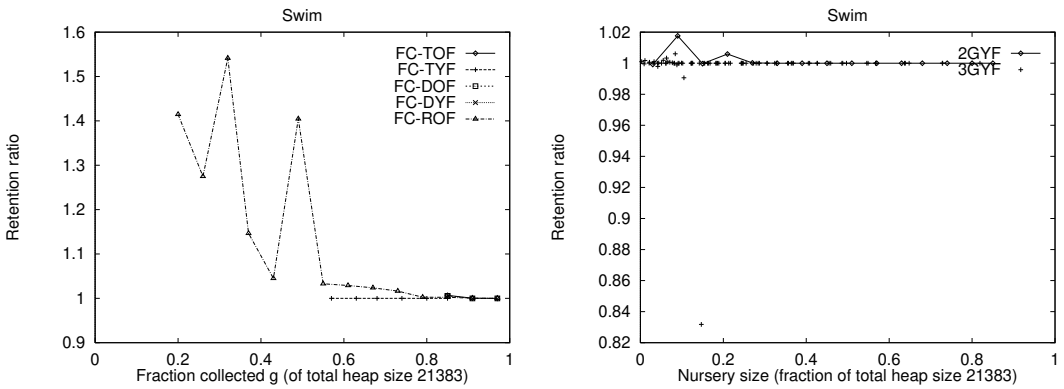
(a) FC schemes (b) GYF schemes

Figure 5.180. Excess retention ratios: Lambda-Fact6, $V = 66671$.



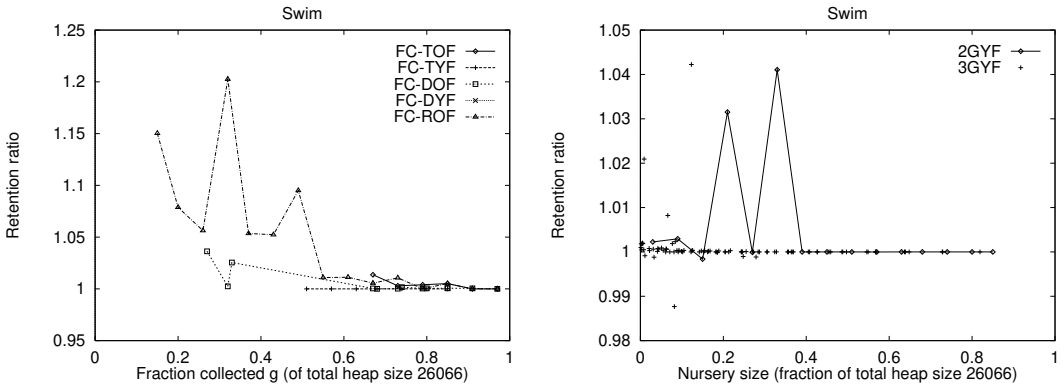
(a) FC schemes (b) GYF schemes

Figure 5.181. Excess retention ratios: Lambda-Fact6, $V = 81272$.



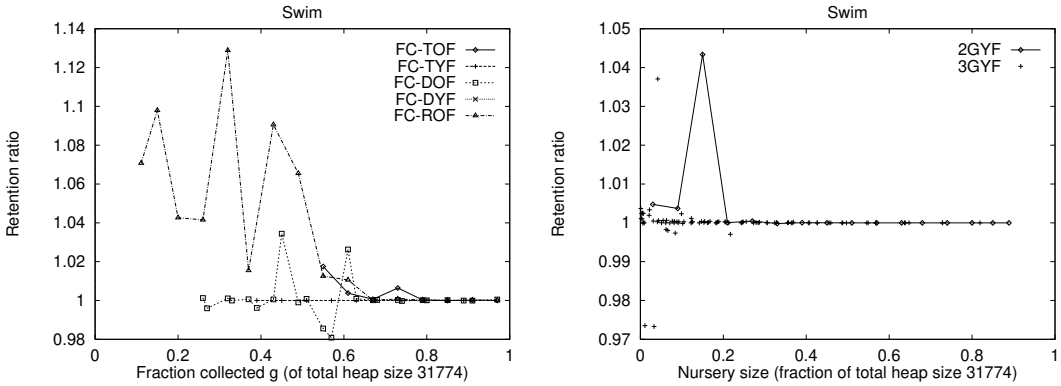
(a) FC schemes (b) GYF schemes

Figure 5.182. Excess retention ratios: Swim, $V = 21383$.



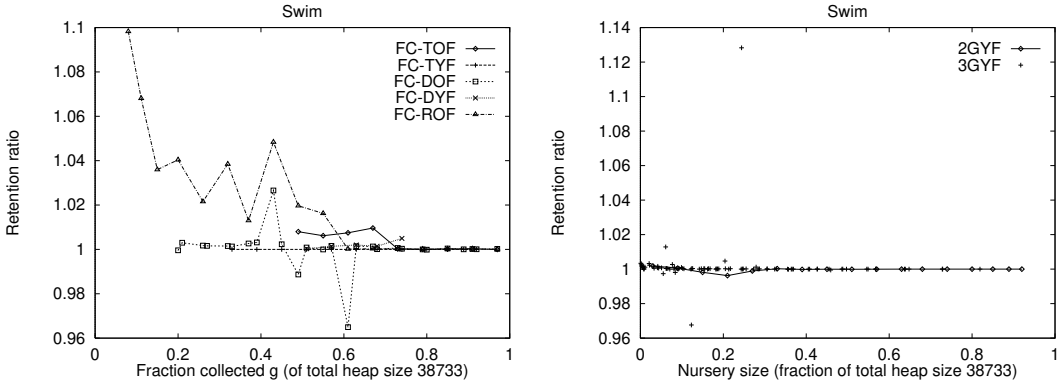
(a) FC schemes (b) GYF schemes

Figure 5.183. Excess retention ratios: Swim, $V = 26066$.



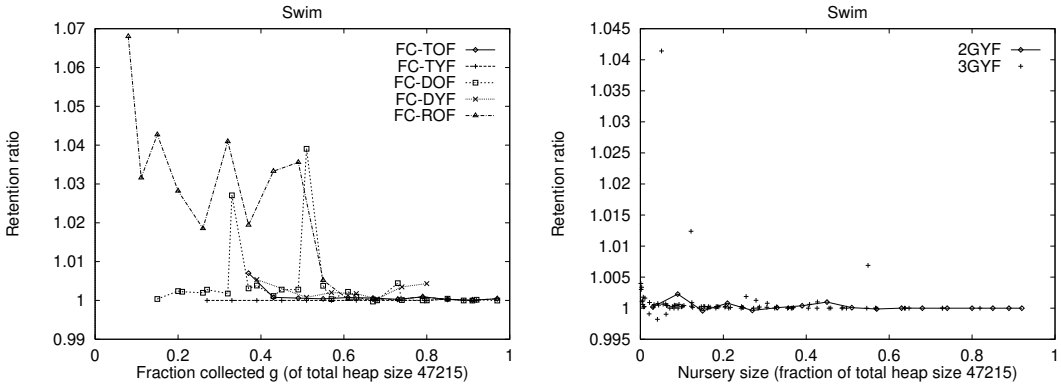
(a) FC schemes (b) GYF schemes

Figure 5.184. Excess retention ratios: Swim, $V = 31774$.



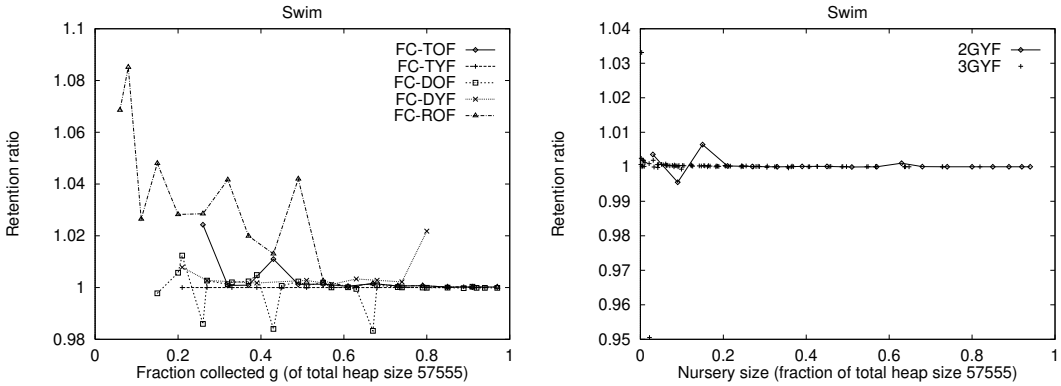
(a) FC schemes (b) GYF schemes

Figure 5.185. Excess retention ratios: Swim, $V = 38733$.



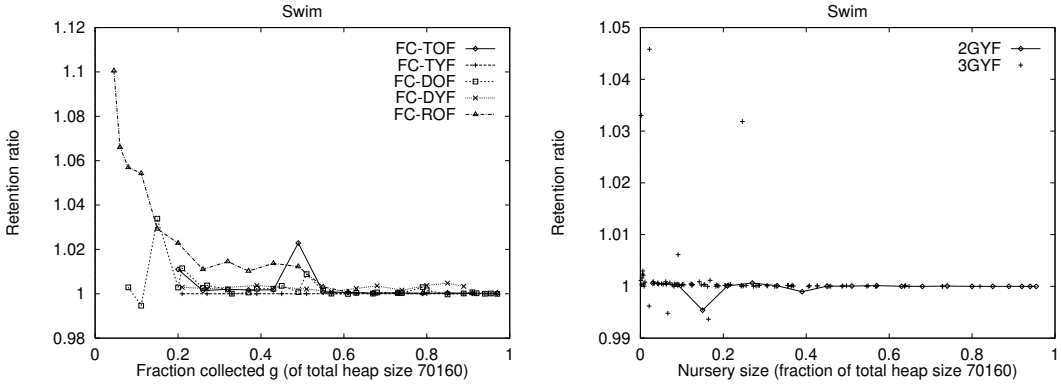
(a) FC schemes (b) GYF schemes

Figure 5.186. Excess retention ratios: Swim, $V = 47215$.



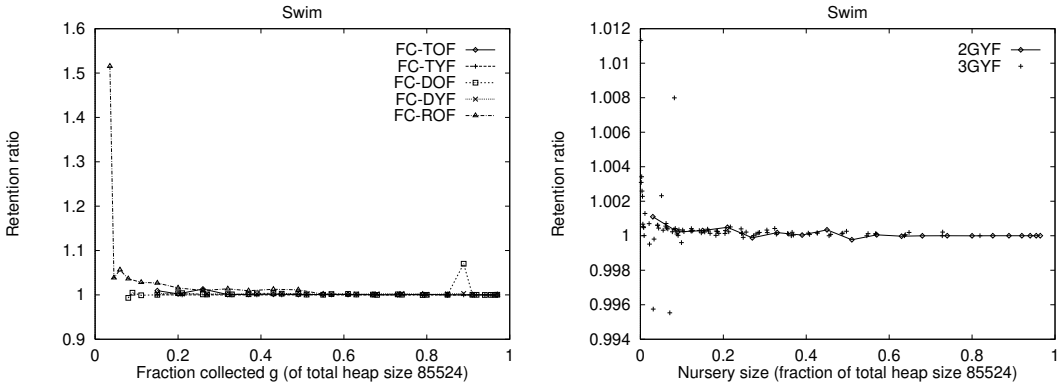
(a) FC schemes (b) GYF schemes

Figure 5.187. Excess retention ratios: Swim, $V = 57555$.



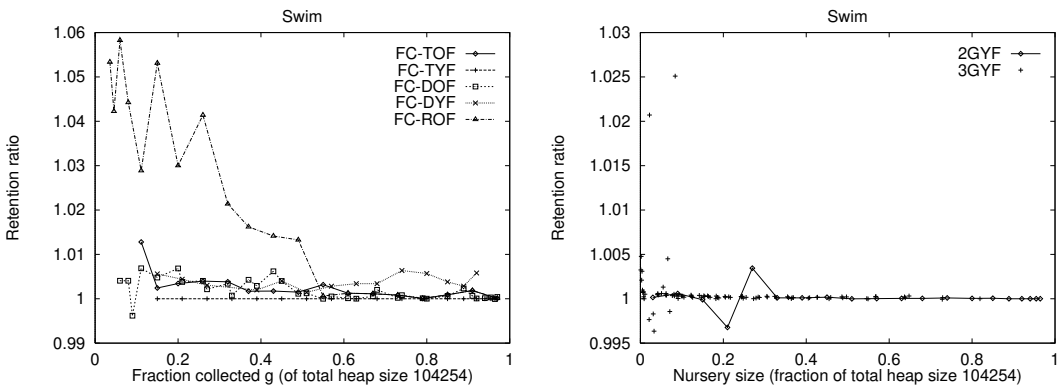
(a) FC schemes (b) GYF schemes

Figure 5.188. Excess retention ratios: Swim, $V = 70160$.



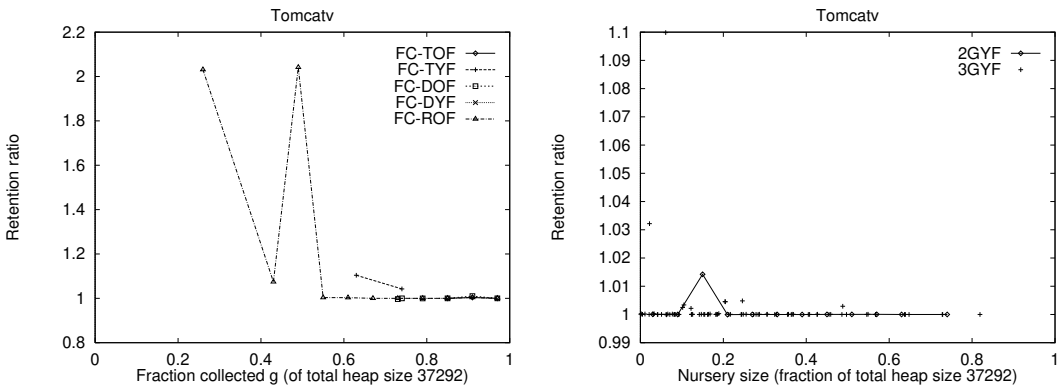
(a) FC schemes (b) GYF schemes

Figure 5.189. Excess retention ratios: Swim, $V = 85524$.



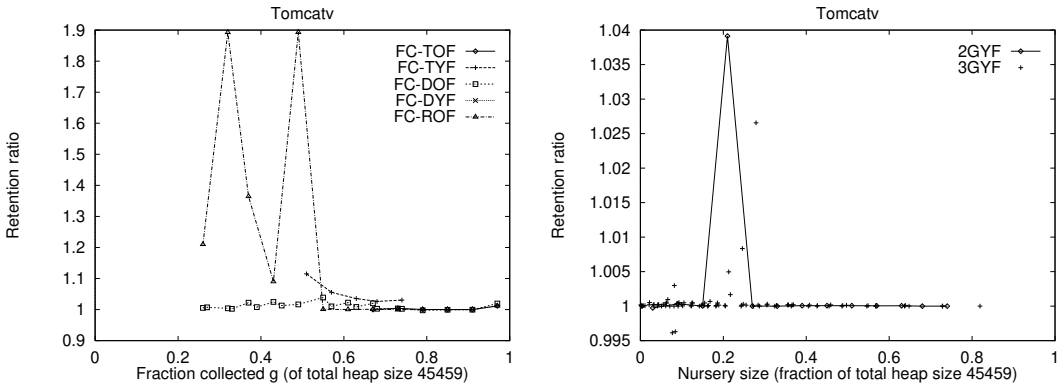
(a) FC schemes (b) GYF schemes

Figure 5.190. Excess retention ratios: Swim, $V = 104254$.



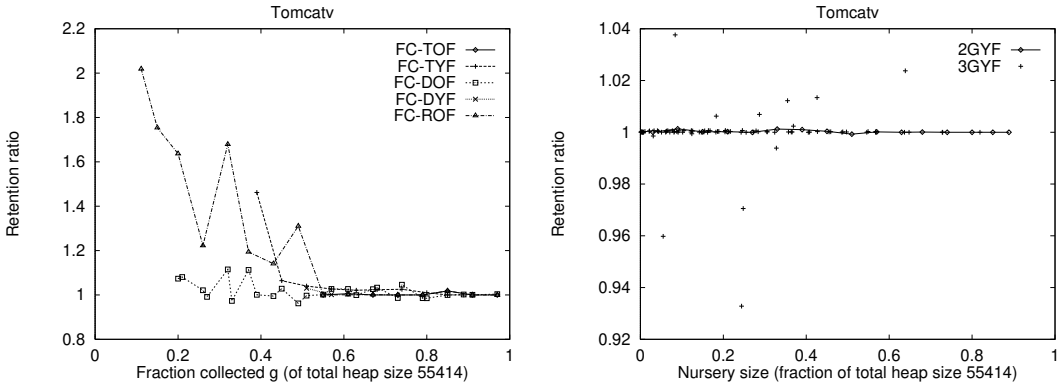
(a) FC schemes (b) GYF schemes

Figure 5.191. Excess retention ratios: Tomcatv, $V = 37292$.



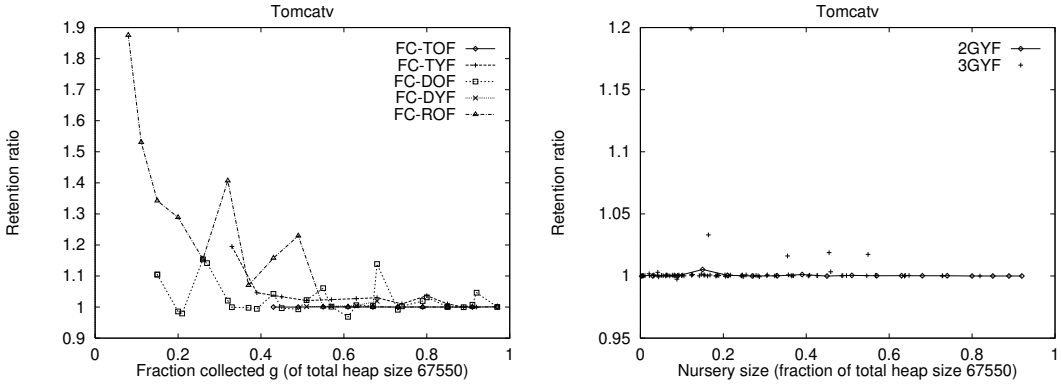
(a) FC schemes (b) GYF schemes

Figure 5.192. Excess retention ratios: Tomcatv, $V = 45459$.



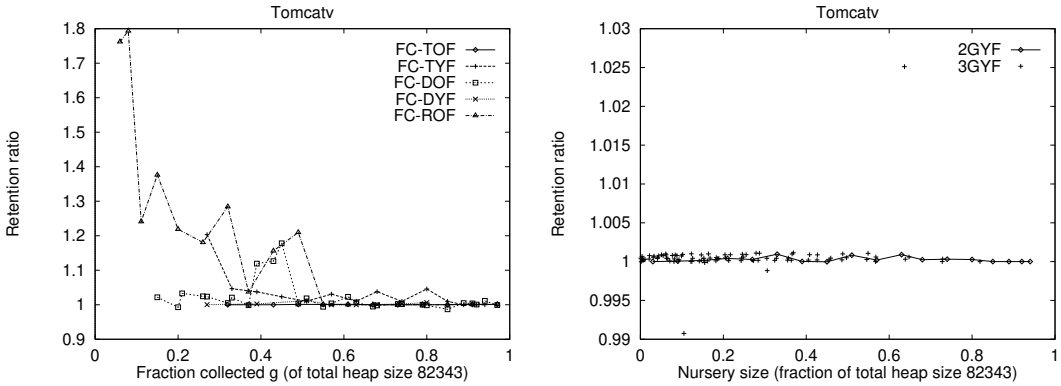
(a) FC schemes (b) GYF schemes

Figure 5.193. Excess retention ratios: Tomcatv, $V = 55414$.



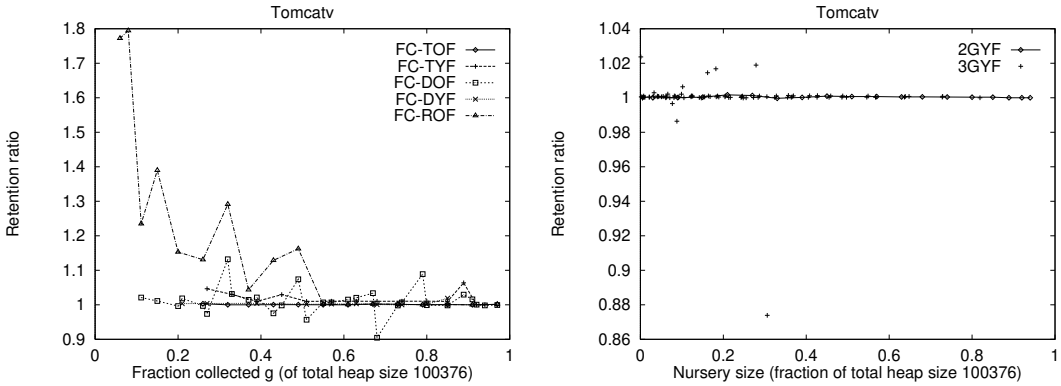
(a) FC schemes (b) GYF schemes

Figure 5.194. Excess retention ratios: Tomcatv, $V = 67550$.



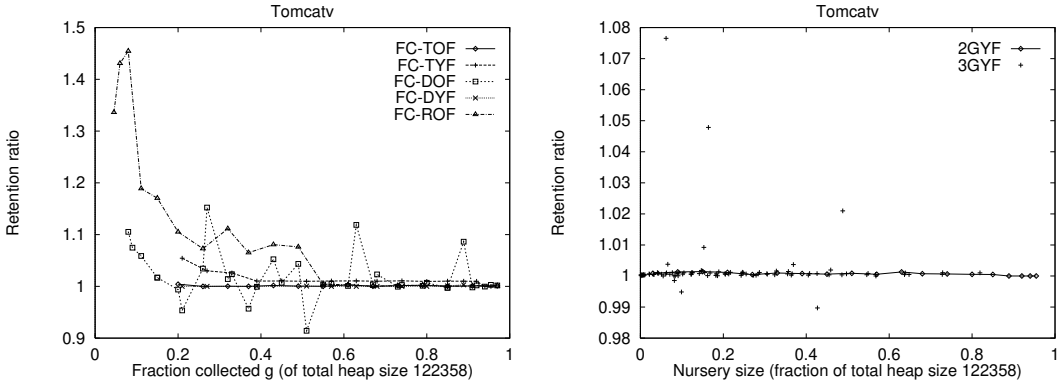
(a) FC schemes (b) GYF schemes

Figure 5.195. Excess retention ratios: Tomcatv, $V = 82343$.



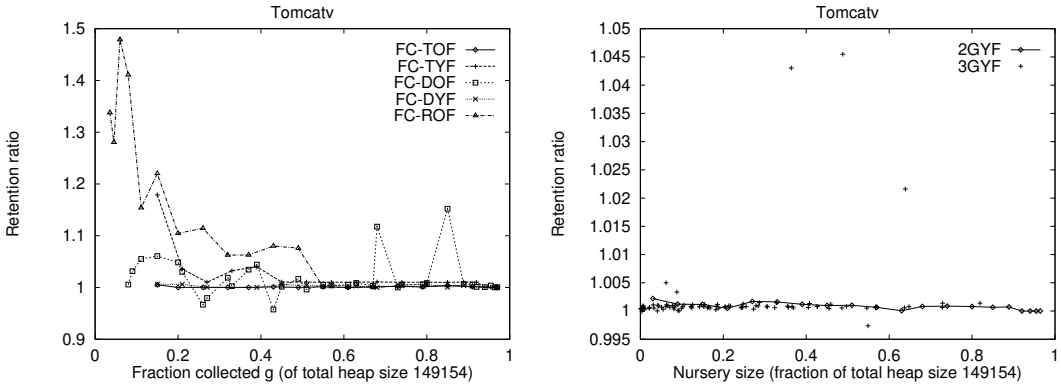
(a) FC schemes (b) GYF schemes

Figure 5.196. Excess retention ratios: Tomcatv, $V = 100376$.



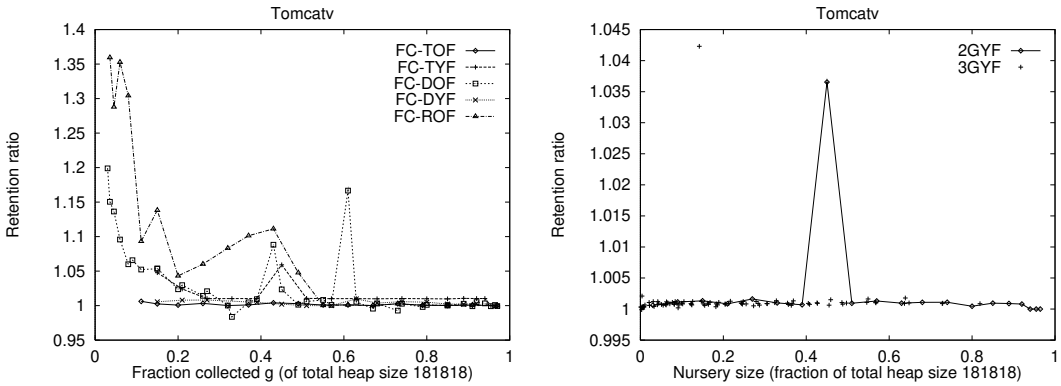
(a) FC schemes (b) GYF schemes

Figure 5.197. Excess retention ratios: Tomcatv, $V = 122358$.



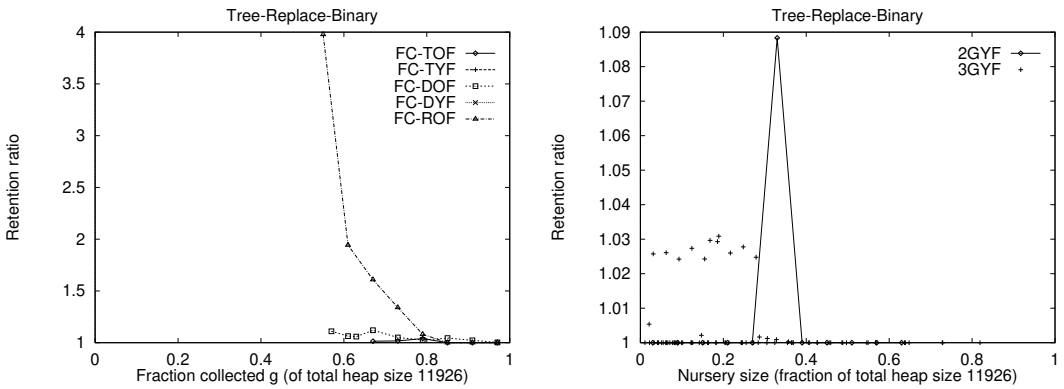
(a) FC schemes (b) GYF schemes

Figure 5.198. Excess retention ratios: Tomcatv, $V = 149154$.



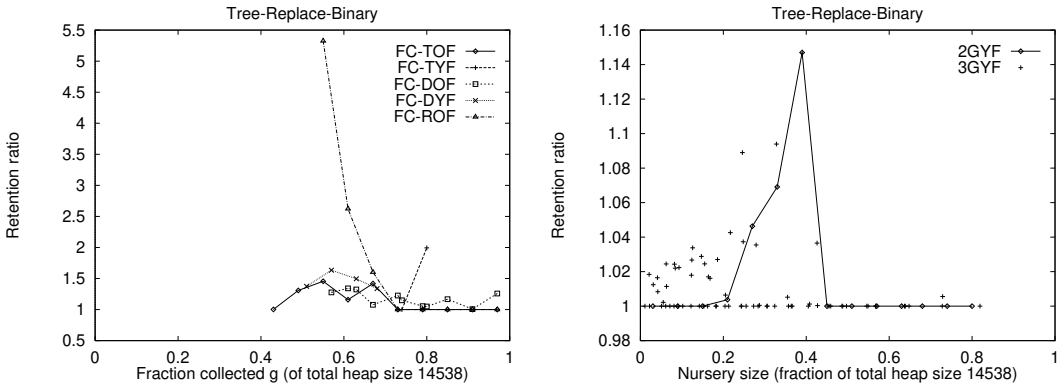
(a) FC schemes (b) GYF schemes

Figure 5.199. Excess retention ratios: Tomcatv, $V = 181818$.



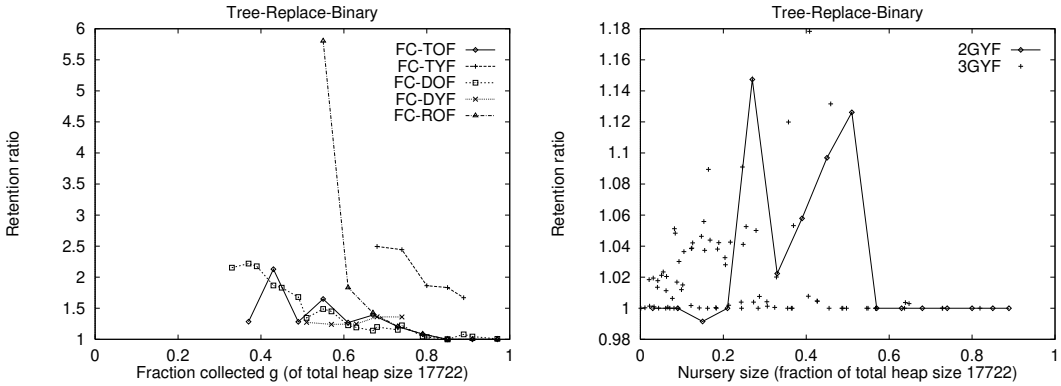
(a) FC schemes (b) GYF schemes

Figure 5.200. Excess retention ratios: Tree-Replace-Binary, $V = 11926$.



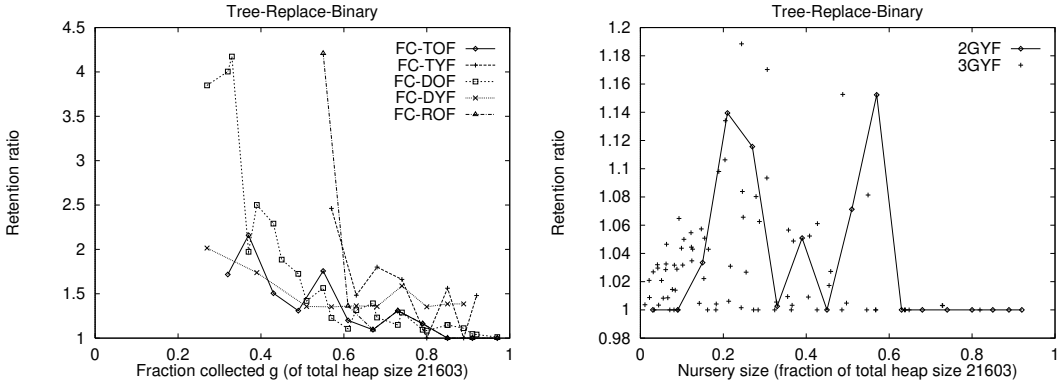
(a) FC schemes (b) GYF schemes

Figure 5.201. Excess retention ratios: Tree-Replace-Binary, $V = 14538$.



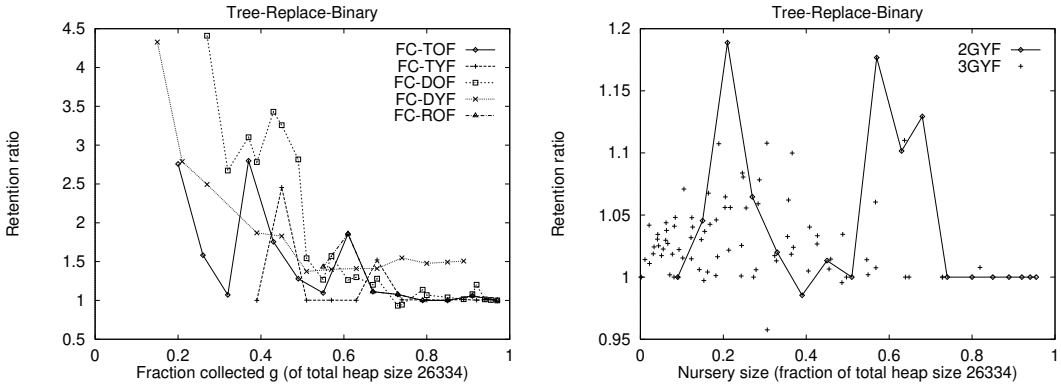
(a) FC schemes (b) GYF schemes

Figure 5.202. Excess retention ratios: Tree-Replace-Binary, $V = 17722$.



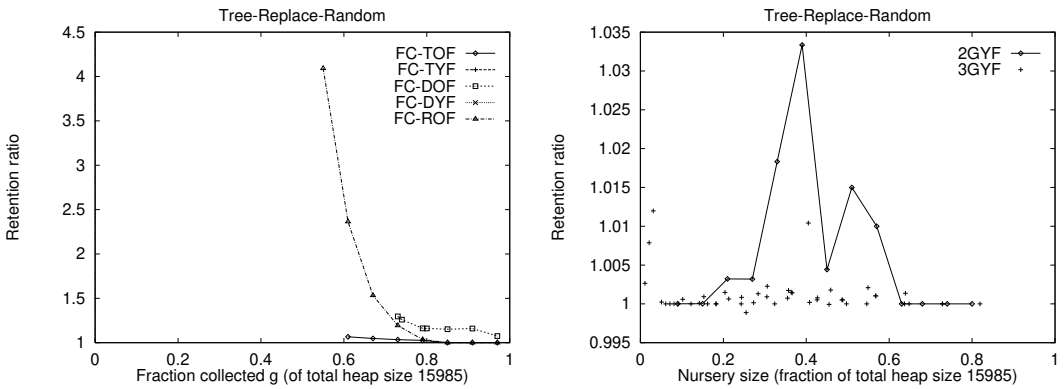
(a) FC schemes (b) GYF schemes

Figure 5.203. Excess retention ratios: Tree-Replace-Binary, $V = 21603$.



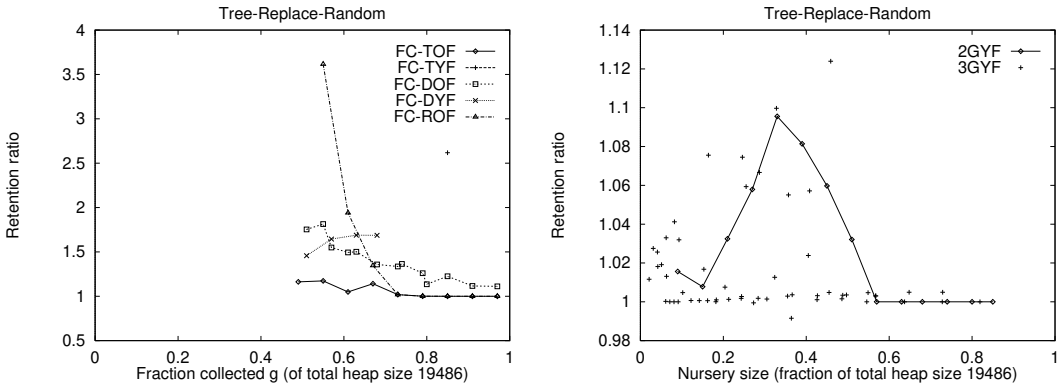
(a) FC schemes (b) GYF schemes

Figure 5.204. Excess retention ratios: Tree-Replace-Binary, $V = 26334$.



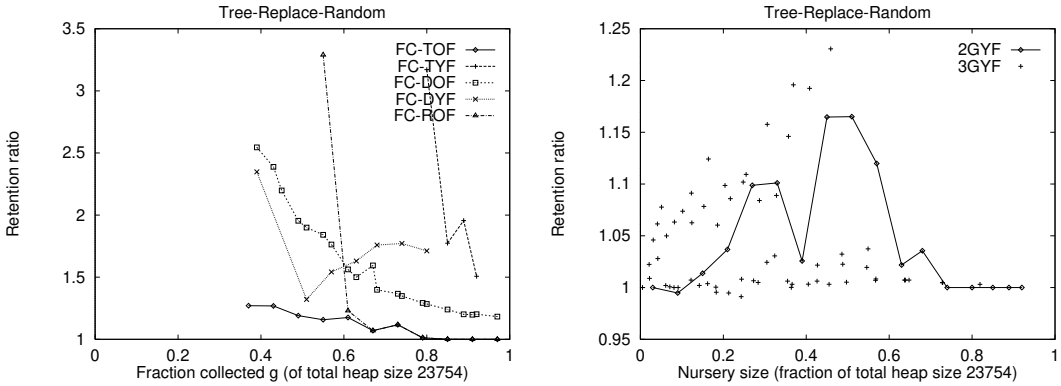
(a) FC schemes (b) GYF schemes

Figure 5.205. Excess retention ratios: Tree-Replace-Random, $V = 15985$.



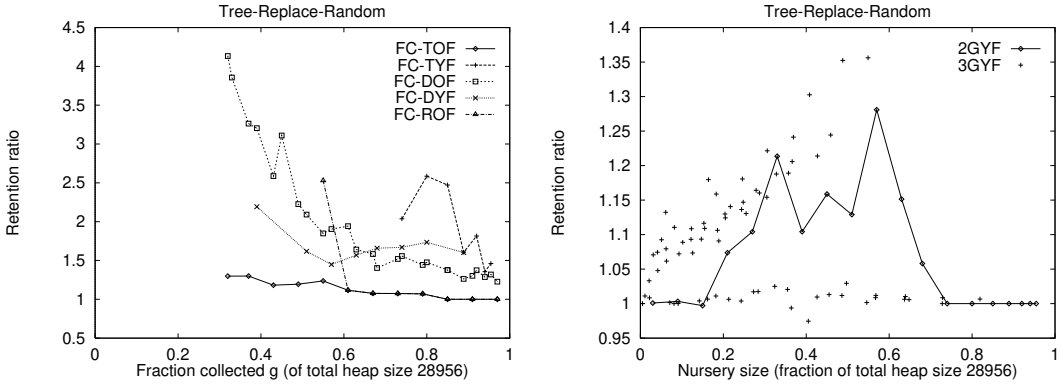
(a) FC schemes (b) GYF schemes

Figure 5.206. Excess retention ratios: Tree-Replace-Random, $V = 19486$.



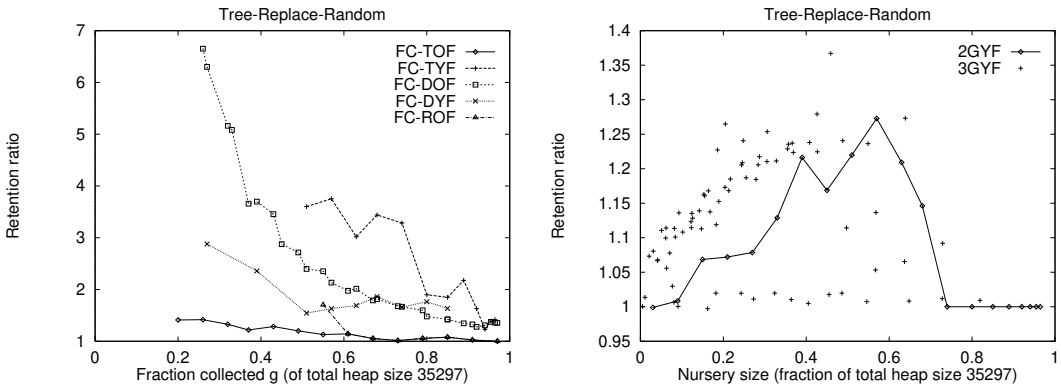
(a) FC schemes (b) GYF schemes

Figure 5.207. Excess retention ratios: Tree-Replace-Random, $V = 23754$.



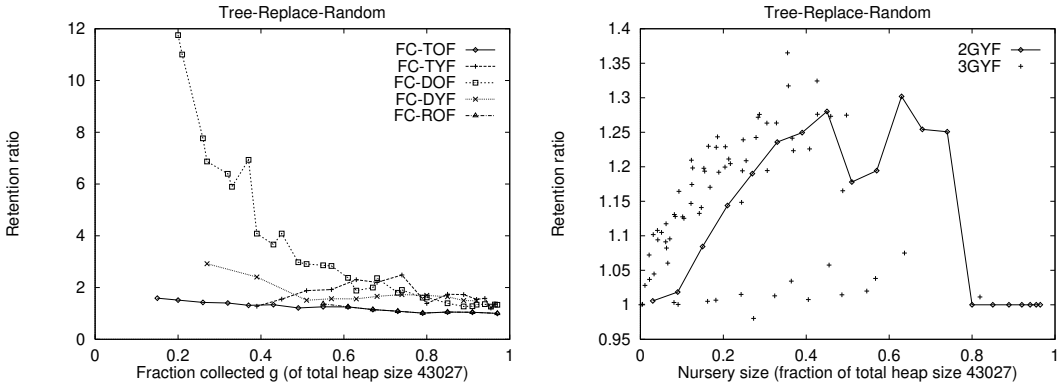
(a) FC schemes (b) GYF schemes

Figure 5.208. Excess retention ratios: Tree-Replace-Random, $V = 28956$.



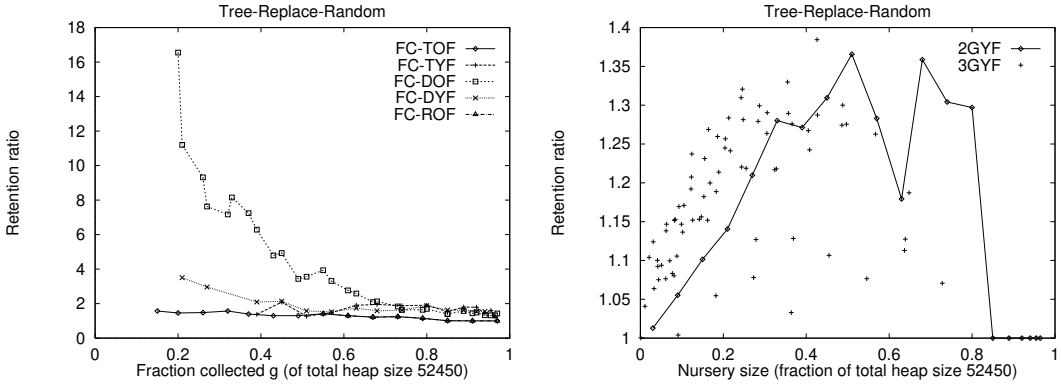
(a) FC schemes (b) GYF schemes

Figure 5.209. Excess retention ratios: Tree-Replace-Random, $V = 35297$.



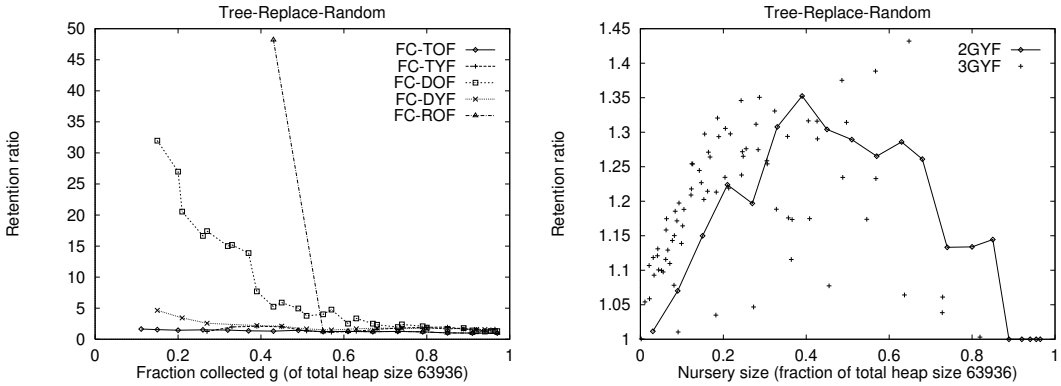
(a) FC schemes (b) GYF schemes

Figure 5.210. Excess retention ratios: Tree-Replace-Random, $V = 43027$.



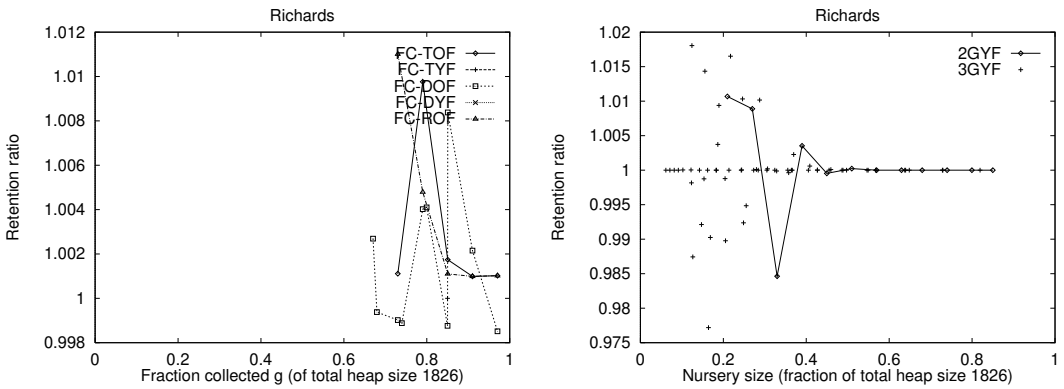
(a) FC schemes (b) GYF schemes

Figure 5.211. Excess retention ratios: Tree-Replace-Random, $V = 52450$.



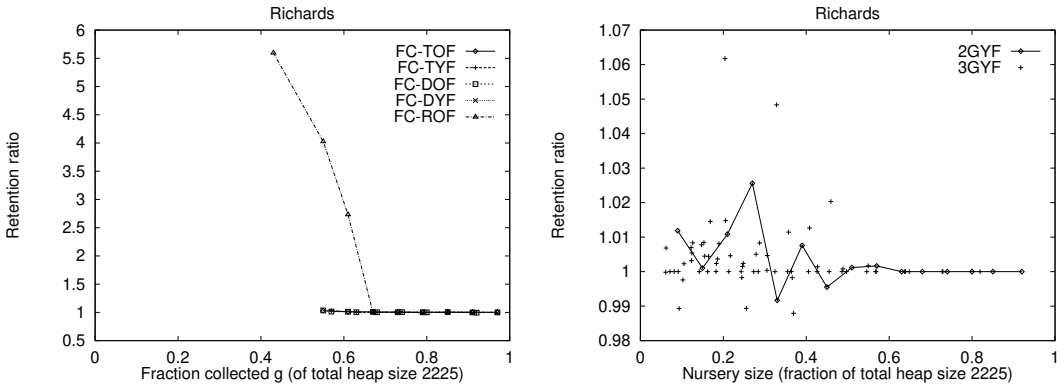
(a) FC schemes (b) GYF schemes

Figure 5.212. Excess retention ratios: Tree-Replace-Random, $V = 63936$.



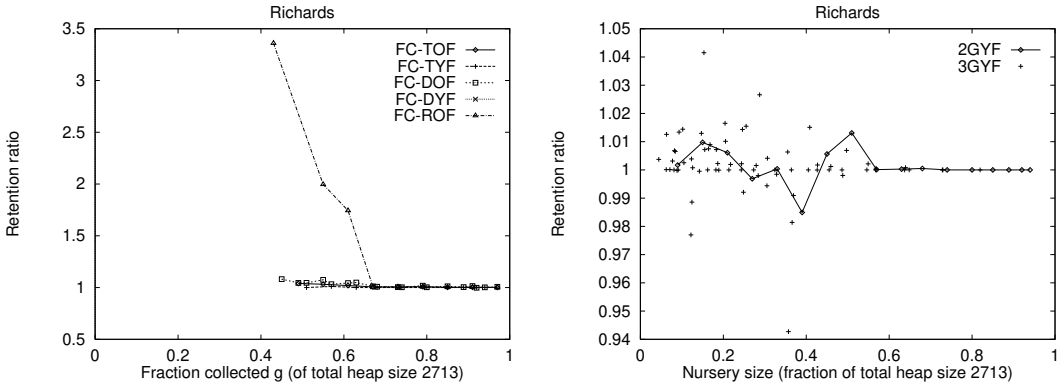
(a) FC schemes (b) GYF schemes

Figure 5.213. Excess retention ratios: Richards, $V = 1826$.



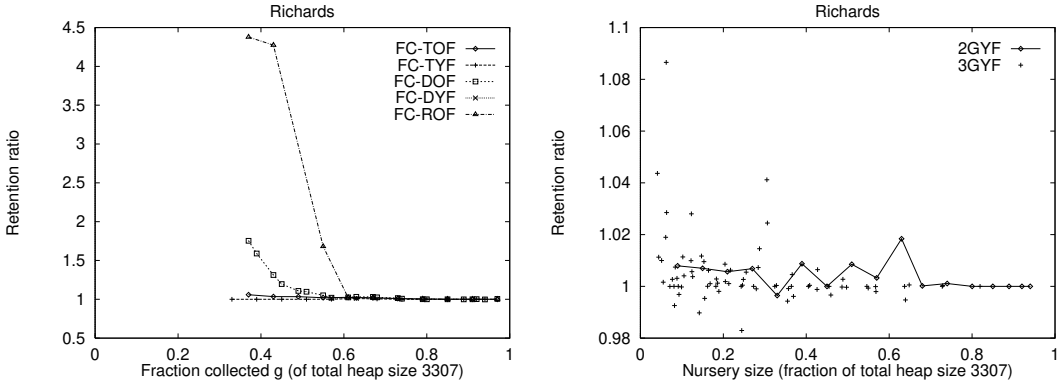
(a) FC schemes (b) GYF schemes

Figure 5.214. Excess retention ratios: Richards, $V = 2225$.



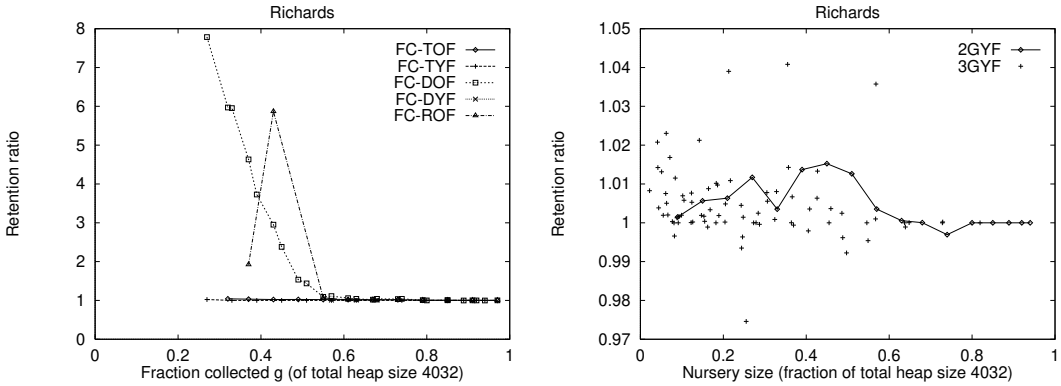
(a) FC schemes (b) GYF schemes

Figure 5.215. Excess retention ratios: Richards, $V = 2713$.



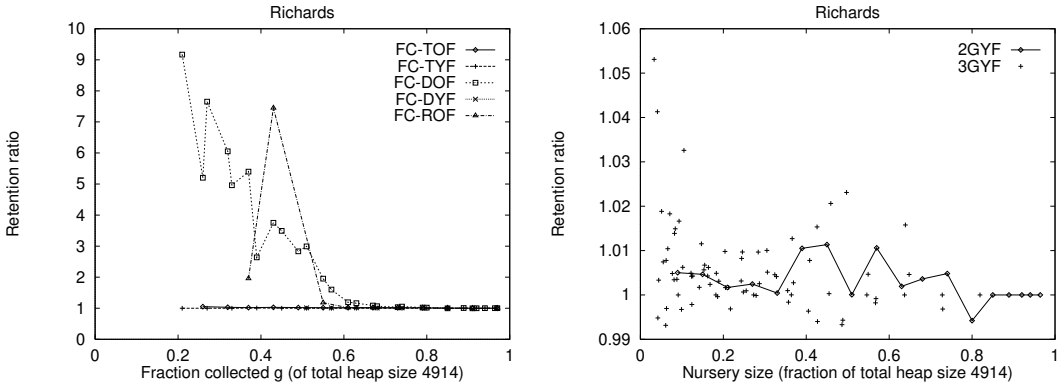
(a) FC schemes (b) GYF schemes

Figure 5.216. Excess retention ratios: Richards, $V = 3307$.



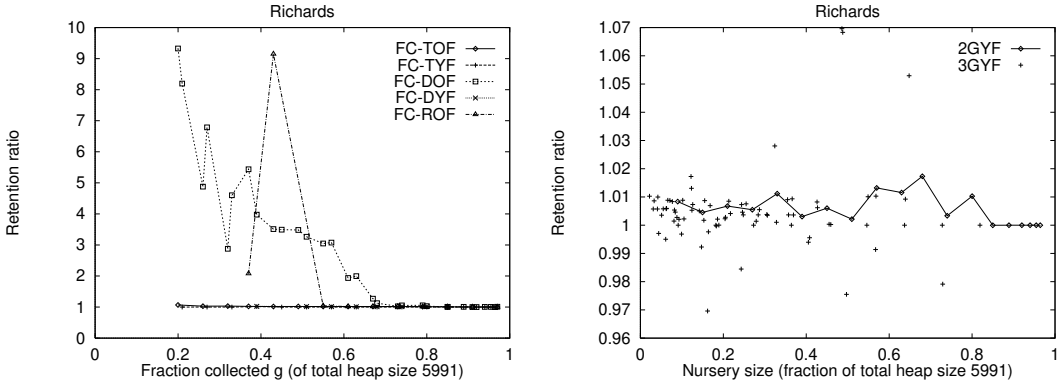
(a) FC schemes (b) GYF schemes

Figure 5.217. Excess retention ratios: Richards, $V = 4032$.



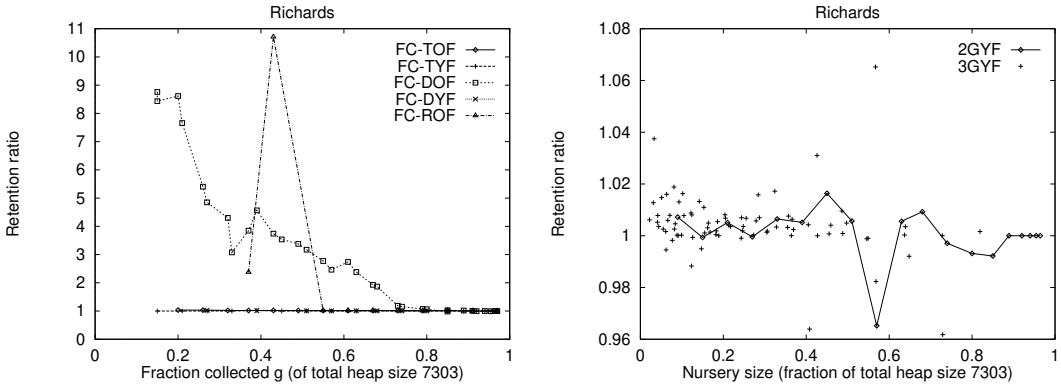
(a) FC schemes (b) GYF schemes

Figure 5.218. Excess retention ratios: Richards, $V = 4914$.



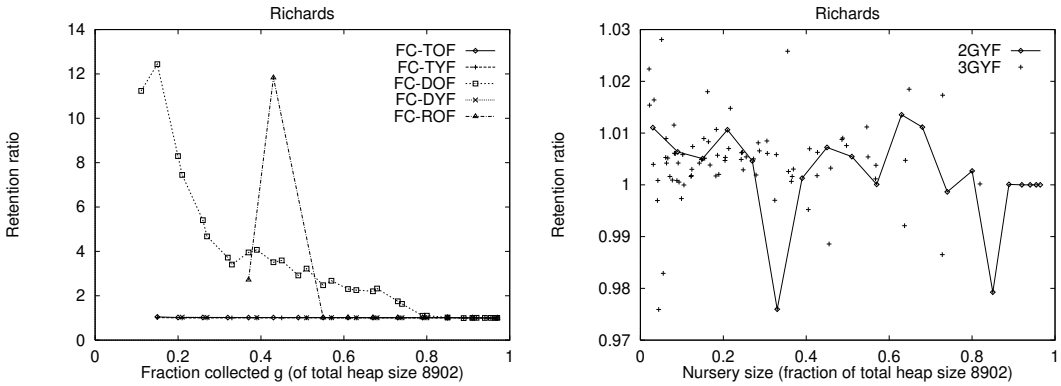
(a) FC schemes (b) GYF schemes

Figure 5.219. Excess retention ratios: Richards, $V = 5991$.



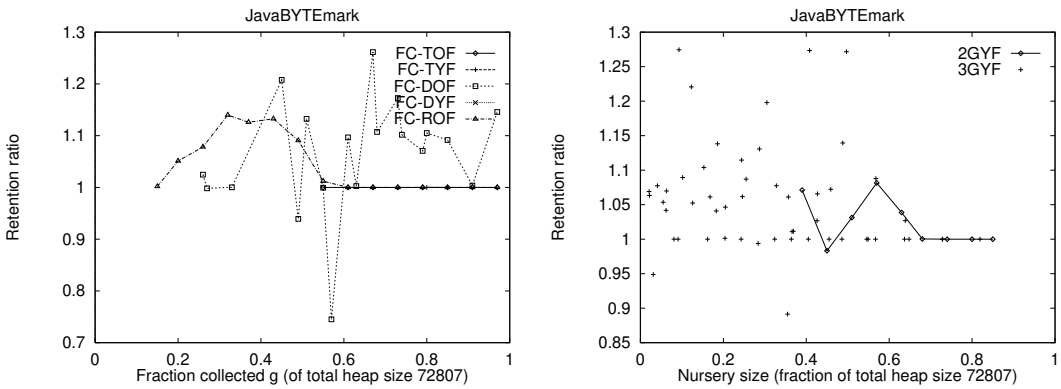
(a) FC schemes (b) GYF schemes

Figure 5.220. Excess retention ratios: Richards, $V = 7303$.



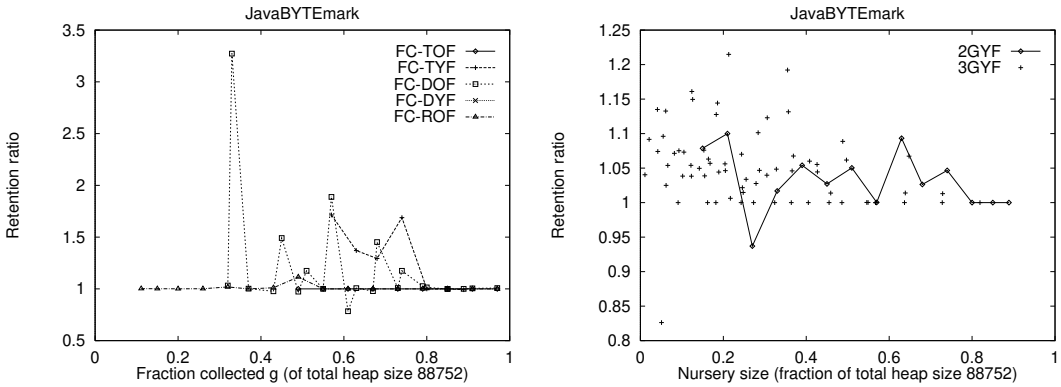
(a) FC schemes (b) GYF schemes

Figure 5.221. Excess retention ratios: Richards, $V = 8902$.



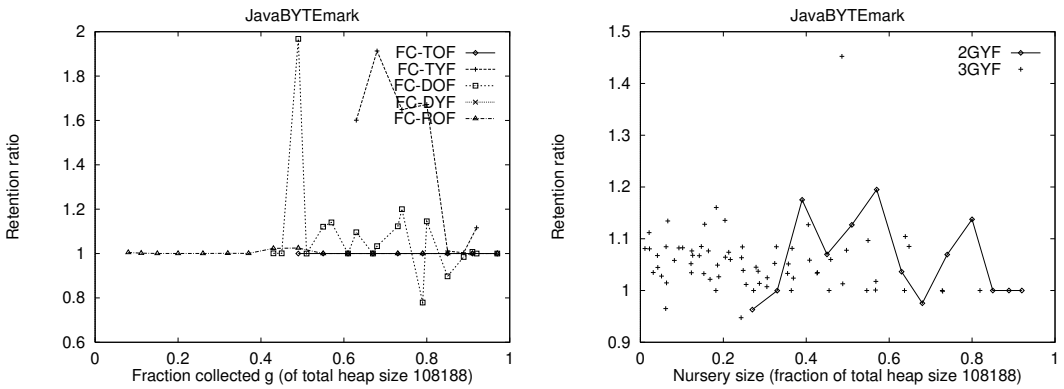
(a) FC schemes (b) GYF schemes

Figure 5.222. Excess retention ratios: JavaBYTEmark, $V = 72807$.



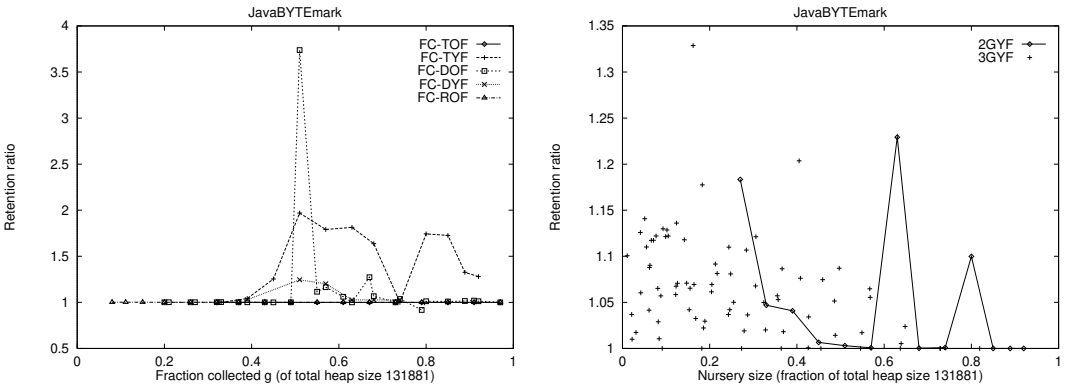
(a) FC schemes (b) GYF schemes

Figure 5.223. Excess retention ratios: JavaBYTEmark, $V = 88752$.



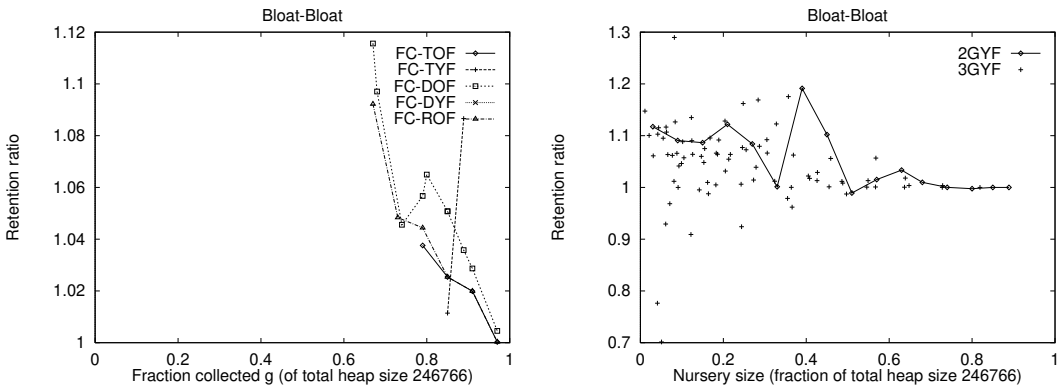
(a) FC schemes (b) GYF schemes

Figure 5.224. Excess retention ratios: JavaBYTEmark, $V = 108188$.



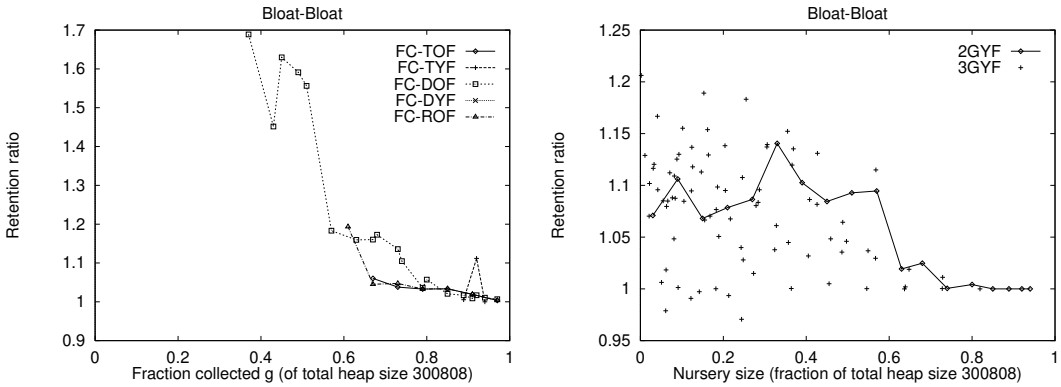
(a) FC schemes (b) GYF schemes

Figure 5.225. Excess retention ratios: JavaBYTEmark, $V = 131881$.



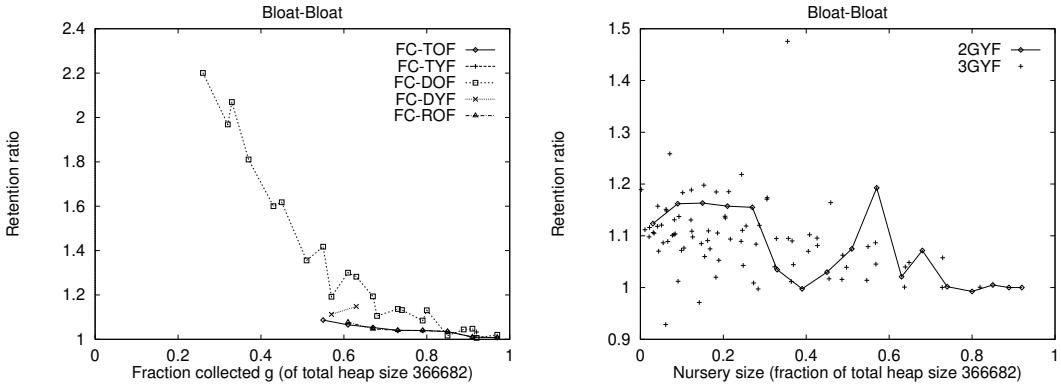
(a) FC schemes (b) GYF schemes

Figure 5.226. Excess retention ratios: Bloat-Bloat, $V = 246766$.



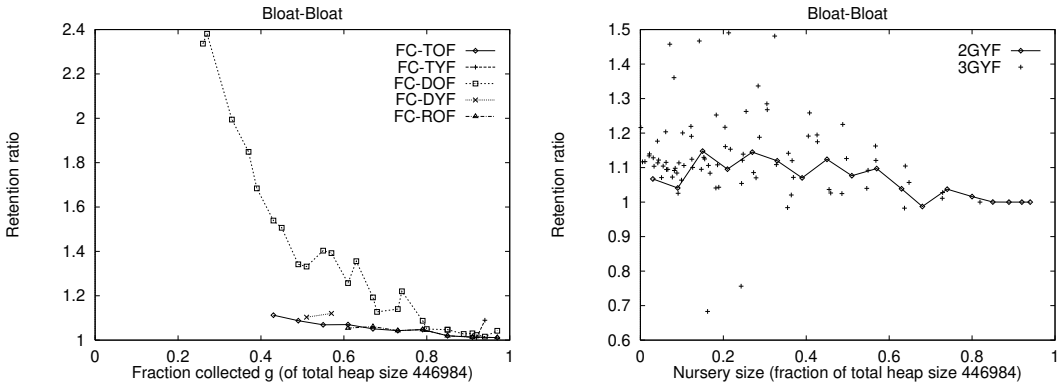
(a) FC schemes (b) GYF schemes

Figure 5.227. Excess retention ratios: Bloat-Bloat, $V = 300808$.



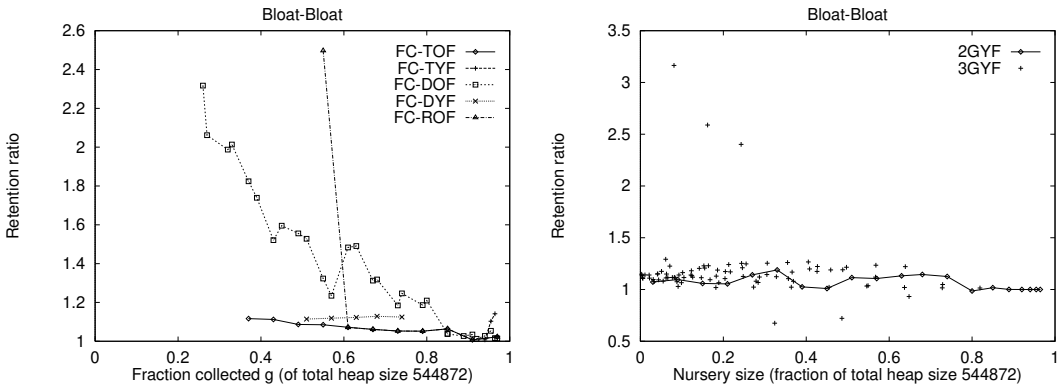
(a) FC schemes (b) GYF schemes

Figure 5.228. Excess retention ratios: Bloat-Bloat, $V = 366682$.



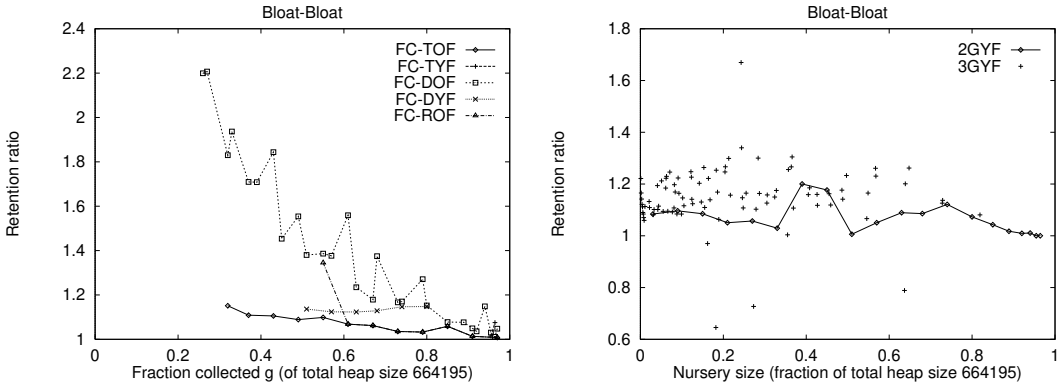
(a) FC schemes (b) GYF schemes

Figure 5.229. Excess retention ratios: Bloat-Bloat, $V = 446984$.



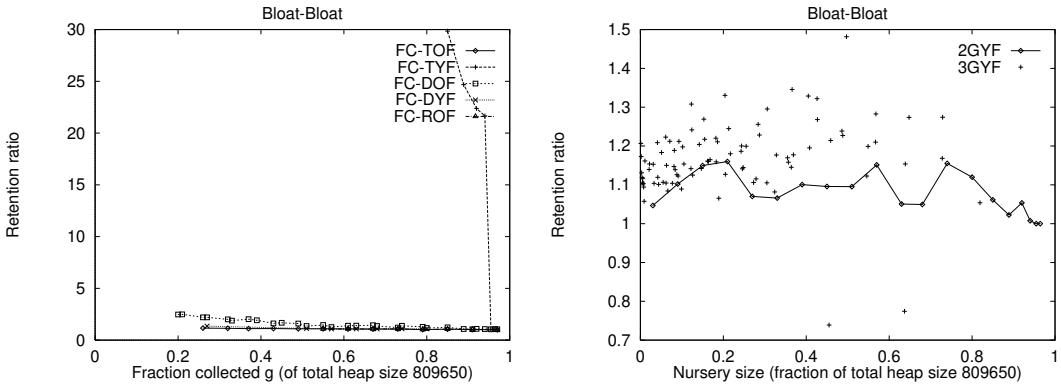
(a) FC schemes (b) GYF schemes

Figure 5.230. Excess retention ratios: Bloat-Bloat, $V = 544872$.



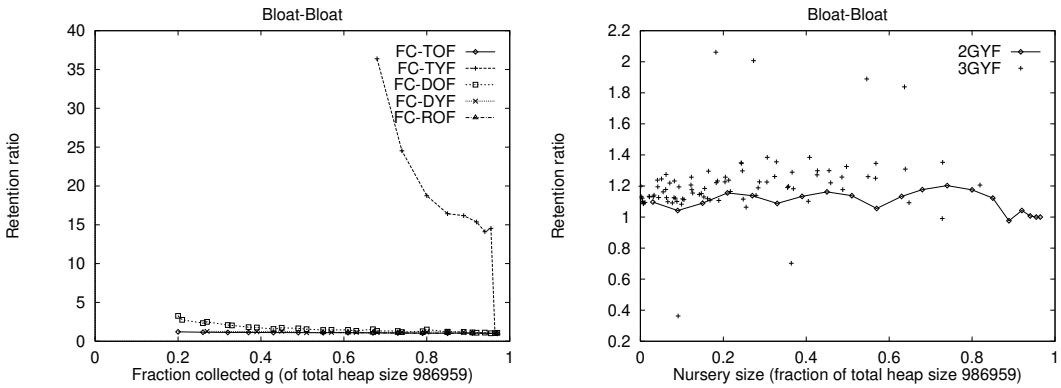
(a) FC schemes (b) GYF schemes

Figure 5.231. Excess retention ratios: Bloat-Bloat, $V = 664195$.



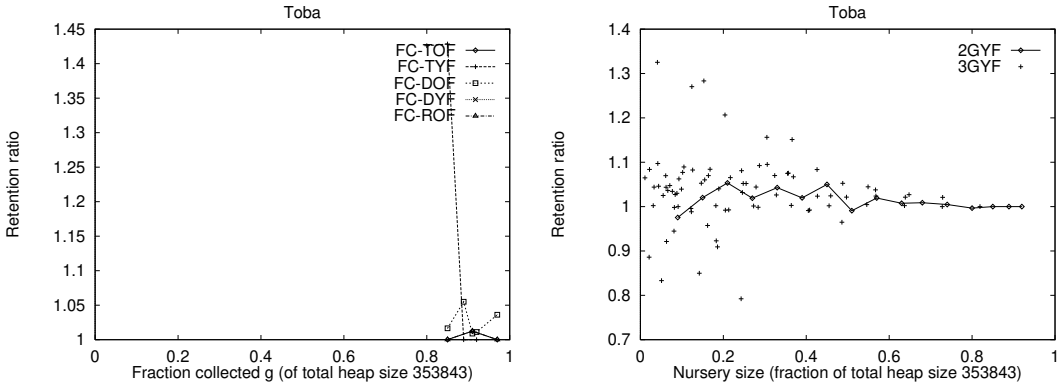
(a) FC schemes (b) GYF schemes

Figure 5.232. Excess retention ratios: Bloat-Bloat, $V = 809650$.



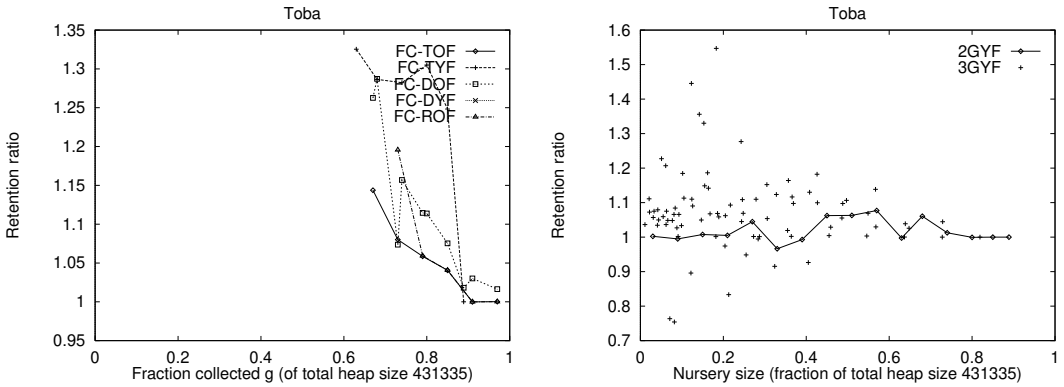
(a) FC schemes (b) GYF schemes

Figure 5.233. Excess retention ratios: Bloat-Bloat, $V = 986959$.



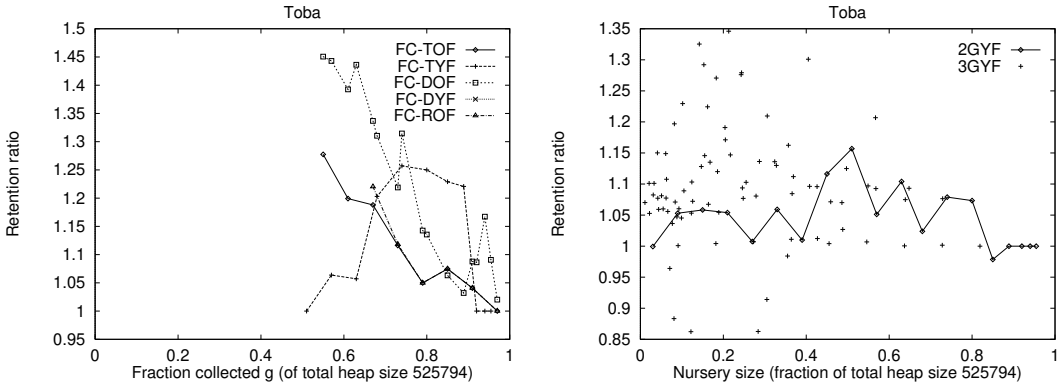
(a) FC schemes (b) GYF schemes

Figure 5.234. Excess retention ratios: Toba, $V = 353843$.



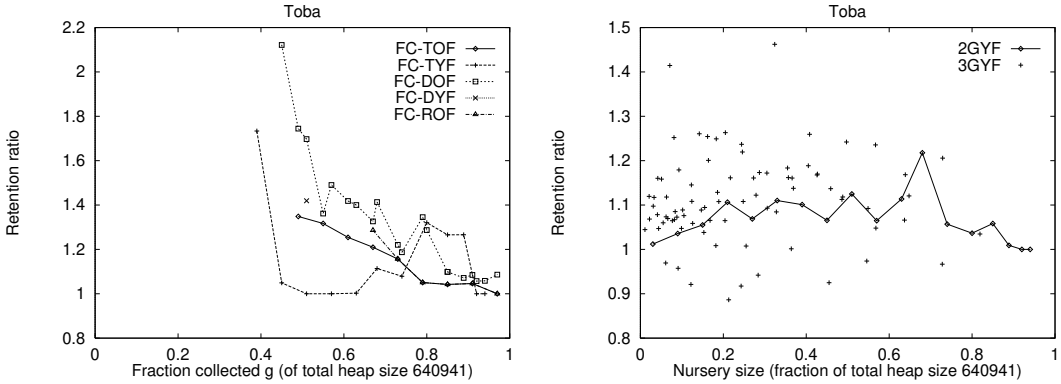
(a) FC schemes (b) GYF schemes

Figure 5.235. Excess retention ratios: Toba, $V = 431335$.



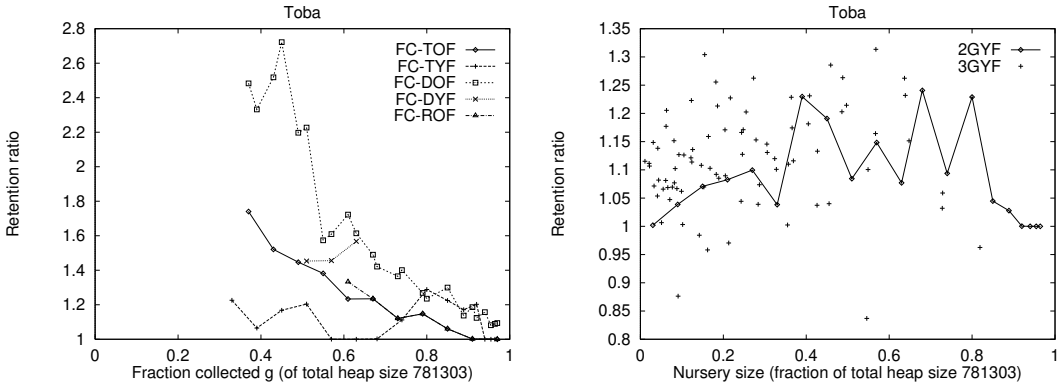
(a) FC schemes (b) GYF schemes

Figure 5.236. Excess retention ratios: Toba, $V = 525794$.



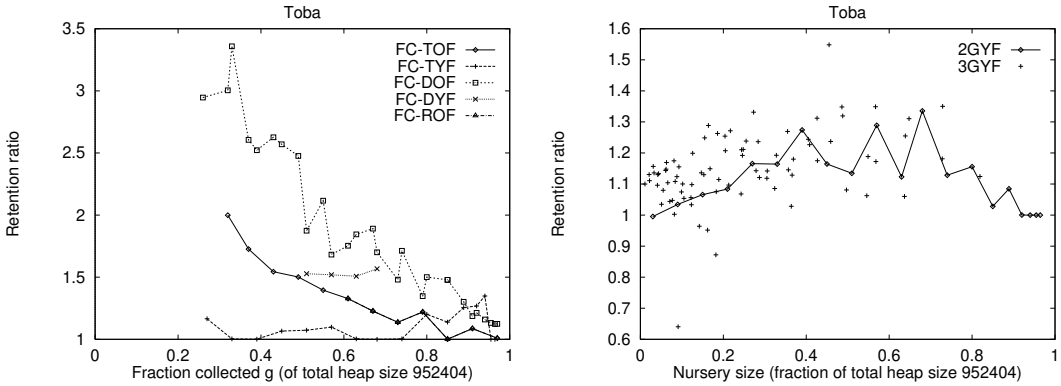
(a) FC schemes (b) GYF schemes

Figure 5.237. Excess retention ratios: Toba, $V = 640941$.



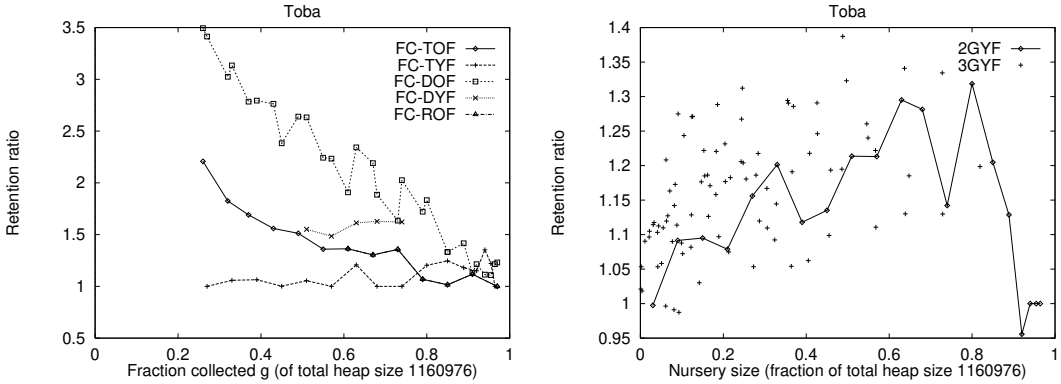
(a) FC schemes (b) GYF schemes

Figure 5.238. Excess retention ratios: Toba, $V = 781303$.



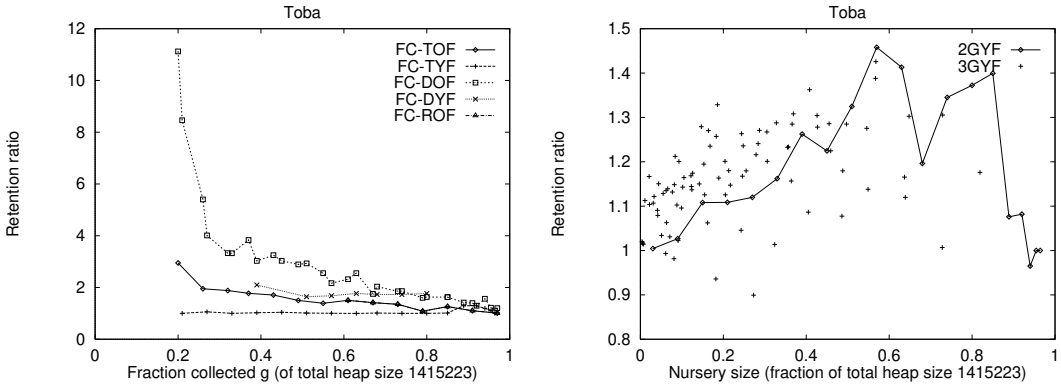
(a) FC schemes (b) GYF schemes

Figure 5.239. Excess retention ratios: Toba, $V = 952404$.



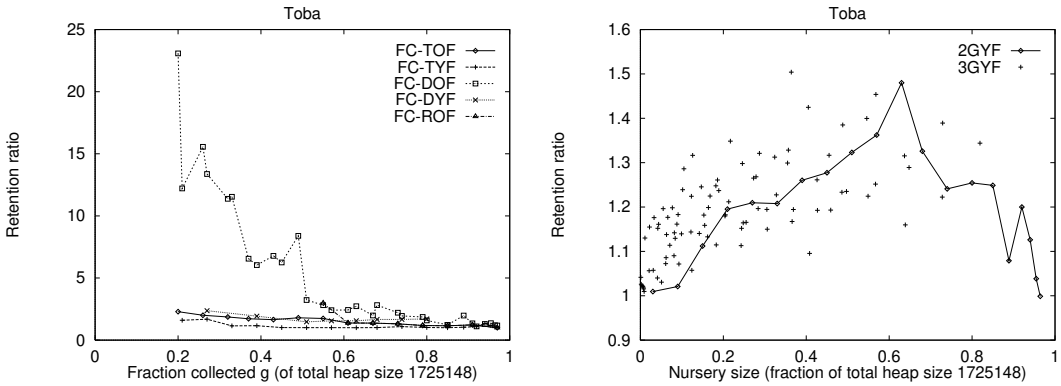
(a) FC schemes (b) GYF schemes

Figure 5.240. Excess retention ratios: Toba, $V = 1160976$.



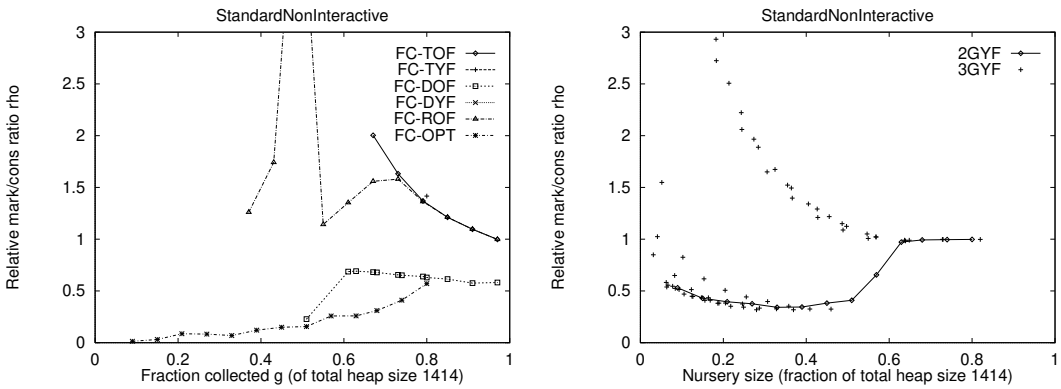
(a) FC schemes (b) GYF schemes

Figure 5.241. Excess retention ratios: Toba, $V = 1415223$.



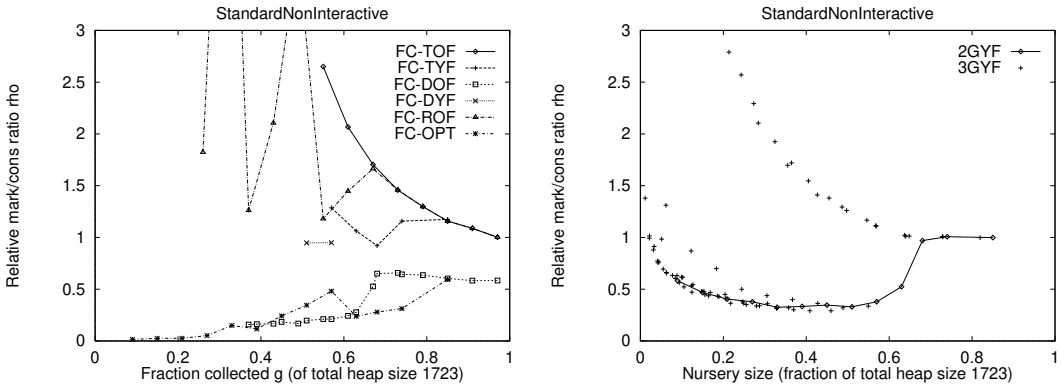
(a) FC schemes (b) GYF schemes

Figure 5.242. Excess retention ratios: Toba, $V = 1725148$.



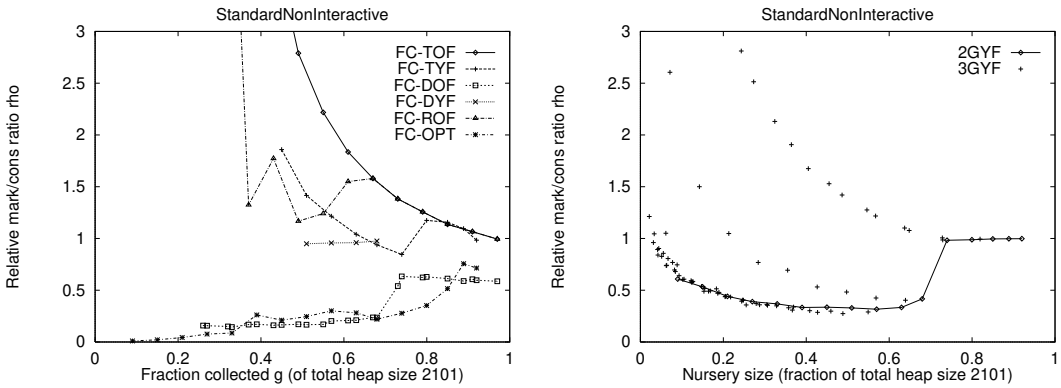
(a) FC schemes including OPT (b) GYF schemes

Figure 5.243. Copying cost, with FC-OPT: StandardNonInteractive, $V = 1414$.



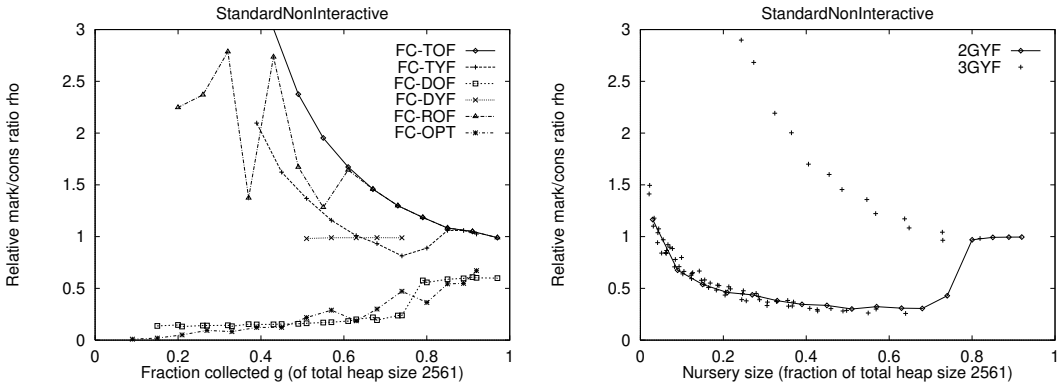
(a) FC schemes including OPT (b) GYF schemes

Figure 5.244. Copying cost, with FC-OPT: StandardNonInteractive, $V = 1723$.



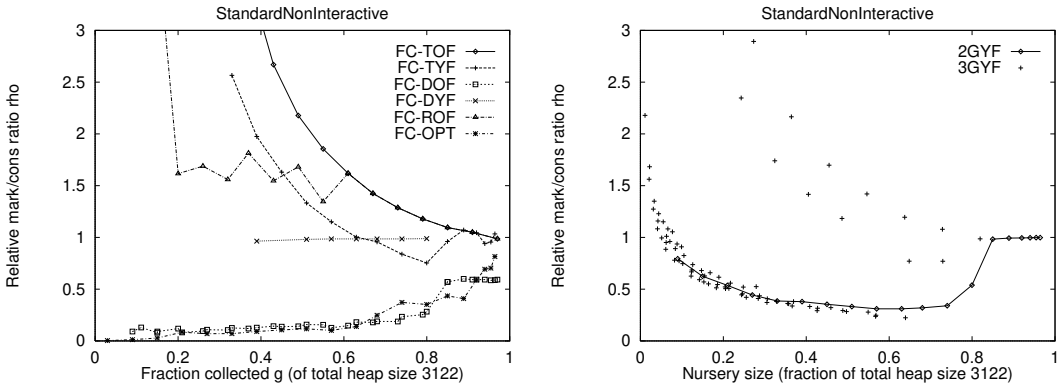
(a) FC schemes including OPT (b) GYF schemes

Figure 5.245. Copying cost, with FC-OPT: StandardNonInteractive, $V = 2101$.



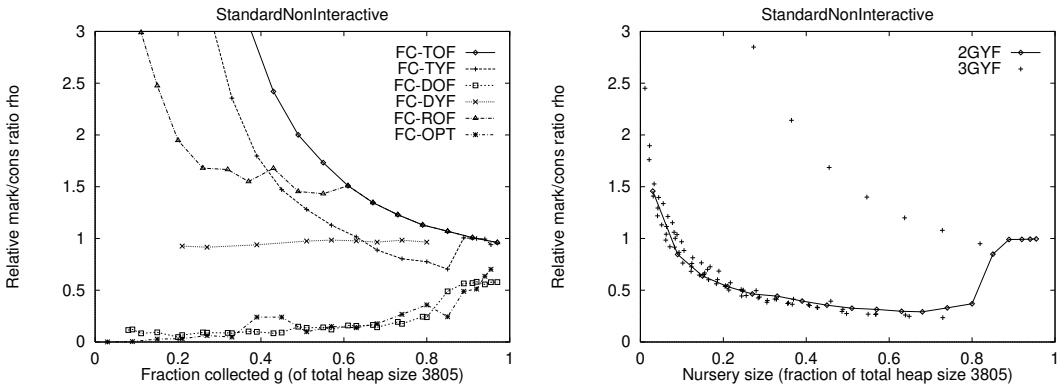
(a) FC schemes including OPT (b) GYF schemes

Figure 5.246. Copying cost, with FC-OPT: StandardNonInteractive, $V = 2561$.



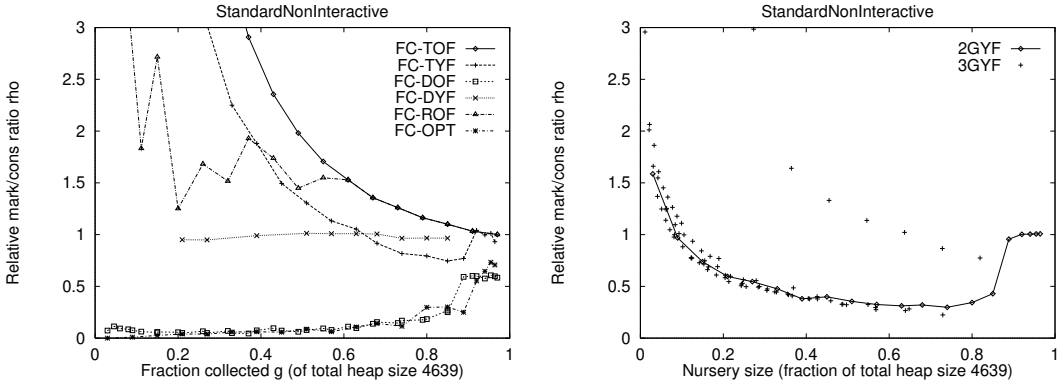
(a) FC schemes including OPT (b) GYF schemes

Figure 5.247. Copying cost, with FC-OPT: StandardNonInteractive, $V = 3122$.



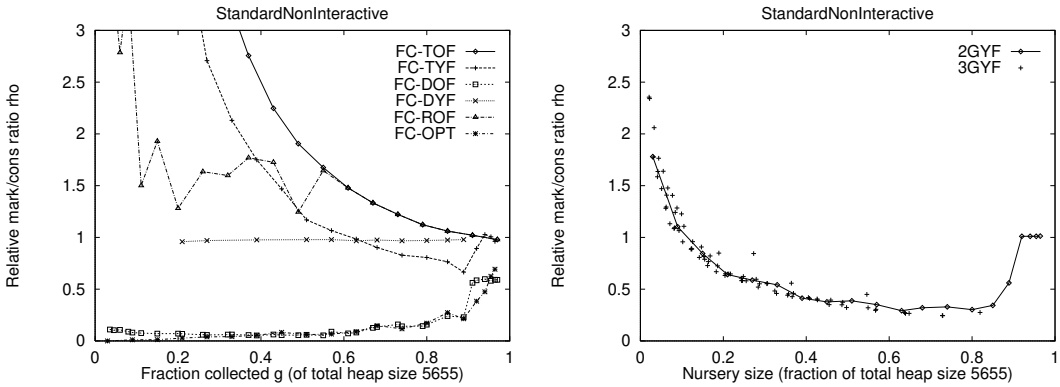
(a) FC schemes including OPT (b) GYF schemes

Figure 5.248. Copying cost, with FC-OPT: StandardNonInteractive, $V = 3805$.



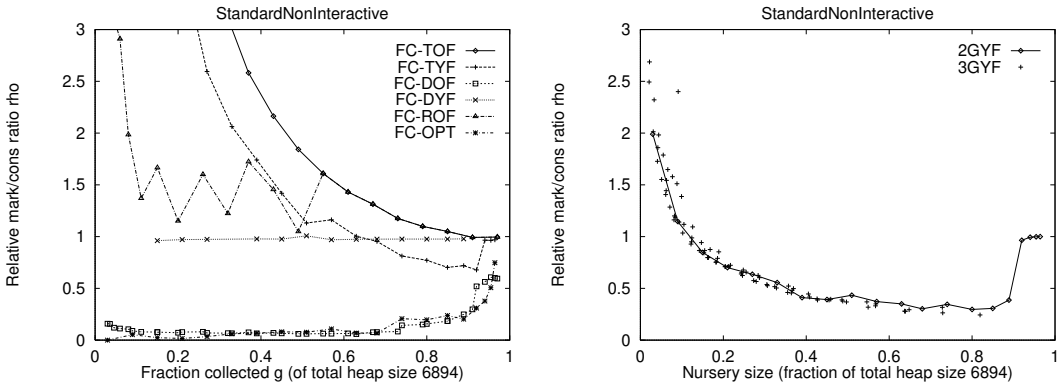
(a) FC schemes including OPT (b) GYF schemes

Figure 5.249. Copying cost, with FC-OPT: StandardNonInteractive, $V = 4639$.



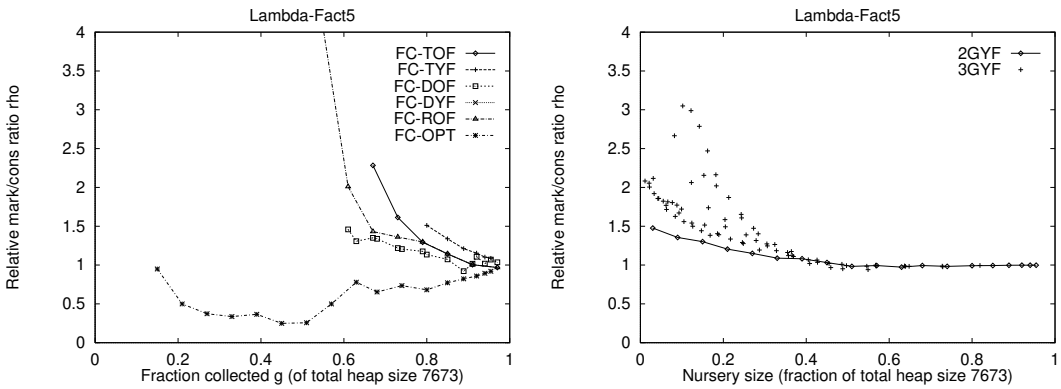
(a) FC schemes including OPT (b) GYF schemes

Figure 5.250. Copying cost, with FC-OPT: StandardNonInteractive, $V = 5655$.



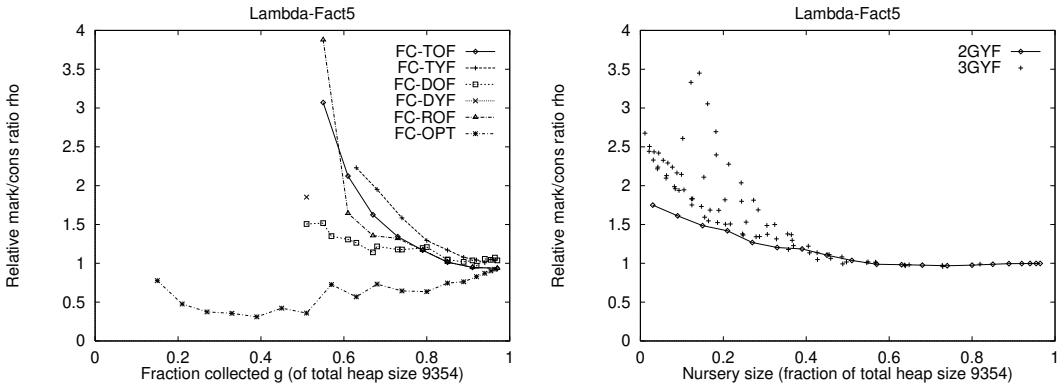
(a) FC schemes including OPT (b) GYF schemes

Figure 5.251. Copying cost, with FC-OPT: StandardNonInteractive, $V = 6894$.



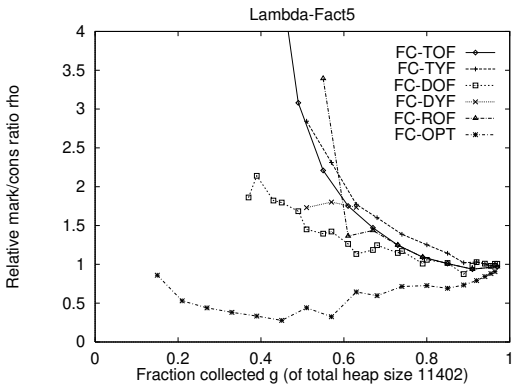
(a) FC schemes including OPT (b) GYF schemes

Figure 5.252. Copying cost, with FC-OPT: Lambda-Fact5, $V = 7673$.

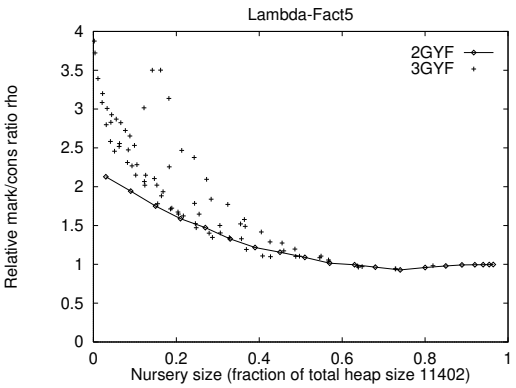


(a) FC schemes including OPT (b) GYF schemes

Figure 5.253. Copying cost, with FC-OPT: Lambda-Fact5, $V = 9354$.

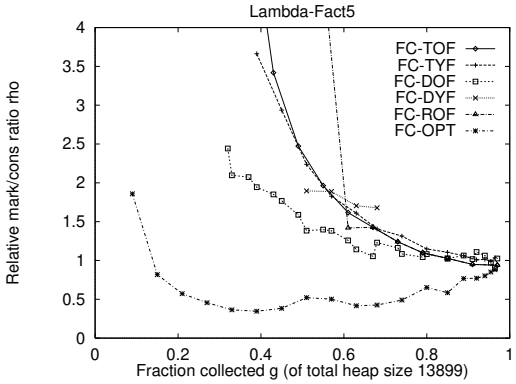


(a) FC schemes including OPT

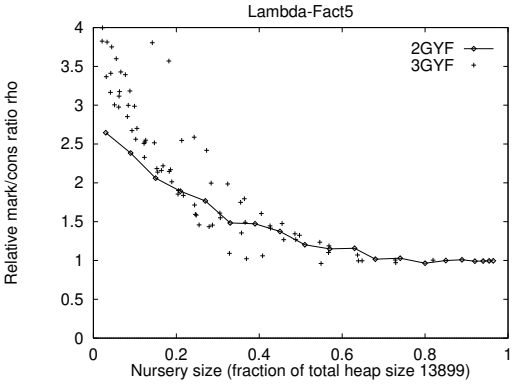


(b) GYF schemes

Figure 5.254. Copying cost, with FC-OPT: Lambda-Fact5, $V = 11402$.

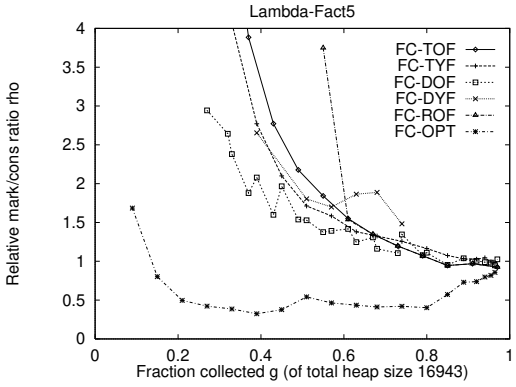


(a) FC schemes including OPT

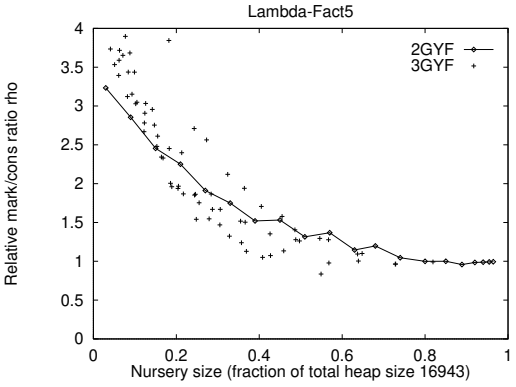


(b) GYF schemes

Figure 5.255. Copying cost, with FC-OPT: Lambda-Fact5, $V = 13899$.

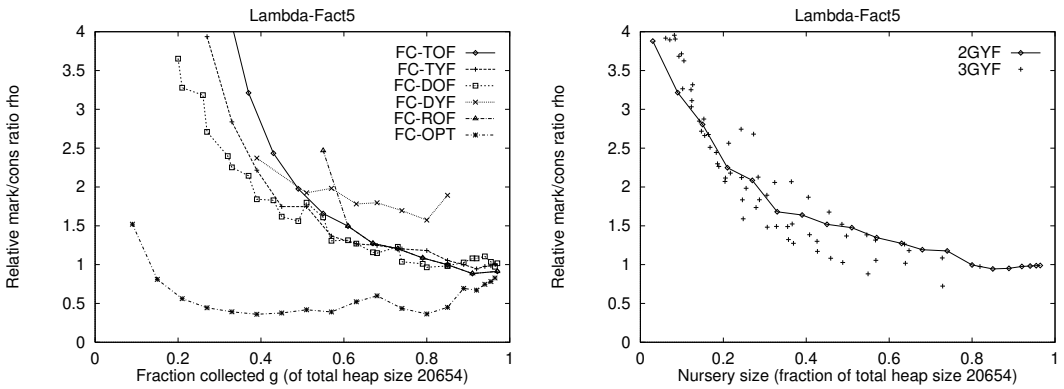


(a) FC schemes including OPT



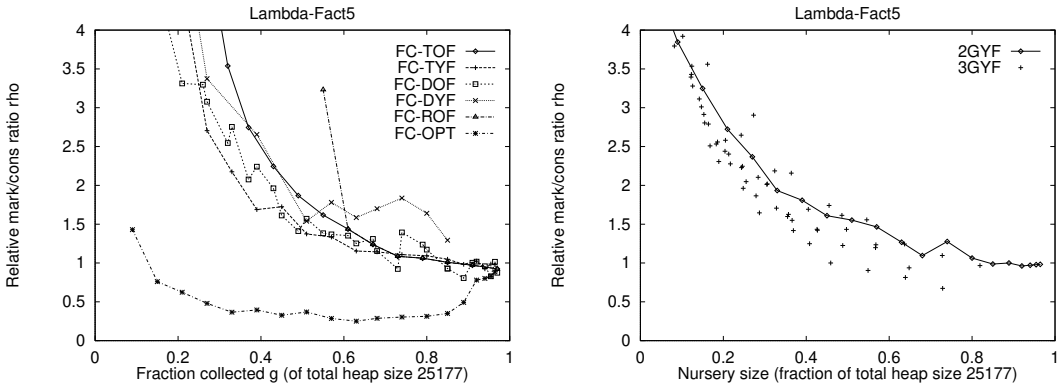
(b) GYF schemes

Figure 5.256. Copying cost, with FC-OPT: Lambda-Fact5, $V = 16943$.



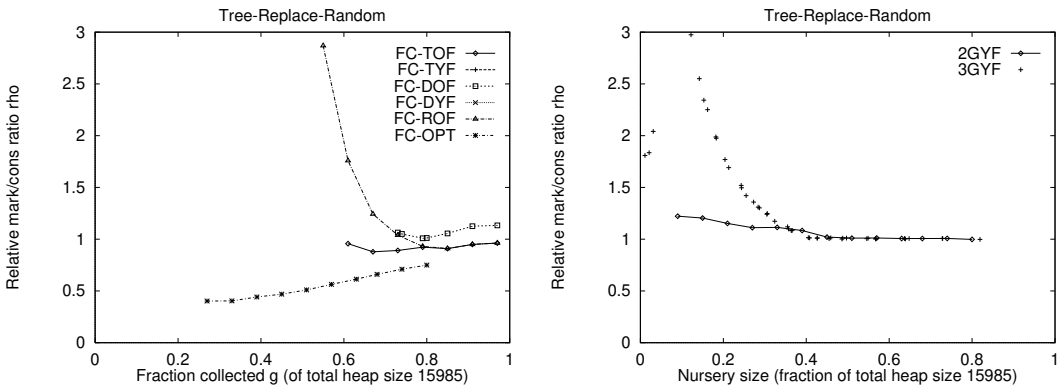
(a) FC schemes including OPT (b) GYF schemes

Figure 5.257. Copying cost, with FC-OPT: Lambda-Fact5, $V = 20654$.



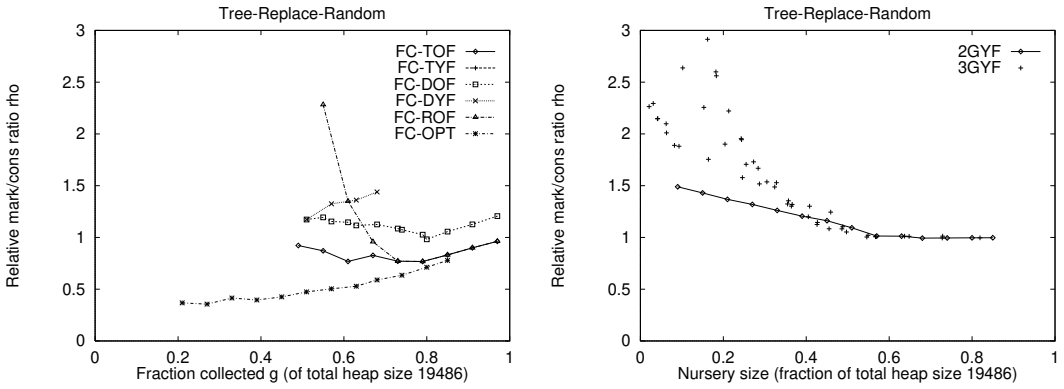
(a) FC schemes including OPT (b) GYF schemes

Figure 5.258. Copying cost, with FC-OPT: Lambda-Fact5, $V = 25177$.



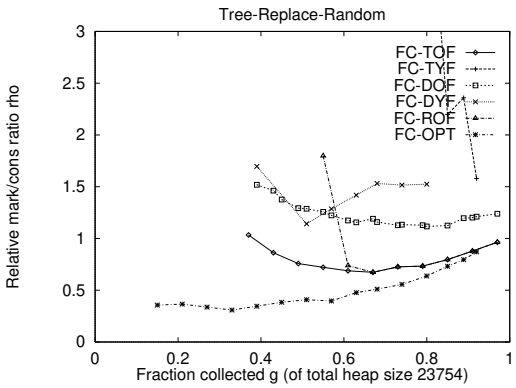
(a) FC schemes including OPT (b) GYF schemes

Figure 5.259. Copying cost, with FC-OPT: Tree-Replace-Random, $V = 15985$.

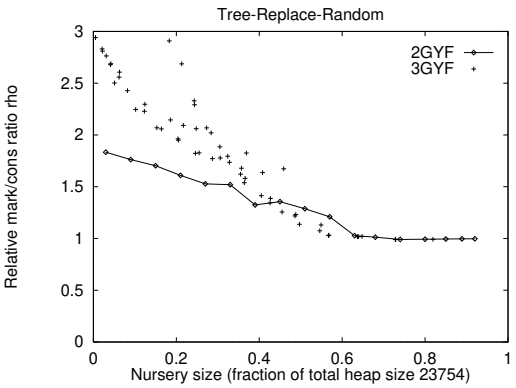


(a) FC schemes including OPT (b) GYF schemes

Figure 5.260. Copying cost, with FC-OPT: Tree-Replace-Random, $V = 19486$.

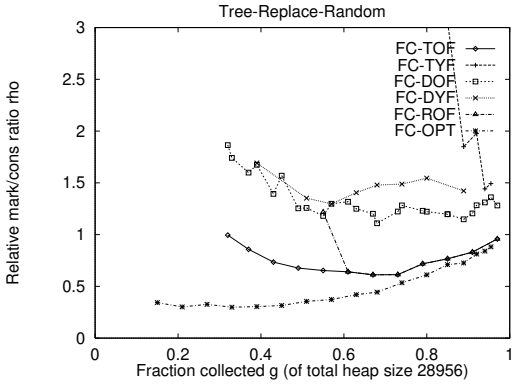


(a) FC schemes including OPT

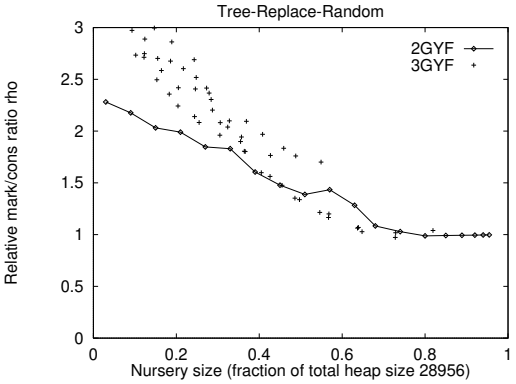


(b) GYF schemes

Figure 5.261. Copying cost, with FC-OPT: Tree-Replace-Random, $V = 23754$.

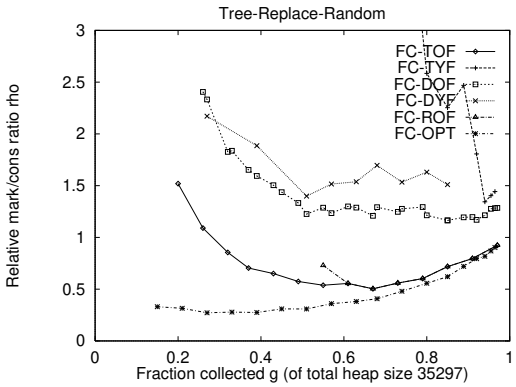


(a) FC schemes including OPT

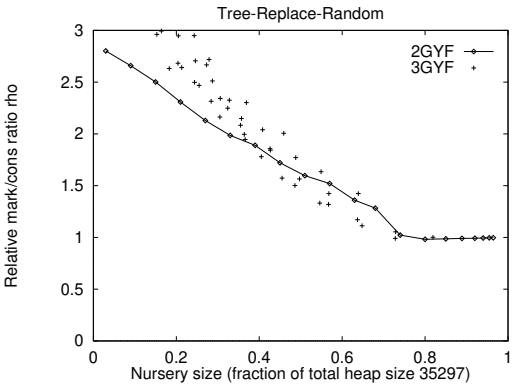


(b) GYF schemes

Figure 5.262. Copying cost, with FC-OPT: Tree-Replace-Random, $V = 28956$.

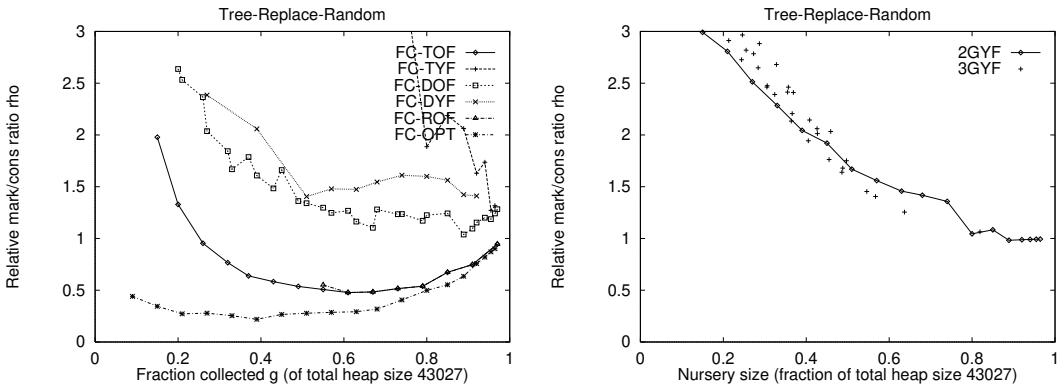


(a) FC schemes including OPT



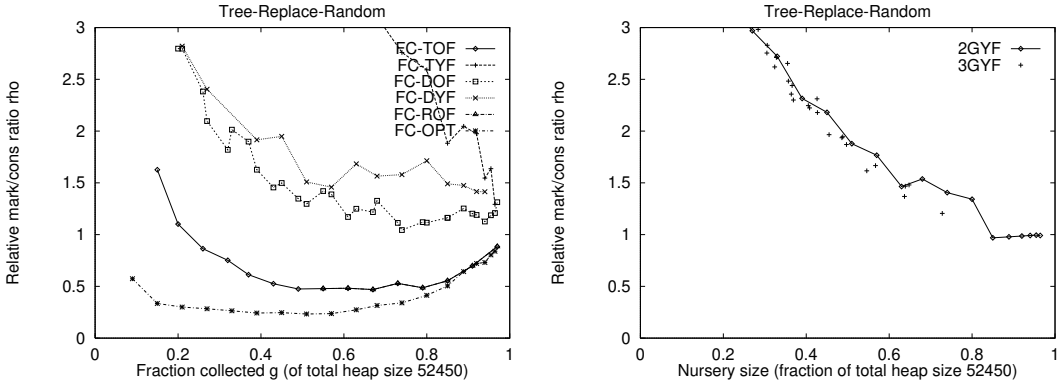
(b) GYF schemes

Figure 5.263. Copying cost, with FC-OPT: Tree-Replace-Random, $V = 35297$.



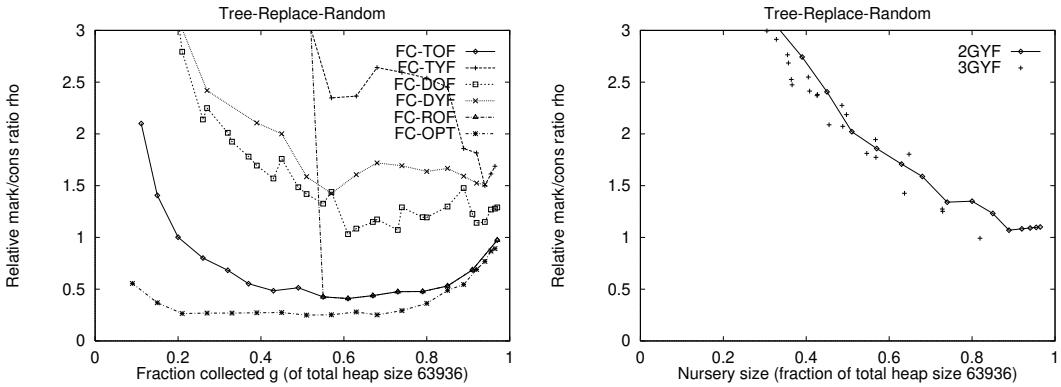
(a) FC schemes including OPT (b) GYF schemes

Figure 5.264. Copying cost, with FC-OPT: Tree-Replace-Random, $V = 43027$.



(a) FC schemes including OPT (b) GYF schemes

Figure 5.265. Copying cost, with FC-OPT: Tree-Replace-Random, $V = 52450$.



(a) FC schemes including OPT (b) GYF schemes

Figure 5.266. Copying cost, with FC-OPT: Tree-Replace-Random, $V = 63936$.

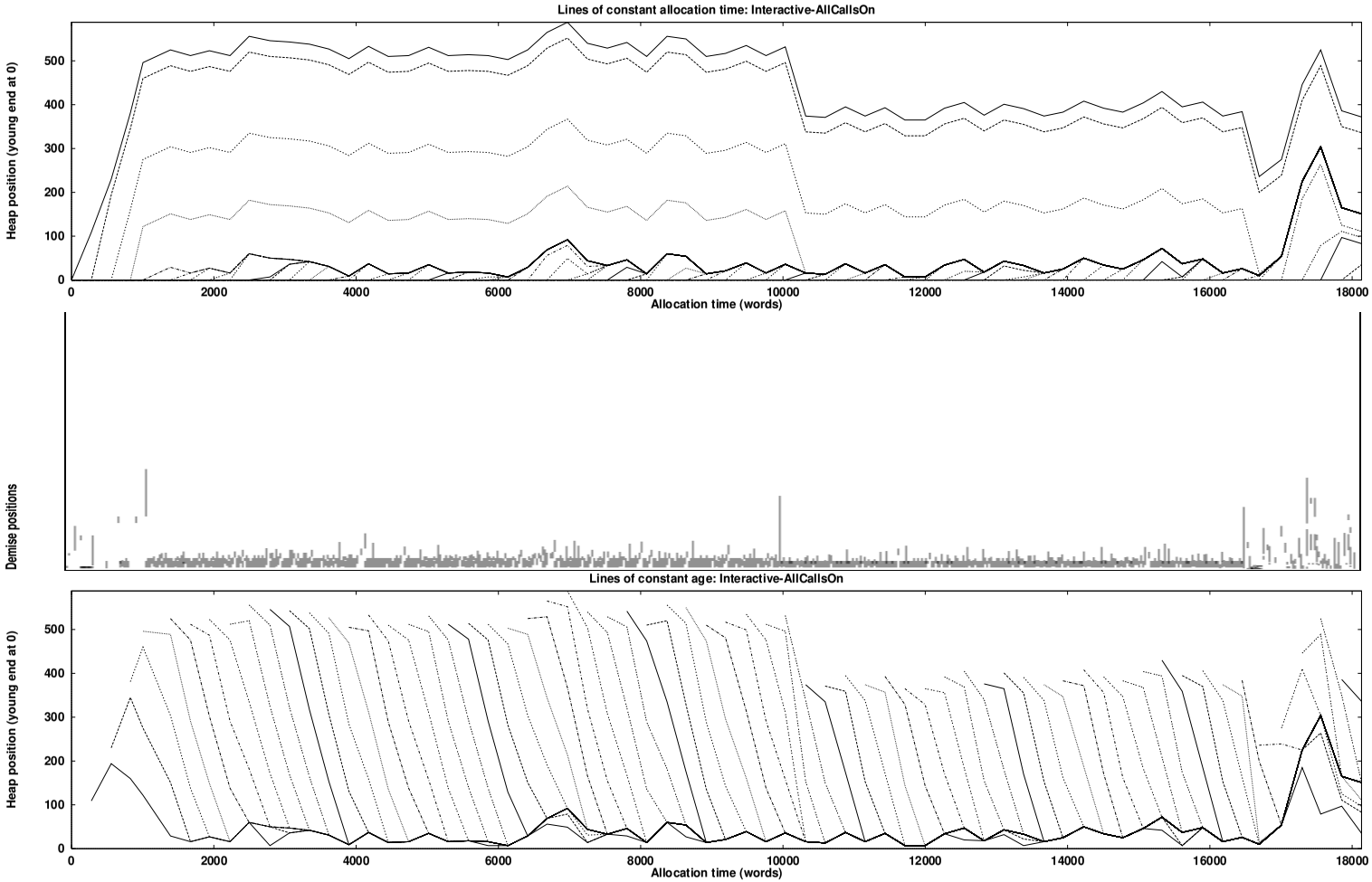


Figure 5.267. Demise graphs: Interactive-AllCallsOn.

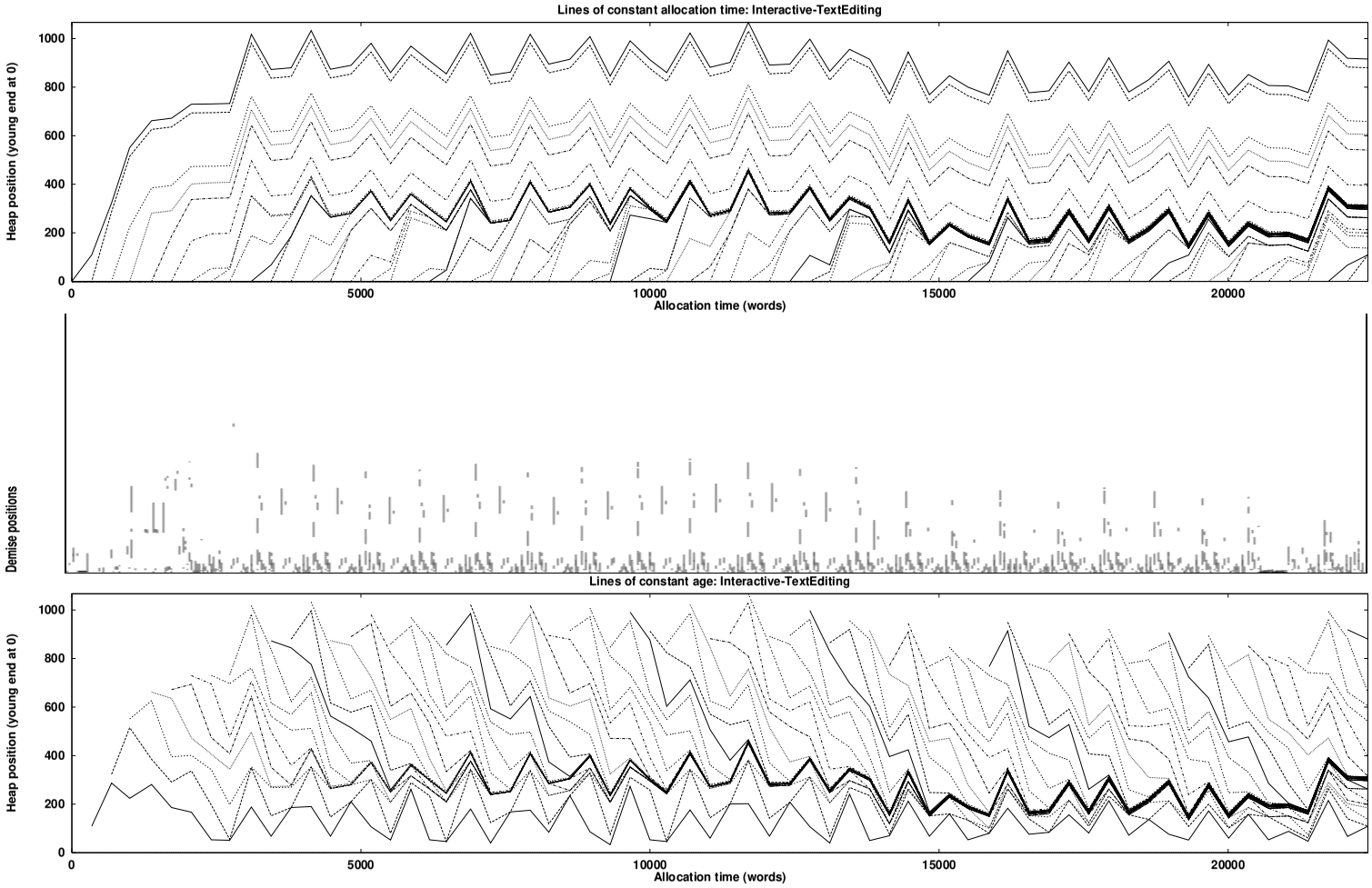


Figure 5.268. Demise graphs: Interactive-TextEditing.

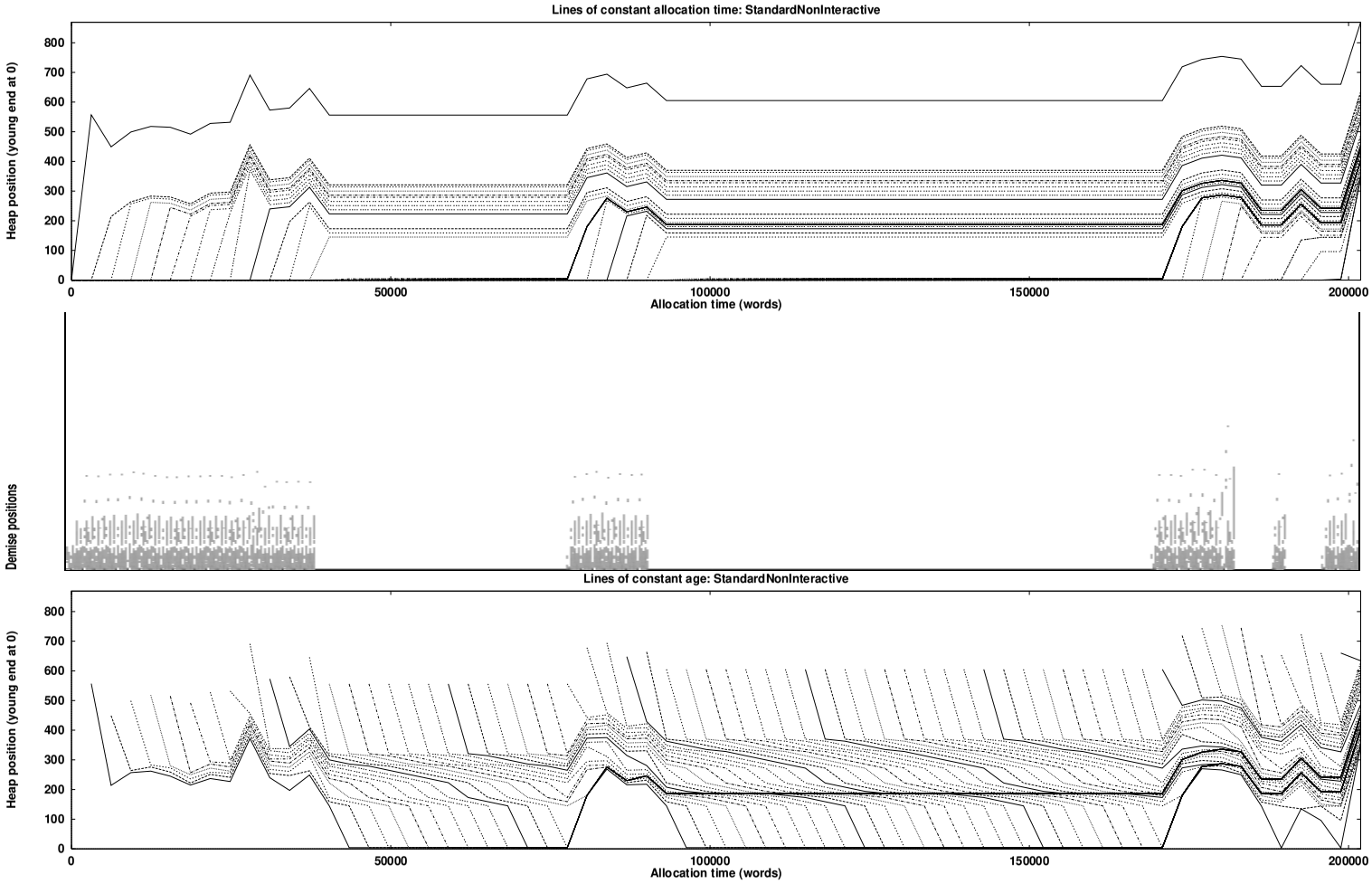


Figure 5.269. Demise graphs: StandardNonInteractive.

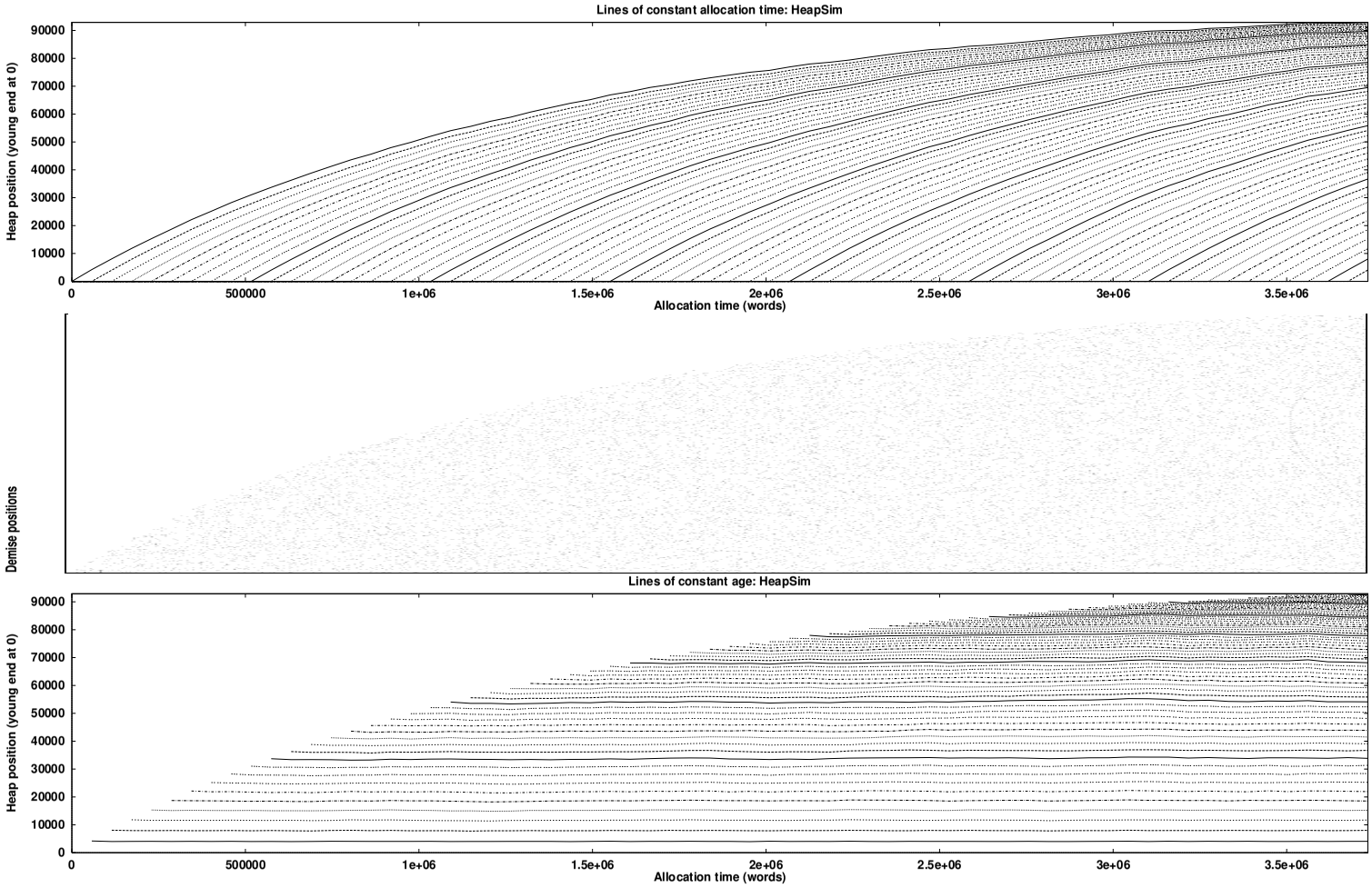


Figure 5.270. Demise graphs: HeapSim.

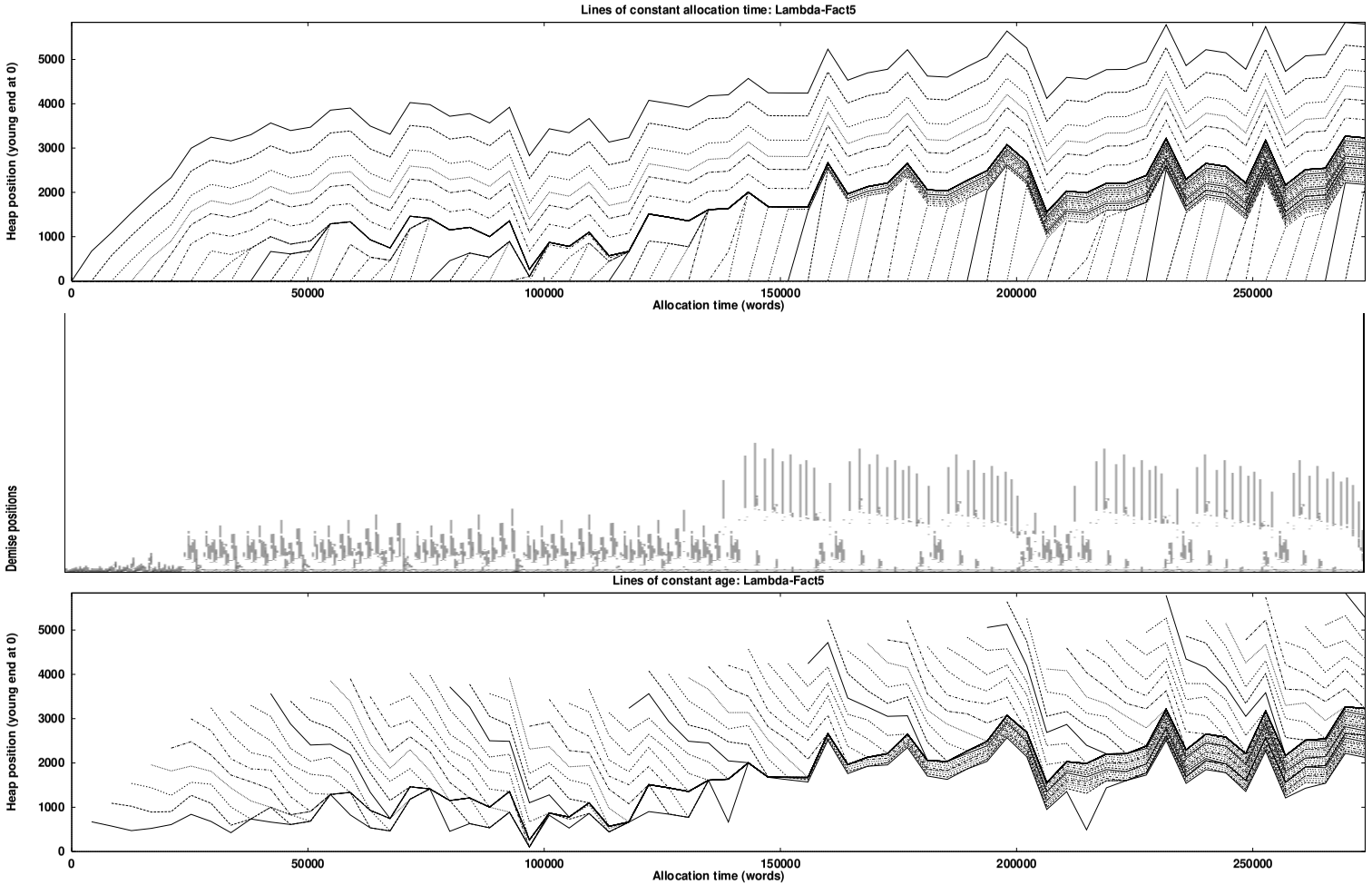


Figure 5.271. Demise graphs: Lambda-Fact5.

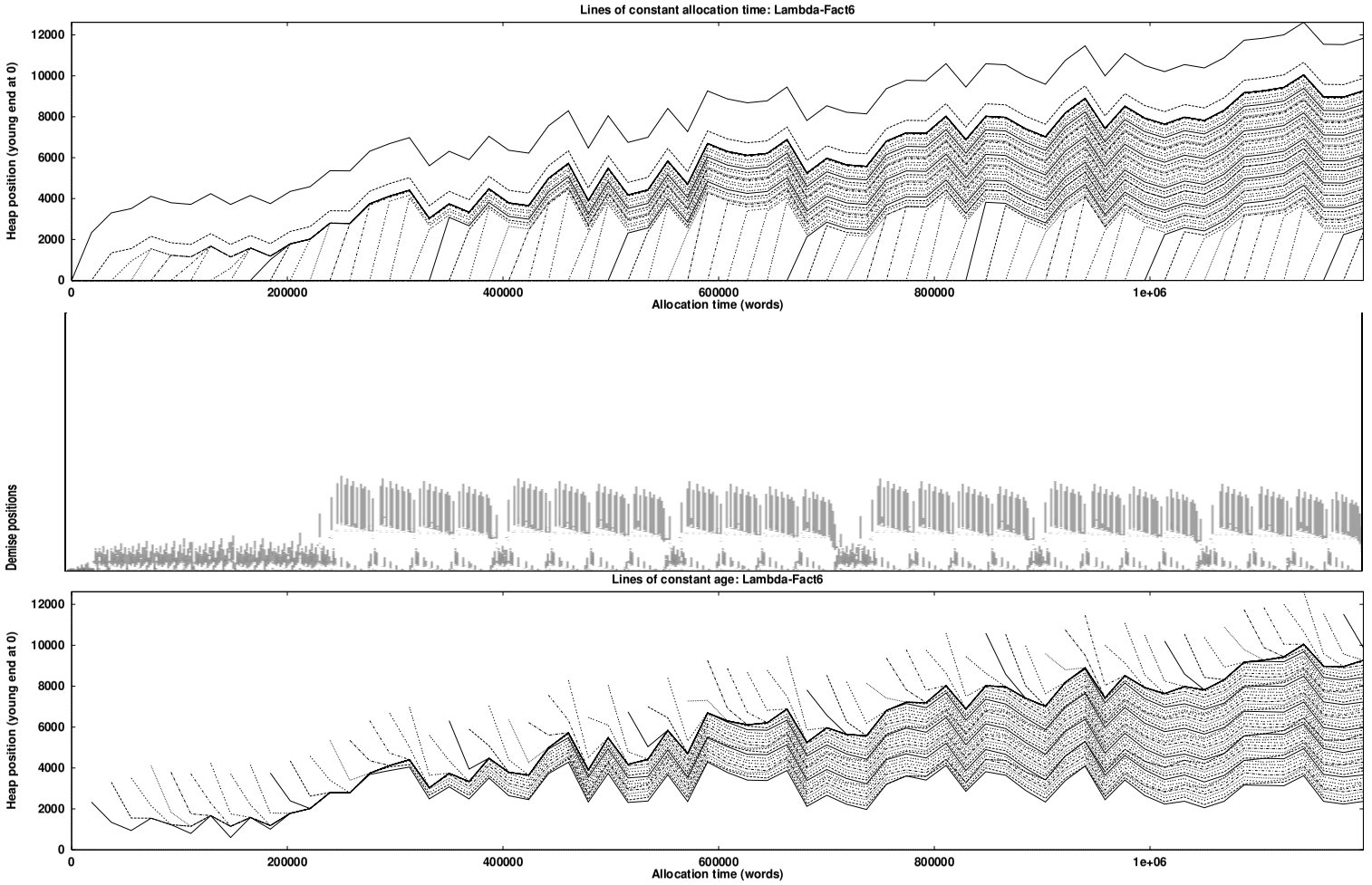


Figure 5.272. Demise graphs: Lambda-Fact6.

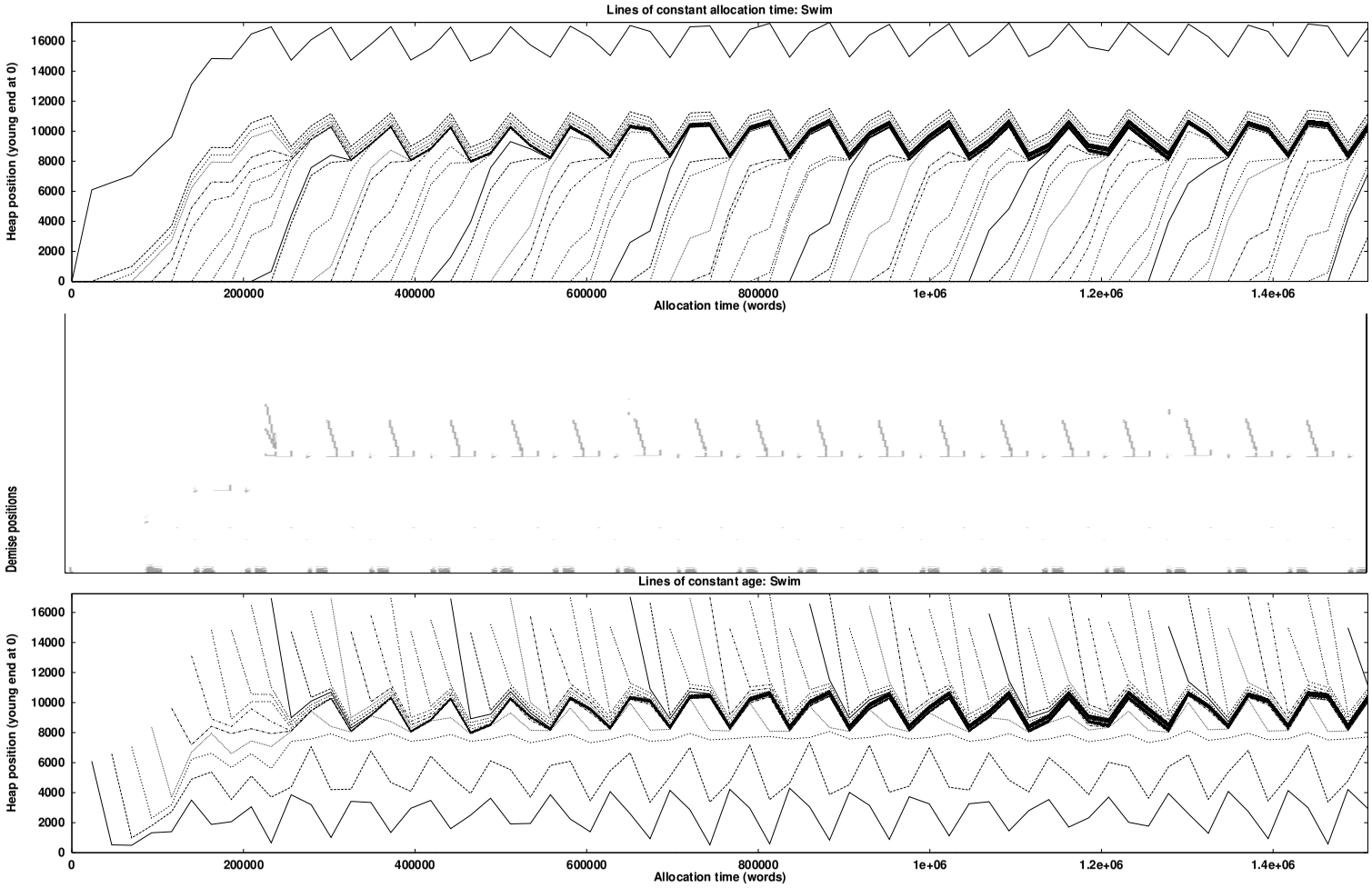


Figure 5.273. Demise graphs: Swim.

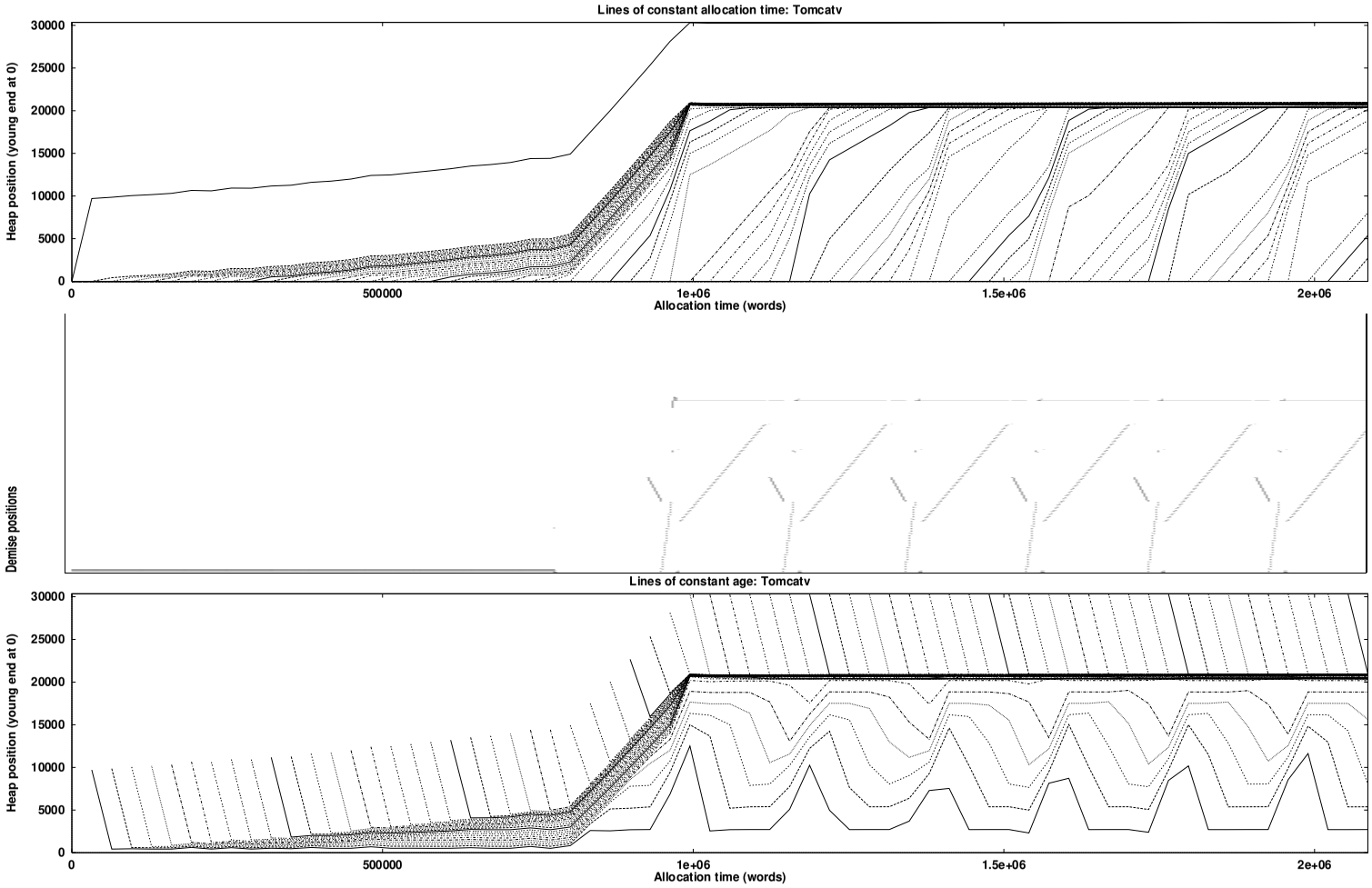


Figure 5.274. Demise graphs: Tomcatv.

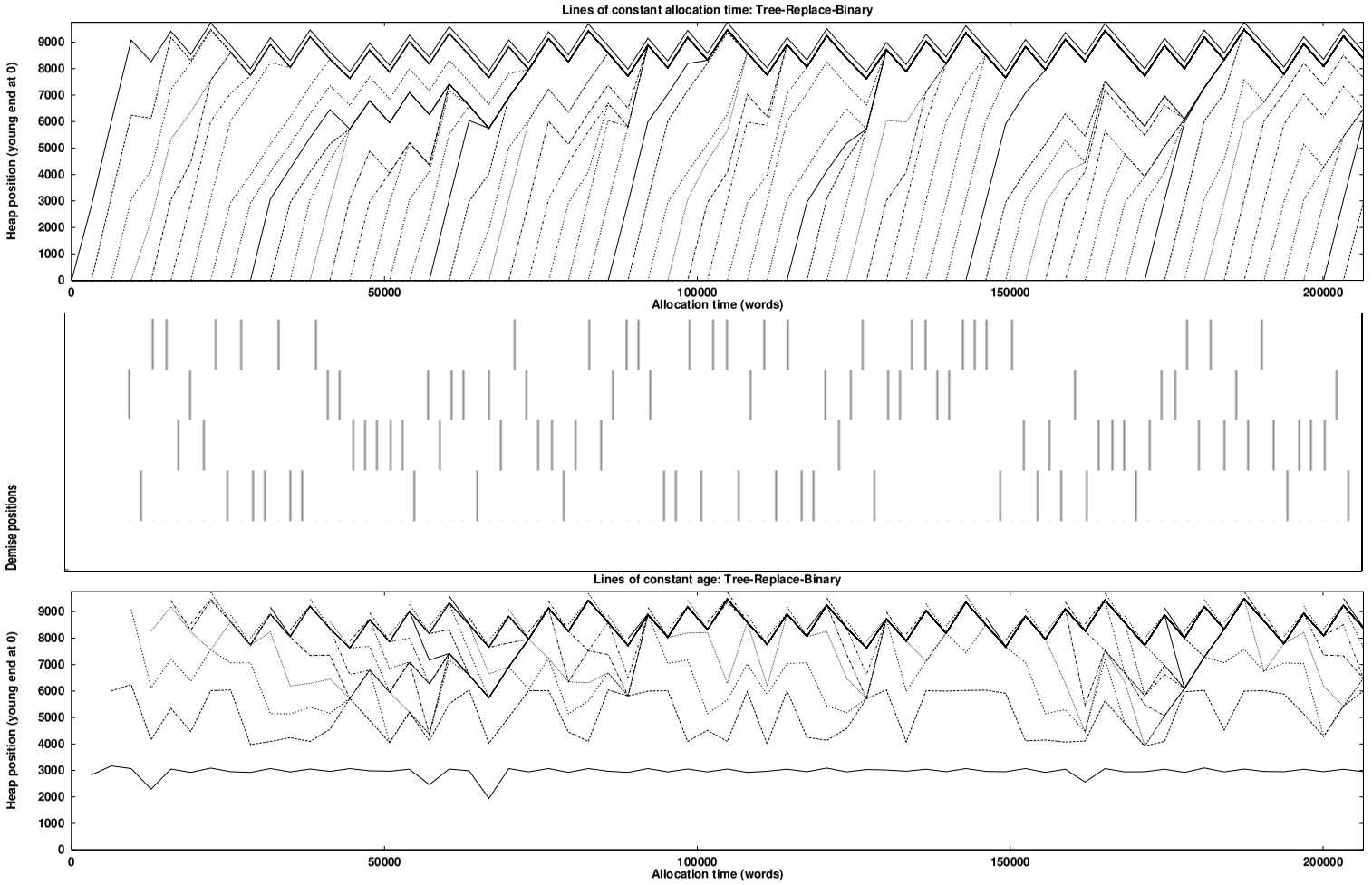


Figure 5.275. Demise graphs: Tree-Replace-Binary.

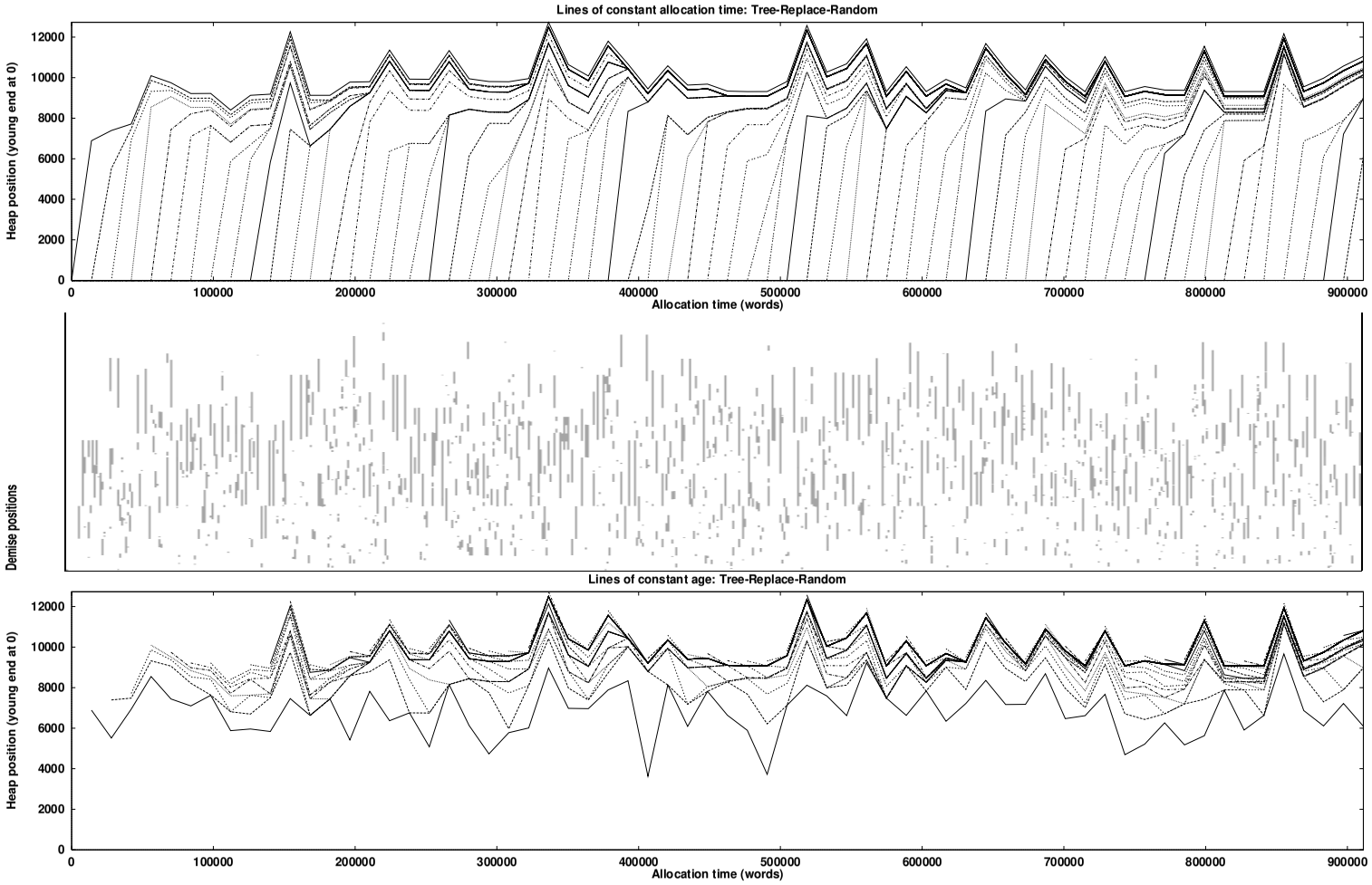


Figure 5.276. Demise graphs: Tree-Replace-Random.

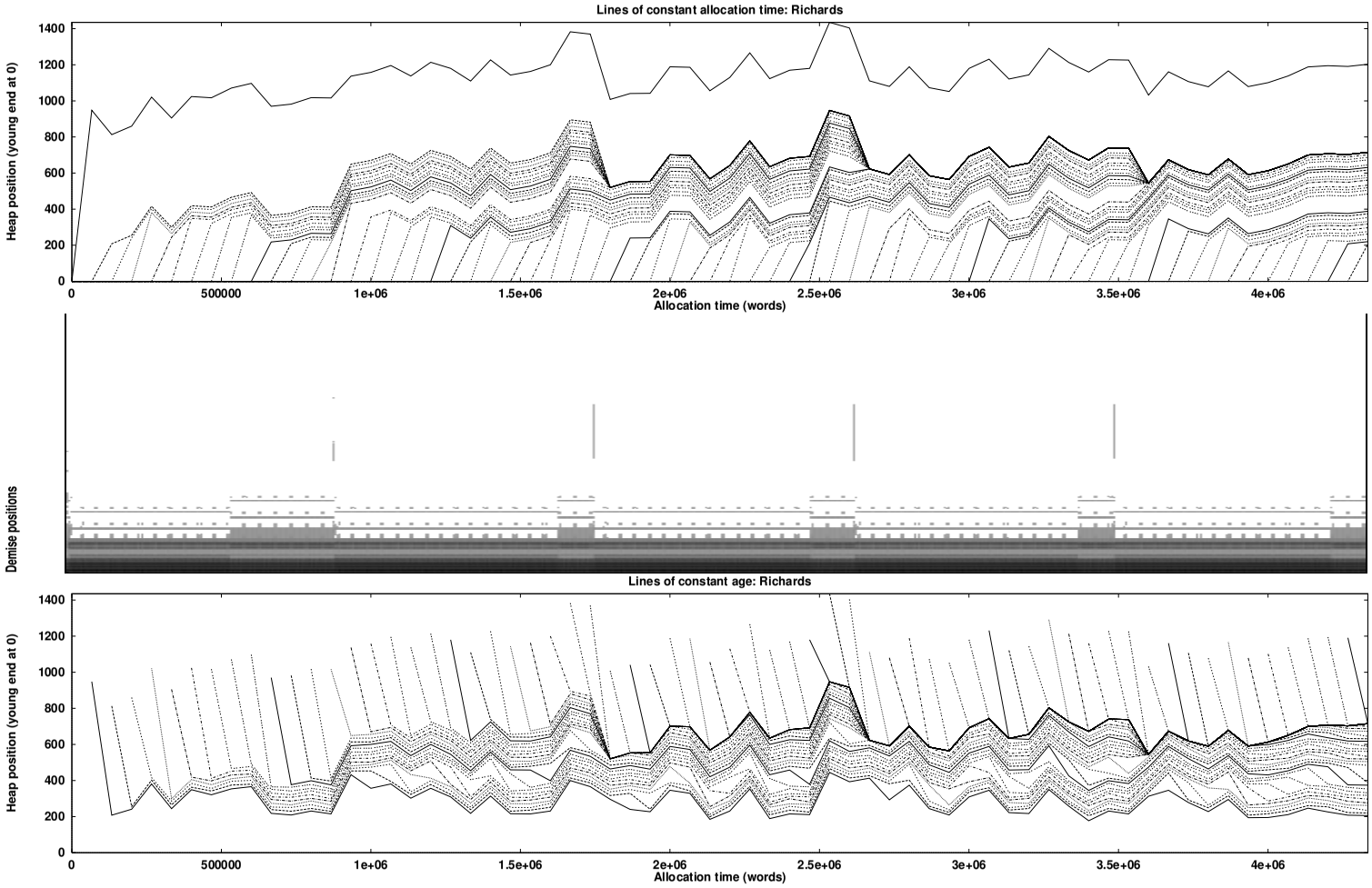


Figure 5.277. Demise graphs: Richards.

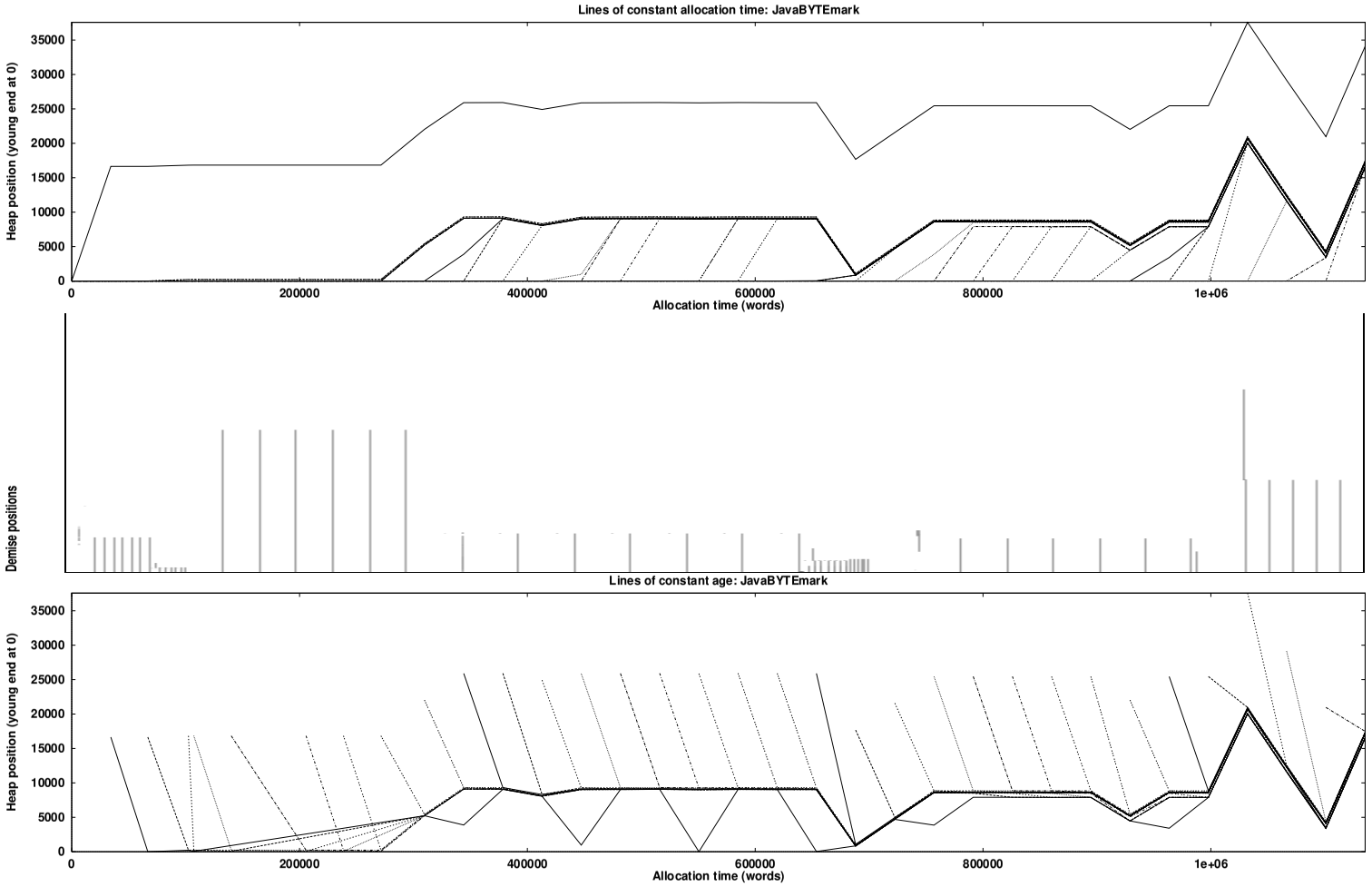


Figure 5.278. Demise graphs: JavaBYTEmark.

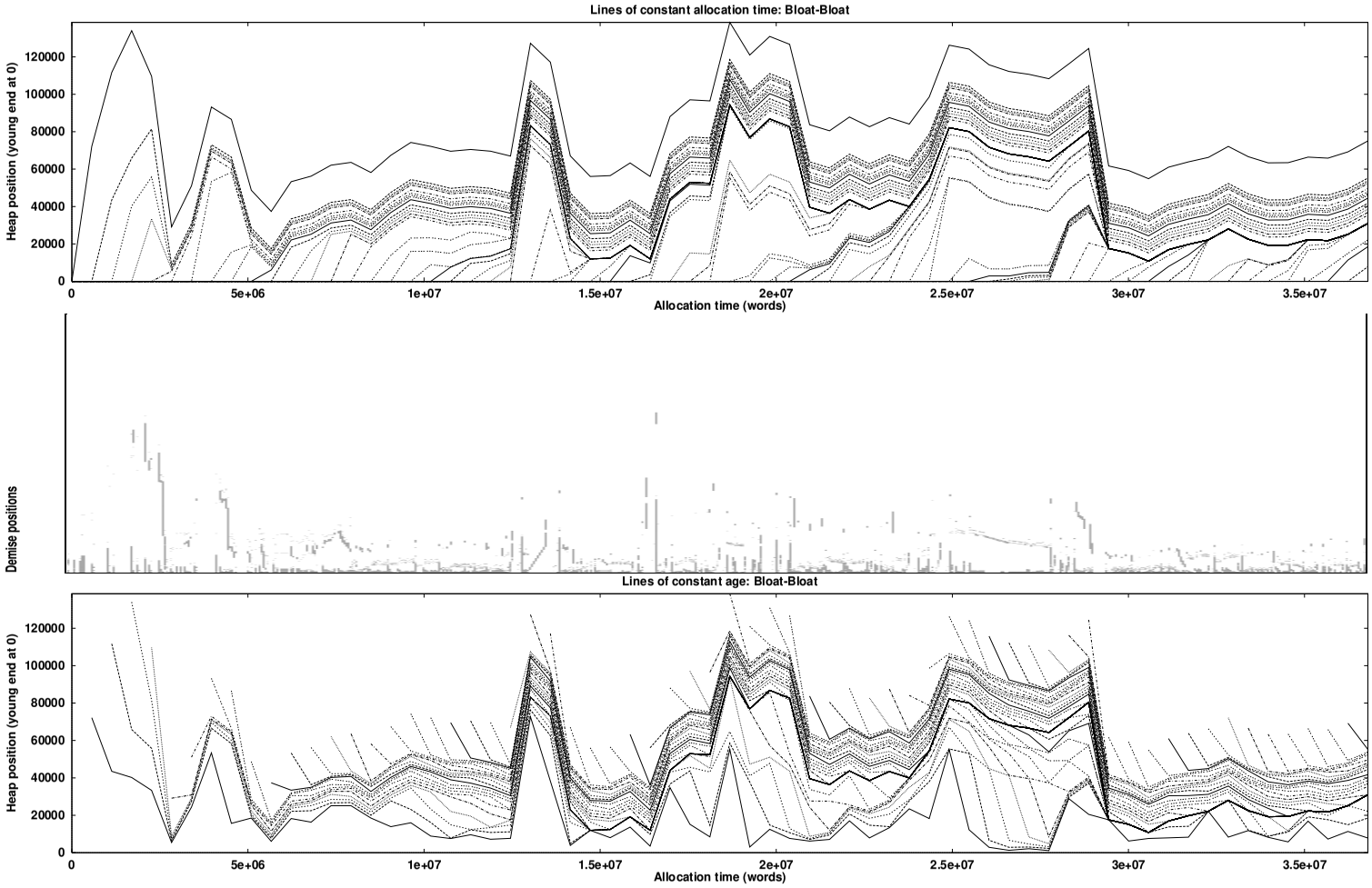


Figure 5.279. Demise graphs: Bloat-Bloat.

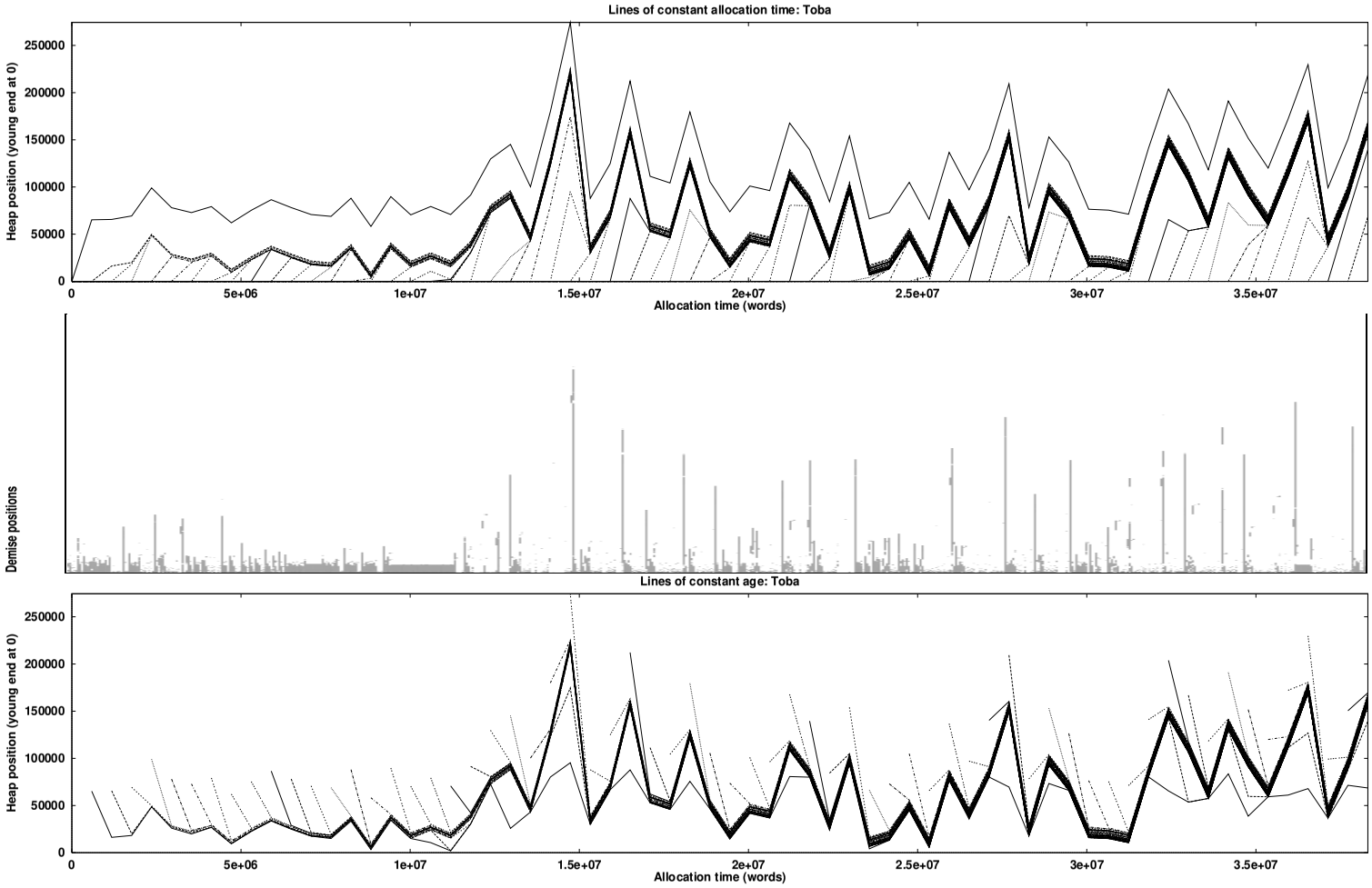


Figure 5.280. Demise graphs: Toba.

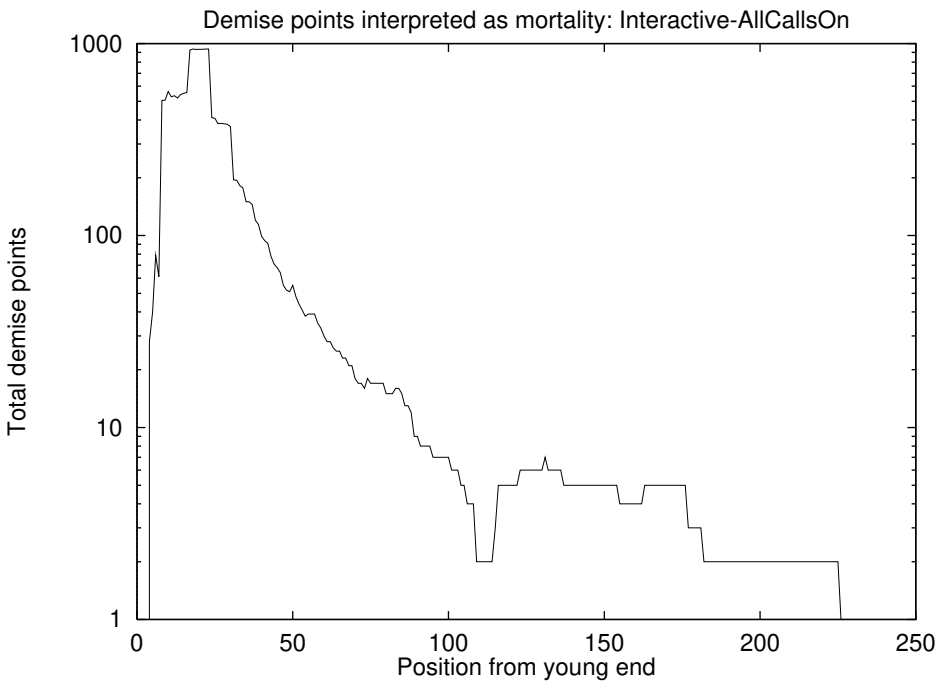


Figure 5.281. Demise position as mortality: Interactive-AllCallsOn.

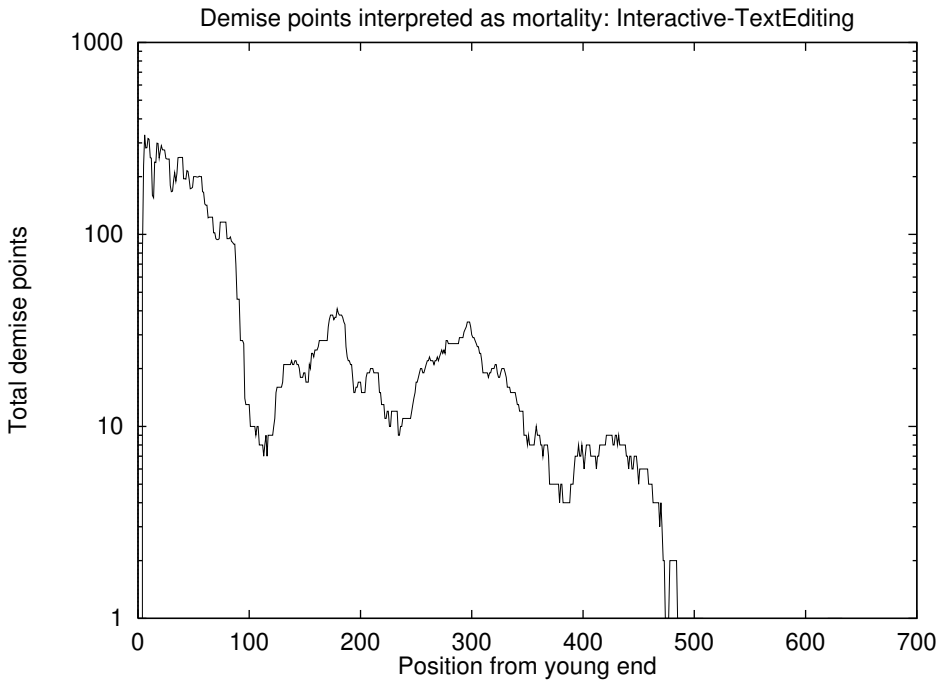


Figure 5.282. Demise position as mortality: Interactive-TextEditing.

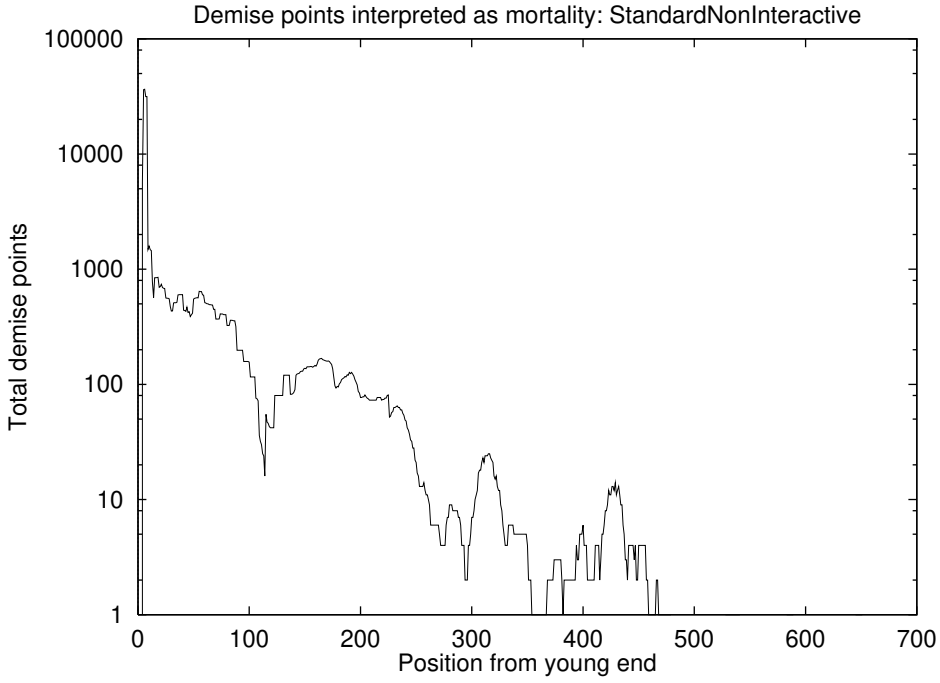


Figure 5.283. Demise position as mortality: StandardNonInteractive.

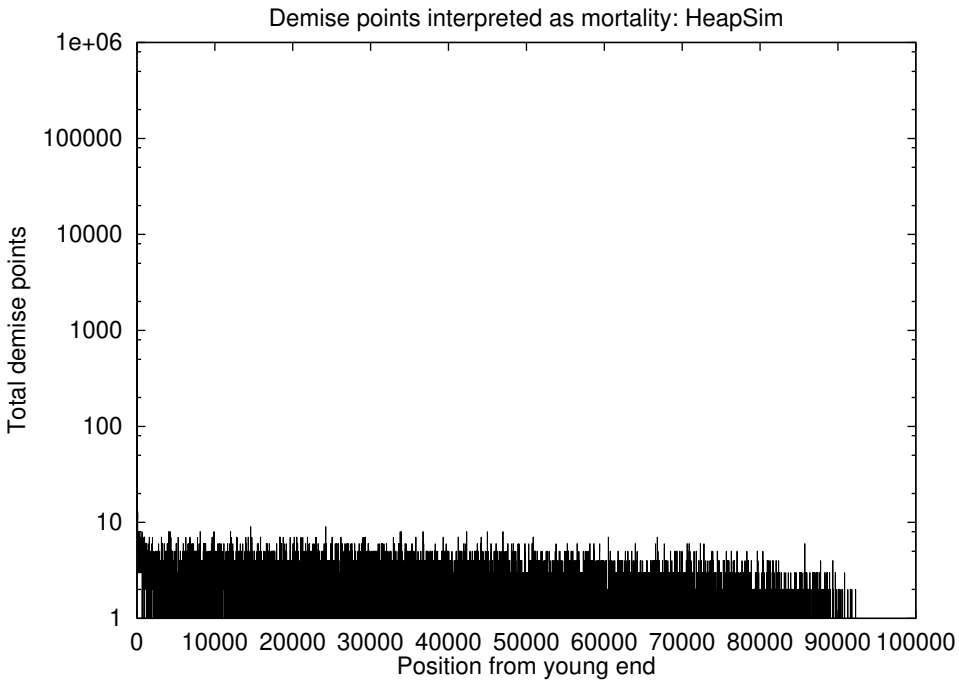


Figure 5.284. Demise position as mortality: HeapSim.

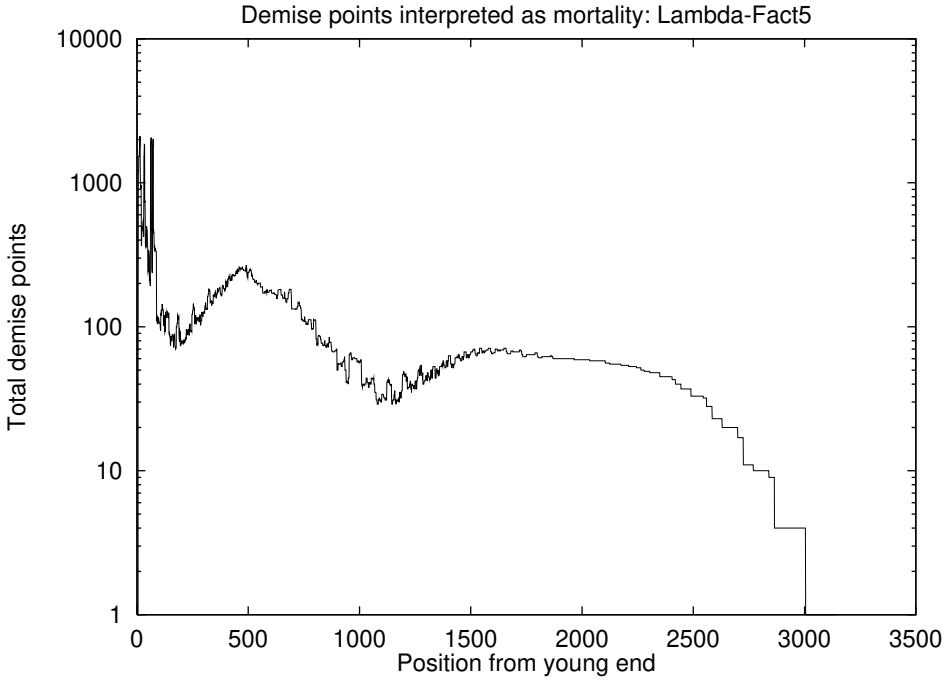


Figure 5.285. Demise position as mortality: Lambda-Fact5.

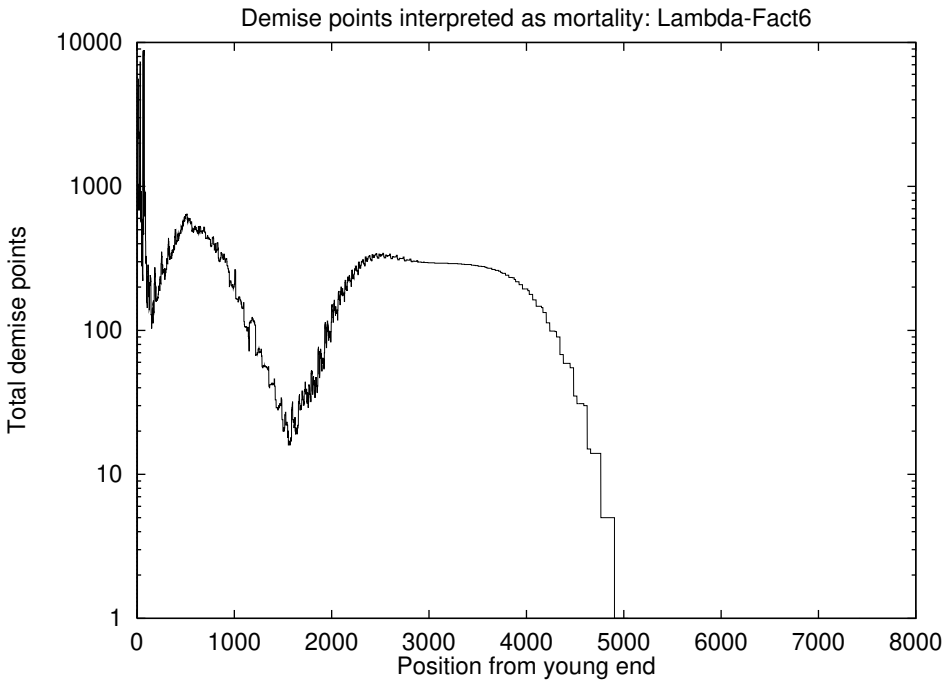


Figure 5.286. Demise position as mortality: Lambda-Fact6.

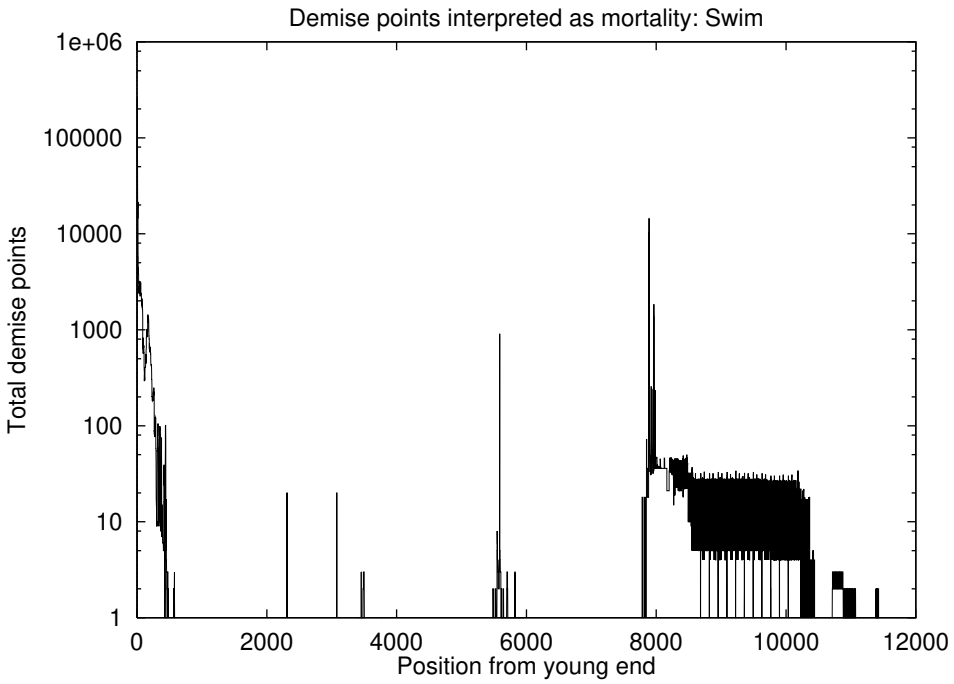


Figure 5.287. Demise position as mortality: Swim.

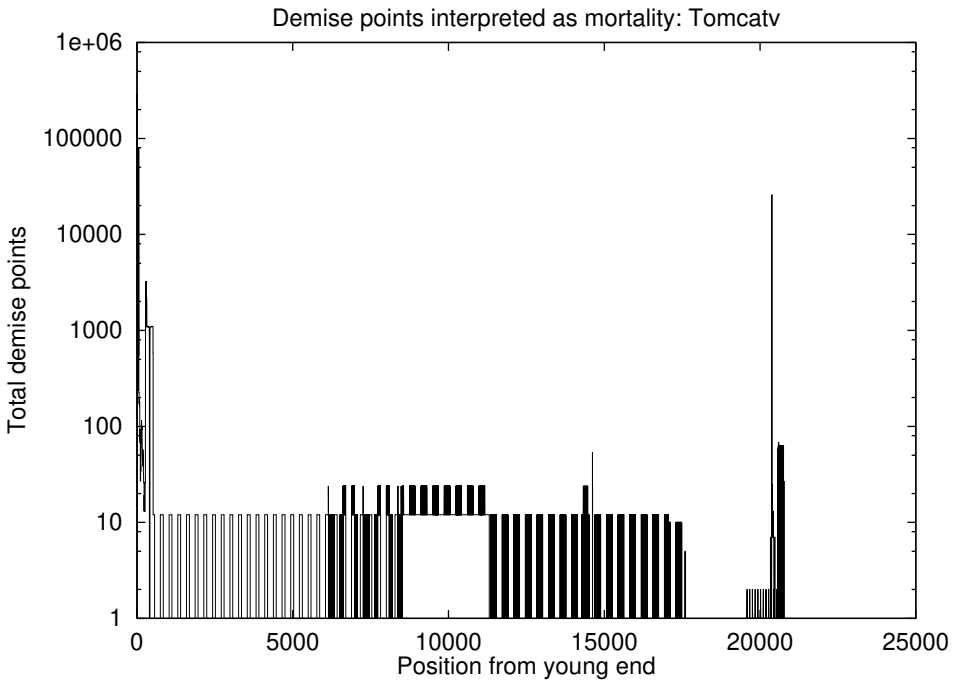


Figure 5.288. Demise position as mortality: Tomcatv.

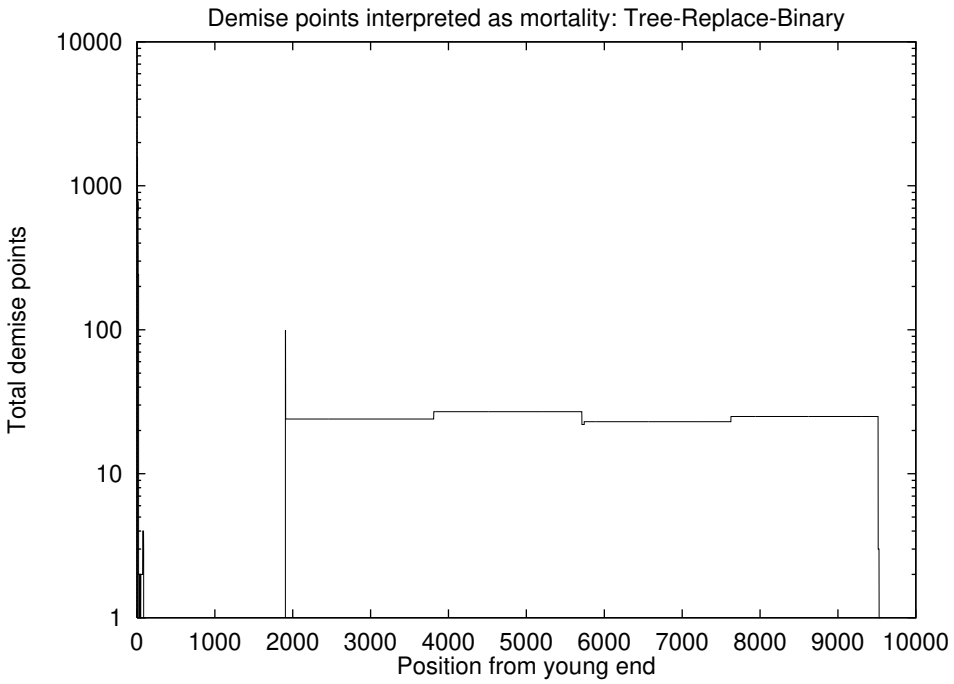


Figure 5.289. Demise position as mortality: Tree-Replace-Binary.

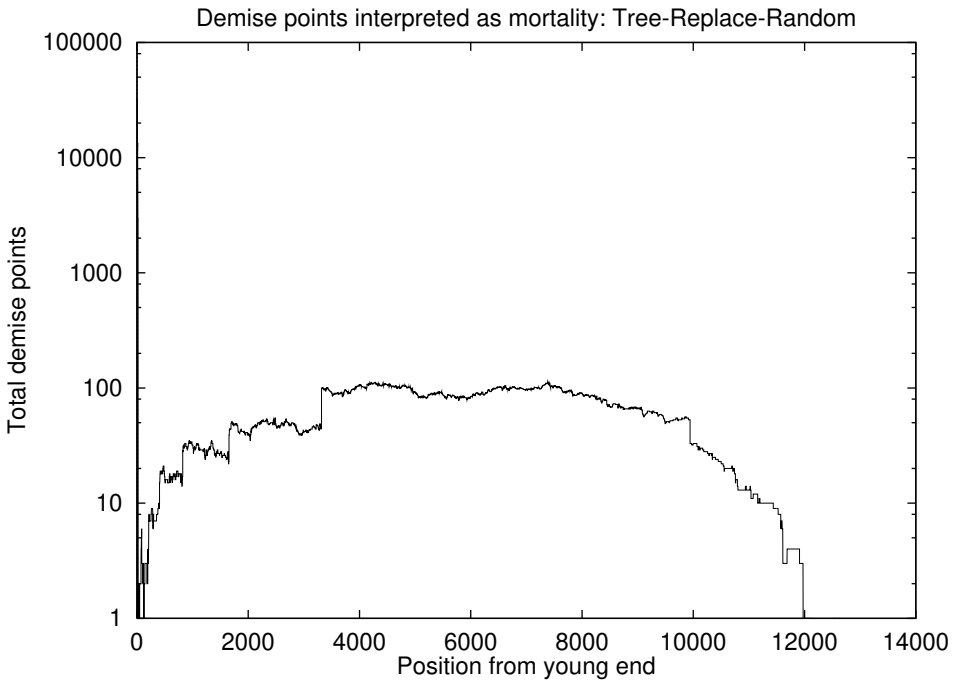


Figure 5.290. Demise position as mortality: Tree-Replace-Random.

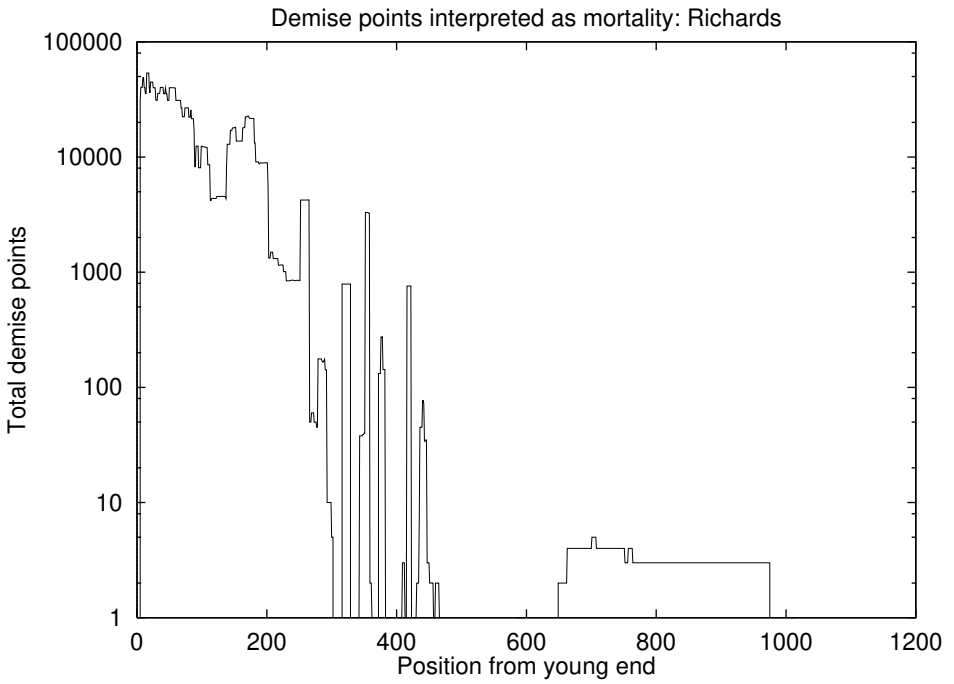


Figure 5.291. Demise position as mortality: Richards.

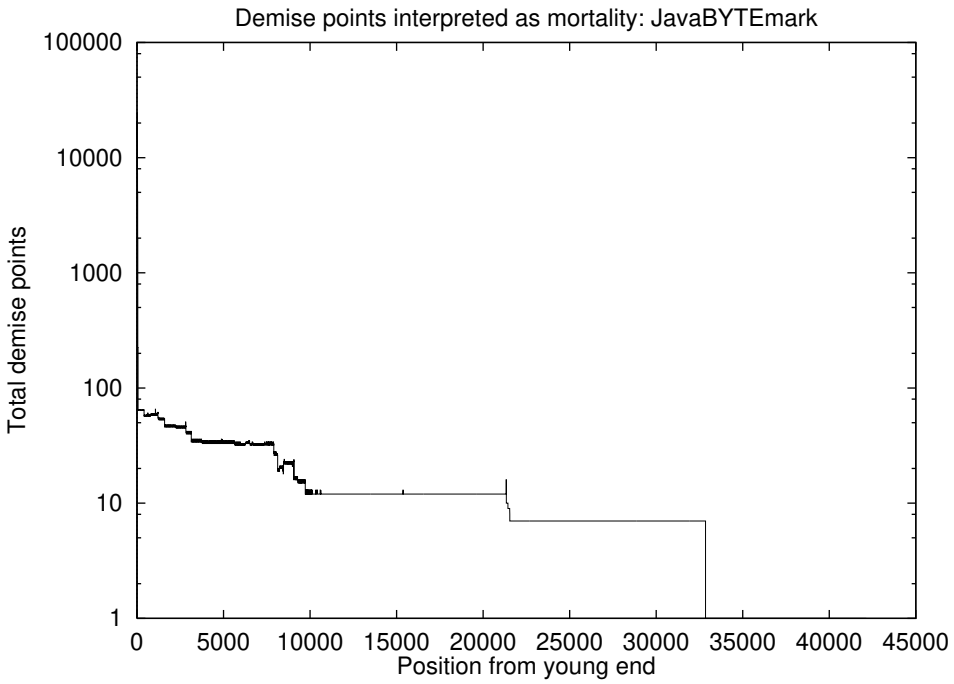


Figure 5.292. Demise position as mortality: JavaBYTEmark.

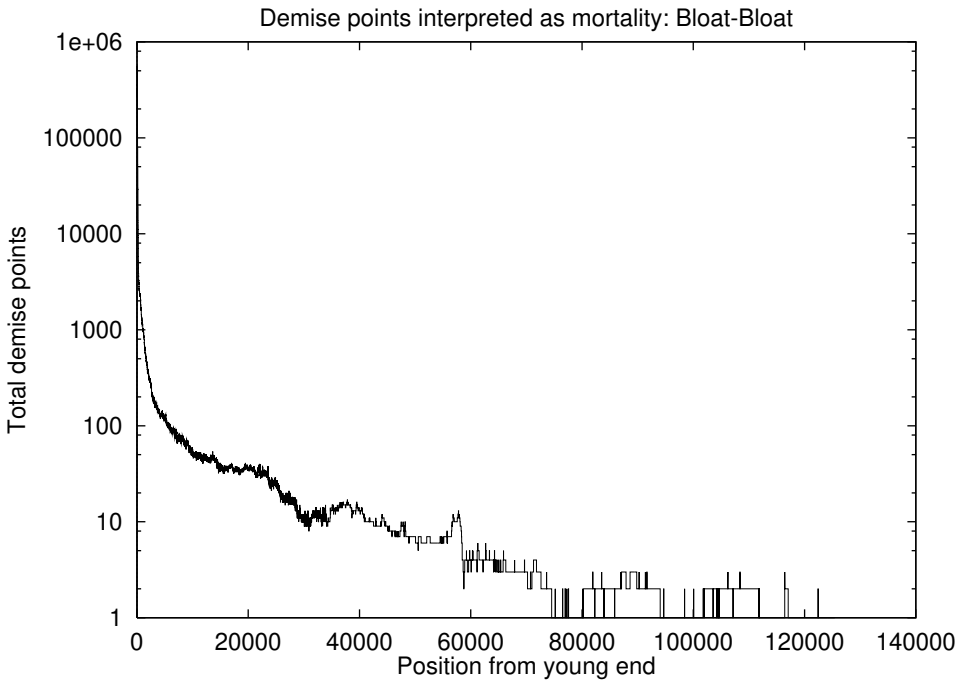


Figure 5.293. Demise position as mortality: Bloat-Bloat.

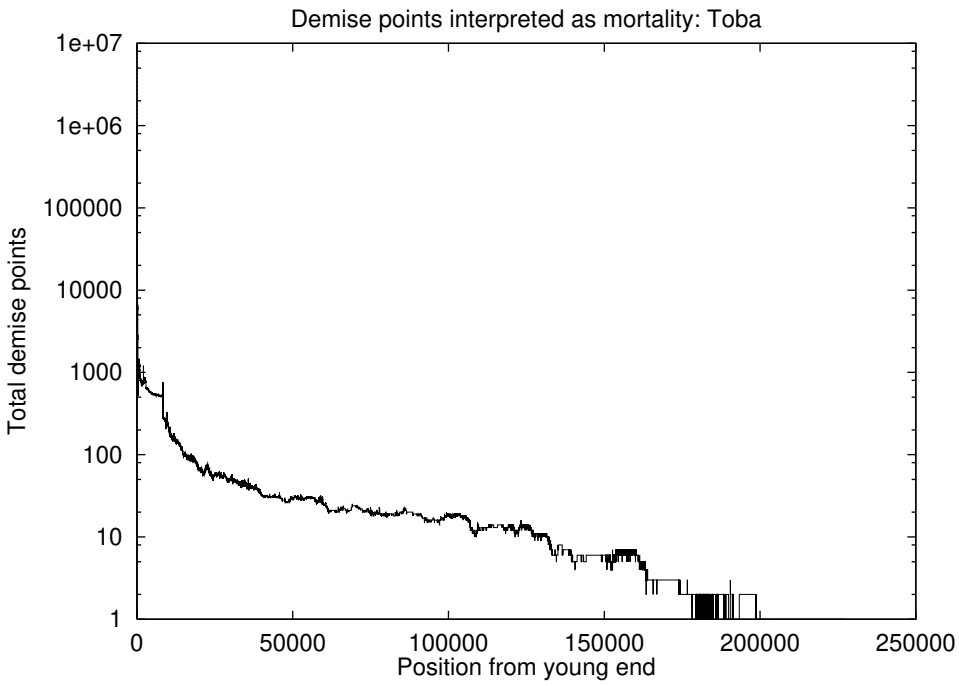


Figure 5.294. Demise position as mortality: Toba.

CHAPTER 6

EVALUATION: COSTS OTHER THAN COPYING

In this chapter we examine the non-copying costs of collection, focusing on the cost of maintaining the information needed for accurate region-based collection: the write barrier and remembered set overheads. The evaluation uses a realistic, block-allocation collection simulator, based on our prototype implementation. We compare the generational youngest-first collector with the deferred older-first algorithm, which was shown in the preceding chapter to be the most promising with respect to copying cost among the alternatives to generational collection that we considered.

6.1 Method for evaluating the pointer maintenance cost

We use an instrumented version of the block simulator (Section 4.3.3) to obtain the statistics of pointer management operations. We discussed in Section 3.2 the design of pointer management for age-based collection, and noted alternative models of pointer store filtering and of remembered set organization. The actual costs of operations depend on these alternative design decisions. Indeed, the operation counts are not invariant themselves: if two filtering tests are executed in sequence, the outcome of the first influences the number of times the second is executed. The division of labor between the run time and the garbage collection time is also a matter of choice. Here we use for the generational collector the traditional model of operation with a unified remembered set and generational filtering, and we show that it is very efficient. For the DOF collector we use block-local and directional filtering. All of the operation counts reported are accurate for these design choices, while some of them are also universally applicable regardless of the design choice.

For the write barrier, we report the operation counts for pointer stores, and the number of non-null pointer stores. Non-null stores are further divided into stores filtered by the specified filtering mechanism, and those not filtered. The non-filtered stores are divided into unique and duplicate, with respect to the prior content of the remembered set into which the stores are inserted. Note that the write-barrier statistics include both the run time and the garbage-collection time pointer stores.

For remembered set processing at garbage-collection time, we report the number of pointers from the uncollected region into the collected region, as well as the number of remembered set entries actually processed, which can be larger.

6.1.1 Results

In Table 6.1 we report the statistics that are independent of the collection algorithm or its configuration. The columns "Words alloc." and "Ptr. st." repeat the information of Table 4.1. The column "Alloc./st." gives the ratio of the number of words allocated to the number of pointer stores for the program. This value is an indication, albeit crude, of the frequency of pointer manipulations, or the "object-orientedness" of the program, as well as of the expected influence of pointer maintenance operations on overall memory management performance. The last two columns give the number of non-null pointer stores as an absolute number, and as a percentage of all pointer stores. In many implementations of the write barrier, the check for a null pointer is a separate first step. As we can see in our numbers, there are few null pointers, and this test rarely eliminates the pointer from consideration. Hence a write barrier that integrates the null-pointer test into a general address test, as proposed in Section 3.2.4, is to be preferred.

In Tables 6.2–6.15, pp. 295–382 (at the end of this chapter) we report the pointer management statistics for the different configurations of the FC-DOF and the 2GYF collectors. The first group of tables (Tables 6.2–6.8) is for the FC-DOF collector. The first three columns in Tables 6.2–6.8 indicate the configuration. The first column, having values between 9 and 20,

Table 6.1. Properties of benchmarks used: run-time allocation and pointer stores

Benchmark	Words alloc.	Ptr. st.	Alloc./st.	Non-null ptr. st.	%
Smalltalk					
Interactive-AllCallsOn	18 359	855	21.47	134	15.7
Interactive-TextEditing	22 747	3 509	6.48	3 241	92.4
StandardNonInteractive	204 954	7 251	28.26	5 882	81.1
HeapSim	3 791 306	30 298	125.13	30 233	99.8
Lambda-Fact5	277 940	91 877	3.03	87 403	95.1
Lambda-Fact6	1 216 247	404 670	3.01	387 037	95.6
Swim	1 533 642	134 355	11.41	120 427	89.6
Tomcatv	2 117 849	286 032	7.40	285 056	99.6
Tree-Replace-Binary	209 600	39 642	5.29	32 986	83.2
Tree-Replace-Random	925 236	168 513	5.49	140 029	83.1
Richards	4 400 543	763 626	5.76	611 767	80.1
Java					
JavaBYTEmark	1 161 949	49 061	23.68	48 835	99.5
Bloat-Bloat	37 364 458	4 927 497	7.58	4 376 798	88.8
Toba	38 897 724	3 027 982	12.85	2 944 672	97.2
Geometric mean			10.27		80.2

gives the block size: for value k, the block size is 2^k bytes. The second column gives the allotted heap size, in terms of blocks. The third column gives the size of the collected region (the window size), also in terms of blocks. The fourth column gives the absolute copying cost as the number of words copied. The fifth column gives the number of collector invocations. The sixth column gives the number of remembered set insertion operations at the write barrier, i.e., the number of pointer stores that were *not* filtered out. The seventh column gives the number of insertion operations that are not duplicates, and hence actually add entries to remembered sets. The eighth column gives the *sum* of the sizes of all remembered sets processed at garbage collection time (i.e., the sets belonging to blocks within the collected region). The ninth column gives the sum over all collections of the sizes of filtered unions of sets within the collected region.

The design used to produce the statistics is the following. Each block has its own remembered set (Section 3.2.2). The write barrier code is executed on every pointer store, both at run time and at garbage collection time. The write barrier consists of three filtering checks: first,

the check for null pointers; second, the check for block-local pointers; and third, the directional check (as in Figures 3.15 and 3.16). If none of the checks eliminates the pointer store, then the write-barrier code inserts it into the remembered set of the pointer target block, and adds it to the tally for table column six, "W.b. insert." The remembered set may already contain an entry recording the same pointer source address; otherwise, a new entry is added to the remembered set, and to the tally for table column seven, "W.b. add."

At collection time, following the processing of global roots, the collector processed the remembered sets of the blocks in the collected region one by one. Table column eight, "R.s. proc," reports the sum of the sizes of all remembered sets processed over all collections. On the other hand, the sum of the sizes of remembered sets for the collected region can be greater than the number of pointers actually crossing into the collected region from the uncollected region. The latter number is computed as the union of the remembered sets involved, less any pointers that have both source and target in the collected region. The sum of these numbers over all collections is reported in column nine, "R.s. proc."

The second group of tables (Tables 6.9–6.15) is for the 2GYF collector. The first four columns in Tables 6.9–6.15 indicate the configuration. The first column gives the block size, the second column gives the allotted heap size in blocks, the third column gives the size of the younger generation (the nursery) in blocks, and the fourth column gives the size of the older generation in blocks. The fifth column gives the absolute copying cost as the number of words copied. The sixth column gives the number of collector invocations. The seventh column gives the number of remembered set insertion operations at the write barrier, and the eighth gives the number of these that are not duplicates. The ninth column gives the sum of the sizes of processed remembered sets.

The design used to produce the statistics is the following. There is a single remembered set, and it records the pointers from the older generation into the younger generation. The write barrier code is executed on every pointer store at run time. No write barrier is needed at garbage collection time, because, with only two generations, all objects are in the oldest

generation following a collection. The write barrier consists of two filtering checks: first, the check for null pointers; and second, the generational check. If neither filter eliminates the pointer store, it is inserted into the remembered set, and added to the tally for column seven, "W.b. insert." If the pointer source is not already present in the remembered set, a new entry is added to the set, and to column eight, "W.b. add."

At collection time, the collector processes the single remembered set. Table column nine, "R.s. proc", reports the sum, over all collection, of the sizes of the remembered set. There is no need for a "R.s. union" column for 2GYF, because its meaning is subsumed by "R.s. proc."

Each row of a table corresponds to one configuration of a collector, and thus to one simulation run. The configurations are ordered by block size, and for each block size, they are ordered by heap size. For a given heap size, the configurations are ordered by the size of the collected region (in the case of FC-DOF), and by the size of the younger generation (in the case of 2GYF).

Certain rows are printed in boldface for ease of reference in both the FC-DOF and the 2GYF tables; we shall use these configurations as examples to discuss the comparative costs of the two schemes.

6.1.2 Block size

We first examine the effect of block size on copying and pointer management costs. For the FC-DOF collector, we find that increasing the block size reduces the operation counts for the write barrier and remembered set processing. Consider benchmark Lambda-Fact5 in Table 6.4, p. 310. We shall compare block sizes 2^9 bytes and 2^{10} bytes, and, in particular, configurations, printed in italic boldface font, for block size 2^9 bytes and heap size 240 blocks, and for block size 2^{10} bytes and heap size 120 blocks. These configurations are pairwise identical with respect to heap size and collected region size.

The differences in copying cost are small in some pairs, but significant in others, such as 30236 words copied for configuration $[9, 240, 108]$ vs. 44507 words copied for configuration

[10, 120, 54]. There are two reasons for the differences. First, blocks of different size cause different fragmentation at block boundaries. However, since blocks are already substantially larger than typical object sizes present in the trace, the effect of fragmentation should be small, and the effect of differences in fragmentation even smaller; moreover, larger blocks should suffer less from fragmentation than smaller blocks, whereas we see that the copying cost is lower for smaller blocks. The second reason, which applies here, is that positioning of the collection window in the FC-DOF scheme is constrained by block boundaries and is less flexible with larger blocks, and, as Section 5.4.5.2 has shown, window positioning is important for good performance of FC-DOF.

If we focus on pairs with similar copying cost, we note that larger blocks enjoy lower counts in columns "W.b. insert," "W.b. add," and "R.s. proc," which is to be expected. With larger blocks, a greater portion of pointer stores are block-local, hence filtered out. Note, however, that doubling the block size reduces the number of write-barrier and remembered set operations by only about 10% in these examples. The column "R.s. union" is not reduced by increasing block size since the number of external pointers into the collected area does not depend on its internal division.

For the 2GYF collector, Table 6.4, p. 310, again comparing the configurations for block size 2^9 bytes and heap size 240 blocks, and for block size 2^{10} bytes and heap size 120 blocks, we find that differences in copying cost are generally not large, and in some cases favor larger, whereas in other cases smaller blocks. The pointer management operation counts are not systematically affected by block size, which is to be expected, since the generational write barrier check ignores block boundaries, except to the extent that it is indirectly influenced by fragmentation.

6.1.3 Comparison between collectors

We compare the FC-DOF and the 2GYF collector with respect to copying and pointer management costs. Consider the benchmark StandardNonInteractive with a block size of 2^9 bytes

and heap size of 25 blocks, i.e., 3200 words (Table 6.2, p. 295). Among the configurations of the FC-DOF collector for this heap size, the lowest copying cost of 3902 words is achieved with a collected region size set to 4 blocks. We abbreviate this configuration as $[9, 25, 4]$. Among the configurations of the 2GYF collector for this heap size (Table 6.9, p. 335), the lowest copying cost of 15699 words is achieved with a younger generation set to 17 blocks, and older generation set to 8 blocks, abbreviated $[9, 25, 17, 8]$. Thus the copying cost of the FC-DOF collector is significantly (4.02 times) lower than that of the 2GYF collector, which confirms, in the context of a block-based implementation, the observations we made in Sections 5.1 and 5.2.

The number of pointer management operations, however, is significantly higher for FC-DOF than for 2GYF. The number of non-filtered pointer stores is $\frac{688}{13} = 52.9$ times greater, and the number of processed remembered set entries is $\frac{615}{12} = 51.5$ times greater. It should come as no surprise that pointer management costs are higher in the FC-DOF scheme. The "R.s. union" column for FC-DOF indicates that the total number of pointers into the collected region (summed over all collections) is 392, whereas the corresponding number for 2GYF is just 12. Therefore, regardless of the efficiency of pointer filtering, when collected regions are chosen in the FC-DOF manner, the number of incoming pointers is high. The causes are in the small size of the region (and large size of the remainder), and the fact that the region visits all parts of the heap regularly, and some parts of the heap have many incoming pointers; we investigate pointer structure further in the following Section 6.2.

We can remark then on the efficiency of the FC-DOF filtering scheme of Section 3.2.2: out of 7251 pointer stores in StandardNonInteractive (Table 6.1), only 688 or 9.5% are inserted into remembered sets. This reduction in the number of pointer that are recorded is certainly not as good as 13 or 0.2% for the generational filtering of 2GYF. On the other hand, we noted that keeping a large number of pointers is a necessary price to pay in order to choose collected regions in a particular way. The true measure of the efficiency of the filtering scheme is therefore given by the ratio of the "R.s. proc" column and the "R.s. union" column: $\frac{615}{392} = 1.56$, thus 56% more pointers than is absolutely necessary are processed at garbage collection time.

We see that the copying cost is lower for the FC-DOF collector, and the pointer management cost is lower for the 2GYF collector. We must look at absolute numbers to determine which collector achieves a lower total cost. The difference in copying cost, in favor of FC-DOF, is 11797 words copied. The difference in pointer management cost, in favor of 2GYF, is 675 remembered set insertions (636 non-duplicate), and 603 processed entries at garbage collection time. Which of these is more expensive depends on the details of implementation. For the sake of a simple analysis, we subsume the different components of pointer management under the single appellation of *pointer processing cost*, which we associate with the highest operation count, that of the non-filtered pointer stores, thus erring conservatively in favor of the scheme with lower pointer management cost. If the cost of processing a pointer is greater than $\frac{11797}{675} = 17.5$ times the cost of copying a word at garbage collection time, then the 2GYF collector will have the advantage for benchmark StandardNonInteractive. (These break-even ratios are summarized for all examined configurations in Figures 6.1– 6.9, pp. 389–393 which we discuss below.)

If we are interested in *incremental* operation of the collector, then we should compare configurations that result in similar (and *large*) numbers of collector invocations. In this case, if the 2GYF collector is configured with the younger generation at 4 blocks, and the older generations at 21 blocks, then it will perform 418 collections, the same as the FC-DOF collector with the collection window of 4 blocks. The copying cost advantage of FC-DOF is now greater, 23510 words, but the pointer management advantage of 2GYF is only slightly smaller, 673 remembered set insertions (633 non-duplicate), and 600 processed entries at garbage collection time; the pointer/copy cost ratio would have to be over 34.9 for 2GYF to have the advantage.

Similar results obtain for the larger block size of 2^{10}: compare the configuration $[10, 13, 3]$ of FC-DOF with configuration $[10, 13, 7, 6]$ of 2GYF.

We turn our attention to benchmark HeapSim (Tables 6.3, p. 301 and 6.10, p. 343), at block size 2^{13} bytes and heap size 123 blocks, or 251904 words. The best configuration of FC-DOF is $[13, 123, 4]$, with copying cost 262984, with 4645 non-filtered pointer stores (of which 3731

non-duplicate), and 3297 processed entries at garbage collection time. The remembered set union total is 1628, thus the filtering efficiency is assessed at an overhead of 102% more processed pointers than is absolutely necessary. The best 2GYF configuration is $[13, 123, 41, 82]$, with copying cost 341702, with 22 non-filtered pointer stores (of which 12 non-duplicate), and 12 processed entries at garbage collection time. The copying cost advantage of FC-DOF is 89798 words; the pointer/copy cost ratio would have to be over 19.4 for 2GYF to have the overall advantage. Similar results obtain for the larger block sizes of 2^{14} (compare the configuration $[14, 62, 3]$ of FC-DOF with configuration $[14, 62, 20, 42]$ of 2GYF) and 2^{16} (compare $[16, 16, 4]$ with $[16, 16, 6, 10]$).

In benchmark Richards (Tables 6.5, p. 322 and 6.12, p. 370), we consider the block size 2^{10} bytes, and heap size 20 blocks, or 5120 words. The best configuration of FC-DOF is $[10, 20, 8]$, with copying cost 56758, with 30270 non-filtered pointer stores (of which 29775 non-duplicate), and 29686 processed entries at garbage collection time. The remembered set union total is 11354, thus the filtering achieves an overhead of 161%. The best 2GYF configuration is $[10, 20, 11, 9]$, with copying cost 631920, with 3754 non-filtered pointer stores (of which 1541 non-duplicate), and 1541 processed entries at garbage collection time. The copying cost advantage of FC-DOF is 575162, and its pointer management disadvantage is 28614 insertions (26021 for non-duplicates) and 28145 processed entries. The pointer/copy cost ratio would have to be over 20.1 for 2GYF to have the overall advantage. With twice as large blocks (configurations $[11, 10, 4]$ and $[11, 10, 6, 4]$) the results are similar, and the pointer/copy cost ratio break-even point is 27.0.

The situation is somewhat different for Lambda-Fact5 (Tables 6.4, p. 310 and 6.11, p. 354). Recall that no collector worked much better than non-generational for this benchmark in object-level simulations of Section 5.1, and that differences among collectors were small. We now look at block size 2^9 bytes and heap size 90 blocks, or 11520 words. The best configuration of FC-DOF is $[9, 90, 67]$, with copying cost 134723, with 21778 non-filtered pointer stores (of which 21769 non-duplicate), and 21087 processed entries at garbage collection time. The

remembered set union total is 4365, which means that filtering is unusually inefficient here, at an overhead of 383%. The best 2GYF configuration is $[9, 90, 67, 23]$, with copying cost 158419, with 801 non-filtered pointer stores (of which 558 non-duplicate), and 557 processed entries at garbage collection time. The copying cost advantage of FC-DOF is 23696, but its disadvantage is 20977 insertions (21211 for non-duplicates) and 20530 processed entries. It is sufficient for the pointer/copy cost ratio to be over 1.12 for 2GYF to have the overall advantage.

Similarly, in Bloat-Bloat (Tables 6.7, p. 327 and 6.14, p. 376), we find that the copying cost difference between the collectors is very small, but the pointer management costs are noticeably higher for FC-DOF. For instance, with a block size of 2^{16}, and a heap size of 28 blocks, or 458752 words, the best configuration of FC-DOF is $[16, 28, 14]$, with copying cost 2308104 and 387199 non-filtered pointer stores, whereas the best configuration of 2GYF is $[16, 28, 13, 15]$, with copying cost 2498540 and 33304 non-filtered pointer stores. In this case it is sufficient for the pointer/copy cost ratio to be over 0.53 for 2GYF to have lower total cost.

In Toba (Tables 6.8, p. 331 and 6.15, p. 382), with 2^{16}-byte blocks and a heap size of 40 blocks, the difference in copying cost in favor of FC-DOF configuration $[16, 40, 25]$ is 756188 words, and the difference in pointer cost in favor of 2GYF configuration $[16, 40, 23, 17]$ is 288379 operations, giving a pointer/copy cost ratio threshold of 2.62.

Benchmark JavaBYTEmark stands out (Tables 6.6, p. 326 and 6.13, p. 375). We look at block size 2^{18} bytes and a heap size of 6 blocks, or 393216 words. One of the (tied) best configurations of FC-DOF is $[18, 6, 3]$, with copying cost 17634 words, with 23 non-filtered pointer stores (out of 49061 pointer stores), 19 non-duplicate, and 19 processed entries at garbage collection time. The best 2GYF configuration is $[18, 6, 5, 1]$, with copying cost 37842 words, with 135 non-filtered pointer stores (of which 102 non-duplicate), and 16 processed entries at garbage collection time. Here both collectors have negligible pointer management costs, but FC-DOF is better, by more than a factor of 2, with respect to copying cost.

A summary of the break-even ratios of pointer-operation cost to word-copying cost is presented in Figures 6.1–6.11, pp. 389–394 (at the end of this chapter), for all simulated heap

sizes, including the exemplars discussed above. In these graphs, the horizontal axis indicates the heap size (in words), and the vertical axis indicates the break-even ratio, shown on a logarithmic scale. Each graph contains several data lines, one for each block size. For example, Figure 6.1 has three lines, for block sizes 2^9, 2^{10}, and 2^{11} words. If in an implementation the ratio of the cost of a pointer operation to the per-word cost of copying is above the point in the graph, then 2GYF has a lower cost overall than FC-DOF.

6.1.4 Additional observations

In addition to the comparison between FC-DOF and 2GYF, we can observe a trade-off between copying cost and pointer-maintenance cost in the configurations of FC-DOF itself. Consider for instance the configurations $[16, 28, *]$ of benchmark Bloat-Bloat. As the size of collected region is increased from 11 to 26 blocks, the copying cost first briefly falls from $2.6 \cdot 10^6$ at 11 blocks to the minimum of $2.3 \cdot 10^6$ at 13 blocks (the configuration we compared against 2GYF above), then rises to $5.4 \cdot 10^6$ at 24 blocks, and briefly falls to $5.2 \cdot 10^6$ at 26 blocks. Although the copying cost is not strictly monotone increasing, it shows a clear increasing trend. At the same time, the "R.s. union" column shows a decreasing trend, from 106683 entries in the remembered set union with collected region of 11 blocks, down to 29426 with 26 blocks. The actual processing costs in the "R.s. proc" column are also nearly monotone decreasing.

An examination of all the tabulated results for the FC-DOF collector reveals that there are remarkably few duplicate entries: the values in the "W.b. add" column are close to those in the "W.b. insert" column. Therefore, it is not essential to eliminate duplicates early in pointer management. A simple sequential store buffer representation of the remembered set, which does not remove duplicates, may be adequate.

With the single generational remembered set of the 2GYF collector, the number of duplicates is significant: the values in the "W.b. add" column are typically well below those in the "W.b. insert" column. When the proportion of duplicate entries is large, it is worth eliminating

them early and cheaply, e.g., by using a card-marking write barrier, with cards flushed into sets without duplicates.

Since the proportion of pointer stores retained is higher in the FC-DOF collector than in the 2GYF collector, the amount of auxiliary space needed for the remembered sets is higher, which raises the concern about the *total* memory usage of the collector. It turns out, however, that the space needed for remembered sets is typically below 2% of heap size. In particular, assuming that a remembered set entry uses one word, the total size of remembered sets for all heap blocks is 0.62% of total heap size for StandardNonInteractive (in the configuration discussed above), 0.49% for Richards, 1.38% for Bloat-Bloat, and 1.02% for Toba. Therefore, the space overhead of pointer maintenance in FC-DOF is low.

The measured values for individual copying and pointer-tracking operations (Section 3.2.5) permit a more detailed estimation of the total cost of collection. However, our trace-based simulation does not simulate stack contents, and therefore, while its copying counts are accurate, it cannot model the pointer-scanning counts: the number of scanned pointers in to-space that do not result in object copying depends on the contents of the stack. We can establish lower and upper limits, however. The lower limit corresponds to approximately one duplicate pointer per object, that one being the pointer to the class object. The upper limit corresponds to all words being duplicate pointers. We assume individual operation costs as in Section 3.2.5, and, for a given heap size, we allow each collector to assume its best configuration with respect to the total cost estimate. For the heap sizes as in the examples discussed above, the lower and upper bounds for the ratios of total processor cycles spent on garbage collection by 2GYF and by DOF are as follows: for benchmark StandardNonInteractive, between 2.66 and 3.28; for benchmark Richards, between 3.84 and 5.70; for benchmark Lambda-Fact5, between 0.98 and 1.04; for benchmark JavaBYTEmark, between 2.16 and 2.48; for benchmark Toba, between 0.98 and 1.08; for benchmark Bloat-Bloat, between 0.94 and 0.98. Therefore, pointer management costs do not significantly alter the qualitative performance relationship between the two collectors that we established when considering the amounts copied in Chapter 5: DOF

remains significantly better than 2GYF on one set of benchmarks, while not significantly worse than 2GYF on the remaining benchmarks.

6.2 Methods for examining the directions of pointers in the heap

An evaluation of the cost of remembered-set maintenance provides us with raw numbers, such as the number of executed write-barrier checks, the number of processed remembered-set entries, and an estimate of the time penalty of these operations. Upon seeing these numbers, we can assess the effect of pointer tracking on different collectors, but we are at a loss about the origins of the effect. Just as it is necessary to recognize where (in which region of the heap) the high copying cost was incurred, it is necessary to recognize *where* in the heap the pointer-tracking costs arise, if we are to understand and remedy the situation. Our purpose is to establish quantitative measurements where the literature so far provides speculative statements.

6.2.1 Pointer stores in the ideal heap

We begin with an examination of pointer store directions in an absolute sense, without reference to a specific collection algorithm, assuming instead that the heap is perfectly collected, as we did for the heap profiles in Section 5.4.3. For each non-null pointer store, we note the heap position of both the pointer source and the pointer target. Recall that the heap position of an object is expressed as the distance from the young end of the heap, i.e., the amount of data that was allocated since that object was allocated and is still live. We define pointer *distance* as the difference between the heap position of the pointer target and the heap position of the pointer source. Hence, the distance is positive for younger-to-older pointers, and negative for older-to-younger pointers. Aggregating over all stores, the distribution of the positions is shown in Figures 6.12–6.25, pp. 395–408 (at the end of this chapter). Each figure contains three plots: (a) a histogram of the distribution of heap positions of pointer sources; (b) a histogram of the distribution of heap positions of pointer targets; and (c) a cumulative distribution graph for pointer distances and directions.

The pointer source histograms for all programs have a peak in the low positions, the youngest objects. A majority of pointer stores are into the most recently allocated objects. We emphasize that the histogram excludes null pointer stores, therefore the default initializations of pointer fields to the null pointer do not skew the distribution. Only the actual initializations by assignment are reflected in the histogram. The pointer *target* distributions also exhibit sharp peaks for the youngest objects, and indeed the fact that pointer distances are very short means that the source and target distributions must be similar. Thus a majority of pointers are from one object in the youngest part of the heap to another.

We now understand that the generational write barrier works well because for most pointer stores, both the source and the target are in the youngest generation, and thus the store need not be recorded. The putative explanation, that older-to-younger pointers are rare relative to younger-to-older pointers, is neither true (as the pointer distances graphs show, both directions are common), nor relevant.

Two programs with exceptional behavior are Swim and Tomcatv; they have secondary peaks of the pointer source distribution among the oldest objects. Both of these programs use a common implementation of floating-point matrices. A floating-point value in Smalltalk is not represented as an immediate register value, because the necessary tag bit(s) would both limit precision and require expensive conversions to the hardware floating-point format. Instead, a level of indirection is used, and each floating-point value is allocated in the heap. Computation creates new floating-point objects and updates the elements of a matrix to point to them. Since the matrix itself is allocated early in program execution, computation produces a stream of old-to-new pointer stores This peculiarity of the Smalltalk floating-point representation, and similar situations with frequently updated data structures with *boxed* representations in general and their interaction with the write barrier, have been noted before [Wilson, 1990; Chambers, 1992; Wilson, 1992] as a potential problem for the generational write barrier.

6.2.2 Pointer stores and remembered sets in age-based collectors

We now examine the pointer distributions that arise as a result of the execution of particular collection algorithms. Our frame of reference is an age-ordered heap, divided into blocks of equal size. The heap position of the youngest block (i.e., the current allocation block) is numbered 0, and older blocks are given successive numbers. We are interested in the distribution of pointer-tracking events by heap position. In other words, we are interested, having observed the positional distribution with an ideally collected heap, in any bias that the actual collection scheme introduces.

The resulting pointer source and target distributions are shown in plots (a)–(d) of Figures 6.26–6.31, pp. 409–414, for the benchmarks and their configurations discussed in the preceding Section 6.1.3. Plots (a) and (c) on the left reflect all non-null pointer stores, whereas plots (b) and (d) on the right reflect only those stores not eliminated by filtering, i.e., those inserted into remembered sets. Plots (a) and (b) show the histogram of positions of the pointer source, whereas plots (c) and (d) show the histogram of positions of the pointer target.

Additionally, the bottom plots (e) of Figures 6.26–6.31 show the positional distribution of the remembered set entries chosen for processing at garbage collection time.

The plots (a) and (c) show the same patterns as the corresponding plots (a) and (b) for the perfectly collected heap in the preceding section. Therefore, the FC-DOF collector, owing to its regular full sweeps, does not maintain large quantities of uncollected ("tenured") garbage, which, if present, would visibly distort the distribution relative to the perfectly collected heap.

Of greater interest is the distribution of heap position among the *non-filtered* pointer stores, in plots (b) and (d). First note that the absolute numbers are much smaller, since filtering eliminates most pointer stores.

A majority of the non-filtered stores still have their source in the youngest block (position 0). The non-filtered targets in the youngest block are few, because of the block-local filtering. However, there are many targets tailing off from block 1 towards older blocks—the directional filter could not eliminate the pointers from block 0 into these older blocks. In Lambda-Fact5

there is also a large number of pointers with source in block 0 and target among the very oldest objects.

When a filter cannot eliminate a pointer store, it inserts it into the remembered set of the target block. Thus, most pointers are inserted into the remembered sets of blocks at low positions (but not position 0). However, some time elapses between the pointer store and the time when the target block becomes part of the collected region of a collection, and during that time the block ages and moves toward higher heap positions. Therefore, in the distribution of heap positions of *processed* remembered set entries, the peak is in the middle or higher (older) regions. In Lambda-Fact5, for instance, just the oldest of the 90 blocks accounts for 30% of all processed remembered set entries (Figure 6.29(e), p. 412). Recall that the oldest blocks contain permanent data (Figure 5.271, p. 258), hence collections immediately following the resetting of the window not only incur a high copying cost, but also a high pointer processing cost.

Benchmark JavaBYTEmark is an exception: the filtering is so successful that very few pointers remain non-filtered, and, with their number so small, it is not warranted to discuss their positional distribution.

6.3 Method for evaluating the collection start-up cost

In addition to the copying cost of collection, which is directly related to the efficiency of the collector in finding low-survival regions, and the remembered-set maintenance cost, which is related to the ability of the collector to approximate the structure of the uncollected region safely and cheaply, there are other costs that are incurred on every collection.

First, there is a direct overhead of switching system activity from the mutator to the collector. This overhead depends on the implementation details. In some implementations, the collection is triggered by a page fault on accessing a protected page past the end of the available nursery space, or it may involve saving architectural state because the mutator is written in a different language from the garbage collector itself, and in these cases the start-up may

be costly. On the other hand, this direct overhead is minimal in virtual machine implementations. The change of program focus from the mutator to the collector also has an adverse effect on reference locality—the performance of the instruction cache, and possibly the data cache, degrades.

Second, each collection requires establishing the global roots. These are found in the registers (machine or virtual) and in the stack (if the implementation uses an explicit stack). Whether this task is a large fraction of total garbage collection cost depends on the implementation: in the case of an explicit stack, has the implementation provided stack maps in a form that permits efficient stack scanning [Diwan *et al.*, 1992]?[1]

The cost of stack scanning is not directly dependent of the garbage collection algorithm that is subsequently employed. However, the cost of one stack scan depends on the stack state (how many activation records there are on the stack), hence it depends on the moment at which it is examined. Since different collector algorithms result in different moments of collector invocation, there is an indirect dependence of total scanning cost on collection algorithm, beyond the mere number of invocations, but with many invocations spread over the execution, with somewhat asynchronous points of collection, this additional dependence is probably very weak.

In summary, there is a certain fixed cost to be paid for each collector invocation. If we are primarily concerned with the total running time of the program, then we must be careful in the choice of the garbage collection algorithm lest it cause too many collector invocations. For many programs, we found that the lowest copying costs are obtained with smaller collected regions or smaller nurseries, and consequently with larger numbers of collector invocations. Clearly, when some rather large number is reached [Stefanović and Moss, 1994], the invocation costs will dominate.

---

[1] In the case of a stack hidden within the heap, stack scanning is implicitly accomplished during the Cheney scan phase of garbage collection, so it is difficult to isolate the stack scanning cost.

If, on the other hand, the primary consideration is that of minimizing pause times, then the invocation cost represents a fixed component that cannot be avoided. Since this cost is independent of the collection algorithm, except as mentioned above, it is still appropriate to choose an algorithm that minimizes the other costs.

6.3.1 Results

The invocation cost is shown in Figures 6.32–6.135, pp. 415–459. The vertical axis shows the number of collector invocations *relative* to the non-generational collector. (This permits us to use the same scale for all plots for a given benchmark.) As before, the horizontal axis in the FC plots gives the size of the collected region expressed as the fraction g of heap size. In the GYF plots, the horizontal axis gives the size of the nursery as a fraction of heap size.

A very regular pattern emerges, for all benchmarks, and all heap sizes: when the size of the collected region (or nursery size) is close to the full heap size, the behavior is non-generational and the relative invocation count is close to 1; but as the size of the collected region decreases, the invocation count increases. This result is intuitively clear: the amount of garbage found by a collection cannot be larger than the size of the collected region, thus the number of collector invocations must rise as the reciprocal of that size, if we ignore for a moment the influence of that size on the proportion of survivors.

6.4 Summary

We have shown that pointer-maintenance costs in a generational collector are very low, confirming past research. The analysis of pointer stores reveals that the explanation lies in the location of stored pointers (in recently allocated data) and not in the temporal direction of pointers (younger-to-older or older-to-younger).

The FC-DOF collector, because it collects smaller regions in all parts of the heap, incurs a considerably higher pointer maintenance cost, both in the higher number of pointer stores that must be recorded at run time (even after efficient filtering), and in the high number of incoming

pointers processed at garbage collection time. Our trade-off analysis suggests that the copying cost advantage of DOF outweighs the pointer cost disadvantage for several tested programs, but the final verdict must await a full implementation.

Let us note, however, that the focus of our study on the application of collectors to the entire heap precluded an investigation of the space of older (mature) objects only, in which the distributions of pointer stores as well as the distribution of lifetimes is less favorable to the generational approach (to the extent that this can be plausibly inferred from observations of relatively longer-lived objects in our programs). Moreover, it can be argued, pending further measurements, that, with remembered set costs having already been absorbed in the mature object space [Hudson and Moss, 1992], deferred older-first becomes a collection *policy* worth exploring. It is a different policy from the *train* algorithm [Hudson and Moss, 1992; Seligmann and Grarup, 1995]; in particular, it does not guarantee reclamation of arbitrary cycles. On the other hand, the train algorithm leaves certain promotion decisions for objects moved out of a train underspecified; whereas locality (for the purpose of exploiting *key object opportunism* [Hayes, 1991]) is suggested as a criterion for making these decisions (thus: moving an object close to the object that keeps it live by pointing to it), age is a permissible criterion as well.

Table 6.2. Costs of block FC-DOF for StandardNonInteractive

Configuration			Copied	Invoc	W.b. insert	W.b. add	R.s. proc	R.s. union
9	12	10	98850	301	2668	2662	2645	371
9	14	6	20028	304	693	674	666	358
9	14	7	26720	269	1022	1005	986	440
9	14	9	81824	269	2333	2318	2292	592
9	14	10	80452	250	2365	2342	2322	474
9	14	12	73403	222	1705	1699	1686	234
9	17	6	14719	296	704	675	662	406
9	17	7	15403	255	623	597	584	343
9	17	8	14888	222	621	600	591	306
9	17	9	17476	201	608	593	572	219
9	17	10	19831	182	758	744	738	236
9	17	11	25396	171	897	881	875	465
9	17	12	56139	182	1751	1733	1720	431
9	17	13	56103	172	1375	1369	1315	179
9	17	14	56389	167	1505	1500	1483	171
9	17	15	53896	159	1394	1391	1375	190
9	17	16	51752	152	1007	1006	996	25
9	21	6	9878	288	640	607	582	329
9	21	7	9729	247	598	567	553	301
9	21	8	10506	217	575	544	534	262
9	21	9	10585	193	531	509	453	232
9	21	11	11634	159	514	499	486	282
9	21	12	13141	147	607	589	571	328
9	21	13	14320	137	517	504	497	227
9	21	14	15793	128	504	495	490	162
9	21	16	39201	125	1005	998	950	228
9	21	17	39948	122	1062	1056	1047	198
9	21	18	38121	117	834	833	821	164
9	21	19	37421	113	724	723	707	41
9	**25**	**4**	**3902**	**418**	**688**	**648**	**615**	**392**
9	25	5	4456	335	678	637	614	395
9	25	7	5563	241	668	630	601	308
9	25	8	7085	213	582	547	529	237
9	25	10	7141	170	562	531	514	239
9	25	11	8247	156	503	478	469	226
9	25	13	8773	132	426	410	397	212
9	25	14	8506	123	386	374	361	222
9	25	16	9578	108	368	364	347	187
9	25	17	10837	103	340	333	320	122
9	25	18	12331	97	429	424	411	164
9	25	20	30557	96	832	828	812	179
9	25	21	31288	94	859	854	804	137
9	25	22	31622	92	849	845	838	122
9	25	23	30533	89	694	688	675	47
9	25	24	30304	87	615	612	598	74
9	30	3	2246	550	664	623	595	444
9	30	4	2389	413	669	628	602	369
9	30	6	3519	278	678	640	619	360

continued on next page

Table 6.2. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
9	30	8	3210	208	658	619	599	246
9	30	10	3953	167	648	609	589	285
9	30	12	3827	139	555	522	505	198
9	30	13	3594	128	544	515	493	201
9	30	15	4885	112	482	458	445	241
9	30	17	7070	100	423	403	385	222
9	30	19	6247	89	289	279	272	166
9	30	20	8405	86	373	363	299	147
9	30	22	9205	78	227	218	209	120
9	30	24	9733	72	191	187	174	82
9	30	26	23200	72	540	534	526	128
9	30	27	24382	71	553	550	542	52
9	30	28	23502	70	471	470	465	92
9	30	29	24267	69	444	444	438	22
9	37	3	1621	546	683	642	611	462
9	37	6	1495	273	679	638	601	344
9	37	8	1486	205	682	641	603	254
9	37	10	1464	164	673	632	597	273
9	37	12	1560	137	672	631	598	225
9	37	14	2177	118	626	590	568	222
9	37	17	2317	97	496	468	451	152
9	37	19	2887	87	428	405	389	147
9	37	21	3155	79	372	355	336	164
9	37	23	3760	72	280	269	265	109
9	37	25	4376	67	228	218	206	118
9	37	27	5138	62	158	152	147	80
9	37	30	7571	57	273	270	259	147
9	37	31	8662	55	261	256	246	78
9	37	33	17392	55	398	396	391	71
9	37	34	18962	55	468	467	454	54
9	37	35	18851	54	469	467	451	43
9	37	36	18994	53	382	382	370	22
9	45	4	1492	407	698	657	610	383
9	45	7	1465	233	698	657	608	345
9	45	9	1376	181	698	657	607	297
9	45	12	1356	136	697	656	604	243
9	45	15	1405	109	665	625	588	331
9	45	18	1314	91	658	620	591	191
9	45	20	1349	82	605	569	547	183
9	45	23	1277	71	501	472	438	131
9	45	26	1484	63	362	343	320	109
9	45	28	2632	59	329	313	301	144
9	45	31	3011	53	279	272	256	114
9	45	33	3707	50	186	181	170	65
9	45	36	5089	47	124	118	108	35
9	45	38	6551	44	225	221	214	70
9	45	40	14201	44	299	296	284	87
9	45	41	14880	44	345	342	329	29

continued on next page

Table 6.2. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
9	45	42	14768	43	326	323	320	53
9	45	43	13976	42	272	272	264	48
9	54	2	1777	811	719	678	631	537
9	54	5	1508	324	717	676	629	428
9	54	8	1440	203	723	682	629	289
9	54	11	1414	148	714	673	623	278
9	54	15	1387	108	693	653	601	346
9	54	18	1288	90	692	652	603	201
9	54	21	1279	78	688	648	615	186
9	54	24	1235	68	668	629	588	141
9	54	28	1237	58	514	483	448	133
9	54	31	1267	53	414	393	366	135
9	54	34	1546	48	291	276	251	94
9	54	37	1556	44	222	211	195	80
9	54	40	2654	41	212	204	193	104
9	54	43	3743	39	116	113	97	48
9	54	46	6161	37	108	104	82	27
9	54	48	6683	35	152	151	143	78
9	54	50	11511	35	255	253	245	63
9	54	51	12149	35	303	301	285	20
9	54	52	11703	34	243	241	195	11
10	6	5	94809	294	1854	1851	1833	372
10	7	6	71219	217	1081	1073	1061	154
10	9	4	12907	214	517	500	495	360
10	9	5	17226	175	491	479	468	207
10	9	6	51354	170	1074	1062	1061	389
10	9	7	50572	153	1049	1041	1030	255
10	9	8	47608	142	808	806	791	67
10	11	3	8530	278	603	582	565	388
10	11	4	8571	209	519	496	491	330
10	11	5	9271	168	435	422	417	294
10	11	6	11251	141	419	408	397	242
10	11	7	13004	122	374	367	359	246
10	11	8	35565	119	799	796	786	195
10	11	9	36631	111	650	647	634	140
10	11	10	36273	105	604	603	594	156
10	**13**	**3**	**4236**	**272**	**531**	**504**	**502**	**315**
10	13	4	5581	205	546	517	504	287
10	13	5	6553	165	563	538	528	310
10	13	6	6637	138	446	428	427	263
10	13	7	7851	119	305	293	291	160
10	13	8	6745	103	347	344	293	170
10	13	9	9446	93	288	285	237	148
10	13	10	27375	91	473	470	461	149
10	13	11	28988	87	616	615	609	180
10	13	12	28514	83	501	500	487	130
10	15	3	2557	269	557	529	513	332
10	15	4	3302	203	590	564	551	322

continued on next page

Table 6.2. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
10	15	5	3311	162	566	541	528	281
10	15	6	3607	135	548	525	515	241
10	15	7	3374	116	398	378	372	227
10	15	8	4807	102	313	302	301	191
10	15	9	6164	92	267	261	252	152
10	15	10	7375	83	238	230	215	148
10	15	11	9205	76	320	313	306	184
10	15	12	23645	75	491	485	474	143
10	15	13	23797	71	420	418	416	82
10	15	14	24146	69	440	439	426	58
10	19	2	1820	400	585	557	534	425
10	19	3	1591	266	589	561	534	358
10	19	4	1602	200	591	563	533	311
10	19	5	1570	160	587	559	528	278
10	19	6	1520	133	583	555	524	236
10	19	7	1432	114	550	525	509	244
10	19	9	1782	89	415	395	389	163
10	19	10	2988	81	339	321	313	157
10	19	11	2912	73	290	278	256	131
10	19	12	4234	68	256	248	242	130
10	19	13	4377	63	217	209	206	134
10	19	14	6130	59	245	239	236	143
10	19	15	5266	55	132	130	129	73
10	19	16	16607	54	279	278	275	76
10	19	17	18372	53	473	469	455	157
10	19	18	18033	51	299	297	296	57
10	23	2	1798	398	602	574	540	434
10	23	3	1569	265	605	577	539	366
10	23	5	1524	159	607	579	538	295
10	23	6	1474	133	608	580	547	259
10	23	8	1465	100	584	557	540	216
10	23	9	1462	89	568	541	527	194
10	23	10	1388	80	540	515	501	187
10	23	12	1717	67	401	385	372	124
10	23	13	1915	62	315	304	296	119
10	23	14	1306	57	274	262	256	136
10	23	16	2822	50	159	155	147	80
10	23	17	4219	48	111	107	101	56
10	23	18	4546	45	176	170	170	89
10	23	20	13522	43	255	251	241	113
10	23	21	13615	41	311	310	310	101
10	23	22	13359	40	191	190	184	19
10	27	2	1788	396	623	595	560	454
10	27	4	1538	198	630	602	560	344
10	27	6	1452	132	630	602	557	272
10	27	7	1340	113	598	571	539	275
10	27	9	1366	88	589	562	535	208
10	27	11	1326	72	582	556	531	193

continued on next page

Table 6.2. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
10	27	12	1294	66	572	547	523	148
10	27	14	1204	57	433	413	402	151
10	27	15	1305	53	351	336	322	128
10	27	17	1319	47	257	244	236	109
10	27	18	1614	44	210	201	195	95
10	27	20	2601	40	187	181	169	114
10	27	22	4323	37	186	181	171	100
10	27	23	4753	35	231	229	216	141
10	27	24	10737	35	230	228	224	97
10	27	25	11380	34	218	218	216	63
10	27	26	11676	34	159	159	155	17
11	4	3	65841	206	867	857	806	441
11	5	3	48669	166	776	765	762	341
11	5	4	44080	135	472	470	470	232
11	6	2	6708	206	385	374	372	297
11	6	3	9426	139	397	387	379	291
11	6	4	35048	117	666	663	657	329
11	6	5	33374	102	289	286	286	177
11	7	2	3504	202	403	393	386	278
11	7	3	4999	136	346	336	336	221
11	7	4	6905	103	251	249	247	196
11	7	5	28399	91	504	501	501	245
11	7	6	26087	80	358	356	352	114
11	8	2	2119	200	421	407	378	271
11	8	3	2951	134	380	370	359	204
11	8	4	4086	101	283	280	272	191
11	8	5	6241	82	136	134	130	94
11	8	6	21036	73	346	343	337	151
11	8	7	22735	67	272	270	268	69
11	10	2	1392	198	446	432	393	294
11	10	3	1372	132	445	431	392	241
11	10	4	1290	99	410	398	379	171
11	10	5	2137	80	269	263	258	138
11	10	6	3129	67	157	154	149	102
11	10	7	4339	58	123	122	122	94
11	10	8	14946	53	107	107	103	51
11	10	9	16282	50	83	83	82	39
11	12	2	1392	197	468	454	413	314
11	12	3	1352	132	447	433	410	259
11	12	4	1282	99	437	424	401	191
11	12	5	1258	79	414	402	383	160
11	12	6	1236	66	314	307	294	139
11	12	7	1470	57	236	229	227	146
11	12	8	2622	50	143	139	138	74
11	12	9	3653	45	128	127	126	91
11	12	10	12324	42	152	151	151	80
11	12	11	13546	40	110	110	109	46
11	14	2	1366	196	482	468	415	320

continued on next page

Table 6.2. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
11	14	3	1352	131	464	450	424	276
11	14	4	1268	98	457	444	407	207
11	14	5	1258	79	447	434	411	180
11	14	6	1228	66	441	429	408	161
11	14	7	1217	56	355	345	316	157
11	14	8	1154	49	287	278	254	104
11	14	9	1655	44	178	175	174	98
11	14	10	2531	40	126	121	120	85
11	14	11	3763	37	91	91	85	59
11	14	12	9620	34	140	137	129	78
11	14	13	11007	33	84	83	81	56
12	3	2	37115	118	276	275	275	275
12	4	2	5110	101	272	267	260	209
12	4	3	23469	74	232	228	227	225
12	5	2	1429	99	298	291	285	184
12	5	3	4417	67	175	173	169	164
12	5	4	17690	54	151	150	148	147
12	6	2	1342	99	327	319	310	208
12	6	3	1232	66	204	200	198	116
12	6	4	3679	50	113	112	112	71
12	6	5	13746	42	127	127	127	125
12	7	2	1320	98	347	339	321	219
12	7	3	1210	66	324	316	311	141
12	7	4	1295	49	176	174	167	119
12	7	5	2375	40	108	108	108	99
12	7	6	11784	35	213	212	212	212
13	3	2	17281	54	250	248	248	247
13	4	2	1223	49	144	139	135	131
13	4	3	10752	34	187	185	177	177

Table 6.3. Costs of block FC-DOF for HeapSim

Configuration			Copied	Invoc	W.b. insert	W.b. add	R.s. proc	R.s. union
13	56	48	5327558	134	229815	229769	225707	75
13	56	50	4878471	123	211206	211162	207048	71
13	56	52	4603637	116	198078	198034	193762	65
13	56	53	4440427	112	191022	190978	187058	63
13	56	54	4274724	108	184076	184032	179898	63
13	68	6	529350	341	10125	9204	7274	2528
13	68	10	588982	208	13767	12846	10113	2744
13	68	14	668317	151	17993	17199	14666	2795
13	68	18	832717	122	27313	26514	22811	3977
13	68	22	1345262	112	51121	50337	46630	7659
13	68	27	2168396	107	87428	86892	83335	10645
13	68	31	2194904	96	90304	89751	85921	7255
13	68	35	3250597	98	136637	136084	132424	9942
13	68	39	3570427	99	149910	149362	145490	4543
13	68	43	3532656	95	151487	150949	146974	1981
13	68	46	3343271	89	142525	141987	137756	700
13	68	50	3128299	82	133954	133910	129392	52
13	68	54	2873891	75	121245	121201	117114	48
13	68	58	2697368	69	113692	113650	109466	45
13	68	60	2587184	66	109339	109297	105317	43
13	68	63	2484252	63	104596	104565	100311	42
13	68	64	2403352	62	102571	102540	98539	40
13	68	65	2409081	61	103261	103230	99108	41
13	68	66	2326937	60	99026	98960	94853	40
13	83	7	421530	283	7670	6730	5229	1802
13	83	12	418704	165	9241	8423	5764	1236
13	83	17	502316	119	12889	12071	9258	1545
13	83	22	497638	92	14342	13524	10105	1584
13	83	27	580774	77	18915	18362	14538	1557
13	83	32	655284	66	22787	22234	18286	1762
13	83	37	895854	60	32883	32335	28315	2132
13	83	42	1208816	57	49085	48537	44685	1676
13	83	47	1918684	60	79729	79683	75683	47
13	83	52	1869868	55	76792	76748	72232	42
13	83	56	1904614	53	79547	79503	75356	40
13	83	61	1798912	49	75354	75323	70722	37
13	83	66	1773195	47	72834	72809	68653	34
13	83	71	1676040	44	67654	67636	63125	32
13	83	74	1650759	42	67342	67324	63743	31
13	83	76	1572842	41	66925	66907	62993	31
13	83	78	1635741	41	67429	67411	63417	30
13	83	79	1548874	40	61941	61923	58077	29
13	83	80	1554664	40	63290	63283	58785	20
13	101	9	331754	213	6587	5727	3896	1375
13	101	15	331560	128	7359	6621	3975	1079
13	101	21	333382	92	7590	6852	4111	1090
13	101	27	413435	73	9948	9456	6908	855
13	101	33	411742	60	11566	11074	7285	782

continued on next page

Table 6.3. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
13	101	39	492751	52	14677	14185	10231	846
13	101	45	486986	45	14810	14318	10319	905
13	101	52	638726	40	22554	22554	17191	226
13	101	58	873771	38	32773	32773	29127	29
13	101	64	1169229	37	46766	46766	42974	25
13	101	69	1287495	36	51013	51013	47110	22
13	101	75	1269543	34	50473	50473	46656	19
13	101	81	1236771	32	51036	51036	47323	18
13	101	86	1224301	31	49283	49283	45273	16
13	101	90	1202810	30	49002	49002	45104	14
13	101	93	1122663	29	46960	46960	41948	14
13	101	95	1180671	29	47195	47195	43382	14
13	101	96	1184501	29	48290	48290	44716	15
13	101	97	1095749	28	43952	43952	39771	14
13	**123**	**4**	**262984**	**465**	**4645**	**3731**	**3297**	**1628**
13	123	11	323488	172	6681	5821	4267	1403
13	123	18	323284	105	7535	6797	4506	980
13	123	26	323098	73	8895	8403	4627	539
13	123	33	325582	58	8152	7660	4847	551
13	123	41	327370	47	9488	8996	4783	681
13	123	48	324917	40	7788	7788	3812	77
13	123	55	405206	36	12158	12158	7759	240
13	123	63	398933	31	12113	12113	7417	192
13	123	70	477079	29	15965	15965	11683	24
13	123	77	535990	26	20078	20078	16137	20
13	123	84	838665	26	33417	33417	28955	18
13	123	91	900843	25	34462	34462	31190	15
13	123	98	941868	24	35447	35447	31789	14
13	123	105	928914	23	36702	36702	32423	12
13	123	109	902469	22	35888	35888	31958	12
13	123	113	913267	22	36972	36972	32766	12
13	123	116	880675	21	33190	33190	29506	11
13	123	117	882820	21	33127	33127	29888	11
13	123	119	887955	21	34990	34990	30754	11
13	150	4	235342	455	4616	3702	2989	1543
13	150	13	236684	140	5150	4412	2755	884
13	150	22	238944	83	5828	5090	2725	1034
13	150	31	240390	59	5762	5270	2269	494
13	150	40	270026	46	7484	6992	4671	712
13	150	49	311629	38	8344	8344	5361	161
13	150	58	310068	32	9080	9080	5258	74
13	150	67	313274	28	7798	7798	4697	42
13	150	76	317015	25	8711	8711	4642	127
13	150	86	397160	23	12786	12786	7864	20
13	150	94	392270	21	12771	12771	8720	18
13	150	102	451379	19	13099	13099	10465	15
13	150	111	602409	19	20151	20151	15649	12
13	150	120	715389	18	27360	27360	23323	10

continued on next page

Table 6.3. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
13	150	127	698011	17	29014	29014	24914	10
13	150	133	687710	17	24386	24386	19714	9
13	150	138	682529	16	27022	27022	23593	9
13	150	141	658355	16	24383	24383	19985	8
13	150	143	662166	16	25583	25583	21097	7
13	150	145	666004	16	26918	26918	22626	8
13	182	5	218646	356	4614	3700	2948	1506
13	182	16	221298	112	5202	4464	2724	789
13	182	27	221454	66	5377	4885	2388	447
13	182	38	224282	47	6342	5850	2382	579
13	182	49	228770	37	5597	5597	1648	34
13	182	60	228854	30	6175	6175	1464	28
13	182	71	237158	26	6485	6485	1235	25
13	182	82	307481	23	8553	8553	6168	122
13	182	93	304693	20	8461	8461	5748	37
13	182	104	305546	18	9321	9321	5238	17
13	182	115	300217	16	8737	8737	5161	15
13	182	124	385192	16	13160	13160	9014	13
13	182	135	365004	14	10807	10807	7837	11
13	182	146	504015	14	16857	16857	12244	9
13	182	155	548267	13	22566	22566	18841	8
13	182	162	539117	13	20255	20255	16776	7
13	182	167	545998	13	20653	20653	16221	7
13	182	171	508107	12	16674	16674	12150	6
13	182	174	511535	12	19287	19287	14996	6
13	182	175	512429	12	14577	14577	10903	6
13	222	7	199978	248	4456	3596	2834	1252
13	222	20	202424	87	5053	4315	2668	949
13	222	33	205140	53	5425	4933	2349	517
13	222	47	204892	37	5085	5085	1659	36
13	222	60	206750	29	5825	5825	1647	27
13	222	73	210396	24	6116	6116	1601	24
13	222	87	211360	20	6851	6851	1419	20
13	222	100	221162	18	7174	7174	1203	18
13	222	113	224604	16	7277	7277	913	16
13	222	127	222393	14	6888	6888	629	13
13	222	140	290073	13	7703	7703	6128	12
13	222	151	286996	12	9325	9325	5869	10
13	222	164	283598	11	10095	10095	5006	9
13	222	178	357878	11	10878	10878	7369	8
13	222	189	399981	10	13202	13202	9945	6
13	222	197	408207	10	16842	16842	10732	6
13	222	204	466652	10	16451	16451	12950	5
13	222	209	451551	10	15611	15611	12263	5
13	222	212	453891	10	17300	17300	14135	3
13	222	214	455336	10	15654	15654	12373	5
13	270	8	179142	209	4356	3496	2772	1126
13	270	24	181040	70	4503	4011	2291	406

continued on next page

Table 6.3. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
13	270	40	183696	42	5207	4715	2276	601
13	270	57	187684	30	5005	5005	1620	30
13	270	73	186344	23	5668	5668	1576	23
13	270	89	190224	19	5995	5995	1572	19
13	270	105	190834	16	6667	6667	1499	15
13	270	121	194878	14	6913	6913	1320	14
13	270	138	210132	13	7246	7246	1055	13
13	270	154	199616	11	7363	7363	752	11
13	270	170	203188	10	7003	7003	540	10
13	270	184	280703	10	9740	9740	6684	9
13	270	200	273859	9	9693	9693	6172	8
13	270	216	263844	8	9200	9200	5927	7
13	270	229	273585	8	9587	9587	5108	6
13	270	240	335381	8	13085	13085	8045	4
13	270	248	392686	8	16898	16898	11479	5
13	270	254	397683	8	12546	12546	9841	4
13	270	258	382987	8	16627	16627	10102	4
13	270	260	356163	7	12586	12586	9538	3
14	28	24	5550430	138	204080	204034	200590	75
14	28	25	5054931	127	184323	184279	180576	69
14	28	26	4686691	118	170751	170707	167123	69
14	28	27	4358808	110	159056	159012	155314	64
14	34	5	645248	210	10407	9492	8243	3174
14	34	7	754665	154	14781	13985	11779	3275
14	34	9	918166	125	21572	20771	17977	4375
14	34	23	3396793	90	125453	124915	121170	700
14	34	25	3123537	82	113742	113698	109997	51
14	34	27	2868354	75	105154	105110	101446	49
14	34	29	2696212	70	100025	99983	96481	46
14	34	30	2584574	67	93413	93371	89592	43
14	34	31	2560361	65	91205	91174	87359	41
14	34	32	2396401	62	85477	85446	81935	41
14	34	33	2321905	60	84661	84636	81254	40
14	42	6	418586	165	6079	5258	3848	1231
14	42	9	499816	112	8744	7923	6043	1544
14	42	11	497767	92	8286	7465	5577	1573
14	42	14	578612	74	13443	12890	10120	1756
14	42	16	655663	66	16151	15598	12969	1619
14	42	19	980222	60	28627	28079	24961	2098
14	42	21	1208610	57	40001	39453	36071	1677
14	42	24	1909889	59	68207	68161	64563	46
14	42	26	1871502	55	65784	65740	62234	42
14	42	29	1831696	51	65338	65296	61615	38
14	42	31	1776685	48	61938	61907	58209	36
14	42	34	1692207	45	58128	58110	54792	32
14	42	36	1679846	43	59304	59286	55438	32
14	42	37	1651400	42	58429	58411	54520	32
14	42	39	1545659	40	55238	55220	51481	29

continued on next page

Table 6.3. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
14	42	40	1551810	39	54678	54671	50834	23
14	51	5	331998	192	4510	3656	2572	1334
14	51	8	330962	120	4650	3910	2560	880
14	51	11	330793	87	5332	4592	2611	1156
14	51	14	411850	70	7655	7163	4885	841
14	51	17	410324	58	8211	7719	4898	706
14	51	20	488486	50	11328	10836	8088	834
14	51	23	485034	44	10295	9803	6778	829
14	51	26	638147	40	16715	16715	13649	222
14	51	29	786953	37	23931	23931	20042	29
14	51	32	1168684	37	40093	40093	36952	24
14	51	35	1205305	35	39330	39330	35731	21
14	51	38	1252689	34	42615	42615	39011	19
14	51	41	1242597	32	41974	41974	38154	18
14	51	43	1225824	31	43300	43300	39796	17
14	51	45	1202816	30	39831	39831	35944	16
14	51	47	1177971	29	38347	38347	34783	15
14	51	48	1092712	28	36536	36536	32276	14
14	51	49	1099447	28	35996	35996	32128	13
14	**62**	**2**	**261134**	**465**	**3318**	**2420**	**2200**	**1596**
14	62	6	323628	158	4103	3363	2598	968
14	62	9	323398	105	4986	4246	2831	986
14	62	13	323016	73	4577	4085	2729	515
14	62	17	323656	56	5730	5238	2813	569
14	62	20	326552	48	5757	5265	2817	736
14	62	24	324853	40	3906	3906	1916	72
14	62	28	403189	35	6609	6609	3773	138
14	62	32	474790	31	10820	10820	7829	76
14	62	35	478933	29	10874	10874	7340	26
14	62	39	625909	27	17390	17390	14789	22
14	62	42	838536	26	27520	27520	23023	17
14	62	46	816180	24	27486	27486	24268	15
14	62	50	858927	23	28176	28176	23716	11
14	62	53	841732	22	27276	27276	23639	12
14	62	55	906078	22	29425	29425	25445	12
14	62	57	823756	21	26996	26996	22692	11
14	62	58	881096	21	29095	29095	25289	11
14	62	59	885920	21	29549	29549	26149	11
14	62	60	890286	21	30626	30626	26072	10
14	75	2	235470	455	3336	2438	2059	1534
14	75	7	237046	130	3447	2707	1859	935
14	75	11	239184	83	3828	3088	1844	1031
14	75	16	240352	57	3619	3127	1442	504
14	75	20	314834	47	5290	4798	3248	743
14	75	25	317502	38	3965	3965	2601	123
14	75	29	318051	33	5973	5973	2919	71
14	75	34	316763	28	5882	5882	2812	78
14	75	38	316858	25	6376	6376	2586	121

continued on next page

Table 6.3. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
14	75	43	397200	23	9104	9104	6137	21
14	75	47	392112	21	8653	8653	5426	19
14	75	51	451306	19	12409	12409	8424	15
14	75	55	599085	19	17036	17036	13567	12
14	75	60	714868	18	22066	22066	18422	11
14	75	64	699476	17	21148	21148	17498	9
14	75	67	689011	17	21851	21851	17552	9
14	75	69	682475	16	21351	21351	17511	9
14	75	70	656226	16	20039	20039	16145	8
14	75	72	663917	16	21995	21995	16860	6
14	91	3	219210	297	3238	2384	1986	1243
14	91	8	221572	112	3458	2718	1845	786
14	91	14	223110	64	3359	2867	1523	457
14	91	19	224464	47	3744	3252	1515	581
14	91	25	227398	36	2939	2939	802	35
14	91	30	229080	30	3117	3117	720	29
14	91	35	233764	26	3307	3307	618	26
14	91	41	307534	23	5134	5134	3429	107
14	91	46	302250	20	5820	5820	3185	22
14	91	52	305415	18	6783	6783	3188	16
14	91	57	298143	16	5359	5359	3085	15
14	91	62	385090	16	9360	9360	6411	14
14	91	67	363339	14	7790	7790	5325	11
14	91	73	503458	14	15791	15791	12751	9
14	91	77	546225	13	17261	17261	13730	6
14	91	81	538573	13	18057	18057	15413	7
14	91	84	546730	13	16078	16078	12934	7
14	91	86	508737	12	16459	16459	12021	6
14	91	87	511166	12	16299	16299	11642	6
14	91	88	513513	12	16121	16121	12674	6
14	111	3	199794	289	3208	2354	1962	1235
14	111	10	202544	87	3418	2678	1817	949
14	111	17	203910	51	3583	3091	1491	528
14	111	23	207684	38	3708	3216	1492	677
14	111	30	206990	29	2943	2943	814	28
14	111	37	213916	24	3481	3481	794	24
14	111	43	220342	21	3838	3838	720	20
14	111	50	221422	18	4107	4107	595	17
14	111	57	227066	16	4439	4439	438	16
14	111	63	220788	14	4532	4532	317	12
14	111	70	290344	13	7923	7923	4232	12
14	111	75	285262	12	6848	6848	3463	11
14	111	82	283616	11	8281	8281	3238	9
14	111	89	357832	11	9015	9015	5919	8
14	111	94	398542	10	12974	12974	8263	5
14	111	99	460576	10	15273	15273	11766	6
14	111	102	466149	10	13152	13152	10640	5
14	111	104	450465	10	14516	14516	10762	5

continued on next page

Table 6.3. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
14	111	106	453677	10	12947	12947	9556	5
14	111	107	455024	10	14487	14487	10353	5
14	135	4	179400	209	3168	2314	1931	1120
14	135	12	181268	70	2823	2331	1470	406
14	135	20	183990	42	3366	2874	1462	602
14	135	28	184422	30	2375	2375	787	29
14	135	36	192066	24	3152	3152	802	22
14	135	45	192790	19	3355	3355	784	18
14	135	53	192960	16	3976	3976	742	15
14	135	61	197078	14	4122	4122	650	14
14	135	69	210294	13	4710	4710	520	13
14	135	77	199868	11	4708	4708	370	9
14	135	85	203356	10	4726	4726	265	10
14	135	92	280776	10	7846	7846	4774	9
14	135	100	273982	9	7142	7142	4256	7
14	135	108	264050	8	7512	7512	4177	6
14	135	115	329110	8	9547	9547	6676	6
14	135	120	335362	8	9323	9323	6549	5
14	135	124	392671	8	12228	12228	7281	5
14	135	127	380687	8	10648	10648	7749	4
14	135	129	383053	8	10559	10559	7528	4
14	135	130	356199	7	11790	11790	7854	4
16	9	7	2668451	70	40142	40083	38628	45
16	9	8	2392160	61	35413	35374	33837	40
16	11	7	1808389	51	26142	26083	24687	38
16	11	8	1757214	47	24343	24304	22969	32
16	11	9	1637526	42	22724	22706	21081	31
16	11	10	1548682	39	22613	22606	21056	23
16	13	3	412539	82	2633	2191	1700	764
16	13	4	406817	61	3028	2586	1807	701
16	13	5	483723	50	3670	3228	2439	681
16	13	6	638167	43	5367	5367	4338	354
16	13	7	1100806	41	14427	14427	12843	29
16	13	8	1295537	38	17455	17455	15959	24
16	13	9	1254035	35	17768	17768	16246	20
16	13	10	1234528	33	17133	17133	15583	18
16	13	11	1191068	30	17076	17076	15397	15
16	13	12	1182383	29	15694	15694	14318	13
16	16	3	322217	79	2090	1648	1113	534
16	**16**	**4**	**321217**	**59**	**1923**	**1481**	**1012**	**545**
16	16	5	325307	48	2060	1618	1154	734
16	16	6	360866	40	1817	1817	1327	174
16	16	7	400007	35	2270	2270	1290	93
16	16	8	475297	31	3417	3417	2306	27
16	16	9	546689	28	6415	6415	4923	24
16	16	10	680905	26	7440	7440	6169	18
16	16	11	886071	25	10958	10958	9867	16
16	16	12	931577	24	12285	12285	10833	14

continued on next page

Table 6.3. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
16	16	13	833872	22	10694	10694	8968	12
16	16	14	818485	21	9929	9929	8493	11
16	16	15	888553	21	11325	11325	9806	11
16	19	2	237998	114	1891	1273	1030	792
16	19	3	239140	76	1622	1180	797	412
16	19	4	244689	57	1661	1219	873	539
16	19	5	315112	47	2683	2241	1383	732
16	19	6	314018	39	1362	1362	685	126
16	19	7	316405	34	1581	1581	695	62
16	19	9	312103	26	1644	1644	610	79
16	19	10	389188	24	2705	2705	1846	22
16	19	11	384936	22	3387	3387	2058	19
16	19	12	448205	20	3630	3630	2036	17
16	19	13	522589	19	6188	6188	4774	13
16	19	14	659237	19	8480	8480	7144	12
16	19	15	712567	18	8467	8467	7369	11
16	19	16	673846	17	7269	7269	5507	9
16	19	17	735980	17	8719	8719	6742	9
16	19	18	659932	16	8721	8721	6855	7
16	23	2	222054	112	1879	1261	1027	791
16	23	3	223972	75	1521	1079	801	411
16	23	5	227186	45	1717	1275	798	605
16	23	6	230838	38	788	788	206	37
16	23	8	229100	28	977	977	171	28
16	23	9	231782	25	803	803	147	23
16	23	10	301284	23	1770	1770	973	116
16	23	12	300676	19	1801	1801	932	18
16	23	13	306067	18	2208	2208	919	16
16	23	14	308492	17	2035	2035	847	14
16	23	16	372624	15	3396	3396	2084	12
16	23	17	426714	14	3752	3752	2610	10
16	23	18	559924	14	6952	6952	5929	9
16	23	20	533817	13	5414	5414	4000	7
16	23	21	590446	13	6527	6527	4238	5
16	23	22	509956	12	6419	6419	4761	6
16	28	3	204122	73	1502	1060	795	409
16	28	4	206604	55	1590	1148	795	504
16	28	6	209778	37	643	643	211	36
16	28	8	214660	28	932	932	210	28
16	28	9	216674	25	743	743	204	24
16	28	11	214488	20	1035	1035	175	20
16	28	13	218216	17	1066	1066	138	16
16	28	14	222556	16	1182	1182	115	15
16	28	16	287541	14	2279	2279	1405	13
16	28	18	295992	13	1834	1834	927	12
16	28	19	288303	12	2249	2249	818	11
16	28	21	348937	11	4393	4393	2287	7
16	28	22	419059	11	4366	4366	3188	8

continued on next page

Table 6.3. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
16	28	24	401427	10	4838	4838	3716	6
16	28	25	460841	10	4919	4919	3664	6
16	28	26	448400	10	5458	5458	3515	5
16	28	27	455716	10	5214	5214	3742	5
16	34	3	181628	70	1470	1028	786	406
16	34	5	184300	42	1604	1162	784	602
16	34	7	184866	30	635	635	198	28
16	34	9	192702	24	816	816	202	23
16	34	11	199138	20	877	877	204	20
16	34	13	202080	17	953	953	194	16
16	34	15	193266	14	966	966	163	13
16	34	17	207230	13	1602	1602	135	13
16	34	19	216870	12	1076	1076	98	12
16	34	21	222948	11	1470	1470	70	9
16	34	23	280968	10	1846	1846	1428	9
16	34	25	274090	9	2298	2298	1228	7
16	34	27	261107	8	1944	1944	1051	6
16	34	29	327756	8	4021	4021	2610	5
16	34	30	384174	8	4272	4272	2859	5
16	34	31	391287	8	4589	4589	3023	5
16	34	32	381945	8	4174	4174	3100	4
16	34	33	357620	7	4707	4707	2906	4
18	4	3	994314	26	1984	1984	1798	13
18	5	3	716660	22	1481	1481	1219	14
18	5	4	715983	18	1270	1270	1098	9
18	6	2	301908	29	513	513	282	106
18	6	3	294933	19	239	239	104	16
18	6	4	484641	15	851	851	490	10
18	6	5	571520	13	1351	1351	987	7
18	7	2	219224	28	186	186	53	27
18	7	3	277118	19	363	363	296	133
18	7	4	284000	14	335	335	203	13
18	7	5	468578	12	966	966	649	8
18	7	6	510907	11	1137	1137	773	6
18	9	2	182914	26	102	102	50	25
18	9	3	195020	18	146	146	51	17
18	9	4	191294	13	188	188	39	13
18	9	5	208406	11	288	288	23	10
18	9	6	259985	9	280	280	123	7
18	9	7	315384	8	686	686	397	5
18	9	8	349081	7	578	578	219	3

Table 6.4. Costs of block FC-DOF for Lambda-Fact5

Configuration			Copied	Invoc	W.b. insert	W.b. add	R.s. proc	R.s. union
9	60	51	482991	158	64550	64522	63936	8485
9	60	53	450215	147	60235	60218	59583	6852
9	60	55	405639	134	52834	52818	52100	4356
9	60	56	374651	125	48390	48364	47787	3095
9	60	57	381807	125	48949	48929	48300	2632
9	74	55	265528	89	39148	39135	38387	7066
9	74	59	240676	80	34351	34329	33295	5180
9	74	63	234167	76	31835	31827	31239	3690
9	74	66	220311	72	28890	28882	27829	2773
9	74	68	234085	73	29860	29846	29139	2266
9	74	70	253979	75	31514	31507	30835	1861
9	74	71	254287	75	31191	31177	30598	1470
9	90	41	175200	90	28705	28673	27814	9271
9	90	46	142305	74	23904	23875	22604	6300
9	90	51	186703	76	28381	28366	27778	7595
9	90	57	166633	63	27758	27734	26685	6252
9	90	61	142518	56	24510	24487	23817	5688
9	**90**	**67**	**134723**	**51**	**21778**	**21769**	**21087**	**4365**
9	90	72	144858	50	21317	21307	20511	3583
9	90	77	142746	48	20197	20196	19633	2961
9	90	80	145369	47	19577	19572	18935	2237
9	90	83	158048	48	20227	20220	19415	1704
9	90	85	161111	48	20296	20292	19429	1401
9	90	86	163742	48	20587	20581	19669	1197
9	90	87	175843	49	21677	21667	20938	996
9	109	36	130099	90	24199	24154	23261	8062
9	109	43	145471	78	25416	25380	24623	8706
9	109	49	114712	64	21234	21196	20236	6104
9	109	56	108150	55	19976	19946	19185	5030
9	109	62	129899	53	22551	22524	21314	5840
9	109	69	117367	46	19598	19586	18870	4522
9	109	74	111536	42	18610	18588	17917	3873
9	109	81	104571	38	17338	17324	16714	3483
9	109	87	106880	36	16819	16812	16224	2813
9	109	93	117771	36	16595	16589	16034	2291
9	109	97	116277	35	16183	16178	15644	1970
9	109	100	115132	34	14899	14895	13992	1564
9	109	102	117070	34	14735	14733	13759	1342
9	109	104	128408	35	16222	16217	15642	1137
9	109	105	119759	34	15079	15070	14066	928
9	133	36	145531	92	27529	27461	26460	10061
9	133	44	83796	64	19257	19206	17881	5351
9	133	52	87488	55	19423	19376	18692	5443
9	133	60	80920	47	17522	17485	15784	4084
9	133	68	83801	42	17613	17584	16223	4069
9	133	76	93625	39	17474	17447	16041	4003
9	133	84	79042	34	14759	14746	14105	3225
9	133	90	75734	31	13887	13864	12821	2924

continued on next page

Table 6.4. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
9	133	98	82075	29	14001	13973	12520	2490
9	133	106	76978	26	12414	12411	11811	2553
9	133	113	76246	25	11785	11774	10704	1773
9	133	118	77326	25	10945	10932	10155	1432
9	133	122	88379	25	12093	12090	11382	1467
9	133	125	91362	25	11908	11904	11213	1182
9	133	127	94684	25	11966	11964	11267	1012
9	133	128	97783	25	12000	11993	11229	855
9	162	34	85476	82	21042	20984	19750	7169
9	162	44	65589	59	18049	18004	16627	4646
9	162	53	60983	49	17424	17379	16240	4119
9	162	63	68838	42	17351	17302	16292	4738
9	162	73	58747	36	15237	15198	14069	3242
9	162	83	51175	31	12907	12871	11986	2510
9	162	92	48529	27	11519	11487	10359	2238
9	162	102	53510	25	11613	11598	10672	2473
9	162	110	57471	24	11266	11253	10613	2338
9	162	120	61997	22	11140	11130	9712	1974
9	162	130	57740	20	9676	9670	9051	2240
9	162	138	52695	19	8274	8272	7566	1621
9	162	144	54891	18	8076	8071	7401	1210
9	162	149	69425	19	9286	9281	8724	999
9	162	152	63426	18	8141	8133	7112	830
9	162	155	73311	19	9192	9188	8610	808
9	162	156	74063	19	9246	9245	8668	703
9	197	30	83293	90	21369	21314	19673	7093
9	197	41	71507	64	19537	19479	18284	5925
9	197	53	63492	48	17825	17784	15770	4156
9	197	65	49691	38	16700	16657	15462	3904
9	197	77	46234	31	14826	14781	13438	3064
9	197	89	51685	28	13886	13846	12533	2515
9	197	100	47475	25	12533	12508	11229	2299
9	197	112	58411	23	12603	12577	11691	2556
9	197	124	33154	19	8694	8670	7845	1964
9	197	134	47276	19	9663	9642	8732	2010
9	197	146	44645	17	8582	8563	7875	1732
9	197	158	47460	16	7806	7795	7069	1426
9	197	167	44523	15	7207	7195	6333	1334
9	197	175	42019	14	6376	6366	5792	1170
9	197	181	51983	14	7065	7058	6152	797
9	197	185	45132	14	6195	6188	5282	684
9	197	188	46604	14	6163	6157	5177	554
9	197	190	49161	14	6136	6127	5249	437
9	***240***	***36***	***66165***	***70***	***19519***	***19471***	***17934***	***6155***
9	***240***	***50***	***41801***	***47***	***16228***	***16182***	***14478***	***3655***
9	240	65	33296	35	14906	14868	13495	3242
9	240	79	34702	29	14312	14271	12886	2583
9	***240***	***94***	***31745***	***24***	***13478***	***13441***	***12024***	***2076***

continued on next page

Table 6.4. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
9	***240***	***108***	***30236***	***21***	***12254***	***12212***	***10939***	***2036***
9	***240***	***122***	***25749***	***18***	***10882***	***10847***	***9659***	***2004***
9	240	137	30186	17	9299	9269	7724	1626
9	240	151	35671	16	9111	9093	8335	1851
9	240	163	42159	15	8566	8546	7722	1595
9	***240***	***178***	***31946***	***13***	***6434***	***6417***	***5705***	***1471***
9	***240***	***192***	***25854***	***12***	***5053***	***5032***	***4348***	***1235***
9	***240***	***204***	***44661***	***12***	***6569***	***6558***	***5913***	***1126***
9	240	213	47529	12	6748	6740	5962	1118
9	240	221	41271	11	5845	5840	5298	911
9	***240***	***226***	***40648***	***11***	***5629***	***5625***	***5079***	***763***
9	240	229	43100	11	5507	5502	4969	611
9	240	231	43731	11	5693	5690	5042	514
9	293	26	78960	99	21310	21258	19489	8452
9	293	44	48386	53	17405	17360	15482	4541
9	293	62	37377	36	16073	16035	14537	4192
9	293	79	23859	27	14113	14069	11653	1756
9	293	97	29906	23	13251	13213	10122	1325
9	293	114	23008	19	11573	11538	9232	1259
9	293	132	17806	16	9961	9928	8140	1394
9	293	149	14761	14	8126	8098	6537	1049
9	293	167	24379	13	8897	8867	7816	1804
9	293	185	23699	12	7459	7436	6635	1620
9	293	199	27496	11	6902	6888	5988	1518
9	293	217	22390	10	5177	5171	4224	1308
9	293	234	20283	9	3954	3950	2881	901
9	293	249	28566	9	4807	4800	4139	1102
9	293	260	28779	9	4340	4338	3224	762
9	293	270	32277	9	5056	5051	4479	850
9	293	275	27154	8	3874	3874	2996	586
9	293	280	31270	8	4028	4027	2989	457
9	293	283	32261	8	4059	4057	3092	342
10	30	25	413467	140	49345	49302	48789	7362
10	30	27	403753	133	47579	47553	47095	4848
10	30	28	373729	123	43097	43043	42504	3196
10	30	29	371550	121	41716	41666	41095	2005
10	37	25	274770	94	38269	38228	37648	9775
10	37	27	276068	91	36893	36866	36346	7770
10	37	30	232838	77	29235	29210	28283	4889
10	37	31	226615	75	27888	27871	27155	4122
10	37	33	224004	72	25777	25761	25145	2791
10	37	34	240211	73	27598	27580	26913	2326
10	37	35	251142	74	28007	27979	27158	1791
10	37	36	241075	71	26506	26475	25780	1163
10	45	20	164353	88	24582	24531	23893	8801
10	45	23	174005	79	25285	25231	24662	7996
10	45	26	183599	73	25605	25576	25092	7753
10	45	28	163207	64	23497	23474	22473	6106

continued on next page

Table 6.4. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
10	45	31	143931	54	21823	21798	21283	5593
10	45	33	135780	51	19365	19350	18749	4495
10	45	36	132782	48	17453	17439	16710	3462
10	45	38	143000	48	18127	18116	17654	3110
10	45	40	149725	47	17587	17576	16995	2241
10	45	41	150563	47	17589	17570	17079	1826
10	45	42	165713	48	18853	18839	18010	1519
10	45	43	162230	47	17610	17590	17067	1208
10	55	18	141268	92	23348	23277	22611	9077
10	55	21	139135	78	22736	22687	21725	8510
10	55	25	108241	61	17963	17921	17323	5816
10	55	28	94932	52	16670	16620	15995	4571
10	55	31	131827	53	19945	19900	19384	5974
10	55	35	111711	44	17523	17493	16939	4450
10	55	37	121373	43	18252	18224	17694	4324
10	55	41	96466	36	14323	14299	13389	3293
10	55	44	95266	34	13168	13142	12489	2736
10	55	47	105484	34	13671	13667	13162	2248
10	55	49	115473	34	13820	13812	12936	1988
10	55	51	115346	33	13377	13371	12749	1425
10	55	52	117751	33	13119	13113	12572	1162
10	55	53	114722	33	12845	12832	12295	814
10	67	18	94665	80	18692	18632	17833	7045
10	67	22	76280	61	16769	16690	15821	5128
10	67	26	89151	54	17429	17368	16594	5356
10	67	30	81977	46	16195	16135	15530	5019
10	67	34	74846	40	14951	14919	14237	3936
10	67	38	89887	38	15497	15462	14699	4046
10	67	42	88137	35	14287	14252	13588	3341
10	67	46	78012	31	12574	12544	11534	2652
10	67	50	78667	28	12309	12283	11727	2836
10	67	54	82301	26	12054	12051	11365	2348
10	67	57	80103	25	10835	10824	10007	1878
10	67	60	86334	25	10813	10809	10313	1761
10	67	62	81114	24	9920	9913	9023	1452
10	67	63	87887	24	10274	10265	9181	1196
10	67	64	77969	23	8725	8722	8174	964
10	67	65	87186	24	9340	9333	8407	773
10	81	17	102887	85	19627	19563	18609	7689
10	81	22	82760	62	17820	17737	16621	6157
10	81	27	62544	47	15502	15424	14251	4137
10	81	32	56052	39	14342	14266	13329	3683
10	81	36	53868	35	13471	13417	12527	3228
10	81	41	49660	30	11584	11535	10726	2667
10	81	46	48996	27	10522	10478	9419	2297
10	81	51	56192	25	10421	10411	9698	2612
10	81	55	51621	23	9064	9045	7898	1913
10	81	60	64744	22	10487	10476	9908	2510

continued on next page

Table 6.4. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
10	81	65	56707	20	8667	8659	8117	2207
10	81	69	58466	19	8153	8147	7338	1630
10	81	72	58056	18	7554	7539	6979	1235
10	81	75	58761	18	6885	6884	6178	1034
10	81	76	60930	18	7078	7077	6374	909
10	81	77	67034	18	7368	7365	6471	796
10	81	78	63364	18	6859	6856	5981	591
10	99	15	85812	90	18822	18738	17125	7497
10	99	21	70749	62	16691	16604	15711	5995
10	99	27	58089	46	15205	15137	13708	4059
10	99	33	50312	37	14275	14209	12945	3638
10	99	39	49336	31	13276	13226	12021	3017
10	99	45	50256	28	11952	11892	10813	2608
10	99	50	34620	23	10237	10206	9411	2321
10	99	56	47989	22	11249	11210	10517	2510
10	99	62	35662	19	7882	7853	7141	2002
10	99	67	38189	18	7204	7171	6485	1492
10	99	73	50288	17	7885	7855	6983	1533
10	99	79	51449	16	7337	7321	6117	1441
10	99	84	44387	15	6163	6148	5505	1317
10	99	88	43015	14	5653	5643	5166	1185
10	99	91	50874	14	6224	6215	5361	920
10	99	93	46843	14	5559	5547	4816	715
10	99	94	42192	13	4841	4828	4269	597
10	99	95	52765	14	5865	5849	4677	541
10	***120***	***18***	***67961***	***70***	***17141***	***17066***	***15714***	***6208***
10	***120***	***25***	***45457***	***47***	***15051***	***14976***	***13139***	***3948***
10	120	32	39774	36	13748	13679	12101	3377
10	120	40	42015	29	12914	12841	11206	2549
10	***120***	***47***	***31755***	***24***	***11596***	***11529***	***10429***	***2235***
10	***120***	***54***	***44507***	***22***	***12722***	***12661***	***11701***	***3086***
10	***120***	***61***	***29142***	***18***	***9674***	***9616***	***8399***	***2045***
10	120	68	30930	17	8168	8118	7007	1801
10	120	76	28072	15	6390	6356	5060	1480
10	120	82	35830	14	7088	7067	6379	1806
10	***120***	***89***	***28455***	***13***	***4925***	***4897***	***3826***	***1053***
10	***120***	***96***	***29801***	***12***	***4942***	***4909***	***4326***	***1169***
10	***120***	***102***	***40734***	***12***	***5553***	***5544***	***5017***	***1196***
10	120	107	39246	11	5162	5155	4633	1079
10	120	110	43997	11	5670	5666	5195	952
10	***120***	***113***	***43036***	***11***	***5422***	***5417***	***4880***	***728***
10	120	115	45794	11	5214	5211	4692	579
10	120	116	44810	11	5200	5196	4626	500
10	147	22	53825	54	16023	15948	13952	4570
10	147	31	43380	37	14301	14233	12926	4352
10	147	40	29473	27	12099	12036	9321	1587
10	147	49	38363	23	12550	12477	11145	3049
10	147	57	15188	18	10460	10391	8715	1242

continued on next page

Table 6.4. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
10	147	66	34553	17	11039	10974	9783	2572
10	147	75	14996	14	7097	7059	5602	1282
10	147	84	22470	13	6305	6269	3862	1065
10	147	93	24926	12	6485	6450	5739	1708
10	147	100	26119	11	5847	5825	4897	1510
10	147	109	24320	10	4870	4855	4145	1288
10	147	118	30488	10	4564	4555	3457	989
10	147	125	26697	9	4120	4112	3541	1101
10	147	131	27711	9	3801	3798	3079	836
10	147	135	25222	8	3501	3497	2727	755
10	147	138	29172	8	3619	3614	2764	593
10	147	140	26278	8	3065	3063	2221	504
10	147	142	28396	8	3184	3176	2486	398
11	15	13	431757	143	43137	43078	42681	6637
11	15	14	378194	125	35981	35887	35472	3126
11	19	12	269224	94	33513	33451	32968	10953
11	19	13	243451	85	29191	29150	28588	8317
11	19	14	234426	79	27004	26969	26572	6543
11	19	15	222525	74	24047	24023	23592	5236
11	19	16	212122	70	21567	21540	21099	3860
11	19	17	207472	67	20201	20169	19771	2554
11	19	18	239124	70	22091	22035	21264	1876
11	23	10	167714	88	21243	21176	20651	9458
11	23	12	154241	71	19774	19733	19234	7268
11	23	13	160422	67	20549	20490	20002	7003
11	23	14	160361	61	22011	21963	21036	6734
11	23	16	142079	52	18437	18402	17977	5339
11	23	17	126472	48	15582	15561	14829	4263
11	23	18	131943	47	15078	15061	14651	3800
11	23	20	140907	46	14740	14726	14184	2406
11	23	21	146792	45	14023	13993	13485	1813
11	23	22	162628	46	14910	14869	14163	1152
11	28	9	145732	92	20747	20663	20075	9428
11	28	11	123114	71	17891	17829	17148	7322
11	28	13	103339	57	15272	15223	14518	5343
11	28	14	90147	51	13690	13643	13163	4514
11	28	16	125480	50	16990	16945	16356	5803
11	28	18	109625	42	15361	15324	14848	4561
11	28	19	118606	42	15200	15170	14566	4263
11	28	21	97113	35	12601	12583	11990	3255
11	28	22	97734	34	11649	11624	11101	2935
11	28	24	101153	33	10919	10902	10431	2329
11	28	25	99978	32	10210	10200	9585	1863
11	28	26	108051	32	10515	10482	9799	1276
11	28	27	122335	33	11301	11272	10860	826
11	34	9	85693	77	15245	15159	14170	6168
11	34	11	102174	67	16258	16183	15436	6077
11	34	13	86370	53	14772	14702	14081	5339

continued on next page

Table 6.4. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
11	34	15	89398	47	14626	14560	13849	5271
11	34	17	67724	39	11982	11929	11414	3463
11	34	19	77648	37	12046	12009	11530	3652
11	34	21	89841	34	13736	13700	13198	3552
11	34	23	74615	30	10878	10854	10334	2829
11	34	25	71798	27	10095	10055	9298	2641
11	34	27	75607	25	9996	9978	9522	2516
11	34	29	78289	24	9012	8999	8594	1930
11	34	30	77970	24	8753	8746	8124	1968
11	34	31	87856	24	9014	8995	8113	1542
11	34	32	85303	24	8112	8097	7396	1141
11	34	33	86419	23	7992	7966	7363	603
11	41	9	83107	75	15421	15340	14492	6553
11	41	11	79209	61	15120	15051	14261	5941
11	41	14	63796	46	12829	12757	11872	3889
11	41	16	61235	40	12832	12763	11652	3733
11	41	18	60681	36	11889	11834	10824	3377
11	41	21	50128	29	9967	9936	8872	2495
11	41	23	53992	27	10720	10664	9872	2648
11	41	26	65265	25	10384	10362	9810	2766
11	41	28	55087	23	8528	8504	7593	1992
11	41	30	63740	22	9124	9095	8622	2571
11	41	33	51679	19	6670	6664	6164	1862
11	41	35	51433	18	6343	6330	5922	1584
11	41	36	60458	18	6643	6626	6186	1263
11	41	38	61572	18	6162	6161	5445	1002
11	41	39	64626	18	5987	5977	5285	690
11	41	40	70213	18	6352	6340	5436	408
11	50	7	93244	98	16873	16783	15090	7183
11	50	10	61072	63	13714	13633	12424	4796
11	50	13	55306	47	13225	13150	12021	4039
11	50	16	47928	37	12373	12292	11533	3920
11	50	19	46665	32	11620	11554	10753	3132
11	50	22	46782	27	10495	10433	9514	2624
11	50	25	42470	24	9221	9165	8383	2341
11	50	29	49765	21	9485	9452	8862	2545
11	50	31	33955	19	6876	6845	6261	2027
11	50	34	36649	17	6336	6306	4847	1395
11	50	37	48064	17	6789	6770	5993	1675
11	50	40	51775	16	6454	6431	5888	1464
11	50	42	40997	14	5289	5279	4823	1440
11	50	44	46033	14	5321	5317	4882	1283
11	50	46	40498	13	4239	4224	3766	794
11	50	47	44452	13	4492	4480	4092	718
11	50	48	46232	13	4304	4298	3826	485
11	60	9	54991	66	13588	13504	11091	4449
11	60	13	52234	46	13190	13107	10888	3608
11	60	16	39105	36	11695	11624	10311	3404

continued on next page

Table 6.4. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
11	60	20	38311	29	11188	11108	9835	2528
11	60	23	27747	24	10507	10434	9467	2202
11	60	27	28628	21	8469	8406	7694	2124
11	60	31	23958	18	7203	7146	6548	2071
11	60	34	30706	17	6769	6720	5712	1665
11	60	38	29127	15	5397	5368	4165	1394
11	60	41	32819	14	5990	5969	5424	1828
11	60	44	32912	13	5300	5277	4719	1479
11	60	48	33392	12	4584	4568	4103	1237
11	60	51	38238	12	4652	4631	4173	1171
11	60	53	45355	11	5304	5295	4901	1061
11	60	55	39819	11	4531	4524	4117	996
11	60	56	42597	11	4526	4517	4073	797
11	60	57	43320	11	4368	4357	3933	598
11	60	58	55003	11	5352	5337	4904	490
11	74	7	66815	87	15380	15309	12487	5610
11	74	11	45855	52	12927	12846	11510	4353
11	74	16	26849	33	11137	11059	8709	2339
11	74	20	24287	26	10111	10032	7914	1557
11	74	24	32234	23	10878	10800	9845	3174
11	74	29	16576	18	8734	8652	7106	1411
11	74	33	17851	16	7457	7384	5810	1332
11	74	38	14261	14	5945	5898	4676	1210
11	74	42	11675	12	4790	4744	3715	1089
11	74	47	24822	12	5714	5677	5086	1785
11	74	50	30310	11	5659	5624	4628	1536
11	74	55	27587	10	4608	4586	3536	1222
11	74	59	25582	9	3686	3674	3109	1021
11	74	63	27985	9	3818	3813	3354	1128
11	74	66	37345	9	4349	4341	3935	955
11	74	68	28138	8	3371	3368	2732	786
11	74	70	28588	8	3031	3025	2388	568
11	74	71	28589	8	2986	2982	2294	449
12	8	6	350603	125	29798	29768	29268	9174
12	8	7	319064	109	24354	24292	23999	4292
12	10	7	192928	71	19127	19090	18736	6620
12	10	8	194677	66	16797	16744	16313	4219
12	10	9	197019	62	15110	15070	14440	2564
12	12	6	143697	69	15785	15716	14772	6979
12	12	7	159261	62	17701	17649	17298	6950
12	12	8	133458	50	14920	14890	14492	5624
12	12	9	136656	47	13076	13044	12683	4301
12	12	10	126248	43	10959	10939	10612	2952
12	12	11	135853	42	10161	10113	9589	1655
12	14	6	107314	63	13306	13250	12836	6180
12	14	7	100297	52	12689	12649	12301	5235
12	14	8	114691	48	13446	13395	12966	5243
12	14	9	107279	42	12563	12528	12175	4668

continued on next page

Table 6.4. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
12	14	10	96781	36	10816	10801	10132	3924
12	14	11	97999	34	9737	9724	9372	3097
12	14	12	111782	34	9797	9774	9147	2413
12	14	13	105064	32	8051	8019	7456	1270
12	17	5	103712	72	14276	14213	13765	7796
12	17	6	82799	57	11742	11694	10943	5232
12	17	7	94594	51	12424	12374	11555	5166
12	17	8	75320	42	10633	10587	9368	3823
12	17	9	75520	37	11188	11138	9879	3727
12	17	10	74801	34	9710	9671	8773	3230
12	17	11	93809	33	11512	11471	10978	3806
12	17	12	67076	28	8412	8387	7978	2881
12	17	13	71324	26	7983	7964	7568	2763
12	17	14	70520	24	7140	7124	6815	2089
12	17	15	80119	24	7058	7036	6508	1908
12	17	16	84172	24	6544	6520	5947	1057
12	21	6	72101	55	11384	11337	9977	4664
12	21	7	61487	46	10539	10498	9963	4061
12	21	8	57707	39	10487	10426	9688	3938
12	21	9	37995	33	7960	7916	7210	2461
12	21	11	54443	29	8148	8114	7568	2611
12	21	12	49637	26	8535	8495	8029	2825
12	21	13	61646	25	8834	8801	8453	2853
12	21	14	54239	22	7868	7843	6469	2082
12	21	16	55890	20	6384	6366	6036	2157
12	21	17	64325	19	6708	6687	6346	2005
12	21	18	61080	18	6075	6068	5722	1526
12	21	19	60369	18	5197	5181	4811	1188
12	21	20	71269	18	5534	5509	4909	765
12	25	4	77884	82	12412	12366	11709	7014
12	25	5	56303	61	10959	10916	9854	5120
12	25	7	50136	43	9901	9863	9209	3810
12	25	8	47785	37	10120	10073	9204	3737
12	25	10	43612	30	8865	8830	8173	2880
12	25	11	38743	26	8472	8431	7780	2510
12	25	13	39330	23	7164	7133	6434	2084
12	25	14	35707	21	6401	6377	5890	1845
12	25	16	43811	19	6738	6711	6347	2258
12	25	17	44226	18	6032	6011	5666	1891
12	25	19	47087	16	5571	5548	5186	1820
12	25	20	43811	15	4813	4789	4405	1567
12	25	21	41482	14	4357	4341	3952	1398
12	25	22	42378	14	4094	4088	3615	1273
12	25	23	45008	14	3940	3935	3536	900
12	25	24	51313	14	4082	4069	3491	552
12	30	6	49634	49	10571	10530	8784	3840
12	30	8	34807	35	9081	9046	8058	3153
12	30	10	31697	28	8219	8183	7381	2605

continued on next page

Table 6.4. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
12	30	12	39032	24	8512	8475	7285	2486
12	30	13	34892	22	7877	7844	6866	2307
12	30	15	28503	19	6874	6850	6013	2285
12	30	17	36949	17	7057	7031	6460	2341
12	30	19	29892	15	4499	4463	3339	1308
12	30	20	29711	14	4951	4925	4469	1847
12	30	22	29856	13	4330	4312	3994	1650
12	30	24	38171	12	4453	4427	3551	1145
12	30	26	44924	12	4607	4590	4264	1193
12	30	27	43373	11	4225	4219	3875	1112
12	30	28	42616	11	3772	3757	3481	818
12	30	29	50945	11	4070	4049	3722	477
12	37	6	43031	47	10560	10520	9338	4446
12	37	8	24086	33	8733	8694	6951	2188
12	37	10	22872	26	8063	8025	6307	1696
12	37	12	18524	22	7929	7890	6268	1634
12	37	14	23328	19	7254	7219	5471	1297
12	37	17	33147	16	7129	7098	6229	2392
12	37	19	28888	14	6510	6480	5723	2175
12	37	21	16918	12	4164	4137	2923	1069
12	37	23	24737	12	5063	5046	4490	1802
12	37	25	26430	11	4584	4565	3816	1602
12	37	27	21904	10	3618	3604	3043	1330
12	37	30	20979	9	2825	2812	1864	989
12	37	31	25883	9	3324	3315	2925	1209
12	37	33	26626	8	2918	2908	2630	1020
12	37	34	27876	8	2712	2706	2289	759
12	37	35	27639	8	2423	2418	1894	543
12	37	36	26898	8	2224	2220	1695	340
13	5	4	192281	68	12927	12879	12483	4470
13	6	4	125611	50	11806	11774	11212	5730
13	6	5	140073	46	9332	9283	8891	3064
13	7	3	118202	63	12299	12227	11924	7022
13	7	4	106151	46	11961	11895	11089	5520
13	7	5	98109	37	8861	8828	8534	4143
13	7	6	108177	34	7456	7416	7013	2380
13	9	3	81423	56	10075	10025	9661	6229
13	9	4	66654	40	8366	8310	7882	4071
13	9	5	66910	33	8159	8108	7524	3505
13	9	6	66437	27	7422	7384	6641	2955
13	9	7	71970	24	6498	6469	6216	2668
13	9	8	69371	22	4898	4874	4355	1589
13	11	3	68466	54	8585	8532	7760	4853
13	11	4	59482	40	8089	8024	7546	4146
13	11	5	55206	31	7319	7272	6552	3065
13	11	6	48701	26	6523	6479	6024	2675
13	11	7	51569	22	5835	5791	5257	2244
13	11	8	42842	19	4788	4772	4462	2268

continued on next page

Table 6.4. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
13	11	9	48116	17	4255	4244	3864	1783
13	11	10	60072	17	4085	4055	3798	1143
13	13	3	56357	51	8017	7963	7034	4384
13	13	4	48859	37	7777	7723	7111	3873
13	13	5	32582	28	6120	6081	4997	2171
13	13	6	39822	24	6330	6280	5832	2587
13	13	7	41569	21	5682	5650	4844	2081
13	13	8	40464	18	5788	5741	5439	2503
13	13	9	47855	17	5076	5038	4662	1750
13	13	10	43550	15	4627	4591	4309	1866
13	13	11	35903	13	3308	3294	3063	1614
13	13	12	40475	13	2928	2915	2673	942
13	15	3	60884	51	8898	8826	7997	5212
13	15	4	41546	36	7404	7350	6292	3309
13	15	5	35515	28	6786	6747	5911	2739
13	15	6	34873	24	6398	6358	5467	2335
13	15	7	43649	21	6621	6588	6149	2851
13	15	8	25747	17	4750	4712	4106	1862
13	15	9	31584	16	4849	4809	4503	1997
13	15	10	29817	14	4148	4115	3771	1822
13	15	11	29519	13	3495	3470	3027	1613
13	15	12	34703	12	3733	3701	3386	1567
13	15	13	46445	12	3929	3908	3598	1295
13	15	14	40364	11	2951	2937	2705	799
13	19	3	52378	48	8488	8452	7222	4677
13	19	4	40610	35	7970	7932	7074	4114
13	19	5	30539	27	6574	6539	4312	1771
13	19	6	21481	22	5570	5536	4199	1669
13	19	7	30781	19	6451	6409	5780	2727
13	19	9	28156	15	5674	5622	5035	2128
13	19	10	29981	14	5505	5479	5015	2395
13	19	11	12512	12	2826	2803	2007	1159
13	19	12	29637	12	4625	4600	4123	1830
13	19	13	29543	11	3961	3928	3393	1589
13	19	14	25240	10	3223	3207	2392	1283
13	19	15	22794	9	2688	2671	1757	1063
13	19	16	26176	9	2872	2846	2552	1202
13	19	17	24559	8	2258	2241	1916	940
13	19	18	29613	8	2136	2126	1488	533
14	4	3	79627	30	5084	5054	4958	3616
14	5	2	58797	39	6727	6681	6535	4538
14	5	3	58414	26	6277	6233	5725	3667
14	5	4	59836	20	3856	3814	3512	2305
14	6	2	56151	38	6146	6098	5785	4201
14	6	3	46005	25	5612	5554	5184	2887
14	6	4	42698	19	4379	4359	4148	2866
14	6	5	42501	15	2661	2639	2582	1857
14	7	2	52419	37	6178	6121	5560	4017

continued on next page

Table 6.4. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
14	7	3	30011	23	4872	4824	3397	1986
14	7	4	31424	18	3941	3902	3560	2274
14	7	5	38391	15	3710	3683	3526	2315
14	7	6	33801	12	2282	2275	2128	1464
14	8	2	52510	37	6637	6577	6450	4815
14	8	3	31608	23	4830	4798	4240	2527
14	8	4	36405	18	5297	5260	5137	2947
14	8	5	27256	14	3471	3425	3309	2177
14	8	6	30832	12	3163	3115	3012	1978
14	8	7	32305	10	1986	1963	1846	1120
14	10	2	40988	35	5936	5908	5310	4037
14	10	3	20405	21	4124	4098	3001	1743
14	10	4	18643	16	3468	3439	2511	1584
14	10	5	15783	13	2666	2647	1899	1362
14	10	6	26854	11	3691	3647	3361	1990
14	10	7	18980	9	2239	2218	1958	1375
14	10	8	20762	8	2246	2225	2158	1512
14	10	9	22377	8	1443	1438	1291	936

Table 6.5. Costs of block FC-DOF for Richards

Configuration			Copied	Invoc	W.b. insert	W.b. add	R.s. proc	R.s. union
10	8	7	4215818	6748	166442	165991	165962	29704
10	9	6	3560761	6033	224511	224253	224139	85263
10	9	7	3107563	5243	160663	160465	160445	62526
10	9	8	2949754	4776	124361	124153	124117	44701
10	11	6	1796246	4199	164308	164081	163970	101725
10	11	7	2167571	3869	171319	171175	171138	96578
10	11	8	2120608	3593	146440	146316	146212	72900
10	11	9	2031689	3346	112517	112361	112315	51316
10	11	10	2088334	3235	91799	91669	91653	28690
10	13	6	564276	3369	56466	56146	56108	39866
10	13	7	820548	3037	76252	76016	75977	44532
10	13	8	1071155	2780	100407	100221	100175	57013
10	13	9	1520018	2704	117510	117344	117238	66263
10	13	10	1490926	2541	91617	91490	91463	44914
10	13	11	1430346	2400	68850	68733	68641	28609
10	13	12	1447058	2321	53418	53303	53274	17603
10	16	6	112585	3060	35269	34695	34643	17048
10	16	7	149586	2645	32271	31872	31822	18470
10	16	8	221826	2349	33786	33424	33384	20702
10	16	9	384428	2163	44565	44280	44238	25974
10	16	10	575071	2026	57236	57055	57013	32640
10	16	11	832751	1930	72666	72490	72304	41303
10	16	12	1034594	1851	81350	81235	81186	46198
10	16	13	1056175	1790	65717	65594	65564	30996
10	16	14	1042023	1728	47641	47547	47521	23910
10	16	15	1060282	1679	37229	37159	37130	6921
10	20	7	58762	2592	34394	33880	33859	12994
10	**20**	**8**	**56758**	**2266**	**30270**	**29775**	**29686**	**11354**
10	20	9	63860	2018	27222	26817	26774	11971
10	20	10	73656	1821	24861	24251	24207	10662
10	20	11	102806	1666	23803	23446	23407	11646
10	20	13	264118	1459	32355	32017	31954	17582
10	20	14	384488	1390	38980	38738	38615	22046
10	20	15	579959	1349	50819	50652	50606	29365
10	20	16	728358	1309	55975	55882	55839	31920
10	20	17	757425	1280	49796	49714	49604	23033
10	20	18	750381	1245	35863	35785	35743	16243
10	20	19	772130	1222	29366	29285	29181	5780
10	24	5	55492	3624	39026	38442	38406	16703
10	24	6	48697	3016	37595	37092	37062	14140
10	24	8	39664	2257	34035	33557	33537	10733
10	24	9	41374	2007	32249	31804	31778	10526
10	24	11	41688	1643	25321	24789	24739	8916
10	24	12	42735	1506	22770	22241	22167	8637
10	24	14	55973	1295	18949	18428	18357	7807
10	24	15	75396	1214	18619	18218	18174	8536
10	24	16	113499	1148	19490	19093	19056	10084
10	24	18	304598	1063	33775	33482	33428	17215

continued on next page

Table 6.5. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
10	24	19	446289	1036	42320	42147	42105	22207
10	24	20	561608	1012	43692	43587	43545	24537
10	24	21	584271	994	37891	37803	37765	17676
10	24	22	586645	974	29086	29008	28910	12285
10	24	23	601428	957	23886	23797	23763	5663
10	29	6	42671	3011	37865	37501	37455	14030
10	29	8	34294	2254	36295	35671	35632	10828
10	29	10	31422	1803	33604	32881	32854	8999
10	29	11	29885	1638	31501	30822	30802	8322
10	29	13	29980	1386	26137	25687	25597	7422
10	29	15	30670	1202	21344	20653	20629	6757
10	29	17	32813	1061	17836	16970	16923	6194
10	29	18	36460	1003	16193	15764	15744	5872
10	29	20	51256	906	13965	13479	13427	6007
10	29	21	97402	871	17829	17475	17434	8127
10	29	23	229499	819	26070	25798	25690	12879
10	29	25	447567	792	37155	37060	37018	19661
10	29	26	453930	776	29844	29760	29654	14496
10	29	27	455726	763	23868	23785	23743	10267
10	29	28	468146	754	19675	19572	19428	5196
10	35	5	44012	3613	39032	38793	38740	16733
10	35	7	33331	2575	37633	37386	37336	12139
10	35	9	29016	2000	36775	35996	35913	9862
10	35	12	25088	1499	33276	32306	32246	7689
10	35	14	22895	1284	29578	29130	29064	6708
10	35	16	20952	1123	25694	24654	24597	5835
10	35	18	20528	999	21213	20693	20670	5213
10	35	20	23635	900	18072	17264	17191	5279
10	35	22	24723	818	15743	14765	14745	4727
10	35	24	28844	750	12825	12230	12206	4546
10	35	26	47556	696	12738	12310	12265	5162
10	35	28	105561	655	14920	14469	14426	7025
10	35	30	268686	632	27551	27341	27156	14156
10	35	31	349719	624	29292	29175	29064	15150
10	35	32	362738	617	25211	25114	25075	11029
10	35	33	364440	608	20346	20222	20195	8199
10	35	34	377486	602	16265	16143	16023	4769
11	5	4	2607278	4238	136997	136799	136792	74576
11	6	4	1825363	3179	120304	120216	120062	80030
11	6	5	1716882	2841	68341	68229	68223	38585
11	7	4	742328	2572	76840	76660	76500	57204
11	7	5	1315292	2320	84630	84510	84441	58413
11	7	6	1315623	2144	67758	67626	67622	38726
11	8	4	263423	2348	32925	32674	32673	23903
11	8	5	756734	2071	69569	69420	69339	52137
11	8	6	1053458	1844	69325	69248	69244	45830
11	8	7	1038680	1725	39576	39500	39494	22229
11	10	3	61314	3001	28802	28445	28429	15152

continued on next page

Table 6.5. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
11	**10**	**4**	**58925**	**2249**	**23376**	**23006**	**22999**	**12323**
11	10	5	77938	1807	19834	19521	19488	12226
11	10	6	163327	1533	22129	21843	21838	15251
11	10	7	522495	1414	49238	49110	49090	37374
11	10	8	735425	1298	45700	45642	45537	31958
11	10	9	745285	1240	28594	28525	28452	16112
11	12	3	50506	2992	28909	28439	28416	14248
11	12	4	41607	2240	26138	25777	25768	11347
11	12	5	41257	1792	22227	21654	21633	9586
11	12	6	44502	1495	17972	17490	17489	8804
11	12	7	58953	1285	15461	15139	15137	9005
11	12	8	126032	1140	17677	17327	17283	11539
11	12	9	403177	1073	38070	37900	37895	27670
11	12	10	568759	1004	36430	36371	36371	24917
11	12	11	584958	970	23214	23140	23064	12899
11	15	3	43031	2987	28898	28659	28624	14168
11	15	4	35148	2237	27726	27252	27231	10809
11	15	5	29928	1787	25891	25704	25685	9330
11	15	6	28728	1489	22999	22528	22497	7751
11	15	7	28283	1276	18989	18878	18851	7262
11	15	8	28633	1117	15919	15809	15803	6664
11	15	9	32941	994	13298	12949	12949	6046
11	15	10	42759	896	12158	11871	11867	6353
11	15	11	109530	827	14633	14498	14496	9278
11	15	12	273085	783	29439	29289	29282	20446
11	15	13	425487	750	27945	27868	27833	18354
11	15	14	440125	730	21478	21388	21381	11520
11	18	3	40306	2985	29224	29016	28966	14178
11	18	4	30893	2234	29011	28856	28810	11250
11	18	5	27759	1786	27370	27262	27192	9119
11	18	6	25549	1488	25073	24972	24941	7717
11	18	7	23194	1275	24214	23908	23871	6867
11	18	8	21905	1115	20982	20609	20588	6180
11	18	9	20231	991	17975	17350	17332	5483
11	18	10	21998	892	14792	14586	14565	5216
11	18	11	23021	811	12614	12494	12485	4963
11	18	12	27057	744	11277	10978	10974	4830
11	18	13	35897	688	10573	10064	10060	4994
11	18	14	84739	646	12386	12073	12071	6896
11	18	15	235712	622	23414	23294	23291	16452
11	18	16	342653	599	22977	22909	22909	14826
11	18	17	353705	585	16018	15930	15885	8347
12	4	3	1053156	1846	57992	57970	57901	46194
12	5	3	336380	1573	42813	42802	42733	34873
12	5	4	749115	1312	42614	42606	42605	33073
12	6	2	36110	2218	16642	16634	16634	10823
12	6	3	35754	1479	11374	11363	11363	8181
12	6	4	255904	1161	33492	33485	33483	26842

continued on next page

Table 6.5. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
12	6	5	564323	1000	31030	31030	31025	25044
12	8	2	29173	2213	18060	18052	18044	10601
12	8	3	24289	1474	15498	15491	15486	7605
12	8	4	21654	1105	11278	11273	11271	6036
12	8	5	24781	885	8584	8581	8581	5175
12	8	6	160906	759	22007	22006	21994	16808
12	8	7	389311	689	21686	21679	21678	17017
12	9	2	28462	2212	18220	18211	18201	10495
12	9	3	22670	1473	16264	16258	16246	7520
12	9	4	19458	1104	13232	13229	13222	5857
12	9	5	18824	883	9926	9922	9922	5044
12	9	6	22365	737	7948	7946	7946	4436
12	9	7	143021	648	20042	20042	20041	14713
12	9	8	337952	596	19043	19037	19036	14721
13	3	2	719438	1255	62732	62199	62199	61701
13	4	2	29725	1084	10847	9069	9062	7908
13	4	3	455645	793	39304	38633	38633	38001
13	5	2	20421	1082	11465	9667	9667	6840
13	5	3	22602	721	8765	6993	6991	5547
13	5	4	330441	579	29433	28715	28715	28023

Table 6.6. Costs of block FC-DOF for JavaBYTEmark

Configuration			Copied	Invoc	W.b. insert	W.b. add	R.s. proc	R.s. union
18	3	2	145533	10	15	15	15	12
18	4	2	17634	9	56	43	43	23
18	4	3	60291	6	33	28	28	10
18	5	2	17634	9	38	33	33	23
18	5	3	17634	6	38	33	33	15
18	5	4	69040	5	17	14	9	5
18	6	2	17634	8	23	19	13	8
18	**6**	**3**	**17634**	**6**	**23**	**19**	**19**	**13**
18	6	4	17634	4	32	25	19	8
18	6	5	63987	4	29	24	24	7

Table 6.7. Costs of block FC-DOF for Bloat-Bloat

Configuration			Copied	Invoc	W.b. insert	W.b. add	R.s. proc	R.s. union
16	16	11	12876471	295	697181	581742	579250	87920
16	16	12	13085042	279	688374	578839	575994	87195
16	16	13	12461415	263	610242	519298	517184	66705
16	16	14	11862567	250	503342	457599	455031	46732
16	19	10	4757169	257	421318	318856	314477	92788
16	19	12	7389650	230	519561	414061	409521	85385
16	19	13	8676499	221	543285	436564	433020	82552
16	19	14	9014138	210	495443	409878	407223	68459
16	19	15	9437192	202	472295	397909	395184	55891
16	19	16	9300046	194	444606	383907	381170	45472
16	19	17	8845748	186	380115	347275	344679	36538
16	19	18	8599180	180	355629	329192	325826	25616
16	23	10	3281896	247	435491	309373	302432	102333
16	23	13	3365409	190	389935	275092	270386	78510
16	23	14	4102366	180	388507	274431	268658	67722
16	23	16	5403719	164	329094	274892	271390	53579
16	23	17	6076636	158	404255	316466	313734	59772
16	23	18	6871866	154	354608	303957	301454	47008
16	23	20	6976001	145	345792	294824	292266	38575
16	23	21	6793647	140	287276	259172	256675	27186
16	28	11	2636562	221	500105	328117	322775	106683
16	28	13	2462227	186	432114	278789	272445	89736
16	**28**	**14**	**2308104**	**172**	**387199**	**259105**	**254044**	**82203**
16	28	16	2370084	151	262564	198459	190897	56246
16	28	18	2594284	135	227942	177462	173344	45733
16	28	19	2971513	129	270221	207777	205245	53744
16	28	21	4185737	121	288240	234322	231459	48123
16	28	22	4737228	118	292317	240525	238062	40629
16	28	24	5419853	112	281177	241342	237944	35775
16	28	25	5417102	110	251690	223261	220050	30480
16	28	26	5226810	107	244684	220074	217212	29426
16	34	11	2192541	217	576701	368060	353858	123741
16	34	13	2373520	185	517779	334863	323173	106029
16	34	15	2179527	160	465836	301691	293087	93951
16	34	17	2100045	140	376493	262244	255204	77093
16	34	19	1980670	125	303203	212029	200805	61920
16	34	21	2087321	114	268910	186589	183095	56086
16	34	23	2208534	104	225715	171745	165140	53348
16	34	25	2422813	97	203739	156587	152282	47004
16	34	27	3397026	92	256724	191829	189044	41483
16	34	29	4084392	88	218667	185636	181895	28390
16	34	30	4257942	87	232291	187859	184743	27701
16	34	31	4263674	85	197347	173381	170256	23806
16	34	32	4150617	84	171133	161101	158616	16439
16	34	33	4110114	82	165909	159934	155973	11502
16	41	9	2557773	267	665434	439201	406936	176233
16	41	11	2199908	217	630212	387455	376662	139670
16	41	14	1933209	169	531921	334913	322383	101384

continued on next page

Table 6.7. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
16	41	16	1934193	148	523459	328822	316784	97436
16	41	18	1812181	131	439550	283148	270958	80269
16	41	21	1792123	113	376833	247645	242972	71088
16	41	23	1725472	103	282533	198499	190762	62468
16	41	26	1710140	91	243671	171798	163926	51203
16	41	28	1837780	85	195852	148986	140559	43220
16	41	30	1808711	79	178290	134693	130427	37376
16	41	33	2440093	73	168190	135968	132738	25965
16	41	35	2951941	70	165079	138012	134087	21856
16	41	36	3450948	70	207801	169217	166457	27039
16	41	38	3558620	68	165346	147557	143792	16955
16	41	39	3368551	66	149437	136373	133285	15961
16	41	40	3342310	65	136462	126982	124373	12590
16	50	10	2917164	242	697371	465025	435269	148099
16	50	13	1745003	181	629755	397008	371871	130960
16	50	16	1871008	147	573977	358575	342814	98994
16	50	19	1807320	124	503497	324014	309987	92931
16	50	22	1734156	107	481070	315242	300075	92151
16	50	25	1475550	94	412913	261795	251863	74792
16	50	29	1495594	81	263025	171706	157670	46688
16	50	32	1644572	74	223829	167061	162338	47235
16	50	34	1510633	69	200363	147415	140607	41026
16	50	37	1787088	64	203688	151490	144130	41421
16	50	40	1713054	59	130734	106078	103019	26135
16	50	43	2399719	56	171828	136692	133111	25970
16	50	44	2364272	55	143358	120423	116643	22775
16	50	46	2912999	54	141771	129375	126755	16904
16	50	47	2904596	53	135927	124808	122323	12793
16	50	48	2828231	53	116829	108206	106247	10099
16	61	9	2090674	262	696119	445838	419547	176023
16	61	13	1744819	180	660497	405345	392560	138056
16	61	16	1638790	146	607421	381486	363965	115084
16	61	20	1672035	117	511574	334324	318046	92692
16	61	24	1421898	97	576340	352568	340037	88362
16	61	27	1420755	86	438950	285302	271771	76780
16	61	31	1378990	75	369144	239228	220126	69642
16	61	35	1441311	67	359757	222249	214292	61134
16	61	38	1377644	61	252319	177790	168710	44810
16	61	41	1230787	57	196377	135910	126911	35204
16	61	45	1493424	52	147821	121526	117888	34381
16	61	49	1281836	47	105801	84594	76913	22471
16	61	52	1678582	45	123422	97364	94037	22572
16	61	54	2044713	44	129698	109926	106406	21426
16	61	56	2197506	43	127513	103417	100745	16825
16	61	57	2384827	43	132252	108318	105147	19145
16	61	58	2310417	42	111174	98291	95350	11988
16	61	59	2202838	42	92991	85392	82252	8595
16	74	7	2043614	334	800511	510958	451376	219306

continued on next page

Table 6.7. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
16	74	11	2043878	213	695889	449895	395917	158557
16	74	16	1480869	144	697802	423865	393569	120620
16	74	20	1630315	116	658996	426607	388811	116269
16	74	24	1359220	96	589643	382701	356967	91548
16	74	29	1363591	80	488961	313117	279707	72130
16	74	33	1475290	70	473644	296743	284956	74958
16	74	38	1282772	61	301231	203177	169674	48188
16	74	42	1242068	55	335280	223183	212594	59927
16	74	47	1251104	49	251540	165603	151510	48722
16	74	50	1158984	46	236284	146654	130137	38779
16	74	55	1297354	42	190819	127702	117573	32659
16	74	59	1392641	39	202170	128872	124395	33790
16	74	63	1445059	37	149725	108730	101802	21504
16	74	66	1655323	36	111369	87052	82949	17994
16	74	68	1636235	35	84935	73634	70793	12922
16	74	70	1824631	34	79387	72305	69954	9202
16	74	71	1952068	34	86974	78068	75506	8343
18	5	4	9245283	195	229680	162498	160792	56116
18	6	4	7283924	170	273905	193935	191957	80397
18	6	5	6759511	144	148606	107053	106620	35584
18	7	4	3183558	154	188507	137072	136695	66305
18	7	5	5738118	131	216454	156928	156031	56533
18	7	6	5662171	116	150169	118656	117910	42946
18	9	3	2007413	198	326689	202555	198898	116110
18	9	4	1943122	148	257830	166951	163626	80986
18	9	5	2054687	119	215405	134582	132389	66321
18	9	6	2520609	101	129884	96150	94286	49848
18	9	7	4072555	90	150750	104085	102204	43558
18	9	8	4043639	82	102881	75815	74096	25120
18	11	4	1970306	148	357575	217101	211421	110703
18	11	5	1709577	118	270670	174443	169575	82429
18	11	6	1622388	98	211515	125346	121549	67216
18	11	7	1812100	84	138307	98348	94121	47648
18	11	8	2362843	75	140745	91413	91058	45420
18	11	9	3128413	68	106829	76620	75348	25979
18	11	10	3221099	63	67895	52383	51752	16002
18	13	3	1794950	195	390698	237804	228152	136491
18	13	4	1823393	147	366605	223640	219871	108653
18	13	5	1679290	117	323586	207571	204866	91205
18	13	6	1499837	97	233325	166457	162589	83302
18	13	7	1492991	83	244104	146494	142879	68373
18	13	8	1575460	73	151447	100194	98745	52467
18	13	9	1548656	65	112562	79992	76317	38661
18	13	10	1840595	59	79805	60347	59528	31636
18	13	11	2709255	55	87937	65630	64283	24123
18	13	12	2749813	52	66175	51939	50361	15535
18	16	3	1767217	194	422384	265789	256414	152336
18	16	4	1658659	146	395542	254080	242193	127448

continued on next page

Table 6.7. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
18	16	5	1632098	116	356411	222971	213700	97465
18	16	6	1441111	97	346786	220517	214666	89070
18	16	7	1451849	83	297484	189885	185661	68755
18	16	8	1350731	72	227789	150800	144891	67346
18	16	9	1336134	64	185466	122508	116414	56599
18	16	10	1402477	58	153690	104797	97877	49713
18	16	11	1368801	53	157622	101670	97904	47029
18	16	12	1535951	49	117061	79493	78207	37855
18	16	13	1641812	45	79337	55569	54716	27455
18	16	14	2067479	42	64800	49957	49358	17465
18	16	15	2178510	40	52396	43493	43117	11508
18	19	2	1823849	290	469220	292737	273406	209105
18	19	3	1673635	193	452494	279894	268698	161989
18	19	4	1479112	144	425509	263913	250578	130343
18	19	5	1586752	116	401978	254730	242891	112365
18	19	6	1423186	96	366763	239264	229409	107215
18	19	7	1257039	82	331355	212877	203160	81186
18	19	9	1274272	64	262960	159716	152873	61047
18	19	10	1170568	57	232347	136966	131669	55918
18	19	11	1348679	53	204841	128496	121590	53699
18	19	12	1348728	48	150447	104959	101361	45011
18	19	13	1224158	44	122494	85634	83696	37611
18	19	14	1201141	41	106291	68640	63391	33398
18	19	15	1162111	38	75756	51790	49674	23557
18	19	16	1436889	36	70131	47225	45693	18990
18	19	17	1770725	35	48965	38299	37416	14452
18	19	18	1935203	33	93387	58620	57333	20958
20	3	2	3878918	78	123991	66039	66039	65445
20	4	2	1589557	73	165288	90758	90020	80715
20	4	3	2552316	50	60379	27688	27688	26692
20	5	2	1232075	72	167093	101486	100398	78518
20	5	3	1201338	48	87907	53049	52379	46159
20	5	4	1911445	37	48288	21616	21530	19638

Table 6.8. Costs of block FC-DOF for Toba

Configuration			Copied	Invoc	W.b. insert	W.b. add	R.s. proc	R.s. union
16	22	19	15022397	202	691552	690821	688179	61101
16	22	20	14442188	194	650465	649727	648159	42269
16	27	20	10215144	159	523931	522731	521201	70165
16	27	22	10371671	147	506829	505971	504092	61392
16	27	23	10928970	145	508015	507252	505247	50695
16	27	24	10471549	140	468950	468246	466517	38481
16	33	19	6772949	149	391021	389673	387750	77126
16	33	21	6859504	135	393432	392148	390579	71144
16	33	22	7043184	130	390656	389306	387458	66895
16	33	24	7375292	121	399131	397991	395859	69880
16	33	26	7340954	113	381777	380724	378827	56643
16	33	28	7922126	108	373738	372858	370897	41140
16	33	29	8293596	107	387928	387226	386004	33455
16	33	30	7771935	103	345727	345088	343718	33232
16	33	31	8025961	102	353078	352484	338322	21835
16	40	20	6204868	139	394565	393269	381663	106112
16	40	23	5222159	118	325390	324061	315733	68888
16	**40**	**25**	**4929155**	**108**	**294684**	**293340**	**280911**	**64753**
16	40	27	5628181	102	315003	313756	303257	52789
16	40	30	5498713	92	280682	279550	278229	40468
16	40	32	5750122	87	297743	296630	295202	48603
16	40	34	6258225	84	320467	319579	307087	37450
16	40	36	6885023	82	319584	318875	307070	33989
16	40	37	6589158	80	297539	296912	283060	27556
16	40	38	6741542	80	302214	301580	288084	20456
16	48	16	7015919	175	453171	451753	436123	198121
16	48	19	6366145	146	413473	412152	396148	156014
16	48	22	6043575	125	387995	386673	370904	118760
16	48	24	6018127	115	372811	371417	360205	97758
16	48	27	5123821	100	325205	323880	316603	66816
16	48	30	5000839	90	299587	298234	285256	56096
16	48	33	4443304	81	254293	253128	243353	47840
16	48	36	5339933	76	301675	300479	291059	49678
16	48	38	5348156	72	278635	277583	267053	39210
16	48	41	5320769	68	260587	259708	257618	32060
16	48	43	5299091	65	254216	253417	240854	26795
16	48	44	5675534	65	263095	262372	260499	24012
16	48	45	5597408	64	249194	248551	237258	20681
16	48	46	5363695	63	237574	236924	235872	14446
16	59	16	6067396	171	427593	426210	411632	190122
16	59	19	5993059	144	405996	404647	383088	168399
16	59	23	5458426	117	380371	379034	360963	126711
16	59	27	5119388	99	346842	345476	322272	95848
16	59	30	5060803	89	333779	332439	321913	82557
16	59	34	4428231	78	294468	293170	282862	56263
16	59	37	3831748	71	247338	246043	226567	48893
16	59	40	4242984	66	257728	256513	247190	50401
16	59	44	3619011	59	210137	209048	198233	39249

continued on next page

Table 6.8. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
16	59	47	4092301	56	220715	219633	209486	37151
16	59	50	4100409	53	209935	209008	198465	31631
16	59	52	4489595	52	225827	224959	214989	29266
16	59	54	4694964	51	219955	219244	217878	18448
16	59	55	4567134	50	210713	210063	200383	17547
16	59	56	4181649	49	187498	186926	185962	16694
16	59	57	4378152	49	189995	189440	188374	10146
16	71	15	5567821	179	401222	399819	356916	174691
16	71	19	5112476	140	383689	382293	348016	152030
16	71	23	4866178	115	349337	348030	312562	119985
16	71	28	4444597	94	334461	333159	304325	105541
16	71	32	4616234	83	324387	323066	306269	94277
16	71	36	4342026	73	300051	298777	279764	69523
16	71	40	3921500	65	258041	256766	239774	41689
16	71	45	3794354	58	256542	255358	243397	50207
16	71	48	3693642	54	244650	243448	225361	46644
16	71	53	3555322	49	197976	196859	189087	32385
16	71	57	3193516	45	188305	187167	178197	31528
16	71	60	3741124	44	192020	191041	180860	24807
16	71	63	3819069	42	208191	207226	198506	27749
16	71	65	3239254	40	145690	144965	136205	16752
16	71	67	3850305	40	179152	178557	167889	17752
16	71	68	3785459	39	171500	170895	162532	8085
16	87	13	6861644	211	484879	483544	447743	260481
16	87	18	5520477	148	422721	421304	375043	193479
16	87	23	4725415	114	359143	357810	332317	142372
16	87	29	4553976	90	345349	344019	320113	122664
16	87	34	4354855	77	331949	330590	305335	103546
16	87	39	4273716	67	314523	313218	268593	84114
16	87	44	3861602	59	292382	291053	262752	68539
16	87	50	3251451	51	247373	246079	228659	47459
16	87	55	3187167	46	218122	216922	181375	40855
16	87	59	2385002	42	167044	165803	157481	30191
16	87	64	2712068	39	170902	169739	155269	29519
16	87	70	2663283	36	162063	160930	151059	28675
16	87	74	3078241	35	158828	157894	156254	22269
16	87	77	2887515	33	152056	151010	144249	23558
16	87	80	3235180	33	167320	166520	164394	20253
16	87	82	3090178	32	141836	141180	139402	16854
16	87	83	3092943	31	146963	146381	139535	18180
16	87	84	3044820	31	132294	131668	122696	7682
16	106	16	5400088	165	417255	415857	372883	194567
16	106	22	4911542	119	378195	376786	342849	158288
16	106	29	3821524	88	321985	320622	264355	104214
16	106	35	3916903	73	339782	338356	319626	120343
16	106	41	3223219	61	290913	289492	239780	76981
16	106	48	3773352	53	305262	303906	270925	80480
16	106	54	3433005	47	258829	257475	235913	58535

continued on next page

Table 6.8. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
16	106	60	2386269	41	213989	212619	179286	37349
16	106	67	2852379	37	205778	204559	172279	37419
16	106	72	2197131	34	160155	158917	152954	31859
16	106	78	2528417	32	171715	170547	164234	31803
16	106	85	2213776	29	129217	128087	112924	22362
16	106	90	1995010	27	105560	104557	93101	16032
16	106	94	2400718	27	139024	138094	126715	18170
16	106	98	2040224	25	103153	102243	97705	16500
16	106	100	2498239	25	123388	122618	116714	14460
16	106	101	2554660	25	125683	124936	117566	10100
16	106	102	2275252	25	101331	100795	99704	7490
18	7	6	10557914	140	294815	293847	293323	44046
18	9	6	7212395	118	289894	288362	288042	80266
18	9	7	7835864	105	266405	265177	259341	62673
18	9	8	7415219	95	216069	215232	215200	32671
18	10	5	5723150	135	291938	290204	284242	97673
18	10	6	6210093	114	276688	275015	265455	74802
18	10	7	6653768	99	259263	257737	248276	67344
18	10	8	6701407	89	231447	230322	229924	53824
18	10	9	6260650	80	187567	186750	179450	30549
18	12	5	6717821	138	341051	339290	323855	143889
18	12	6	5511308	112	282779	281038	272139	91058
18	12	7	4905544	95	241128	239437	232150	72936
18	12	8	5209000	84	238692	237033	228488	59910
18	12	9	4841445	74	191780	190364	190163	46738
18	12	10	5708098	69	206728	205589	196434	47371
18	12	11	5600359	65	170640	169742	169487	25248
18	15	4	6157757	169	330425	328690	309381	187449
18	15	5	4870721	132	269418	267669	257811	134666
18	15	6	4809979	109	260175	258508	232261	108225
18	15	7	4354547	93	264881	263136	254738	96607
18	15	8	4140150	81	227447	225647	217532	62564
18	15	9	4011778	72	199959	198292	197750	51809
18	15	10	4593828	66	217149	215420	207852	59952
18	15	11	4249132	60	189257	187734	179803	47988
18	15	12	4084660	55	163333	162004	154811	40091
18	15	13	4339548	51	155471	154426	147815	36485
18	15	14	4143888	48	126194	125359	119563	21872
18	18	5	5042347	131	287392	285709	247083	147913
18	18	6	4788418	109	281559	279887	246142	129767
18	18	7	4110896	92	235952	234263	222178	94438
18	18	8	4588074	82	252808	251087	238426	92465
18	18	9	4269130	72	232024	230281	208545	62143
18	18	10	3844825	64	202414	200696	186812	55014
18	18	11	3669324	58	191400	189827	185155	49506
18	18	12	3475408	53	171920	170370	159075	41369
18	18	13	3406012	49	153469	152073	144916	35399
18	18	14	3544328	46	158764	157255	156563	39395

continued on next page

Table 6.8. continued

Configuration			Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc	R.s. union
18	18	15	3329820	43	137208	135835	128533	30683
18	18	16	3287413	40	116152	115149	107842	27048
18	18	17	3836959	39	127504	126662	119261	15473
18	22	5	4917512	130	288535	286734	264264	156815
18	22	6	4612817	108	285171	283426	263832	145352
18	22	7	3973449	91	250471	248765	209590	104140
18	22	9	4147048	71	257294	255593	232516	101679
18	22	10	3777769	64	230397	228692	206185	71029
18	22	11	3640102	58	218415	216708	192405	61628
18	22	13	3035591	48	178309	176616	164597	50916
18	22	14	2828839	45	165702	164029	146328	42830
18	22	15	3121825	42	167960	166327	154361	42704
18	22	16	3267859	40	168708	167178	166278	48558
18	22	18	3142156	35	138456	137147	125392	29120
18	22	19	2831318	33	115127	113811	109332	23176
18	22	20	2980521	32	104751	103811	103716	20637
18	22	21	2722441	31	78071	77183	77033	11231
18	27	6	5018366	108	310200	308441	272225	160868
18	27	7	3843051	90	258424	256720	204359	112017
18	27	9	4176674	71	271754	270022	244730	122197
18	27	11	3627756	57	232750	231047	217383	85758
18	27	12	3574707	53	238065	236361	220664	86245
18	27	14	3162265	45	204877	203242	189230	50264
18	27	15	2744717	41	187890	186165	183113	53894
18	27	17	2581970	36	145799	144224	134180	35182
18	27	18	2518603	34	137983	136392	126686	35129
18	27	20	2505493	31	122418	120866	108873	28921
18	27	22	2801038	28	126559	125271	106120	26725
18	27	23	2306098	27	105465	104160	96805	23457
18	27	24	2194403	26	83865	82530	76205	18399
18	27	25	2656252	25	99236	98176	93425	16770
18	27	26	2380790	24	77629	76918	64985	9296
20	4	3	4703450	55	73387	72316	72303	67128
20	5	2	4223654	80	161229	159328	151462	94810
20	5	3	3947436	53	113939	112157	105296	61217
20	5	4	4042792	40	64723	63594	61042	49981
20	6	2	4017172	79	173008	171092	160611	110782
20	6	3	3249012	52	124241	122363	112022	56659
20	6	4	3137875	39	98298	96696	94284	57238
20	6	5	2965312	31	52494	51474	51420	47026
20	7	2	4090073	79	175125	173155	160420	118410
20	7	3	3339464	52	145188	143172	134171	74379
20	7	4	2501170	38	102923	101006	90905	47702
20	7	5	2258927	31	66914	65123	65082	38710
20	7	6	2334110	26	37746	36860	36860	31767

Table 6.9. Costs of block 2GYF for StandardNonInteractive

Configuration				Copied	Invoc	W.b. insert	W.b. add	R.s. proc
9	12	1	11	68661	1660	44	43	43
9	12	2	10	61484	855	33	31	31
9	12	3	9	58205	575	32	30	30
9	12	4	8	58087	434	18	18	18
9	12	5	7	55902	351	17	17	17
9	12	6	6	58361	294	21	19	19
9	12	7	5	101550	267	18	16	16
9	12	8	4	166802	271	12	12	12
9	12	9	3	166943	269	16	16	16
9	12	10	2	166285	268	11	11	11
9	14	1	13	60948	1660	44	43	43
9	14	2	12	50440	847	22	22	22
9	14	3	11	44807	567	29	27	27
9	14	4	10	42610	425	18	18	18
9	14	5	9	36442	339	17	17	17
9	14	6	8	37997	284	16	16	16
9	14	7	7	38401	245	20	18	18
9	14	8	6	43540	217	20	18	18
9	14	9	5	74875	202	21	21	21
9	14	10	4	122732	201	17	17	17
9	14	12	2	125215	202	15	15	15
9	17	1	16	51698	1660	44	43	43
9	17	2	15	42915	841	31	29	29
9	17	3	14	38716	563	24	24	24
9	17	4	13	33446	422	15	15	15
9	17	5	12	29027	337	15	15	15
9	17	6	11	28362	281	14	14	14
9	17	7	10	27882	240	12	12	12
9	17	8	9	28055	212	16	16	16
9	17	9	8	29946	189	12	12	12
9	17	10	7	28362	170	12	12	12
9	17	11	6	31109	156	9	9	9
9	17	12	5	58231	148	11	11	11
9	17	13	4	89909	148	9	9	9
9	17	14	3	91049	147	9	9	9
9	17	15	2	90940	147	9	9	9
9	17	16	1	90495	147	18	16	16
9	21	1	20	48810	1660	44	43	43
9	21	2	19	38044	838	26	24	24
9	21	3	18	34070	560	21	21	21
9	21	4	17	28600	419	24	22	22
9	21	6	15	24552	279	14	14	14
9	21	7	14	23099	239	17	15	15
9	21	8	13	20912	209	15	15	15
9	21	9	12	20256	186	14	14	14
9	21	11	10	20496	153	10	10	10
9	21	12	9	20191	140	10	10	10
9	21	13	8	21118	130	13	13	13

continued on next page

Table 6.9. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
9	21	14	7	22860	121	14	13	13
9	21	16	5	36625	108	11	11	11
9	21	17	4	66141	109	12	11	11
9	21	18	3	67015	108	13	12	12
9	21	19	2	67067	108	17	16	16
9	25	1	24	45402	1660	44	43	43
9	25	2	23	35483	836	32	30	30
9	25	4	21	27412	418	15	15	15
9	25	5	20	23741	334	14	14	14
9	25	7	18	19498	238	16	16	16
9	25	8	17	19741	209	19	17	17
9	25	10	15	17654	167	11	11	11
9	25	11	14	17208	152	12	12	12
9	25	13	12	15894	129	14	14	14
9	25	14	11	15784	119	10	10	10
9	25	16	9	16045	105	10	8	8
9	**25**	**17**	**8**	**15699**	**99**	**13**	**12**	**12**
9	25	18	7	17661	93	15	14	11
9	25	20	5	32105	86	7	7	7
9	25	21	4	52301	86	9	8	8
9	25	22	3	52990	86	10	9	9
9	25	23	2	53042	86	9	8	8
9	25	24	1	51770	85	10	10	7
9	30	1	29	43928	1660	44	43	43
9	30	3	27	30277	557	29	28	28
9	30	4	26	26449	417	12	12	12
9	30	6	24	21397	278	14	14	14
9	30	8	22	18556	209	9	9	9
9	30	10	20	15389	167	17	17	17
9	30	12	18	15119	139	13	13	13
9	30	13	17	14433	128	13	12	12
9	30	15	15	12765	111	9	9	9
9	30	17	13	13636	98	10	10	10
9	30	19	11	13611	88	9	8	8
9	30	20	10	11694	83	13	12	9
9	30	22	8	12932	76	9	8	8
9	30	24	6	14458	70	10	9	9
9	30	26	4	41158	68	9	8	8
9	30	27	3	41859	68	8	7	7
9	30	28	2	42143	68	11	8	8
9	30	29	1	42258	68	11	9	9
9	37	1	36	41822	1660	44	43	43
9	37	3	34	29189	555	24	22	22
9	37	6	31	20292	278	16	16	16
9	37	8	29	16979	208	14	14	14
9	37	10	27	15318	166	18	16	16
9	37	12	25	13990	138	9	9	9
9	37	14	23	13070	118	15	15	15

continued on next page

Table 6.9. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
9	37	17	20	11364	98	11	10	10
9	37	19	18	10258	87	9	8	8
9	37	21	16	10063	79	7	6	6
9	37	23	14	9294	72	7	6	6
9	37	25	12	9057	66	7	6	6
9	37	27	10	8664	61	8	7	7
9	37	30	7	10502	55	12	10	10
9	37	31	6	11677	54	8	6	6
9	37	33	4	30614	52	11	8	8
9	37	34	3	32404	53	9	7	7
9	37	35	2	32382	53	9	7	7
9	37	36	1	31275	52	11	9	9
9	45	1	44	40736	1660	44	43	43
9	45	4	41	24297	416	13	13	13
9	45	7	38	17924	237	12	12	12
9	45	9	36	15341	185	12	12	12
9	45	12	33	14015	138	12	12	12
9	45	15	30	12351	111	15	13	13
9	45	18	27	10893	92	8	8	8
9	45	20	25	10564	83	11	10	10
9	45	23	22	8960	72	8	7	7
9	45	26	19	8577	64	11	9	9
9	45	28	17	7844	59	8	6	6
9	45	31	14	6509	53	9	7	7
9	45	33	12	7247	50	13	10	7
9	45	36	9	6951	46	6	5	5
9	45	38	7	9219	44	10	7	7
9	45	40	5	15978	42	7	5	5
9	45	41	4	26141	42	7	5	5
9	45	42	3	25786	42	8	6	6
9	45	43	2	25674	42	5	4	4
9	54	2	52	31682	832	28	26	26
9	54	5	49	21191	332	11	11	11
9	54	8	46	15530	208	22	20	20
9	54	11	43	13664	151	11	11	11
9	54	15	39	12187	110	9	9	9
9	54	18	36	10725	92	10	10	10
9	54	21	33	9544	79	9	8	8
9	54	24	30	8414	69	12	10	10
9	54	28	26	7521	59	8	6	6
9	54	31	23	6605	53	7	5	5
9	54	34	20	6066	48	7	5	5
9	54	37	17	5882	44	9	6	6
9	54	40	14	6336	41	8	6	6
9	54	43	11	6354	38	12	8	8
9	54	46	8	6122	36	7	5	5
9	54	48	6	7962	34	5	3	3
9	54	50	4	20310	34	5	3	3

continued on next page

Table 6.9. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
9	54	51	3	20853	34	6	4	4
9	54	52	2	20813	34	4	3	3
10	6	1	5	52015	809	47	0	41
10	6	2	4	56515	430	31	0	27
10	6	3	3	59973	295	27	0	24
10	6	4	2	167107	276	21	0	19
10	6	5	1	171625	275	19	0	17
10	7	1	6	44129	809	47	0	41
10	7	2	5	39398	419	25	0	23
10	7	3	4	36711	280	27	0	22
10	7	4	3	39148	214	23	0	21
10	7	5	2	122983	204	20	0	16
10	7	6	1	126087	203	19	0	15
10	9	1	8	38784	809	47	0	41
10	9	2	7	30657	412	31	0	26
10	9	3	6	25144	275	27	0	24
10	9	4	5	22517	206	18	0	16
10	9	5	4	24128	166	14	0	13
10	9	6	3	26623	140	11	0	11
10	9	7	2	81307	135	12	0	12
10	9	8	1	81959	134	20	0	18
10	11	1	10	36124	809	47	0	41
10	11	2	9	28622	410	27	0	25
10	11	3	8	23663	273	22	0	19
10	11	4	7	21165	205	17	0	14
10	11	5	6	19045	164	14	0	14
10	11	6	5	17685	137	12	0	12
10	11	7	4	16995	118	15	0	14
10	11	8	3	19370	104	12	0	11
10	11	9	2	61354	101	17	0	11
10	11	10	1	62624	101	16	0	11
10	13	1	12	34469	809	47	0	41
10	13	2	11	26663	409	27	0	24
10	13	3	10	21504	272	26	0	23
10	13	4	9	18307	204	18	0	17
10	13	5	8	17733	163	10	0	10
10	13	6	7	16051	136	16	0	14
10	**13**	**7**	**6**	**14239**	**116**	**10**	**0**	**10**
10	13	8	5	14891	102	15	0	11
10	13	9	4	15482	91	14	0	11
10	13	10	3	15544	82	13	0	11
10	13	11	2	47686	80	12	0	9
10	13	12	1	48798	80	10	0	8
10	15	1	14	33122	809	47	0	41
10	15	2	13	25234	408	24	0	23
10	15	3	12	20817	271	21	0	20
10	15	4	11	17860	203	19	0	16
10	15	5	10	16421	163	22	0	19

continued on next page

Table 6.9. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
10	15	6	9	14757	135	20	0	17
10	15	7	8	13974	116	14	0	13
10	15	8	7	12068	101	10	0	9
10	15	9	6	11423	90	16	0	13
10	15	10	5	10993	81	7	0	6
10	15	11	4	12107	74	9	0	7
10	15	12	3	12924	68	12	0	11
10	15	13	2	40572	67	15	0	11
10	15	14	1	41193	67	8	0	7
10	19	1	18	32177	809	47	0	41
10	19	2	17	24097	407	31	0	26
10	19	3	16	20776	271	24	0	21
10	19	4	15	16553	203	19	0	19
10	19	5	14	15571	162	19	0	18
10	19	6	13	13389	135	19	0	18
10	19	7	12	14480	116	14	0	14
10	19	9	10	10272	90	14	0	12
10	19	10	9	10607	81	11	0	10
10	19	11	8	9072	73	13	0	11
10	19	12	7	8399	67	13	0	11
10	19	13	6	8169	62	10	0	7
10	19	14	5	9170	58	14	0	9
10	19	15	4	9195	54	9	0	7
10	19	16	3	10712	51	13	0	9
10	19	17	2	29814	50	9	0	7
10	19	18	1	30882	50	7	0	4
10	23	1	22	30661	809	47	0	41
10	23	2	21	23744	406	26	0	24
10	23	3	20	18500	270	19	0	18
10	23	5	18	14819	162	17	0	16
10	23	6	17	12824	135	14	0	13
10	23	8	15	11015	101	17	0	15
10	23	9	14	9695	90	12	0	12
10	23	10	13	9302	81	14	0	12
10	23	12	11	8109	67	15	0	11
10	23	13	10	7705	62	12	0	8
10	23	14	9	8517	58	9	0	7
10	23	16	7	6397	50	7	0	5
10	23	17	6	7271	47	6	0	5
10	23	18	5	6580	45	13	0	10
10	23	20	3	9505	41	6	0	4
10	23	21	2	23898	40	11	0	8
10	23	22	1	24581	40	9	0	6
10	27	1	26	30043	809	47	0	41
10	27	2	25	23397	406	21	0	19
10	27	4	23	15854	202	21	0	20
10	27	6	21	13077	135	18	0	15
10	27	7	20	12490	115	14	0	14

continued on next page

Table 6.9. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
10	27	9	18	10031	90	12	0	12
10	27	11	16	9247	73	9	0	7
10	27	12	15	7402	67	13	0	10
10	27	14	13	8033	58	11	0	10
10	27	15	12	7116	54	11	0	8
10	27	17	10	6359	47	9	0	6
10	27	18	9	6235	45	9	0	7
10	27	20	7	5414	40	13	0	8
10	27	22	5	5069	36	8	0	6
10	27	23	4	5809	35	5	0	3
10	27	24	3	7721	34	3	0	2
10	27	25	2	19675	33	9	0	6
10	27	26	1	20053	33	9	0	6
11	3	1	2	42337	402	45	33	33
11	3	2	1	233072	386	39	29	29
11	4	1	3	31591	402	45	33	33
11	4	2	2	30579	214	36	27	27
11	4	3	1	118307	197	24	19	19
11	5	1	4	28407	402	45	33	33
11	5	2	3	22474	207	23	17	17
11	5	3	2	20884	140	18	14	14
11	5	4	1	78084	131	21	18	18
11	6	1	5	27035	402	45	33	33
11	6	2	4	18402	205	23	19	19
11	6	3	3	17821	137	23	18	18
11	6	4	2	15745	103	20	16	14
11	6	5	1	57828	98	20	16	14
11	7	1	6	25754	402	45	33	33
11	7	2	5	19308	204	18	15	15
11	7	3	4	15979	136	25	18	18
11	7	4	3	14151	102	17	11	11
11	7	5	2	12819	82	23	17	17
11	7	6	1	46931	79	12	10	10
11	8	1	7	24756	402	45	33	33
11	8	2	6	17335	203	27	21	21
11	8	3	5	14368	135	17	15	15
11	8	4	4	12184	101	28	21	19
11	8	5	3	10529	81	26	19	17
11	8	6	2	10265	68	12	8	8
11	8	7	1	39331	66	14	11	11
11	10	1	9	24114	402	45	33	33
11	10	2	8	17507	203	32	24	24
11	10	3	7	13435	135	23	18	18
11	10	4	6	11396	101	16	14	14
11	10	5	5	10093	81	20	15	15
11	10	6	4	8230	67	18	12	10
11	10	7	3	8534	58	19	13	13
11	10	8	2	9868	51	13	8	8

continued on next page

Table 6.9. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
11	10	9	1	29527	49	15	11	11
11	12	1	11	23601	402	45	33	33
11	12	2	10	15349	202	26	21	21
11	12	3	9	12650	134	26	20	20
11	12	4	8	10936	101	16	14	14
11	12	5	7	9233	80	21	16	16
11	12	6	6	7793	67	14	10	10
11	12	7	5	7953	57	15	10	10
11	12	8	4	5973	50	10	7	7
11	12	9	3	6304	45	11	7	7
11	12	10	2	6389	40	16	11	11
11	12	11	1	23564	39	7	5	5
11	14	1	13	23289	402	45	33	33
11	14	2	12	15858	202	24	18	18
11	14	3	11	13014	134	27	20	20
11	14	4	10	9667	100	22	17	15
11	14	5	9	8019	80	24	18	16
11	14	6	8	7636	67	17	12	12
11	14	7	7	7142	57	8	6	6
11	14	8	6	5395	50	13	8	6
11	14	9	5	5011	44	7	5	5
11	14	10	4	5429	40	14	10	10
11	14	11	3	5079	36	5	4	4
11	14	12	2	5379	33	14	9	9
11	14	13	1	20418	33	11	7	7
12	2	1	1	25089	200	14	0	13
12	3	1	2	18384	200	14	0	13
12	3	2	1	16318	106	9	0	8
12	4	1	3	17152	200	14	0	13
12	4	2	2	12254	102	10	0	8
12	4	3	1	11228	69	7	0	6
12	5	1	4	16261	200	14	0	13
12	5	2	3	11053	101	8	0	8
12	5	3	2	9781	68	8	0	6
12	5	4	1	8171	51	11	0	9
12	6	1	5	15684	200	14	0	13
12	6	2	4	11166	101	13	0	12
12	6	3	3	8358	67	13	0	10
12	6	4	2	5971	50	7	0	5
12	6	5	1	7824	41	6	0	4
12	7	1	6	15771	200	14	0	13
12	7	2	5	9968	100	14	0	12
12	7	3	4	7505	67	13	0	10
12	7	4	3	5515	50	7	0	5
12	7	5	2	5555	40	10	0	7
12	7	6	1	5953	34	3	0	2
13	2	1	1	12024	100	13	10	10
13	3	1	2	10791	100	13	10	10

continued on next page

Table 6.9. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
13	3	2	1	7321	51	12	7	7
13	4	1	3	10331	100	13	10	10
13	4	2	2	5709	50	10	6	6
13	4	3	1	5271	34	13	8	8

Table 6.10. Costs of block 2GYF for HeapSim

Configuration				Copied	Invoc	W.b. insert	W.b. add	R.s. proc
13	56	2	54	870801	930	54	32	32
13	56	5	51	1034112	372	46	36	36
13	56	8	48	1286129	233	67	30	30
13	56	12	44	3941765	159	74	22	22
13	56	15	41	5434667	135	64	24	24
13	56	18	38	6358576	123	63	22	22
13	56	22	34	6820966	113	6	4	4
13	56	25	31	7134689	109	79	18	18
13	56	29	27	7369797	106	46	12	12
13	56	32	24	7496892	104	90	18	18
13	56	35	21	7584273	103	66	18	18
13	56	38	18	7608841	102	61	18	18
13	56	41	15	7703753	102	22	12	12
13	56	45	11	7693569	101	16	9	9
13	56	48	8	7728651	101	94	10	10
13	56	50	6	7747103	101	91	10	10
13	56	52	4	7755799	101	91	10	10
13	56	53	3	7765052	101	91	10	10
13	56	54	2	7766506	101	91	10	10
13	68	2	66	531008	928	54	32	32
13	68	6	62	609541	309	58	22	22
13	68	10	58	688552	185	45	28	28
13	68	14	54	764906	132	46	14	14
13	68	18	50	925821	103	63	22	22
13	68	22	46	1432020	85	6	4	4
13	68	27	41	2935521	72	46	12	12
13	68	31	37	3430263	66	41	16	16
13	68	35	33	3693126	63	66	18	18
13	68	39	29	3925412	61	61	18	18
13	68	43	25	3992511	59	6	4	4
13	68	46	22	4109706	59	22	11	11
13	68	50	18	4123386	58	91	10	10
13	68	54	14	4207035	58	91	10	10
13	68	58	10	4155494	57	82	10	10
13	68	60	8	4175790	57	82	10	10
13	68	63	5	4185104	57	62	10	10
13	68	64	4	4194039	57	62	10	10
13	68	65	3	4195671	57	62	10	10
13	68	66	2	4197194	57	51	10	10
13	83	2	81	452646	927	54	32	32
13	83	7	76	441550	265	39	22	22
13	83	12	71	433508	154	74	22	22
13	83	17	66	517424	109	61	20	20
13	83	22	61	507445	84	6	4	4
13	83	27	56	582750	68	46	12	12
13	83	32	51	744293	58	90	18	18
13	83	37	46	1060740	50	63	18	18
13	83	42	41	1868085	45	6	4	4

continued on next page

Table 6.10. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
13	83	47	36	2216894	42	83	10	10
13	83	52	31	2478769	41	91	10	10
13	83	56	27	2625433	40	91	10	10
13	83	61	22	2650391	39	62	10	10
13	83	66	17	2648219	38	51	10	10
13	83	71	12	2691703	38	38	10	10
13	83	74	9	2717850	38	35	10	10
13	83	76	7	2721748	38	35	10	10
13	83	78	5	2736561	38	35	10	10
13	83	79	4	2736846	38	35	10	10
13	83	80	3	2737385	38	14	4	4
13	101	3	98	364364	618	64	30	30
13	101	9	92	356838	206	57	28	28
13	101	15	86	351203	123	64	24	24
13	101	21	80	437140	88	21	12	12
13	101	27	74	427744	68	46	12	12
13	101	33	68	423309	56	79	18	18
13	101	39	62	498849	47	61	18	18
13	101	45	56	573029	41	16	9	9
13	101	52	49	718667	35	91	10	10
13	101	58	43	1191258	32	82	10	10
13	101	64	37	1548218	30	62	10	10
13	101	69	32	1729766	29	38	10	10
13	101	75	26	1767884	28	35	10	10
13	101	81	20	1913110	28	6	4	4
13	101	86	15	1861959	27	6	4	4
13	101	90	11	1892968	27	6	4	4
13	101	93	8	1897475	27	6	4	4
13	101	95	6	1912600	27	0	0	0
13	101	96	5	1913019	27	0	0	0
13	101	97	4	1913162	27	0	0	0
13	123	4	119	367112	463	64	30	30
13	123	11	112	360290	168	13	9	9
13	123	18	105	356871	103	63	22	22
13	123	26	97	350951	71	75	18	18
13	123	33	90	346567	56	79	18	18
13	**123**	**41**	**82**	**341702**	**45**	**22**	**12**	**12**
13	123	48	75	419620	38	94	10	10
13	123	55	68	409600	33	91	10	10
13	123	63	60	485102	29	62	10	10
13	123	70	53	555359	26	38	10	10
13	123	77	46	695936	24	35	10	10
13	123	84	39	1054992	22	6	4	4
13	123	91	32	1236695	21	6	4	4
13	123	98	25	1350981	21	0	0	0
13	123	105	18	1350562	20	0	0	0
13	123	109	14	1381353	20	0	0	0
13	123	113	10	1387089	20	0	0	0

continued on next page

Table 6.10. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
13	123	116	7	1403346	20	0	0	0
13	123	117	6	1404333	20	0	0	0
13	123	119	4	1405931	20	0	0	0
13	150	4	146	276703	463	64	30	30
13	150	13	137	270671	142	76	26	26
13	150	22	128	358277	84	6	4	4
13	150	31	119	351756	59	41	16	16
13	150	40	110	348642	46	30	12	12
13	150	49	101	340637	37	94	10	10
13	150	58	92	332811	31	82	10	10
13	150	67	83	330118	27	51	10	10
13	150	76	74	409382	24	35	10	10
13	150	86	64	396500	21	6	4	4
13	150	94	56	461782	19	0	0	0
13	150	102	48	533716	18	0	0	0
13	150	111	39	886871	17	0	0	0
13	150	120	30	980039	16	0	0	0
13	150	127	23	1078403	16	0	0	0
13	150	133	17	1021244	15	0	0	0
13	150	138	12	1026050	15	0	0	0
13	150	141	9	1043567	15	0	0	0
13	150	143	7	1045162	15	0	0	0
13	150	145	5	1046595	15	0	0	0
13	182	5	177	275501	370	46	36	36
13	182	16	166	268907	115	69	20	20
13	182	27	155	265779	68	46	12	12
13	182	38	144	261723	48	61	18	18
13	182	49	133	257678	37	94	10	10
13	182	60	122	254023	30	82	10	10
13	182	71	111	340141	26	38	10	10
13	182	82	100	329720	22	6	4	4
13	182	93	89	319356	19	6	4	4
13	182	104	78	313253	17	0	0	0
13	182	115	67	391403	16	0	0	0
13	182	124	58	387396	15	0	0	0
13	182	135	47	431013	13	0	0	0
13	182	146	36	767934	13	0	0	0
13	182	155	27	781055	12	0	0	0
13	182	162	20	819055	12	0	0	0
13	182	167	15	822947	12	0	0	0
13	182	171	11	826290	12	0	0	0
13	182	174	8	845160	12	0	0	0
13	182	175	7	845708	12	0	0	0
13	222	7	215	273644	264	39	22	22
13	222	20	202	267618	92	45	20	20
13	222	33	189	265886	56	79	18	18
13	222	47	175	261249	39	83	10	10
13	222	60	162	254023	30	82	10	10

continued on next page

Table 6.10. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
13	222	73	149	255296	25	35	10	10
13	222	87	135	252564	21	6	4	4
13	222	100	122	246353	18	0	0	0
13	222	113	109	322692	16	0	0	0
13	222	127	95	312849	14	0	0	0
13	222	140	82	312936	13	0	0	0
13	222	151	71	305065	12	0	0	0
13	222	164	58	364597	11	0	0	0
13	222	178	44	477447	10	0	0	0
13	222	189	33	601892	10	0	0	0
13	222	197	25	566560	9	0	0	0
13	222	204	18	604225	9	0	0	0
13	222	209	13	607740	9	0	0	0
13	222	212	10	628480	9	0	0	0
13	222	214	8	629312	9	0	0	0
13	270	8	262	272989	231	67	30	30
13	270	24	246	267984	77	97	19	19
13	270	40	230	263317	46	30	12	12
13	270	57	213	257903	32	82	10	10
13	270	73	197	255296	25	35	10	10
13	270	89	181	245838	20	6	4	4
13	270	105	165	243856	17	0	0	0
13	270	121	149	244499	15	0	0	0
13	270	138	132	238743	13	0	0	0
13	270	154	116	242362	12	0	0	0
13	270	170	100	221404	10	0	0	0
13	270	184	86	305304	10	0	0	0
13	270	200	70	290679	9	0	0	0
13	270	216	54	340288	8	0	0	0
13	270	229	41	457035	8	0	0	0
13	270	240	30	424007	7	0	0	0
13	270	248	22	464167	7	0	0	0
13	270	254	16	467124	7	0	0	0
13	270	258	12	490462	7	0	0	0
13	270	260	10	491197	7	0	0	0
14	28	1	27	869909	926	55	33	33
14	28	3	25	1120294	312	58	22	22
14	28	4	24	1285986	234	67	30	30
14	28	6	22	3944878	161	74	22	22
14	28	8	20	6012802	133	69	20	20
14	28	9	19	6450262	125	64	23	23
14	28	11	17	7077996	116	6	4	4
14	28	13	15	7444919	111	75	18	18
14	28	14	14	7541359	109	46	12	12
14	28	16	12	7744556	107	90	18	18
14	28	18	10	7798127	105	63	18	18
14	28	19	9	7851738	105	61	18	18
14	28	21	7	7895573	104	6	4	4

continued on next page

Table 6.10. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
14	28	22	6	7931478	104	6	4	4
14	28	24	4	7975717	104	94	10	10
14	28	25	3	7995432	104	91	10	10
14	28	26	2	8002797	104	91	10	10
14	28	27	1	7923387	103	91	10	10
14	34	1	33	530093	926	55	33	33
14	34	3	31	610574	310	58	22	22
14	34	5	29	690209	186	45	28	28
14	34	7	27	766825	133	46	14	14
14	34	9	25	925646	103	64	23	23
14	34	11	23	1432355	85	6	4	4
14	34	13	21	2693810	74	75	18	18
14	34	15	19	3378238	68	41	15	15
14	34	17	17	3727929	65	66	18	18
14	34	19	15	3940912	62	61	18	18
14	34	21	13	4110512	61	6	4	4
14	34	23	11	4192636	60	22	11	11
14	34	25	9	4204518	59	91	10	10
14	34	27	7	4287917	59	91	10	10
14	34	29	5	4232454	58	82	10	10
14	34	30	4	4256008	58	82	10	10
14	34	31	3	4261143	58	62	10	10
14	34	32	2	4273504	58	62	10	10
14	34	33	1	4275538	58	51	10	10
14	42	1	41	453109	926	55	33	33
14	42	4	38	440733	232	67	30	30
14	42	6	36	434399	154	74	22	22
14	42	9	33	517276	103	64	23	23
14	42	11	31	507756	84	6	4	4
14	42	14	28	583425	66	46	12	12
14	42	16	26	748562	58	90	18	18
14	42	19	23	1066824	49	61	18	18
14	42	21	21	1715150	45	6	4	4
14	42	24	18	2291863	42	94	10	10
14	42	26	16	2388381	40	91	10	10
14	42	29	13	2550359	39	82	10	10
14	42	31	11	2569868	38	62	10	10
14	42	34	8	2663301	38	38	10	10
14	42	36	6	2697877	38	35	10	10
14	42	37	5	2723672	38	35	10	10
14	42	39	3	2639926	37	35	10	10
14	42	40	2	2650397	37	14	4	4
14	51	2	49	362095	463	64	30	30
14	51	5	46	355580	185	45	28	28
14	51	8	43	350258	115	69	20	20
14	51	11	40	436784	84	6	4	4
14	51	14	37	429303	66	46	12	12
14	51	17	34	420642	54	66	18	18

continued on next page

Table 6.10. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
14	51	20	31	500845	46	30	12	12
14	51	23	28	574248	40	22	11	11
14	51	26	25	636179	35	91	10	10
14	51	29	22	1036920	32	82	10	10
14	51	32	19	1549415	30	62	10	10
14	51	35	16	1736336	29	38	10	10
14	51	38	13	1818516	28	35	10	10
14	51	41	10	1826210	27	6	4	4
14	51	43	8	1866994	27	6	4	4
14	51	45	6	1896779	27	6	4	4
14	51	47	4	1903081	27	0	0	0
14	51	48	3	1917221	27	0	0	0
14	51	49	2	1917641	27	0	0	0
14	62	2	60	367393	463	64	30	30
14	62	6	56	359795	154	74	22	22
14	62	9	53	357443	103	64	23	23
14	62	13	49	351650	71	75	18	18
14	62	17	45	344897	54	66	18	18
14	**62**	**20**	**42**	**342101**	**46**	**30**	**12**	**12**
14	62	24	38	419842	38	94	10	10
14	62	28	34	414183	33	91	10	10
14	62	32	30	489104	29	62	10	10
14	62	35	27	555421	26	38	10	10
14	62	39	23	700272	24	35	10	10
14	62	42	20	992445	22	6	4	4
14	62	46	16	1241519	21	6	4	4
14	62	50	12	1401630	21	0	0	0
14	62	53	9	1354021	20	0	0	0
14	62	55	7	1384797	20	0	0	0
14	62	57	5	1390180	20	0	0	0
14	62	58	4	1393542	20	0	0	0
14	62	59	3	1407944	20	0	0	0
14	62	60	2	1409924	20	0	0	0
14	75	2	73	276798	463	64	30	30
14	75	7	68	270676	132	46	14	14
14	75	11	64	358230	84	6	4	4
14	75	16	59	350861	57	90	18	18
14	75	20	55	348755	46	30	12	12
14	75	25	50	345556	37	91	10	10
14	75	29	46	332826	31	82	10	10
14	75	34	41	332931	27	38	10	10
14	75	38	37	409481	24	35	10	10
14	75	43	32	396615	21	6	4	4
14	75	47	28	461811	19	0	0	0
14	75	51	24	533705	18	0	0	0
14	75	55	20	884634	17	0	0	0
14	75	60	15	979391	16	0	0	0
14	75	64	11	1079097	16	0	0	0

continued on next page

Table 6.10. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
14	75	67	8	1021877	15	0	0	0
14	75	69	6	1025889	15	0	0	0
14	75	70	5	1026730	15	0	0	0
14	75	72	3	1044375	15	0	0	0
14	91	3	88	274399	308	58	22	22
14	91	8	83	269021	115	69	20	20
14	91	14	77	267136	66	46	12	12
14	91	19	72	261765	48	61	18	18
14	91	25	66	263450	37	91	10	10
14	91	30	61	254149	30	82	10	10
14	91	35	56	337675	26	38	10	10
14	91	41	50	329728	22	6	4	4
14	91	46	45	329467	20	6	4	4
14	91	52	39	313322	17	0	0	0
14	91	57	34	393073	16	0	0	0
14	91	62	29	387498	15	0	0	0
14	91	67	24	429283	13	0	0	0
14	91	73	18	767943	13	0	0	0
14	91	77	14	779692	12	0	0	0
14	91	81	10	818263	12	0	0	0
14	91	84	7	823472	12	0	0	0
14	91	86	5	843787	12	0	0	0
14	91	87	4	845022	12	0	0	0
14	91	88	3	845974	12	0	0	0
14	111	3	108	274399	308	58	22	22
14	111	10	101	267806	92	45	20	20
14	111	17	94	264324	54	66	18	18
14	111	23	88	262606	40	22	11	11
14	111	30	81	254149	30	82	10	10
14	111	37	74	258364	25	35	10	10
14	111	43	68	249567	21	6	4	4
14	111	50	61	246441	18	0	0	0
14	111	57	54	324461	16	0	0	0
14	111	63	48	311024	14	0	0	0
14	111	70	41	313082	13	0	0	0
14	111	75	36	303956	12	0	0	0
14	111	82	29	364568	11	0	0	0
14	111	89	22	477572	10	0	0	0
14	111	94	17	601083	10	0	0	0
14	111	99	12	600446	9	0	0	0
14	111	102	9	604158	9	0	0	0
14	111	104	7	607112	9	0	0	0
14	111	106	5	628130	9	0	0	0
14	111	107	4	628904	9	0	0	0
14	135	4	131	273007	231	67	30	30
14	135	12	123	268092	77	97	19	19
14	135	20	115	263433	46	30	12	12
14	135	28	107	261564	33	91	10	10

continued on next page

Table 6.10. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
14	135	36	99	251845	25	35	10	10
14	135	45	90	248316	20	6	4	4
14	135	53	82	245581	17	0	0	0
14	135	61	74	246485	15	0	0	0
14	135	69	66	238777	13	0	0	0
14	135	77	58	242369	12	0	0	0
14	135	85	50	221402	10	0	0	0
14	135	92	43	305397	10	0	0	0
14	135	100	35	290738	9	0	0	0
14	135	108	27	340309	8	0	0	0
14	135	115	20	457726	8	0	0	0
14	135	120	15	424030	7	0	0	0
14	135	124	11	464096	7	0	0	0
14	135	127	8	466794	7	0	0	0
14	135	129	6	490551	7	0	0	0
14	135	130	5	491039	7	0	0	0
16	7	1	6	1284039	231	67	30	30
16	7	2	5	8296588	162	69	20	20
16	7	3	4	9902468	144	97	19	19
16	7	4	3	10319074	138	90	18	18
16	7	5	2	10537601	136	30	12	12
16	7	6	1	10575430	135	94	10	10
16	9	1	8	518621	231	67	30	30
16	9	2	7	684418	118	69	20	20
16	9	3	6	1095064	80	97	19	19
16	9	4	5	3074650	66	90	18	18
16	9	5	4	3729588	60	30	12	12
16	9	6	3	3994675	58	94	10	10
16	9	7	2	4102691	57	91	10	10
16	9	8	1	4103254	56	62	10	10
16	11	1	10	442953	231	67	30	30
16	11	2	9	434323	116	69	20	20
16	11	3	8	513533	78	97	19	19
16	11	4	7	580495	58	90	18	18
16	11	5	6	903826	47	30	12	12
16	11	6	5	1939845	42	94	10	10
16	11	7	4	2332710	39	91	10	10
16	11	8	3	2544224	38	62	10	10
16	11	9	2	2593103	37	35	10	10
16	11	10	1	2655987	37	14	4	4
16	13	1	12	357813	231	67	30	30
16	13	2	11	352895	116	69	20	20
16	13	3	10	435919	77	97	19	19
16	13	4	9	427418	58	90	18	18
16	13	5	8	501287	46	30	12	12
16	13	6	7	576508	39	94	10	10
16	13	7	6	808112	34	91	10	10
16	13	8	5	1413737	30	62	10	10

continued on next page

Table 6.10. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
16	13	9	4	1737251	29	35	10	10
16	13	10	3	1870260	28	14	4	4
16	13	11	2	1956374	28	6	4	4
16	13	12	1	1897467	27	0	0	0
16	16	1	15	363693	231	67	30	30
16	16	2	14	359415	116	69	20	20
16	16	3	13	354255	77	97	19	19
16	16	4	12	350245	58	90	18	18
16	16	5	11	343107	46	30	12	12
16	**16**	**6**	**10**	**336487**	**38**	**94**	**10**	**10**
16	16	7	9	417399	33	91	10	10
16	16	8	8	492475	29	62	10	10
16	16	9	7	483055	26	35	10	10
16	16	10	6	616426	23	14	4	4
16	16	11	5	1069844	22	6	4	4
16	16	12	4	1301969	21	0	0	0
16	16	13	3	1320651	20	0	0	0
16	16	14	2	1368624	20	0	0	0
16	16	15	1	1404722	20	0	0	0
16	19	1	18	273129	231	67	30	30
16	19	2	17	269231	115	69	20	20
16	19	3	16	357476	77	97	19	19
16	19	4	15	355667	58	90	18	18
16	19	5	14	349496	46	30	12	12
16	19	6	13	343165	38	94	10	10
16	19	7	12	341860	33	91	10	10
16	19	9	10	326615	25	35	10	10
16	19	10	9	409354	23	14	4	4
16	19	11	8	399766	21	6	4	4
16	19	12	7	463637	19	0	0	0
16	19	13	6	534989	18	0	0	0
16	19	14	5	887851	17	0	0	0
16	19	15	4	931422	16	0	0	0
16	19	16	3	1074535	16	0	0	0
16	19	17	2	1113409	16	0	0	0
16	19	18	1	1133992	16	0	0	0
16	23	1	22	273129	231	67	30	30
16	23	2	21	269231	115	69	20	20
16	23	3	20	268176	77	97	19	19
16	23	5	18	263459	46	30	12	12
16	23	6	17	259815	38	94	10	10
16	23	8	15	345115	29	62	10	10
16	23	9	14	334223	25	35	10	10
16	23	10	13	336564	23	14	4	4
16	23	12	11	325496	19	0	0	0
16	23	13	10	313479	17	0	0	0
16	23	14	9	311585	16	0	0	0
16	23	16	7	373268	14	0	0	0

continued on next page

Table 6.10. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
16	23	17	6	433102	13	0	0	0
16	23	18	5	647428	13	0	0	0
16	23	20	3	785484	12	0	0	0
16	23	21	2	821942	12	0	0	0
16	23	22	1	843570	12	0	0	0
16	28	1	27	273129	231	67	30	30
16	28	3	25	268176	77	97	19	19
16	28	4	24	262993	57	90	18	18
16	28	6	22	259815	38	94	10	10
16	28	8	20	252325	28	62	10	10
16	28	9	19	251976	25	35	10	10
16	28	11	17	255850	21	6	4	4
16	28	13	15	241944	17	0	0	0
16	28	14	14	243076	16	0	0	0
16	28	16	12	314679	14	0	0	0
16	28	18	10	298119	12	0	0	0
16	28	19	9	306532	12	0	0	0
16	28	21	7	370542	11	0	0	0
16	28	22	6	417712	10	0	0	0
16	28	24	4	654522	10	0	0	0
16	28	25	3	600403	9	0	0	0
16	28	26	2	605405	9	0	0	0
16	28	27	1	629714	9	0	0	0
16	34	1	33	273129	231	67	30	30
16	34	3	31	268176	77	97	19	19
16	34	5	29	263459	46	30	12	12
16	34	7	27	261516	33	91	10	10
16	34	9	25	251976	25	35	10	10
16	34	11	23	255850	21	6	4	4
16	34	13	21	241944	17	0	0	0
16	34	15	19	242910	15	0	0	0
16	34	17	17	236298	13	0	0	0
16	34	19	15	240138	12	0	0	0
16	34	21	13	312169	11	0	0	0
16	34	23	11	305446	10	0	0	0
16	34	25	9	290797	9	0	0	0
16	34	27	7	275320	8	0	0	0
16	34	29	5	458789	8	0	0	0
16	34	30	4	516603	8	0	0	0
16	34	31	3	463834	7	0	0	0
16	34	32	2	467877	7	0	0	0
16	34	33	1	491679	7	0	0	0
18	2	1	1	3106246	57	90	18	18
18	3	1	2	504738	57	90	18	18
18	3	2	1	2986212	46	62	10	10
18	4	1	3	346000	57	90	18	18
18	4	2	2	501336	30	62	10	10
18	4	3	1	1566272	25	0	0	0

continued on next page

Table 6.10. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
18	5	1	4	263046	57	90	18	18
18	5	2	3	340420	29	62	10	10
18	5	3	2	391606	19	0	0	0
18	5	4	1	1067071	17	0	0	0
18	6	1	5	263046	57	90	18	18
18	6	2	4	252384	28	62	10	10
18	6	3	3	326866	19	0	0	0
18	6	4	2	377105	14	0	0	0
18	6	5	1	828746	13	0	0	0
18	7	1	6	263046	57	90	18	18
18	7	2	5	252384	28	62	10	10
18	7	3	4	250860	19	0	0	0
18	7	4	3	314411	14	0	0	0
18	7	5	2	293525	11	0	0	0
18	7	6	1	642845	10	0	0	0
18	9	1	8	263046	57	90	18	18
18	9	2	7	252384	28	62	10	10
18	9	3	6	250860	19	0	0	0
18	9	4	5	240538	14	0	0	0
18	9	5	4	230695	11	0	0	0
18	9	6	3	289227	9	0	0	0
18	9	7	2	282355	8	0	0	0
18	9	8	1	432879	7	0	0	0

Table 6.11. Costs of block 2GYF for Lambda-Fact5

Configuration				Copied	Invoc	W.b. insert	W.b. add	R.s. proc
9	60	2	58	541348	1141	6451	6005	6001
9	60	5	55	519056	478	4018	3511	3491
9	60	9	51	488873	277	3065	2541	2521
9	60	13	47	445425	197	2475	2024	2004
9	60	16	44	430114	166	2209	1798	1778
9	60	20	40	397687	138	1977	1529	1518
9	60	23	37	404469	119	1824	1436	1430
9	60	27	33	387990	105	1685	1317	1311
9	60	31	29	388007	97	1653	1266	1258
9	60	34	26	381103	92	1575	1197	1186
9	60	38	22	384510	90	1522	1130	1119
9	60	41	19	396249	91	1581	1193	1182
9	60	44	16	392140	90	1533	1153	1147
9	60	48	12	387960	89	1554	1158	1147
9	60	51	9	389122	89	1490	1118	1107
9	60	53	7	396863	90	1598	1177	1171
9	60	55	5	390214	89	1516	1144	1133
9	60	56	4	397538	90	1560	1152	1146
9	60	57	3	384240	88	1586	1179	1171
9	74	2	72	416822	1127	6431	5998	5994
9	74	7	67	382338	335	3368	2901	2881
9	74	11	63	356321	216	2570	2102	2089
9	74	16	58	327954	151	2090	1688	1677
9	74	20	54	311479	125	1870	1498	1475
9	74	24	50	286996	107	1701	1333	1322
9	74	29	45	277301	93	1517	1162	1139
9	74	33	41	259576	80	1433	1067	1043
9	74	38	36	247839	69	1291	947	924
9	74	42	32	242288	64	1207	886	863
9	74	47	27	229321	57	1069	756	733
9	74	50	24	237818	57	1138	810	787
9	74	55	19	236205	56	1107	813	790
9	74	59	15	239478	56	1197	844	821
9	74	63	11	243217	56	1174	859	855
9	74	66	8	242333	56	1190	834	811
9	74	68	6	242209	56	1200	844	821
9	74	70	4	242546	56	1190	837	814
9	74	71	3	242905	56	1142	819	796
9	90	3	87	352473	753	5358	4837	4833
9	90	8	82	327816	290	3146	2636	2630
9	90	13	77	294284	180	2280	1881	1864
9	90	19	71	265600	125	1848	1476	1456
9	90	24	66	244875	101	1596	1233	1213
9	90	30	60	217970	82	1455	1103	1080
9	90	35	55	205729	69	1180	880	864
9	90	41	49	192591	61	1101	780	772
9	90	46	44	186906	56	999	694	674
9	90	51	39	177450	50	999	707	698

continued on next page

Table 6.11. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
9	90	57	33	164503	43	875	620	596
9	90	61	29	160294	41	861	603	602
9	**90**	**67**	**23**	**158419**	**39**	**801**	**558**	**557**
9	90	72	18	165209	39	858	601	600
9	90	77	13	165180	39	850	597	573
9	90	80	10	166985	39	827	590	566
9	90	83	7	167880	39	854	600	576
9	90	85	5	167842	39	847	593	569
9	90	86	4	167553	39	896	605	581
9	90	87	3	168016	39	849	583	563
9	109	3	106	316464	749	5388	4856	4850
9	109	10	99	281512	229	2788	2311	2291
9	109	16	93	259119	146	2093	1657	1637
9	109	23	86	225704	101	1647	1238	1225
9	109	29	80	215199	83	1436	1115	1111
9	109	36	73	180159	65	1091	816	800
9	109	43	66	173902	56	1083	783	767
9	109	49	60	161806	49	952	680	671
9	109	56	53	152103	43	966	674	657
9	109	62	47	139504	40	956	639	611
9	109	69	40	132127	37	753	530	510
9	109	74	35	124984	33	696	453	442
9	109	81	28	119697	30	673	465	454
9	109	87	22	118720	29	654	442	431
9	109	93	16	122668	29	609	428	417
9	109	97	12	123560	29	551	405	394
9	109	100	9	124467	29	661	445	432
9	109	102	7	124704	29	677	453	440
9	109	104	5	124923	29	647	438	425
9	109	105	4	125274	29	694	465	451
9	133	4	129	289219	561	4576	4038	4025
9	133	12	121	259453	192	2454	2024	2011
9	133	20	113	215921	114	1707	1352	1340
9	133	28	105	195279	83	1505	1142	1114
9	133	36	97	168687	64	1215	857	833
9	133	44	89	160922	54	1027	745	725
9	133	52	81	139407	45	892	612	596
9	133	60	73	124571	39	834	576	560
9	133	68	65	121266	35	852	581	553
9	133	76	57	116068	31	736	482	470
9	133	84	49	104697	28	621	421	400
9	133	90	43	102357	27	628	413	402
9	133	98	35	95320	24	506	343	325
9	133	106	27	93735	23	583	370	359
9	133	113	20	93061	22	534	338	327
9	133	118	15	92965	22	575	381	364
9	133	122	11	92886	22	532	338	326
9	133	125	8	95138	22	569	385	368

continued on next page

Table 6.11. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
9	133	127	6	95916	22	563	386	369
9	133	128	5	93361	22	531	339	328
9	162	5	157	267402	448	4019	3478	3474
9	162	15	147	224009	151	2099	1689	1678
9	162	24	138	195645	95	1657	1282	1270
9	162	34	128	177581	68	1369	1001	989
9	162	44	118	141140	51	977	688	679
9	162	53	109	131217	43	847	603	587
9	162	63	99	122239	36	820	563	546
9	162	73	89	111256	32	718	479	455
9	162	83	79	98526	27	614	414	396
9	162	92	70	93310	25	639	401	387
9	162	102	60	92391	23	640	405	389
9	162	110	52	77564	21	560	342	323
9	162	120	42	74978	19	453	297	281
9	162	130	32	72327	18	420	290	270
9	162	138	24	66808	17	383	252	228
9	162	144	18	70902	17	414	261	241
9	162	149	13	72442	17	396	251	243
9	162	152	10	72725	17	387	248	240
9	162	155	7	73416	17	378	220	212
9	162	156	6	73626	17	402	248	240
9	197	6	191	251417	372	3595	3083	3063
9	197	18	179	204598	125	1880	1505	1485
9	197	30	167	171722	77	1362	1024	1018
9	197	41	156	157554	56	1151	811	791
9	197	53	144	129481	44	823	580	576
9	197	65	132	109249	35	796	541	521
9	197	77	120	98023	29	645	450	438
9	197	89	108	87951	25	617	372	365
9	197	100	97	81236	22	474	314	310
9	197	112	85	78617	20	537	372	365
9	197	124	73	69774	18	475	282	270
9	197	134	63	66479	17	469	317	293
9	197	146	51	55534	15	385	253	227
9	197	158	39	64936	15	406	241	225
9	197	167	30	61391	14	407	245	224
9	197	175	22	54088	13	346	238	234
9	197	181	16	54172	13	289	188	181
9	197	185	12	54511	13	293	181	174
9	197	188	9	54829	13	317	205	198
9	197	190	7	55363	13	303	195	188
9	240	7	233	241721	319	3324	2837	2833
9	***240***	***22***	***218***	***188739***	***102***	***1635***	***1254***	***1250***
9	***240***	***36***	***204***	***160183***	***63***	***1254***	***931***	***911***
9	***240***	***50***	***190***	***132082***	***45***	***896***	***627***	***613***
9	240	65	175	100249	34	694	453	437
9	240	79	161	100134	28	697	457	441

continued on next page

Table 6.11. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
9	240	94	146	75522	23	500	361	354
9	***240***	***108***	***132***	***77034***	***20***	***499***	***315***	***307***
9	***240***	***122***	***118***	***73309***	***18***	***566***	***352***	***334***
9	240	137	103	58927	16	388	257	241
9	240	151	89	61022	15	397	225	208
9	240	163	77	59596	14	443	257	240
9	***240***	***178***	***62***	***44263***	***12***	***304***	***203***	***192***
9	***240***	***192***	***48***	***49991***	***12***	***285***	***182***	***169***
9	***240***	***204***	***36***	***52861***	***11***	***333***	***186***	***173***
9	240	213	27	41281	10	313	169	160
9	240	221	19	42077	10	361	214	201
9	***240***	***226***	***14***	***42829***	***10***	***278***	***162***	***155***
9	240	229	11	42951	10	269	159	152
9	240	231	9	42440	10	266	155	153
9	293	9	284	230995	249	2874	2413	2389
9	293	26	267	176976	86	1483	1097	1078
9	293	44	249	126540	50	915	664	663
9	293	62	231	113727	36	793	538	534
9	293	79	214	96895	28	690	459	439
9	293	97	196	70936	22	449	304	287
9	293	114	179	73770	19	491	315	306
9	293	132	161	66835	17	394	266	260
9	293	149	144	55446	15	388	246	240
9	293	167	126	49568	13	353	222	204
9	293	185	108	45330	12	269	177	158
9	293	199	94	48040	11	322	188	173
9	293	217	76	50871	10	343	200	191
9	293	234	59	37002	9	260	148	133
9	293	249	44	36729	9	245	167	158
9	293	260	33	29497	8	212	144	131
9	293	270	23	33815	8	234	127	117
9	293	275	18	34113	8	240	147	137
9	293	280	13	35031	8	211	130	120
9	293	283	10	35128	8	244	147	134
10	30	1	29	504721	1096	6475	6045	6040
10	30	3	27	493591	402	3741	3213	3208
10	30	5	25	469702	251	2858	2429	2423
10	30	6	24	455436	215	2631	2191	2170
10	30	8	22	433866	166	2310	1836	1815
10	30	10	20	405701	139	1977	1542	1537
10	30	12	18	396355	114	1699	1339	1330
10	30	14	16	375607	101	1646	1249	1240
10	30	15	15	378252	99	1692	1301	1292
10	30	17	13	370778	90	1532	1168	1159
10	30	19	11	369224	88	1479	1128	1119
10	30	20	10	372191	87	1493	1120	1111
10	30	22	8	372825	87	1472	1113	1104
10	30	24	6	369799	86	1443	1111	1102

continued on next page

Table 6.11. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
10	30	25	5	370411	86	1466	1116	1107
10	30	27	3	373067	86	1505	1135	1126
10	30	28	2	377358	87	1490	1135	1130
10	30	29	1	379146	87	1565	1190	1181
10	37	1	36	402383	1096	6475	6045	6040
10	37	3	34	382942	387	3713	3200	3179
10	37	6	31	362899	203	2574	2188	2177
10	37	8	29	332890	152	2141	1705	1687
10	37	10	27	311970	124	1887	1488	1460
10	37	12	25	275830	102	1555	1210	1192
10	37	14	23	275934	94	1521	1200	1177
10	37	17	20	261423	77	1348	1024	1008
10	37	19	18	232772	68	1216	890	869
10	37	21	16	243743	63	1253	925	904
10	37	23	14	239533	59	1214	862	841
10	37	25	12	236848	57	1156	831	810
10	37	27	10	235513	56	1108	792	771
10	37	30	7	241808	56	1163	839	818
10	37	31	6	242420	56	1187	846	825
10	37	33	4	245172	56	1172	850	829
10	37	34	3	244543	56	1092	785	764
10	37	35	2	244502	56	1170	843	822
10	37	36	1	245127	56	1132	822	801
10	45	1	44	353424	1096	6475	6045	6040
10	45	4	41	323842	289	3065	2576	2562
10	45	7	38	296183	169	2290	1847	1829
10	45	9	36	267218	133	1846	1505	1485
10	45	12	33	250262	100	1681	1279	1268
10	45	15	30	217082	81	1321	980	971
10	45	18	27	209500	67	1204	903	885
10	45	20	25	191227	61	1239	844	835
10	45	23	22	172476	54	1065	759	757
10	45	26	19	169063	48	919	662	641
10	45	28	17	163434	45	918	641	627
10	45	31	14	166869	41	853	612	597
10	45	33	12	160096	39	827	598	596
10	45	36	9	161969	39	801	568	554
10	45	38	7	166677	39	837	592	571
10	45	40	5	159021	38	843	573	571
10	45	41	4	171166	39	866	604	589
10	45	42	3	171026	39	860	603	588
10	45	43	2	160194	38	872	589	587
10	55	2	53	310159	559	4508	3999	3994
10	55	5	50	287090	228	2818	2375	2354
10	55	8	47	253189	146	2070	1670	1665
10	55	12	43	223420	99	1757	1327	1306
10	55	15	40	202294	79	1282	974	968
10	55	18	37	180383	66	1238	891	880

continued on next page

Table 6.11. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
10	55	21	34	172069	57	1059	790	772
10	55	25	30	157822	48	947	659	654
10	55	28	27	149722	43	929	650	639
10	55	31	24	137655	39	843	565	559
10	55	35	20	129809	36	800	552	542
10	55	37	18	120973	33	701	498	484
10	55	41	14	123159	30	639	453	443
10	55	44	11	119421	29	707	455	441
10	55	47	8	123430	29	669	471	457
10	55	49	6	117808	28	679	459	449
10	55	51	4	121022	28	622	405	392
10	55	52	3	126603	29	711	488	474
10	55	53	2	115000	28	702	454	444
10	67	2	65	283390	556	4554	4003	3998
10	67	6	61	250824	189	2446	2010	1996
10	67	10	57	220082	114	1810	1430	1409
10	67	14	53	186661	82	1341	1050	1029
10	67	18	49	171867	64	1128	847	840
10	67	22	45	152373	53	978	714	708
10	67	26	41	141344	45	930	662	657
10	67	30	37	130512	39	825	566	551
10	67	34	33	119034	34	704	462	449
10	67	38	29	108758	31	671	456	435
10	67	42	25	101358	28	698	450	432
10	67	46	21	99658	26	667	440	427
10	67	50	17	84844	23	561	356	339
10	67	54	13	86312	21	574	342	327
10	67	57	10	87889	21	559	360	345
10	67	60	7	85898	21	523	349	332
10	67	62	5	86939	21	506	344	334
10	67	63	4	86724	21	514	350	340
10	67	64	3	86910	21	537	362	352
10	67	65	2	86863	21	512	348	338
10	81	2	79	270331	554	4631	4106	4092
10	81	7	74	232799	162	2261	1823	1802
10	81	12	69	200403	95	1523	1190	1172
10	81	17	64	174292	68	1178	869	855
10	81	22	59	141700	52	993	711	690
10	81	27	54	119754	42	799	583	578
10	81	32	49	114348	36	785	541	513
10	81	36	45	106330	32	732	488	471
10	81	41	40	97090	28	592	408	387
10	81	46	35	90406	24	584	384	370
10	81	51	30	93792	23	602	382	364
10	81	55	26	83496	21	593	360	347
10	81	60	21	85288	20	522	327	307
10	81	65	16	71694	18	399	281	275
10	81	69	12	65364	17	410	254	248

continued on next page

Table 6.11. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
10	81	72	9	70560	17	445	261	255
10	81	75	6	73053	17	407	269	263
10	81	76	5	73296	17	419	278	272
10	81	77	4	73629	17	440	285	279
10	81	78	3	73145	17	461	272	266
10	99	3	96	252476	371	3675	3162	3148
10	99	9	90	206197	124	1892	1504	1498
10	99	15	84	173680	75	1368	965	951
10	99	21	78	144853	54	1109	804	793
10	99	27	72	114794	41	840	598	583
10	99	33	66	107260	34	795	532	516
10	99	39	60	94829	28	646	421	414
10	99	45	54	85306	25	567	348	346
10	99	50	49	89400	22	629	383	366
10	99	56	43	76284	20	501	332	311
10	99	62	37	71390	18	449	296	283
10	99	67	32	64683	17	427	277	267
10	99	73	26	61765	16	354	226	212
10	99	79	20	56394	14	363	243	232
10	99	84	15	55458	13	388	241	224
10	99	88	11	52848	13	311	213	200
10	99	91	8	54754	13	347	235	218
10	99	93	6	56217	13	329	225	212
10	99	94	5	55777	13	354	237	220
10	99	95	4	55592	13	363	244	227
10	120	4	116	235627	276	3154	2689	2675
10	***120***	***11***	***109***	***188284***	***102***	***1667***	***1287***	***1273***
10	***120***	***18***	***102***	***153031***	***62***	***1088***	***833***	***810***
10	***120***	***25***	***95***	***116555***	***44***	***851***	***607***	***587***
10	120	32	88	116392	35	749	501	486
10	120	40	80	98180	28	696	457	436
10	120	47	73	78680	23	546	371	364
10	***120***	***54***	***66***	***86124***	***21***	***544***	***352***	***331***
10	***120***	***61***	***59***	***78551***	***18***	***503***	***324***	***311***
10	120	68	52	60098	16	417	260	250
10	120	76	44	64064	15	382	245	225
10	120	82	38	50010	13	343	210	202
10	***120***	***89***	***31***	***49037***	***12***	***309***	***204***	***194***
10	***120***	***96***	***24***	***34952***	***11***	***220***	***145***	***133***
10	***120***	***102***	***18***	***50741***	***11***	***291***	***182***	***174***
10	120	107	13	43288	10	322	187	179
10	120	110	10	42428	10	322	201	175
10	***120***	***113***	***7***	***41300***	***10***	***262***	***172***	***162***
10	120	115	5	42267	10	281	171	161
10	120	116	4	42398	10	313	185	175
10	147	4	143	233275	277	3088	2623	2602
10	147	13	134	174267	85	1465	1128	1119
10	147	22	125	137977	50	1021	711	709

continued on next page

Table 6.11. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
10	147	31	116	106754	35	720	513	503
10	147	40	107	94787	28	599	406	396
10	147	49	98	78981	22	516	330	313
10	147	57	90	68818	19	430	297	295
10	147	66	81	63941	16	425	273	252
10	147	75	72	59008	14	372	237	229
10	147	84	63	55011	13	342	205	198
10	147	93	54	42206	12	222	154	147
10	147	100	47	46389	11	328	204	197
10	147	109	38	47507	10	312	176	160
10	147	118	29	40519	9	292	162	145
10	147	125	22	34582	9	192	124	118
10	147	131	16	33209	8	210	145	125
10	147	135	12	34964	8	199	125	112
10	147	138	9	34830	8	248	161	143
10	147	140	7	34947	8	244	156	138
10	147	142	5	35807	8	235	149	141
11	15	1	14	467717	545	4571	4034	4024
11	15	2	13	475298	306	3285	2799	2789
11	15	3	12	455922	216	2622	2187	2164
11	15	4	11	443993	169	2370	1911	1901
11	15	5	10	435288	140	1954	1598	1588
11	15	6	9	423367	122	1844	1473	1462
11	15	7	8	398437	105	1666	1276	1265
11	15	8	7	397132	99	1600	1232	1221
11	15	9	6	393226	93	1564	1188	1177
11	15	10	5	389098	91	1565	1185	1174
11	15	11	4	391532	91	1489	1134	1123
11	15	12	3	395961	91	1534	1170	1159
11	15	13	2	398139	91	1604	1200	1189
11	15	14	1	398654	91	1572	1186	1175
11	19	1	18	370511	545	4571	4034	4024
11	19	2	17	363163	292	3287	2747	2737
11	19	3	16	338978	200	2542	2131	2121
11	19	4	15	327001	153	2168	1742	1719
11	19	5	14	302591	127	1947	1549	1523
11	19	6	13	285586	105	1738	1341	1318
11	19	7	12	263147	92	1547	1197	1186
11	19	9	10	257212	75	1354	992	975
11	19	10	9	243522	67	1333	967	950
11	19	11	8	230531	60	1189	849	832
11	19	12	7	230877	57	1167	830	813
11	19	13	6	222246	54	1131	808	791
11	19	14	5	232162	54	1180	840	823
11	19	15	4	232044	54	1168	825	808
11	19	16	3	235653	54	1176	823	806
11	19	17	2	229139	53	1138	810	793
11	19	18	1	228779	53	1149	813	796

continued on next page

Table 6.11. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
11	23	1	22	327404	545	4571	4034	4024
11	23	2	21	318153	286	3102	2655	2632
11	23	3	20	295698	195	2529	2107	2096
11	23	5	18	253774	118	1828	1415	1404
11	23	6	17	244165	102	1689	1281	1258
11	23	8	15	212816	77	1337	1014	988
11	23	9	14	202880	68	1235	901	878
11	23	10	13	183653	61	1081	826	809
11	23	12	11	177460	52	951	723	715
11	23	13	10	170799	49	997	689	672
11	23	14	9	147612	43	847	583	575
11	23	16	7	155945	39	865	614	606
11	23	17	6	145533	37	800	561	553
11	23	18	5	149099	37	797	550	542
11	23	20	3	155944	37	822	551	540
11	23	21	2	157396	37	814	553	545
11	23	22	1	157834	37	808	547	539
11	28	1	27	293010	545	4571	4034	4024
11	28	3	25	274851	193	2555	2155	2145
11	28	4	24	254458	145	2144	1711	1701
11	28	6	22	226542	97	1638	1286	1263
11	28	8	20	190617	74	1336	963	953
11	28	9	19	189565	66	1186	867	857
11	28	11	17	158746	53	976	713	687
11	28	13	15	146329	46	939	655	644
11	28	14	14	140082	43	829	566	556
11	28	16	12	124181	37	718	513	499
11	28	18	10	124900	34	679	484	468
11	28	19	9	120217	32	760	503	480
11	28	21	7	117506	29	680	455	447
11	28	22	6	113070	28	713	455	438
11	28	24	4	115508	28	649	460	437
11	28	25	3	117470	28	670	465	442
11	28	26	2	118514	28	677	470	447
11	28	27	1	119398	28	668	463	440
11	34	1	33	279528	545	4571	4034	4024
11	34	3	31	253264	189	2603	2107	2097
11	34	5	29	217520	114	1708	1356	1333
11	34	7	27	199918	83	1382	1072	1061
11	34	9	25	165250	65	1185	840	832
11	34	11	23	145333	53	1059	724	713
11	34	13	21	130353	44	864	624	616
11	34	15	19	123999	39	823	569	561
11	34	17	17	121405	34	883	585	572
11	34	19	15	109440	31	726	477	454
11	34	21	13	101414	28	739	470	444
11	34	23	11	93235	26	629	410	387
11	34	25	9	95668	24	614	389	381

continued on next page

Table 6.11. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
11	34	27	7	92611	22	505	326	315
11	34	29	5	87358	21	509	342	334
11	34	30	4	88396	21	495	331	323
11	34	31	3	89963	21	497	327	319
11	34	32	2	90399	21	488	318	310
11	34	33	1	90889	21	487	321	313
11	41	1	40	262820	545	4571	4034	4024
11	41	4	37	224937	143	2115	1698	1688
11	41	6	35	198688	95	1576	1202	1179
11	41	9	32	155193	63	1184	871	860
11	41	11	30	146092	51	1037	699	682
11	41	14	27	113557	40	816	590	576
11	41	16	25	101849	35	772	536	516
11	41	18	23	101990	31	675	452	438
11	41	21	20	95449	27	625	397	383
11	41	23	18	91900	25	578	395	382
11	41	26	15	85609	22	607	380	364
11	41	28	13	72534	20	431	293	281
11	41	30	11	80081	19	502	315	295
11	41	33	8	67212	17	426	309	291
11	41	35	6	69432	17	432	256	233
11	41	36	5	66597	16	366	240	219
11	41	38	3	68564	16	377	245	227
11	41	39	2	69026	16	382	256	238
11	41	40	1	69199	16	376	250	232
11	50	1	49	250849	545	4571	4034	4024
11	50	4	46	222880	140	1949	1579	1553
11	50	7	43	176061	81	1424	1068	1051
11	50	10	40	150107	56	1179	839	816
11	50	13	37	115751	43	852	592	575
11	50	16	34	108995	35	808	553	536
11	50	19	31	96094	29	640	420	407
11	50	22	28	87250	25	601	392	381
11	50	25	25	84153	22	581	366	350
11	50	29	21	68436	19	454	287	274
11	50	31	19	77620	18	504	302	288
11	50	34	16	60806	16	413	262	250
11	50	37	13	53895	15	392	259	245
11	50	40	10	56918	14	376	219	210
11	50	42	8	53551	13	350	221	210
11	50	44	6	54148	13	388	244	224
11	50	46	4	53881	13	301	189	178
11	50	47	3	54010	13	304	192	181
11	50	48	2	54619	13	329	223	212
11	60	2	58	237410	276	3063	2606	2596
11	60	5	55	191152	110	1730	1367	1341
11	60	9	51	149451	62	1127	838	815
11	60	13	47	118581	43	884	614	588

continued on next page

Table 6.11. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
11	60	16	44	95876	34	744	508	494
11	60	20	40	93150	28	653	415	398
11	60	23	37	77962	24	520	353	343
11	60	27	33	87116	21	534	353	330
11	60	31	29	75593	18	471	289	281
11	60	34	26	56701	16	365	242	231
11	60	38	22	53276	14	341	218	203
11	60	41	19	53381	13	344	214	206
11	60	44	16	47310	12	324	201	188
11	60	48	12	40984	11	291	158	145
11	60	51	9	50988	11	318	189	179
11	60	53	7	41279	10	348	211	198
11	60	55	5	41443	10	254	156	147
11	60	56	4	41378	10	241	148	139
11	60	57	3	42767	10	316	182	173
11	60	58	2	43068	10	316	190	181
11	74	2	72	232550	275	3084	2606	2583
11	74	7	67	174698	79	1459	1104	1087
11	74	11	63	133291	51	1010	706	695
11	74	16	58	98364	34	723	495	487
11	74	20	54	85201	27	544	353	340
11	74	24	50	71846	23	538	349	341
11	74	29	45	71782	19	496	297	289
11	74	33	41	60370	16	422	281	272
11	74	38	36	59700	14	354	215	213
11	74	42	32	55605	13	400	272	246
11	74	47	27	36852	11	256	158	145
11	74	50	24	44382	11	311	192	184
11	74	55	19	38009	10	267	181	155
11	74	59	15	38994	9	250	154	141
11	74	63	11	31356	8	252	156	137
11	74	66	8	36166	8	273	148	135
11	74	68	6	33072	8	239	159	146
11	74	70	4	33881	8	237	160	147
11	74	71	3	33182	8	212	137	123
12	8	1	7	391274	271	3057	2568	2547
12	8	2	6	399789	164	2294	1851	1830
12	8	3	5	362210	120	1883	1441	1420
12	8	4	4	339700	92	1487	1111	1092
12	8	5	3	350733	83	1558	1143	1124
12	8	6	2	345465	81	1482	1091	1072
12	8	7	1	348829	81	1489	1100	1081
12	10	1	9	327103	271	3057	2568	2547
12	10	2	8	308676	152	2131	1732	1713
12	10	3	7	278388	106	1749	1327	1306
12	10	4	6	254996	83	1374	1031	1012
12	10	5	5	232789	68	1271	956	937
12	10	6	4	212523	56	1020	770	747

continued on next page

Table 6.11. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
12	10	7	3	206265	50	1026	752	729
12	10	8	2	210689	50	1013	727	704
12	10	9	1	213840	50	1016	730	707
12	12	1	11	297240	271	3057	2568	2547
12	12	2	10	270348	147	2176	1714	1695
12	12	3	9	245794	102	1722	1294	1271
12	12	4	8	212952	78	1298	973	962
12	12	5	7	189565	61	1195	848	837
12	12	6	6	176931	53	1082	775	756
12	12	7	5	144679	45	842	603	593
12	12	8	4	145702	39	801	581	567
12	12	9	3	148323	36	739	512	498
12	12	10	2	150258	36	805	550	536
12	12	11	1	152741	36	789	550	536
12	14	1	13	277819	271	3057	2568	2547
12	14	2	12	254597	145	2081	1670	1649
12	14	3	11	229450	99	1638	1234	1215
12	14	4	10	198325	75	1302	959	948
12	14	5	9	172695	59	1108	806	795
12	14	6	8	151306	49	1025	701	691
12	14	7	7	137507	43	835	598	584
12	14	8	6	146970	39	848	634	611
12	14	9	5	125481	35	822	572	549
12	14	10	4	121787	31	757	519	505
12	14	11	3	120946	29	667	452	429
12	14	12	2	126089	29	684	466	443
12	14	13	1	128092	29	706	481	458
12	17	1	16	261385	271	3057	2568	2547
12	17	2	15	233292	142	2032	1636	1617
12	17	3	14	208296	97	1602	1195	1174
12	17	4	13	183445	73	1287	940	919
12	17	5	12	175017	60	1147	848	829
12	17	6	11	141775	48	972	693	674
12	17	7	10	124339	41	793	574	555
12	17	8	9	119844	36	811	564	553
12	17	9	8	97831	32	684	444	442
12	17	10	7	100516	29	671	429	419
12	17	11	6	100024	27	612	428	405
12	17	12	5	89294	25	574	370	360
12	17	13	4	92416	23	589	383	369
12	17	14	3	93215	22	602	389	370
12	17	15	2	86762	21	531	362	343
12	17	16	1	89865	21	522	354	335
12	21	1	20	247258	271	3057	2568	2547
12	21	2	19	217152	140	1980	1588	1569
12	21	3	18	203286	95	1638	1281	1258
12	21	4	17	174774	72	1247	922	901
12	21	6	15	134792	48	1027	712	689

continued on next page

Table 6.11. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
12	21	7	14	122846	41	816	578	568
12	21	8	13	110393	35	738	494	492
12	21	9	12	105155	32	662	429	408
12	21	11	10	96155	26	664	422	412
12	21	12	9	82099	24	598	390	369
12	21	13	8	98561	23	679	432	411
12	21	14	7	85749	21	535	343	324
12	21	16	5	71020	18	408	288	269
12	21	17	4	70335	17	432	266	255
12	21	18	3	65459	16	425	263	255
12	21	19	2	66307	16	393	239	228
12	21	20	1	67838	16	407	242	231
12	25	1	24	238492	271	3057	2568	2547
12	25	2	23	210776	139	2018	1605	1586
12	25	4	21	159405	70	1298	934	923
12	25	5	20	146865	56	1156	814	793
12	25	7	18	110779	40	779	535	514
12	25	8	17	103221	34	670	463	461
12	25	10	15	101145	28	643	451	432
12	25	11	14	92489	26	604	398	379
12	25	13	12	77025	21	564	347	338
12	25	14	11	74428	20	503	324	305
12	25	16	9	55111	17	340	244	235
12	25	17	8	69745	17	456	293	274
12	25	19	6	64529	15	430	258	256
12	25	20	5	61803	14	404	244	228
12	25	21	4	53833	13	353	216	199
12	25	22	3	53352	13	296	202	188
12	25	23	2	55324	13	359	239	229
12	25	24	1	53301	13	318	201	180
12	30	1	29	232094	271	3057	2568	2547
12	30	3	27	188523	94	1570	1182	1161
12	30	4	26	157400	70	1187	871	850
12	30	6	24	126064	46	985	664	645
12	30	8	22	101476	34	674	451	449
12	30	10	20	89780	27	583	401	382
12	30	12	18	76174	23	528	352	329
12	30	13	17	75089	21	496	331	312
12	30	15	15	74592	18	549	331	322
12	30	17	13	61366	16	404	260	241
12	30	19	11	64284	15	419	271	248
12	30	20	10	59810	14	421	249	239
12	30	22	8	51434	13	334	211	192
12	30	24	6	45465	12	381	208	187
12	30	26	4	53604	11	358	219	217
12	30	27	3	41842	10	314	174	172
12	30	28	2	42424	10	316	203	179
12	30	29	1	43007	10	340	216	192

continued on next page

Table 6.11. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
12	37	1	36	228979	271	3057	2568	2547
12	37	3	34	175709	92	1634	1177	1158
12	37	6	31	126257	46	883	636	625
12	37	8	29	106068	35	766	516	495
12	37	10	27	87741	27	664	434	415
12	37	12	25	74354	22	539	351	331
12	37	14	23	72760	19	455	280	272
12	37	17	20	62144	16	395	263	244
12	37	19	18	64272	14	406	244	235
12	37	21	16	47538	13	304	204	202
12	37	23	14	40088	12	244	180	161
12	37	25	12	42758	11	300	198	175
12	37	27	10	44140	10	294	179	159
12	37	30	7	40245	9	228	154	147
12	37	31	6	36432	9	247	145	135
12	37	33	4	35855	8	254	149	137
12	37	34	3	34125	8	255	158	149
12	37	35	2	35532	8	234	151	142
12	37	36	1	33966	8	160	108	100
13	4	1	3	318551	135	1913	1572	1546
13	4	2	2	436811	110	1697	1353	1327
13	4	3	1	436275	99	1624	1275	1249
13	5	1	4	268624	135	1913	1572	1546
13	5	2	3	248575	83	1414	1094	1068
13	5	3	2	256475	64	1221	877	873
13	5	4	1	235847	55	1059	752	748
13	6	1	5	249846	135	1913	1572	1546
13	6	2	4	219588	79	1296	990	964
13	6	3	3	170941	54	909	681	677
13	6	4	2	163912	42	850	614	604
13	6	5	1	159923	38	772	547	537
13	7	1	6	232852	135	1913	1572	1546
13	7	2	5	196924	75	1219	950	946
13	7	3	4	167787	51	959	717	713
13	7	4	3	147305	40	913	654	644
13	7	5	2	127578	32	688	468	450
13	7	6	1	129263	30	661	442	438
13	9	1	8	217996	135	1913	1572	1546
13	9	2	7	183220	73	1255	951	925
13	9	3	6	152942	50	1015	727	723
13	9	4	5	122393	37	741	538	512
13	9	5	4	108763	29	675	455	437
13	9	6	3	88329	25	517	364	354
13	9	7	2	90267	22	510	354	344
13	9	8	1	86228	20	575	358	340
13	11	1	10	209489	135	1913	1572	1546
13	11	2	9	174983	71	1324	981	977
13	11	3	8	142179	48	994	726	700

continued on next page

Table 6.11. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
13	11	4	7	107357	35	696	491	473
13	11	5	6	99178	29	638	444	434
13	11	6	5	83828	24	518	355	329
13	11	7	4	88930	21	518	340	330
13	11	8	3	66720	18	402	277	273
13	11	9	2	60503	16	333	225	214
13	11	10	1	65755	15	375	231	221
13	13	1	12	206908	135	1913	1572	1546
13	13	2	11	164052	70	1135	877	873
13	13	3	10	132097	47	943	679	675
13	13	4	9	101890	35	635	459	455
13	13	5	8	94758	28	612	415	411
13	13	6	7	81135	23	556	376	365
13	13	7	6	80898	20	531	336	318
13	13	8	5	58827	17	395	275	264
13	13	9	4	61626	16	405	261	235
13	13	10	3	52365	14	355	221	203
13	13	11	2	53726	13	280	204	193
13	13	12	1	48820	12	279	191	175
13	15	1	14	200050	135	1913	1572	1546
13	15	2	13	160710	70	1206	939	913
13	15	3	12	126118	46	877	656	646
13	15	4	11	104712	35	710	517	513
13	15	5	10	93149	28	624	425	421
13	15	6	9	69306	23	451	318	314
13	15	7	8	67537	20	459	319	293
13	15	8	7	60100	17	394	282	271
13	15	9	6	59037	15	304	198	181
13	15	10	5	58963	14	444	267	241
13	15	11	4	46299	12	293	198	181
13	15	12	3	46231	12	278	170	166
13	15	13	2	54906	11	371	239	223
13	15	14	1	43530	10	317	182	165
13	19	1	18	197371	135	1913	1572	1546
13	19	2	17	156245	69	1142	870	866
13	19	3	16	126665	46	915	654	644
13	19	4	15	105679	35	733	538	512
13	19	5	14	90124	27	645	443	425
13	19	6	13	73717	22	508	349	332
13	19	7	12	67304	19	429	290	274
13	19	9	10	53492	15	303	203	192
13	19	10	9	55737	13	388	228	218
13	19	11	8	44266	12	318	199	189
13	19	12	7	41152	11	294	190	174
13	19	13	6	48461	10	310	178	163
13	19	14	5	36592	9	243	162	154
13	19	15	4	44822	9	279	181	165
13	19	16	3	34733	8	224	146	126

continued on next page

Table 6.11. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
13	19	17	2	36483	8	199	131	121
13	19	18	1	33264	8	205	112	108
14	2	1	1	237650	67	1162	850	834
14	3	1	2	178182	67	1162	850	834
14	3	2	1	216440	53	996	704	688
14	4	1	3	164455	67	1162	850	834
14	4	2	2	133541	40	819	568	552
14	4	3	1	114907	29	618	420	404
14	5	1	4	155426	67	1162	850	834
14	5	2	3	119311	37	734	500	484
14	5	3	2	81742	25	531	361	345
14	5	4	1	89800	21	515	346	330
14	6	1	5	158110	67	1162	850	834
14	6	2	4	110423	36	690	471	455
14	6	3	3	77122	24	514	339	323
14	6	4	2	63049	18	386	263	247
14	6	5	1	62696	15	379	243	230
14	7	1	6	154939	67	1162	850	834
14	7	2	5	105229	35	707	484	468
14	7	3	4	78542	23	523	327	317
14	7	4	3	74142	18	508	331	315
14	7	5	2	54897	14	331	220	207
14	7	6	1	45839	12	309	193	180
14	8	1	7	148212	67	1162	850	834
14	8	2	6	101723	35	714	489	473
14	8	3	5	75142	23	510	331	315
14	8	4	4	61839	17	410	265	255
14	8	5	3	64250	14	388	245	232
14	8	6	2	47848	11	321	182	167
14	8	7	1	42750	10	299	189	174
14	10	1	9	147116	67	1162	850	834
14	10	2	8	108398	35	793	527	511
14	10	3	7	73366	23	510	334	318
14	10	4	6	68584	17	460	293	277
14	10	5	5	60619	13	416	260	249
14	10	6	4	42176	11	282	173	160
14	10	7	3	34555	9	216	147	139
14	10	8	2	33207	8	241	140	125
14	10	9	1	33189	8	201	116	100

Table 6.12. Costs of block 2GYF for Richards

Configuration				Copied	Invoc	W.b. insert	W.b. add	R.s. proc
10	8	1	7	4776121	17920	4566	4485	4485
10	8	2	6	4831103	10482	4282	4223	4223
10	8	3	5	4899711	7462	4189	4143	4143
10	8	4	4	6554704	6196	4087	4044	4044
10	8	5	3	6856678	6063	4065	4027	4027
10	8	6	2	6829140	6045	4113	4076	4076
10	8	7	1	6825616	6041	4083	4047	4047
10	9	1	8	4117706	17920	4566	4485	4485
10	9	2	7	3743626	9996	4178	4133	4133
10	9	3	6	3714939	6885	4062	4002	4002
10	9	4	5	3538799	5194	3985	3967	3967
10	9	5	4	4732598	4498	3880	3746	3746
10	9	6	3	5020488	4466	3891	3803	3803
10	9	7	2	5024717	4468	3879	3794	3794
10	9	8	1	5001620	4464	3911	3824	3824
10	11	1	10	3343032	17920	4566	4485	4485
10	11	2	9	2814513	9574	4212	4157	4157
10	11	3	8	2364919	6452	4073	4038	4038
10	11	4	7	2041471	4776	3975	3936	3936
10	11	5	6	2211442	3908	3817	3251	3251
10	11	6	5	2480657	3314	3883	2936	2936
10	11	7	4	3277608	3055	3954	2805	2805
10	11	8	3	3449058	3030	3921	2762	2762
10	11	9	2	3441060	3024	3891	2737	2737
10	11	10	1	3440124	3022	3924	2763	2763
10	13	1	12	2987601	17920	4566	4485	4485
10	13	2	11	2427044	9404	4178	4130	4130
10	13	3	10	2050222	6263	4056	4024	4024
10	13	4	9	1652677	4690	3969	3945	3945
10	13	5	8	1591705	3758	3811	3201	3201
10	13	6	7	1462558	3132	3880	2821	2821
10	13	7	6	1488790	2706	3812	2420	2420
10	13	8	5	1649225	2405	3802	2177	2177
10	13	9	4	2358853	2246	3794	2040	2040
10	13	10	3	2502993	2233	3778	2019	2019
10	13	11	2	2518551	2236	3740	1988	1988
10	13	12	1	2510513	2237	3755	2001	2001
10	16	1	15	2712144	17920	4566	4485	4485
10	16	2	14	2148891	9273	4135	4094	4094
10	16	3	13	1766954	6178	4024	3982	3982
10	16	4	12	1403765	4609	3972	3938	3938
10	16	5	11	1281797	3685	3863	3210	3210
10	16	6	10	1118147	3070	3828	2716	2716
10	16	7	9	1001276	2627	3838	2393	2393
10	16	8	8	926526	2303	3809	2102	2102
10	16	9	7	967599	2056	3753	1846	1846
10	16	10	6	1013001	1863	3749	1704	1704
10	16	11	5	1204985	1723	3770	1608	1608

continued on next page

Table 6.12. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
10	16	12	4	1718827	1643	3770	1546	1546
10	16	13	3	1831175	1631	3760	1526	1526
10	16	14	2	1834156	1632	3748	1517	1517
10	16	15	1	1831967	1631	3787	1555	1555
10	20	1	19	2503213	17920	4566	4485	4485
10	20	2	18	1968469	9185	4109	4064	4064
10	20	3	17	1581980	6119	4006	3975	3975
10	20	4	16	1288065	4574	3953	3922	3922
10	20	5	15	1132162	3653	3847	3180	3180
10	20	7	13	842036	2601	3804	2335	2335
10	20	8	12	728287	2272	3798	2065	2065
10	20	9	11	714790	2023	3757	1831	1831
10	20	10	10	661131	1821	3733	1658	1658
10	**20**	**11**	**9**	**631920**	**1656**	**3754**	**1541**	**1541**
10	20	13	7	646652	1408	3723	1319	1319
10	20	14	6	712836	1316	3727	1248	1248
10	20	15	5	884611	1244	3722	1188	1188
10	20	16	4	1264113	1204	3742	1168	1168
10	20	17	3	1348834	1197	3736	1157	1157
10	20	18	2	1345162	1195	3734	1160	1160
10	20	19	1	1344093	1195	3732	1157	1157
10	24	1	23	2375629	17920	4566	4485	4485
10	24	2	22	1843985	9134	4126	4081	4081
10	24	4	20	1220944	4550	3972	3939	3939
10	24	5	19	1049384	3636	3848	3171	3171
10	24	6	18	887402	3024	3830	2695	2695
10	24	8	16	637765	2262	3828	2082	2082
10	24	9	15	627700	2012	3764	1832	1832
10	24	11	13	532549	1645	3766	1544	1544
10	24	12	12	467101	1507	3749	1419	1419
10	24	14	10	477360	1296	3705	1216	1216
10	24	15	9	463291	1209	3702	1141	1141
10	24	16	8	455311	1135	3737	1116	1116
10	24	18	6	560160	1018	3706	988	988
10	24	19	5	694111	973	3724	971	971
10	24	20	4	994477	948	3732	958	958
10	24	21	3	1067309	945	3701	934	934
10	24	22	2	1063662	944	3720	949	949
10	24	23	1	1065258	944	3705	937	937
10	29	1	28	2281821	17920	4566	4485	4485
10	29	3	26	1431668	6055	3964	3935	3935
10	29	4	25	1143851	4532	3949	3917	3917
10	29	6	23	838891	3014	3818	2680	2680
10	29	8	21	556491	2254	3856	2090	2090
10	29	10	19	527573	1804	3751	1658	1658
10	29	11	18	481239	1640	3764	1538	1538
10	29	13	16	416555	1387	3731	1313	1313
10	29	15	14	381543	1202	3715	1146	1146

continued on next page

Table 6.12. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
10	29	17	12	347218	1061	3699	1023	1023
10	29	18	11	353575	1003	3730	999	999
10	29	20	9	325805	903	3714	914	914
10	29	21	8	357251	862	3701	871	871
10	29	23	6	432847	792	3718	825	825
10	29	25	4	791003	748	3685	768	768
10	29	26	3	839343	746	3687	768	768
10	29	27	2	841519	746	3680	760	760
10	29	28	1	838151	746	3705	779	779
10	35	1	34	2222445	17920	4566	4485	4485
10	35	3	32	1383958	6039	3955	3930	3930
10	35	5	30	962469	3614	3830	3149	3149
10	35	7	28	692627	2575	3807	2324	2324
10	35	9	26	548575	2001	3748	1814	1814
10	35	12	23	423565	1500	3730	1401	1401
10	35	14	21	368682	1285	3719	1222	1222
10	35	16	19	323416	1125	3726	1094	1094
10	35	18	17	304221	1000	3723	993	993
10	35	20	15	283258	900	3700	895	895
10	35	22	13	269929	818	3703	837	837
10	35	24	11	249701	750	3724	795	795
10	35	26	9	269243	694	3686	727	727
10	35	28	7	288297	646	3709	699	699
10	35	30	5	433640	608	3675	648	646
10	35	31	4	631348	599	3668	634	634
10	35	32	3	675968	598	3674	637	637
10	35	33	2	675537	598	3679	643	643
10	35	34	1	673119	597	3670	636	636
11	4	1	3	3424861	8888	4316	4170	4170
11	4	2	2	8696827	8205	4262	4132	4132
11	4	3	1	8792578	7802	4224	4108	4108
11	5	1	4	2398681	8888	4316	4170	4170
11	5	2	3	2412548	5184	4021	3947	3947
11	5	3	2	4642961	4301	3987	3774	3774
11	5	4	1	4781096	4229	3996	3711	3711
11	6	1	5	2040481	8888	4316	4170	4170
11	6	2	4	1692238	4841	4008	3928	3928
11	6	3	3	1675450	3304	3926	2951	2951
11	6	4	2	2972053	2863	3847	2582	2582
11	6	5	1	3117662	2794	3864	2550	2550
11	7	1	6	1847645	8888	4316	4170	4170
11	7	2	5	1436990	4716	4037	3958	3958
11	7	3	4	1200888	3149	3889	2823	2823
11	7	4	3	1149594	2397	3798	2156	2156
11	7	5	2	2303501	2176	3816	1998	1998
11	7	6	1	2418731	2152	3817	1976	1976
11	8	1	7	1734422	8888	4316	4170	4170
11	8	2	6	1365213	4661	3936	3898	3898

continued on next page

Table 6.12. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
11	8	3	5	1039329	3093	3912	2795	2795
11	8	4	4	874352	2324	3822	2131	2131
11	8	5	3	966929	1896	3802	1760	1760
11	8	6	2	1795201	1739	3776	1624	1624
11	8	7	1	1905791	1711	3799	1622	1622
11	10	1	9	1596827	8888	4316	4170	4170
11	10	2	8	1198564	4588	4021	3934	3934
11	10	3	7	892715	3044	3898	2748	2748
11	10	4	6	707717	2278	3842	2105	2105
11	10	5	5	613348	1824	3792	1700	1700
11	**10**	**6**	**4**	**588878**	**1527**	**3762**	**1445**	**1445**
11	10	7	3	668509	1328	3769	1287	1287
11	10	8	2	1284229	1249	3739	1206	1206
11	10	9	1	1376638	1237	3749	1207	1207
11	12	1	11	1518730	8888	4316	4170	4170
11	12	2	10	1195129	4560	3985	3925	3925
11	12	3	9	825985	3021	3890	2724	2724
11	12	4	8	642937	2260	3861	2095	2095
11	12	5	7	527816	1806	3882	1753	1753
11	12	6	6	471964	1506	3770	1432	1432
11	12	7	5	435137	1293	3764	1251	1251
11	12	8	4	447431	1137	3753	1123	1123
11	12	9	3	518262	1022	3750	1030	1030
11	12	10	2	1004547	975	3699	957	957
11	12	11	1	1078490	968	3725	974	974
11	15	1	14	1450839	8888	4316	4170	4170
11	15	2	13	1117869	4528	4003	3911	3911
11	15	3	12	775520	3004	3881	2708	2708
11	15	4	11	580270	2247	3868	2092	2092
11	15	5	10	482586	1796	3776	1667	1667
11	15	6	9	414996	1496	3751	1401	1401
11	15	7	8	362876	1282	3752	1240	1240
11	15	8	7	333435	1122	3741	1104	1104
11	15	9	6	309888	998	3726	990	990
11	15	10	5	307668	900	3736	924	924
11	15	11	4	327314	821	3716	839	839
11	15	12	3	385295	760	3722	807	807
11	15	13	2	757413	734	3701	766	766
11	15	14	1	810667	729	3718	771	771
11	18	1	17	1408932	8888	4316	4170	4170
11	18	2	16	1060078	4509	4003	3924	3924
11	18	3	15	746981	2995	3928	2733	2733
11	18	4	14	566895	2241	3840	2071	2071
11	18	5	13	457060	1791	3790	1670	1670
11	18	6	12	388143	1491	3744	1397	1397
11	18	7	11	340638	1278	3744	1221	1221
11	18	8	10	301082	1118	3757	1111	1111
11	18	9	9	276301	994	3731	994	994

continued on next page

Table 6.12. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
11	18	10	8	255256	894	3736	914	914
11	18	11	7	242580	814	3728	844	844
11	18	12	6	236611	746	3714	789	789
11	18	13	5	233405	690	3709	737	737
11	18	14	4	249357	643	3710	696	696
11	18	15	3	301618	604	3704	656	656
11	18	16	2	613056	588	3681	641	641
11	18	17	1	654773	585	3719	657	657
12	2	1	1	4392162	4404	4360	4283	4283
12	3	1	2	1617622	4404	4360	4283	4283
12	3	2	1	4149809	3976	4318	3885	3885
12	4	1	3	1304469	4404	4360	4283	4283
12	4	2	2	908356	2412	4076	2417	2417
12	4	3	1	2120789	2072	3998	2088	2088
12	5	1	4	1190238	4404	4360	4283	4283
12	5	2	3	702541	2313	4030	2305	2305
12	5	3	2	586104	1558	3913	1609	1609
12	5	4	1	1417384	1403	3886	1463	1463
12	6	1	5	1128876	4404	4360	4283	4283
12	6	2	4	629174	2277	4036	2290	2290
12	6	3	3	460320	1516	3891	1558	1558
12	6	4	2	433706	1151	3846	1227	1227
12	6	5	1	1064167	1062	3829	1140	1140
12	8	1	7	1065570	4404	4360	4283	4283
12	8	2	6	569315	2247	4019	2250	2250
12	8	3	5	391781	1493	3896	1546	1546
12	8	4	4	312968	1119	3828	1185	1185
12	8	5	3	274698	898	3804	989	989
12	8	6	2	284271	755	3782	860	860
12	8	7	1	718010	716	3750	802	802
12	9	1	8	1047513	4404	4360	4283	4283
12	9	2	7	553727	2240	4031	2254	2254
12	9	3	6	380387	1488	3889	1532	1532
12	9	4	5	294040	1115	3848	1199	1199
12	9	5	4	249750	892	3799	980	980
12	9	6	3	228398	745	3774	844	844
12	9	7	2	246342	645	3752	741	741
12	9	8	1	614526	616	3746	718	718
13	2	1	1	862635	2156	3824	2006	2006
13	3	1	2	638274	2156	3824	2006	2006
13	3	2	1	480752	1188	3752	1169	1169
13	4	1	3	586894	2156	3824	2006	2006
13	4	2	2	334341	1117	3735	1101	1101
13	4	3	1	308568	765	3692	789	789
13	5	1	4	563813	2156	3824	2006	2006
13	5	2	3	303795	1102	3731	1083	1083
13	5	3	2	222709	736	3711	780	780
13	5	4	1	229407	565	3698	634	634

Table 6.13. Costs of block 2GYF for JavaBYTEmark

Configuration				Copied	Invoc	W.b. insert	W.b. add	R.s. proc
18	2	1	1	150427	20	278	245	155
18	3	1	2	133592	20	278	245	155
18	3	2	1	87789	10	177	127	30
18	4	1	3	133592	20	278	245	155
18	4	2	2	71718	10	158	120	30
18	4	3	1	51493	6	54	19	17
18	5	1	4	133592	20	278	245	155
18	5	2	3	71718	10	158	120	30
18	5	3	2	51493	6	54	19	17
18	5	4	1	46855	5	137	106	16
18	6	1	5	133592	20	278	245	155
18	6	2	4	71718	10	158	120	30
18	6	3	3	51493	6	54	19	17
18	6	4	2	46855	5	137	106	16
18	**6**	**5**	**1**	**37842**	**4**	**135**	**102**	**12**

Table 6.14. Costs of block 2GYF for Bloat-Bloat

Configuration				Copied	Invoc	W.b. insert	W.b. add	R.s. proc
16	16	1	15	5586489	2283	138221	57562	57551
16	16	2	14	4923038	1152	99475	40320	40309
16	16	3	13	4715154	768	81049	34907	34886
16	16	4	12	4769205	577	73694	30406	30395
16	16	5	11	4589356	461	61502	27895	27830
16	16	6	10	4853174	385	56370	24494	24444
16	16	7	9	5420551	331	47851	23193	23109
16	16	8	8	7350601	293	51422	22082	22007
16	16	9	7	10178457	266	52494	22512	22462
16	16	10	6	12715009	247	41721	20085	20035
16	16	11	5	15440731	235	46973	19895	19884
16	16	12	4	17860932	228	41700	19934	19902
16	16	13	3	19771607	227	44477	19375	19343
16	16	14	2	19725327	226	44760	20898	20877
16	19	1	18	5210358	2283	138221	57562	57551
16	19	2	17	4561459	1149	102098	41552	41531
16	19	3	16	4206513	766	83378	36196	36175
16	19	4	15	4035345	574	77174	31419	31369
16	19	5	14	3846856	460	60698	27618	27584
16	19	6	13	3663016	383	59087	25511	25446
16	19	7	12	3658250	328	43972	22859	22784
16	19	9	10	4123333	257	44681	21227	21216
16	19	10	9	4312733	231	43514	19989	19914
16	19	11	8	5610469	211	37932	18615	18511
16	19	12	7	7916579	197	38141	18474	18463
16	19	13	6	9843328	186	42388	18795	18763
16	19	14	5	11835670	179	38067	18845	18780
16	19	15	4	13750934	175	34301	17815	17740
16	19	16	3	14787025	174	31198	17706	17641
16	19	17	2	14995648	173	29905	16968	16909
16	19	18	1	15059949	174	30951	17398	17387
16	23	1	22	4755921	2283	138221	57562	57551
16	23	2	21	4277460	1147	102181	41750	41729
16	23	3	20	3541449	764	77768	34328	34294
16	23	5	18	3261615	459	64261	29390	29379
16	23	6	17	3264059	382	52547	24502	24437
16	23	8	15	3094966	286	44520	22355	22261
16	23	9	14	3164740	255	40299	21421	21346
16	23	10	13	3386693	230	41373	19424	19413
16	23	12	11	3176402	191	40740	17893	17834
16	23	13	10	3415013	177	35112	16928	16844
16	23	14	9	3616462	165	34185	16398	16377
16	23	16	7	5891127	146	26892	15109	15009
16	23	17	6	7105112	140	29045	15254	15179
16	23	18	5	9040941	136	30348	15064	14960
16	23	20	3	11275372	133	28565	14574	14563
16	23	21	2	11226326	132	28489	15143	15049
16	28	1	27	4544476	2283	138221	57562	57551

continued on next page

Table 6.14. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
16	28	3	25	3589931	763	81160	35987	35953
16	28	4	24	3366903	573	68538	29633	29622
16	28	6	22	2862583	381	58255	25603	25528
16	28	8	20	2647460	286	46183	20935	20870
16	28	9	19	2709352	254	41457	20849	20774
16	28	11	17	2634141	208	44445	18340	18256
16	**28**	**13**	**15**	**2498540**	**176**	**33304**	**16469**	**16375**
16	28	14	14	2585478	164	27113	15943	15893
16	28	16	12	3068428	144	32452	15417	15383
16	28	18	10	2701472	127	25602	14853	14737
16	28	19	9	3084709	121	28298	15007	14913
16	28	21	7	4676406	111	27621	13071	12996
16	28	22	6	5947201	107	25784	12594	12505
16	28	24	4	7868865	103	24832	13709	13644
16	28	25	3	8826807	103	24805	13321	13300
16	28	26	2	8747838	102	21833	12826	12769
16	34	1	33	4291518	2283	138221	57562	57551
16	34	3	31	3439548	763	78657	33458	33437
16	34	5	29	2769184	457	60106	28170	28105
16	34	7	27	2528534	326	49531	24670	24586
16	34	9	25	2371660	254	40173	21678	21644
16	34	11	23	2539118	208	43070	18229	18179
16	34	13	21	2296614	176	33782	16538	16504
16	34	15	19	2181897	152	34500	16923	16885
16	34	17	17	2133641	134	29449	15420	15382
16	34	19	15	2216944	120	25067	13516	13373
16	34	21	13	2333964	109	25622	13247	13143
16	34	23	11	2457187	100	27459	12480	12430
16	34	25	9	2671649	92	25743	12927	12843
16	34	27	7	3599351	86	19897	11355	11290
16	34	29	5	5338996	82	21322	10597	10503
16	34	30	4	6291831	81	28603	10834	10730
16	34	31	3	6777718	80	20715	11475	11406
16	34	32	2	6776128	80	21470	11452	11373
16	34	33	1	6794862	80	20580	11934	11854
16	41	1	40	4226891	2283	138221	57562	57551
16	41	4	37	2838193	571	73868	29873	29823
16	41	6	35	2571537	381	53400	24261	24196
16	41	9	32	2361250	254	43615	20838	20788
16	41	11	30	2165799	207	40903	18726	18667
16	41	14	27	1952919	163	29421	17415	17331
16	41	16	25	1833028	142	26706	15331	15215
16	41	18	23	2042049	127	27776	14072	13997
16	41	21	20	2081814	109	28024	13783	13772
16	41	23	18	2061719	99	30216	12104	12015
16	41	26	15	2067274	88	22882	11440	11346
16	41	28	13	1772593	81	19416	11441	11362
16	41	30	11	2065876	76	20434	11096	11033

continued on next page

Table 6.14. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
16	41	33	8	2789307	70	24420	11967	11956
16	41	35	6	3840382	66	24699	9914	9759
16	41	36	5	4395318	65	20141	10664	10567
16	41	38	3	5306929	64	16194	9373	9286
16	41	39	2	5365141	64	14874	8953	8866
16	41	40	1	5323811	64	18263	9603	9540
16	50	1	49	4127288	2283	138221	57562	57551
16	50	4	46	2840778	571	71090	31224	31190
16	50	7	43	2473833	326	48454	23025	22960
16	50	10	40	2223875	228	46367	19463	19379
16	50	13	37	2066038	175	35371	16772	16734
16	50	16	34	1839606	142	26070	14500	14411
16	50	19	31	1707548	120	21918	13348	13264
16	50	22	28	1787221	104	25131	12289	12255
16	50	25	25	1673430	91	21037	12034	11945
16	50	29	21	1830777	78	20259	10602	10495
16	50	32	18	1641796	71	18146	10997	10826
16	50	34	16	1844265	67	20112	10723	10607
16	50	37	13	1785598	62	17858	8802	8752
16	50	40	10	1876355	57	14914	8452	8395
16	50	43	7	2610637	53	16560	9338	9114
16	50	44	6	2751430	52	13988	8081	7886
16	50	46	4	3975994	51	18215	8299	8155
16	50	47	3	4150293	51	12772	7794	7651
16	50	48	2	4183439	51	15369	8165	8022
16	61	2	59	3406263	1143	103154	43019	43008
16	61	5	56	2655779	457	66112	27410	27399
16	61	9	52	2171871	253	38737	20819	20715
16	61	13	48	2036762	175	32764	16590	16531
16	61	16	45	1820670	142	27698	15242	15179
16	61	20	41	1783480	114	27345	14084	14009
16	61	24	37	1689622	95	21805	11651	11592
16	61	27	34	1572276	84	19899	10951	10780
16	61	31	30	1566295	73	19231	10933	10854
16	61	35	26	1500232	65	18748	9446	9346
16	61	38	23	1515549	60	15419	9194	9090
16	61	41	20	1427643	55	16147	9476	9296
16	61	45	16	1527134	50	19135	8898	8733
16	61	49	12	1605443	46	15542	8770	8626
16	61	52	9	1858955	44	16153	7524	7486
16	61	54	7	2110586	42	16600	8565	8428
16	61	56	5	2804623	41	12283	7311	7195
16	61	57	4	3220481	41	12032	7057	6950
16	61	58	3	3390473	41	11662	6885	6826
16	61	59	2	3383290	41	11423	6767	6729
16	74	2	72	3336876	1142	103805	42291	42270
16	74	7	67	2401535	327	50029	23593	23582
16	74	11	63	2014431	207	38607	17949	17855

continued on next page

Table 6.14. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
16	74	16	58	1714904	142	23467	14512	14423
16	74	20	54	1647755	114	27070	13811	13736
16	74	24	50	1501824	95	22471	11632	11567
16	74	29	45	1487522	78	21046	11350	11243
16	74	33	41	1559994	69	20952	10081	10049
16	74	38	36	1328676	60	16421	10059	9975
16	74	42	32	1294318	54	14559	8703	8640
16	74	47	27	1279195	48	12715	8305	8110
16	74	50	24	1228524	45	14028	7571	7327
16	74	55	19	1280122	41	12802	7248	7078
16	74	59	15	1229569	38	11813	6698	6566
16	74	63	11	1370257	36	11484	6504	6397
16	74	66	8	1574113	34	10915	6399	6232
16	74	68	6	2098259	33	16967	6983	6720
16	74	70	4	2665735	33	13181	6309	6165
16	74	71	3	2739037	33	10517	5919	5802
18	4	1	3	4622093	570	72966	29790	29763
18	4	2	2	8302477	314	54421	22866	22788
18	4	3	1	21830991	275	52980	21446	21419
18	5	1	4	3707512	570	72966	29790	29763
18	5	2	3	3632159	293	51088	21941	21863
18	5	3	2	5881079	204	38526	18657	18630
18	5	4	1	13954525	182	38983	17318	17240
18	6	1	5	3307593	570	72966	29790	29763
18	6	2	4	3198184	290	52796	21477	21399
18	6	3	3	3239610	195	44192	18102	18075
18	6	4	2	4548957	150	31893	16515	16472
18	6	5	1	10843589	139	40514	15456	15378
18	7	1	6	3294683	570	72966	29790	29763
18	7	2	5	2805684	289	50132	22121	22094
18	7	3	4	2585395	193	46445	18787	18760
18	7	4	3	2687127	145	31029	16153	16082
18	7	5	2	3683459	119	28146	14697	14670
18	7	6	1	8402643	110	31679	12856	12692
18	9	1	8	3002093	570	72966	29790	29763
18	9	2	7	2478270	287	47775	22259	22181
18	9	3	6	2252843	191	39315	17125	17082
18	9	4	5	2275782	144	36707	15021	14994
18	9	5	4	2095978	115	31195	11948	11905
18	9	6	3	2054571	96	21943	12993	12922
18	9	7	2	3052093	84	28081	11311	11240
18	9	8	1	6015511	79	21583	11164	11071
18	11	1	10	2925048	570	72966	29790	29763
18	11	2	9	2391805	286	47843	21175	21097
18	11	3	8	2031734	191	36649	18128	18101
18	11	4	7	2051116	143	35551	15875	15804
18	11	5	6	1988813	114	27543	13619	13455
18	11	6	5	1848633	95	27382	12787	12716

continued on next page

Table 6.14. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
18	11	7	4	1910252	82	19197	11311	11268
18	11	8	3	1797397	72	22197	10198	10155
18	11	9	2	2288098	65	18059	9680	9653
18	11	10	1	4714305	62	16033	9512	9348
18	13	1	12	2918572	570	72966	29790	29763
18	13	2	11	2366567	286	49328	21484	21406
18	13	3	10	1951855	190	34928	17221	17178
18	13	4	9	1920328	143	25871	15004	14926
18	13	5	8	1841324	114	26412	14005	13934
18	13	6	7	1762570	95	25415	11947	11876
18	13	7	6	1527652	81	18489	11525	11432
18	13	8	5	1708842	71	21830	11883	11790
18	13	9	4	1525684	63	16450	9380	9201
18	13	10	3	1760958	57	22943	8903	8724
18	13	11	2	1884088	52	13341	7958	7776
18	13	12	1	3738735	50	15586	8604	8422
18	16	1	15	2777771	570	72966	29790	29763
18	16	2	14	2303418	286	51733	21781	21754
18	16	3	13	1970752	190	40744	17430	17387
18	16	4	12	1707050	143	24375	15077	15050
18	16	5	11	1790873	114	28575	14669	14626
18	16	6	10	1740203	95	24507	10773	10730
18	16	7	9	1439625	81	17409	10312	10241
18	16	8	8	1679801	71	17193	10285	10214
18	16	9	7	1372134	63	15318	9209	9138
18	16	10	6	1387484	57	14283	8355	8284
18	16	11	5	1450956	52	15358	8857	8814
18	16	12	4	1559009	47	18887	8843	8639
18	16	13	3	1469348	44	18440	6872	6708
18	16	14	2	1827829	41	14593	7162	6983
18	16	15	1	3172849	40	19985	7382	7218
18	19	1	18	2723131	570	72966	29790	29763
18	19	2	17	2085569	285	45139	21990	21912
18	19	3	16	1944007	190	37281	17249	17171
18	19	4	15	1707894	142	27078	14262	14191
18	19	5	14	1675897	114	26343	13775	13697
18	19	6	13	1523025	95	22063	11036	10958
18	19	7	12	1464939	81	21219	10881	10717
18	19	9	10	1468139	63	19207	9892	9821
18	19	10	9	1282650	57	15013	8238	8195
18	19	11	8	1434121	52	18730	8799	8772
18	19	12	7	1286892	47	12241	7601	7422
18	19	13	6	1335495	44	14036	8037	7959
18	19	14	5	1180920	40	12202	7617	7458
18	19	15	4	1468035	38	15174	7267	7196
18	19	16	3	1437407	36	14765	7122	7095
18	19	17	2	1579314	34	12692	6655	6612
18	19	18	1	2426435	33	9682	5760	5717

continued on next page

Table 6.14. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
20	2	1	1	2149218	142	30260	14794	14767
20	3	1	2	1841768	142	30260	14794	14767
20	3	2	1	1770564	74	17152	10452	10425
20	4	1	3	1732852	142	30260	14794	14767
20	4	2	2	1534535	72	20434	10207	10180
20	4	3	1	1600108	49	16149	8357	8330
20	5	1	4	1744298	142	30260	14794	14767
20	5	2	3	1433614	71	17512	9962	9864
20	5	3	2	1255636	48	13795	7832	7805
20	5	4	1	1208917	36	12336	6189	6025

Table 6.15. Costs of block 2GYF for Toba

Configuration				Copied	Invoc	W.b. insert	W.b. add	R.s. proc
16	22	1	21	12591818	2403	84765	82857	82856
16	22	2	20	11116641	1218	52894	51031	51029
16	22	3	19	10960736	813	38693	36930	36924
16	22	5	17	11093420	489	22824	20983	20977
16	22	6	16	12005563	410	23686	21864	21855
16	22	7	15	11443916	353	18766	16901	16894
16	22	9	13	12375984	278	19643	17699	17698
16	22	10	12	13704951	254	13923	12045	12032
16	22	11	11	15039560	233	13125	11207	11197
16	22	13	9	17733236	207	11234	9281	9274
16	22	14	8	19109783	197	8936	7052	7048
16	22	15	7	20250622	191	13478	11508	11504
16	22	16	6	20712410	185	7561	5656	5653
16	22	18	4	22789072	178	9251	7320	7317
16	22	19	3	23199960	178	10547	8637	8624
16	22	20	2	23689464	180	12352	10444	10433
16	27	1	26	11340734	2403	84765	82857	82856
16	27	2	25	9738692	1213	52521	50756	50754
16	27	4	23	8670448	608	31075	29314	29313
16	27	6	21	8282062	405	16000	14204	14202
16	27	7	20	7955917	347	21828	19985	19979
16	27	9	18	8175976	270	12257	10391	10380
16	27	11	16	8601306	222	12580	10687	10683
16	27	12	15	8354492	204	8229	6304	6294
16	27	14	13	9035349	177	7155	5240	5231
16	27	15	12	9575017	165	11475	9516	9509
16	27	17	10	10674668	149	9900	7944	7938
16	27	18	9	11919193	144	12640	10689	10688
16	27	20	7	13105182	134	10718	8767	8756
16	27	22	5	14364131	129	8743	6761	6759
16	27	23	4	15154400	127	9660	7705	7694
16	27	24	3	16033469	128	11758	9807	9805
16	33	1	32	10564261	2403	84765	82857	82856
16	33	3	30	9028210	807	40955	39165	39163
16	33	5	28	8156104	484	22948	21088	21081
16	33	7	26	7198084	346	18376	16571	16564
16	33	9	24	7049347	269	17206	15315	15306
16	33	11	22	6959029	221	14161	12317	12311
16	33	13	20	6394643	187	10545	8642	8640
16	33	15	18	6378196	162	11904	9847	9838
16	33	17	16	7149388	143	8680	6715	6707
16	33	19	14	7131125	129	8304	6348	6344
16	33	21	12	8107715	117	9343	7381	7365
16	33	22	11	7751164	113	5162	3199	3189
16	33	24	9	8671491	105	9494	7524	7512
16	33	26	7	9642086	100	9077	7112	7105
16	33	28	5	10675585	97	7554	5590	5586
16	33	29	4	11675117	96	6032	4030	4026

continued on next page

Table 6.15. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
16	33	30	3	11878209	96	5096	3140	3136
16	33	31	2	11708491	96	5699	3737	3660
16	40	1	39	10432554	2403	84765	82857	82856
16	40	4	36	7559615	604	30687	28889	28887
16	40	6	34	6984337	403	22989	21241	21240
16	40	8	32	6949810	302	17666	15849	15840
16	40	11	29	6959041	221	15441	13595	13589
16	40	13	27	6501847	186	13414	11466	11462
16	40	16	24	5935176	151	6886	4990	4986
16	40	18	22	5919640	135	15694	13748	13747
16	40	20	20	5765725	121	11005	9037	9009
16	**40**	**23**	**17**	**5685343**	**106**	**6305**	**4327**	**4318**
16	40	25	15	5799969	98	7608	5646	5636
16	40	27	13	6444446	91	8725	6721	6717
16	40	30	10	6493485	83	7241	5241	5228
16	40	32	8	7236589	79	5177	3231	3154
16	40	34	6	7213034	76	6507	4546	4540
16	40	36	4	8458649	74	5860	3884	3880
16	40	37	3	8586463	74	5044	3052	2975
16	40	38	2	8609610	74	4895	2910	2833
16	48	1	47	9781248	2403	84765	82857	82856
16	48	4	44	7494067	603	27394	25681	25675
16	48	7	41	6537490	344	14724	12877	12867
16	48	10	38	6175641	241	16706	14780	14767
16	48	13	35	5750707	185	13007	11076	10999
16	48	16	32	5570895	151	9764	7842	7835
16	48	19	29	5655561	127	8151	6211	6207
16	48	22	26	5414857	110	7225	5205	5194
16	48	24	24	5105442	101	7644	5660	5657
16	48	27	21	5149220	90	6613	4631	4627
16	48	30	18	4733472	81	9013	7033	7024
16	48	33	15	5189342	74	4071	2092	2082
16	48	36	12	5387710	68	6844	4847	4834
16	48	38	10	5227761	65	5964	3974	3961
16	48	41	7	6057491	61	4382	2402	2394
16	48	43	5	6225060	60	4168	2173	2166
16	48	44	4	6661744	59	5098	3095	3076
16	48	45	3	6712723	59	4663	2674	2659
16	48	46	2	6836726	59	3909	1921	1902
16	59	2	57	8493361	1206	51896	50043	50042
16	59	5	54	7252753	482	19739	18013	18006
16	59	9	50	6069469	268	17780	15895	15889
16	59	12	47	5965181	201	12681	10779	10772
16	59	16	43	5497062	151	11574	9686	9680
16	59	19	40	5140408	127	8123	6173	6167
16	59	23	36	4657626	104	9120	7175	7166
16	59	27	32	4552709	89	7107	5144	5136
16	59	30	29	4490860	80	10620	8629	8620

continued on next page

Table 6.15. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
16	59	34	25	4261642	71	4628	2640	2625
16	59	37	22	4384321	65	6613	4625	4606
16	59	40	19	4555863	60	5078	3110	3102
16	59	44	15	4223257	55	4810	2834	2812
16	59	47	12	4443330	52	3943	1936	1932
16	59	50	9	4449340	49	4127	2161	2143
16	59	52	7	4644286	48	5661	3647	3639
16	59	54	5	4820466	47	7077	5098	5087
16	59	55	4	5183566	46	4223	2239	2231
16	59	56	3	5565407	47	3330	1364	1363
16	59	57	2	5361499	46	3953	1968	1959
16	71	2	69	8321666	1205	52031	50199	50198
16	71	6	65	6581044	401	22261	20442	20436
16	71	11	60	5715827	219	12724	10822	10816
16	71	15	56	5030439	160	8188	6249	6246
16	71	19	52	4636679	127	5599	3698	3696
16	71	23	48	4715487	104	12675	10737	10667
16	71	28	43	4518681	86	13611	11654	11644
16	71	32	39	4303274	75	9619	7641	7634
16	71	36	35	3961441	67	7770	5790	5782
16	71	40	31	3866383	60	8603	6606	6599
16	71	45	26	4030186	54	6916	4898	4891
16	71	48	23	3527695	50	3762	1775	1743
16	71	53	18	3677191	46	12256	10278	10201
16	71	57	14	3484212	42	3855	1884	1829
16	71	60	11	3753255	40	5338	3377	3344
16	71	63	8	3728420	39	3236	1259	1245
16	71	65	6	3804178	38	3608	1659	1592
16	71	67	4	4002838	37	5144	3165	3098
16	71	68	3	4078398	37	5542	3539	3472
16	87	3	84	7601796	802	32741	31072	31066
16	87	8	79	5905616	300	21196	19360	19349
16	87	13	74	5164095	185	14423	12515	12506
16	87	18	69	4861538	134	6553	4580	4579
16	87	23	64	4490055	104	8656	6727	6657
16	87	29	58	4311287	83	7407	5451	5441
16	87	34	53	4048794	71	7875	5885	5883
16	87	39	48	3896050	61	6980	4997	4983
16	87	44	43	3647852	54	4649	2673	2646
16	87	50	37	3586602	48	7705	5719	5704
16	87	55	32	3354195	44	5254	3243	3236
16	87	59	28	3195691	41	4562	2580	2578
16	87	64	23	3099474	38	6441	4460	4459
16	87	70	17	2824376	34	5383	3394	3361
16	87	74	13	3098261	32	6192	4226	4168
16	87	77	10	3199763	31	3879	1904	1844
16	87	80	7	2653134	30	3252	1289	1101
16	87	82	5	3259716	30	4265	2300	2273

continued on next page

Table 6.15. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
16	87	83	4	3433080	30	8004	6040	6027
16	87	84	3	3462159	30	7215	5246	5176
16	106	3	103	7486683	802	36104	34255	34253
16	106	10	96	5474652	240	12390	10496	10483
16	106	16	90	4681580	150	10516	8571	8558
16	106	22	84	4522876	109	4773	2843	2830
16	106	29	77	4108770	83	5859	3894	3884
16	106	35	71	3988817	68	5116	3122	3114
16	106	41	65	3840313	59	7363	5373	5372
16	106	48	58	3518283	50	4034	2056	2052
16	106	54	52	3239133	44	4360	2370	2307
16	106	60	46	3064149	40	5147	3136	3124
16	106	67	39	3063955	36	5219	3240	3236
16	106	72	34	2427058	33	3894	1897	1868
16	106	78	28	2911797	31	3394	1435	1433
16	106	85	21	2766662	28	3532	1552	1507
16	106	90	16	2555843	27	6111	4120	4109
16	106	94	12	2662781	26	3060	1096	1090
16	106	98	8	2242414	24	2941	983	896
16	106	100	6	2536890	24	4437	2465	2405
16	106	101	5	2607726	24	5781	3797	3764
16	106	102	4	2990573	24	3689	1707	1675
18	6	1	5	8785525	595	25811	24063	24061
18	6	2	4	11367580	318	14194	12402	12400
18	6	3	3	13779210	222	12634	10725	10723
18	6	4	2	18418008	183	9538	7564	7558
18	6	5	1	22253449	174	12444	10549	10543
18	7	1	6	8362063	595	25811	24063	24061
18	7	2	5	8014919	309	16231	14419	14417
18	7	3	4	8441619	208	15082	13181	13172
18	7	4	3	10108059	161	8959	7065	7061
18	7	5	2	13141873	138	11645	9686	9677
18	7	6	1	16152005	132	6675	4735	4726
18	9	1	8	8003237	595	25811	24063	24061
18	9	2	7	7165137	305	15189	13311	13309
18	9	3	6	6765654	203	11410	9474	9470
18	9	4	5	6496695	153	8762	6811	6802
18	9	5	4	6974175	124	6820	4861	4859
18	9	6	3	6966905	104	9226	7233	7218
18	9	7	2	8488453	93	5630	3684	3673
18	9	8	1	10660231	90	8639	6663	6648
18	10	1	9	7791229	595	25811	24063	24061
18	10	2	8	7023681	304	12618	10809	10807
18	10	3	7	6711466	203	9147	7263	7261
18	10	4	6	6009993	152	13521	11589	11580
18	10	5	5	5809424	122	11904	9940	9931
18	10	6	4	5972714	102	6813	4834	4828
18	10	7	3	6487686	89	6147	4181	4166

continued on next page

Table 6.15. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
18	10	8	2	7063723	80	5235	3263	3259
18	10	9	1	9183237	78	5572	3583	3568
18	12	1	11	7335812	595	25811	24063	24061
18	12	2	10	6504339	302	15221	13396	13394
18	12	3	9	6234046	201	9713	7808	7804
18	12	4	8	5638418	151	7403	5475	5466
18	12	5	7	5254524	121	5352	3425	3423
18	12	6	6	5629985	101	6541	4558	4552
18	12	7	5	5212972	87	8797	6796	6794
18	12	8	4	5259212	76	7752	5804	5795
18	12	9	3	5178472	69	7888	5868	5866
18	12	10	2	5899102	64	4857	2846	2844
18	12	11	1	7172210	62	10408	8430	8428
18	15	1	14	7062422	595	25811	24063	24061
18	15	2	13	6327451	301	13313	11456	11454
18	15	3	12	5585677	200	8876	6933	6924
18	15	4	11	5106319	150	8033	6098	6089
18	15	5	10	4533484	120	7918	5965	5956
18	15	6	9	4952296	100	8889	6907	6901
18	15	7	8	4641514	86	5316	3370	3361
18	15	8	7	4341541	75	6280	4279	4273
18	15	9	6	4157497	67	5486	3495	3489
18	15	10	5	4087995	60	4826	2844	2821
18	15	11	4	4298552	55	4197	2221	2206
18	15	12	3	4599576	51	3880	1899	1876
18	15	13	2	4244387	48	4220	2250	2244
18	15	14	1	5396474	47	7631	5643	5639
18	18	1	17	7057831	595	25811	24063	24061
18	18	2	16	5996465	300	14665	12802	12800
18	18	3	15	5543018	200	13017	11102	11100
18	18	4	14	4987962	150	9130	7173	7171
18	18	5	13	4559158	120	6668	4717	4715
18	18	6	12	4487482	100	11268	9299	9297
18	18	7	11	4359409	85	7118	5131	5122
18	18	8	10	4271401	75	5209	3229	3225
18	18	9	9	4046367	67	4940	2942	2940
18	18	10	8	3848551	60	5901	3907	3903
18	18	11	7	3771743	55	4282	2283	2281
18	18	12	6	3767137	50	4285	2313	2304
18	18	13	5	3747181	46	3605	1630	1612
18	18	14	4	3693446	43	3588	1621	1610
18	18	15	3	3709924	41	6677	4707	4698
18	18	16	2	3625418	38	2983	1026	994
18	18	17	1	4230538	37	4930	2917	2875
18	22	1	21	6815179	595	25811	24063	24061
18	22	2	20	5829377	299	17519	15647	15638
18	22	3	19	5299992	199	15791	13856	13852
18	22	5	17	4616395	119	6684	4761	4746

continued on next page

Table 6.15. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
18	22	6	16	4413488	100	7964	6021	6019
18	22	7	15	4283250	85	7128	5138	5123
18	22	9	13	4037535	66	6405	4426	4417
18	22	10	12	3870386	60	5218	3248	3246
18	22	11	11	3743816	54	5668	3649	3640
18	22	13	9	3407540	46	4013	2041	2037
18	22	14	8	3323226	42	5476	3486	3454
18	22	15	7	3099585	40	3600	1628	1624
18	22	16	6	3173735	37	4453	2470	2433
18	22	18	4	2998176	33	2952	982	945
18	22	19	3	2860781	32	4048	2088	2079
18	22	20	2	2966343	30	5775	3823	3777
18	22	21	1	3445157	30	5878	3906	3891
18	27	1	26	6737603	595	25811	24063	24061
18	27	2	25	5998228	299	19823	17991	17989
18	27	4	23	4846012	149	9242	7342	7338
18	27	6	21	4431588	99	7309	5342	5327
18	27	7	20	4084153	85	4811	2832	2826
18	27	9	18	3894656	66	4177	2198	2183
18	27	11	16	3709213	54	4640	2643	2628
18	27	12	15	3487484	49	4503	2510	2481
18	27	14	13	3262968	42	4274	2290	2258
18	27	15	12	2843891	40	4229	2242	2240
18	27	17	10	2912449	35	5951	3982	3971
18	27	18	9	2953653	33	4490	2512	2503
18	27	20	7	2694418	30	4957	2994	2985
18	27	22	5	2615701	27	3955	1976	1958
18	27	23	4	2112657	26	2966	1014	1005
18	27	24	3	2513627	25	2989	1011	988
18	27	25	2	2641062	24	3548	1584	1542
18	27	26	1	2780389	24	3401	1444	1438
20	2	1	1	7144539	148	9442	7539	7527
20	3	1	2	5336057	148	9442	7539	7527
20	3	2	1	5489077	83	4418	2477	2465
20	4	1	3	5138698	148	9442	7539	7527
20	4	2	2	4123868	77	8482	6526	6519
20	4	3	1	4451954	54	7233	5257	5245
20	5	1	4	5029654	148	9442	7539	7527
20	5	2	3	3975144	76	7546	5578	5566
20	5	3	2	3641446	51	3520	1549	1542
20	5	4	1	3583989	39	6719	4756	4715
20	6	1	5	4984286	148	9442	7539	7527
20	6	2	4	4042792	75	4698	2744	2737
20	6	3	3	3592622	50	4367	2394	2370
20	6	4	2	2977875	38	6879	4892	4880
20	6	5	1	2515719	31	3528	1584	1577
20	7	1	6	4880934	148	9442	7539	7527
20	7	2	5	3914284	75	4714	2747	2735

continued on next page

Table 6.15. continued

Configuration				Copied	Invoc	W.b. Insert	W.b. Add	R.s. proc
20	7	3	4	3401516	50	4261	2283	2276
20	7	4	3	3094451	37	3748	1778	1737
20	7	5	2	2650357	30	6596	4645	4621
20	7	6	1	2290351	25	2963	997	923

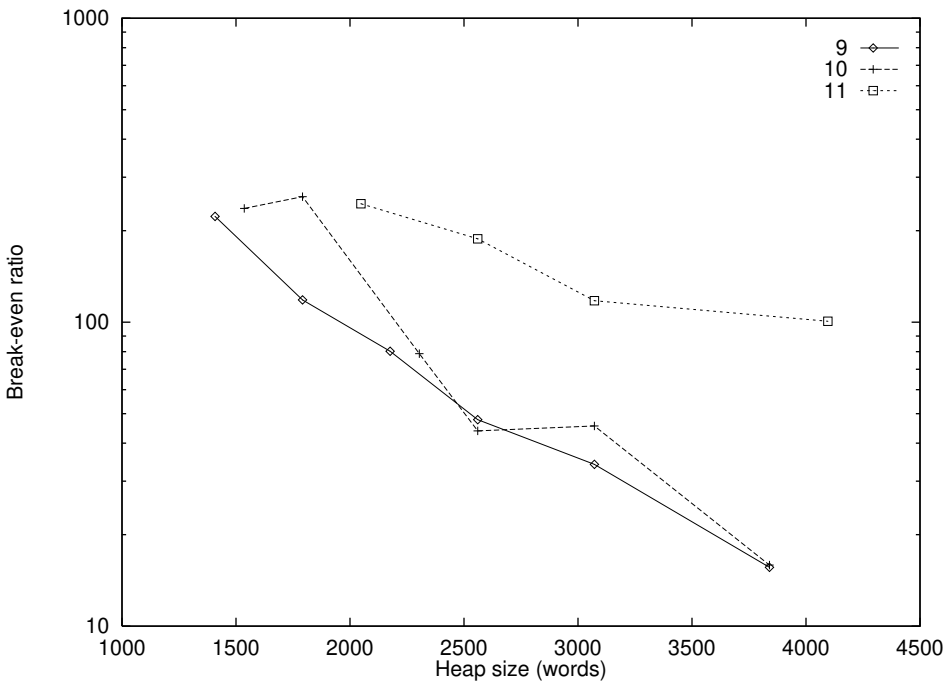


Figure 6.1. Copying and pointer cost tradeoff: Interactive-AllCallsOn.

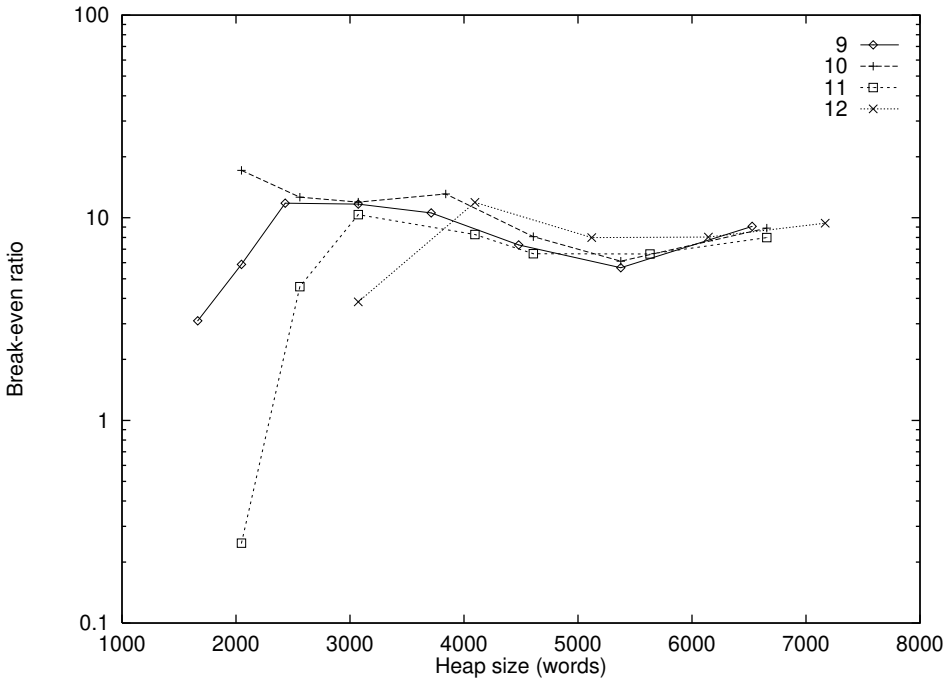


Figure 6.2. Copying and pointer cost tradeoff: Interactive-TextEditing.

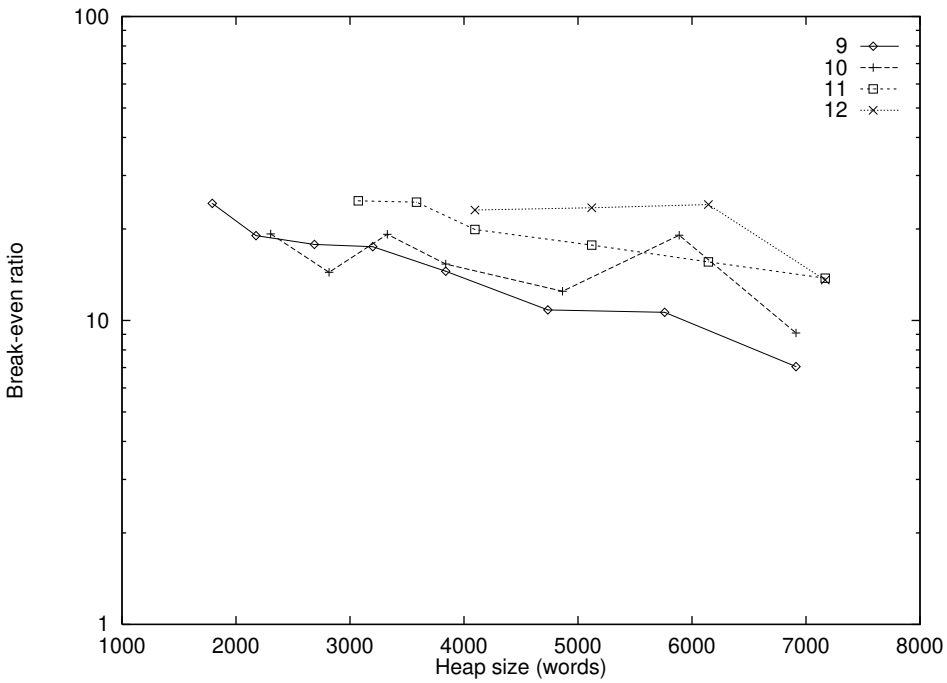


Figure 6.3. Copying and pointer cost tradeoff: StandardNonInteractive.

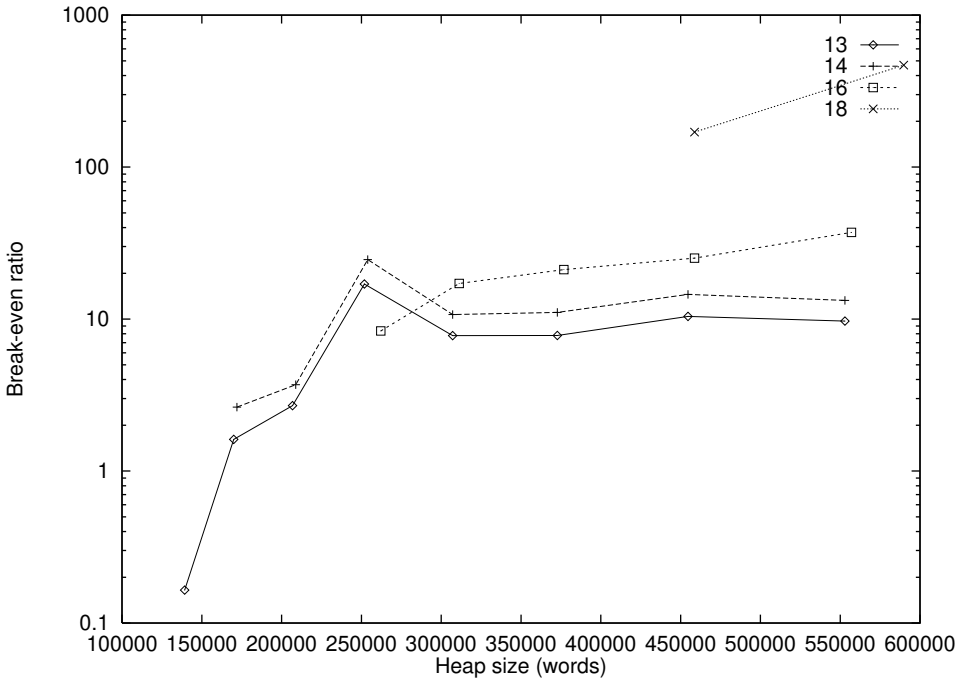


Figure 6.4. Copying and pointer cost tradeoff: HeapSim.

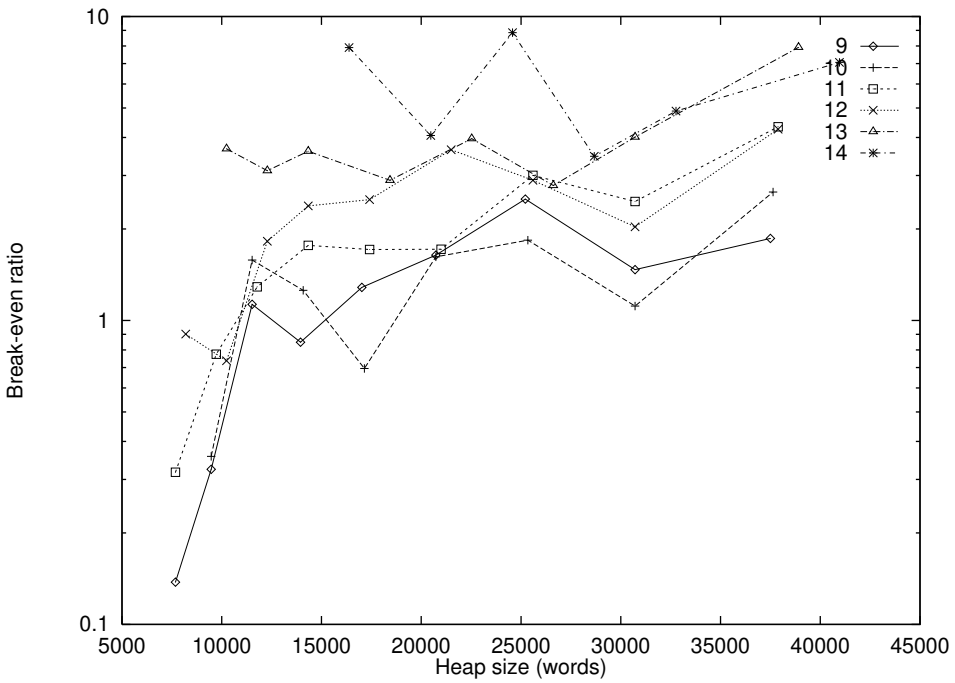


Figure 6.5. Copying and pointer cost tradeoff: Lambda-Fact5.

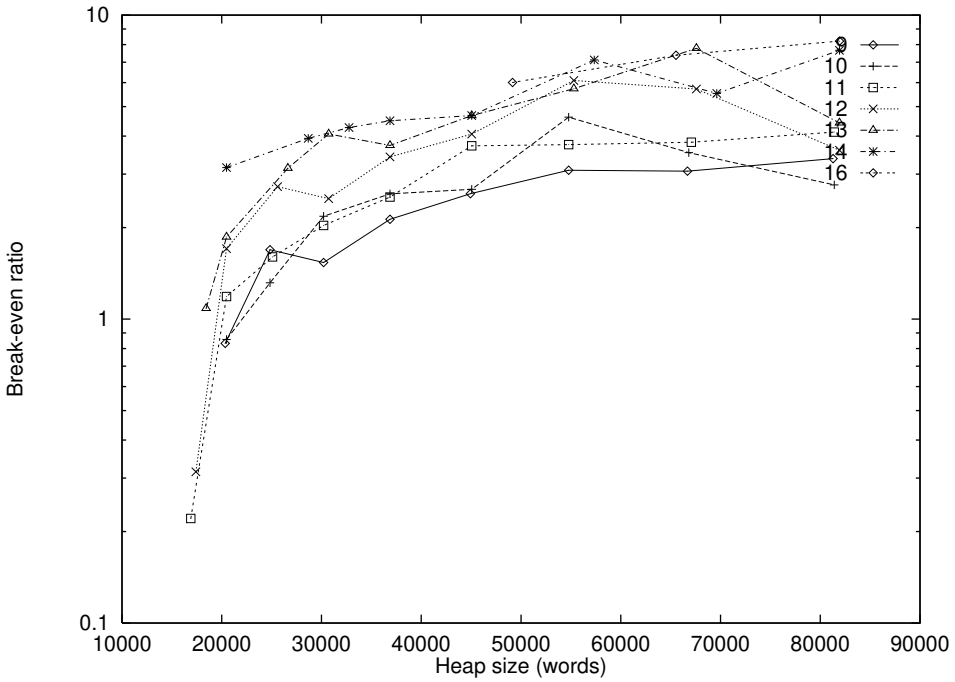


Figure 6.6. Copying and pointer cost tradeoff: Lambda-Fact6.

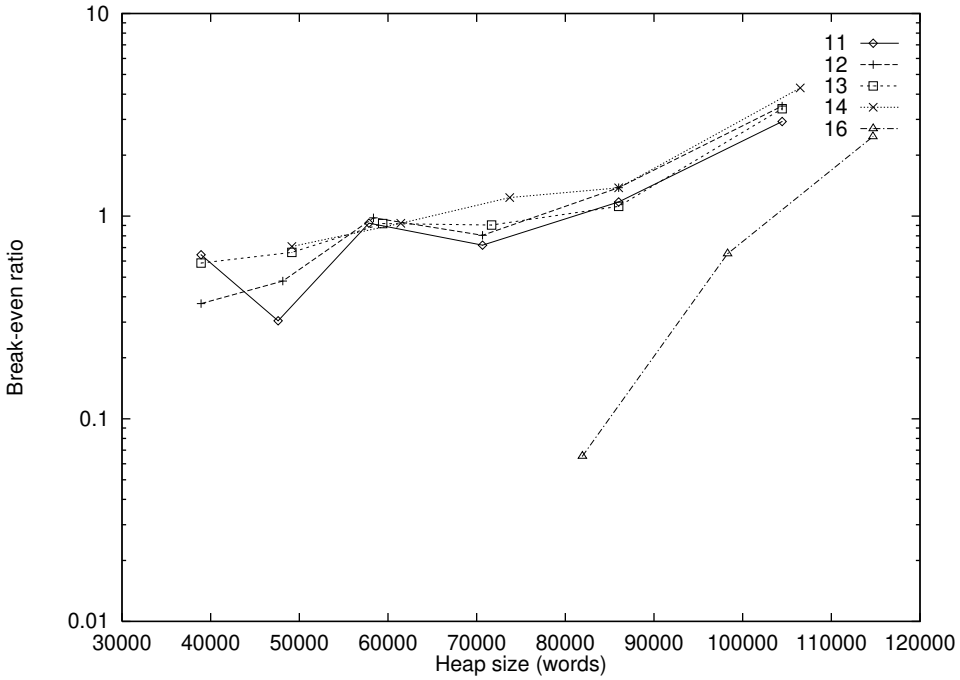


Figure 6.7. Copying and pointer cost tradeoff: Swim.

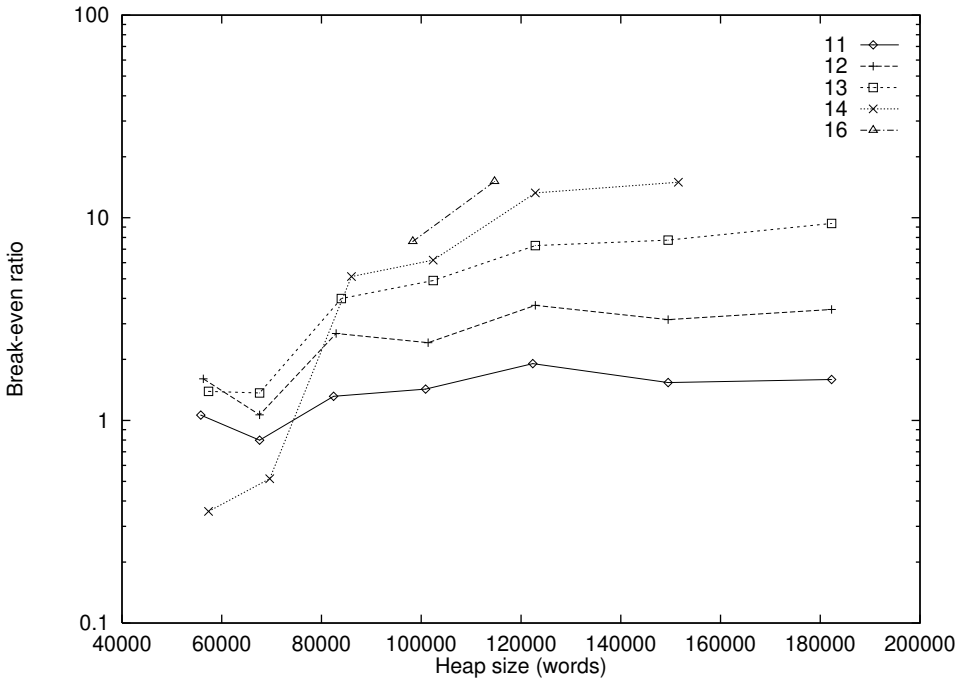


Figure 6.8. Copying and pointer cost tradeoff: Tomcatv.

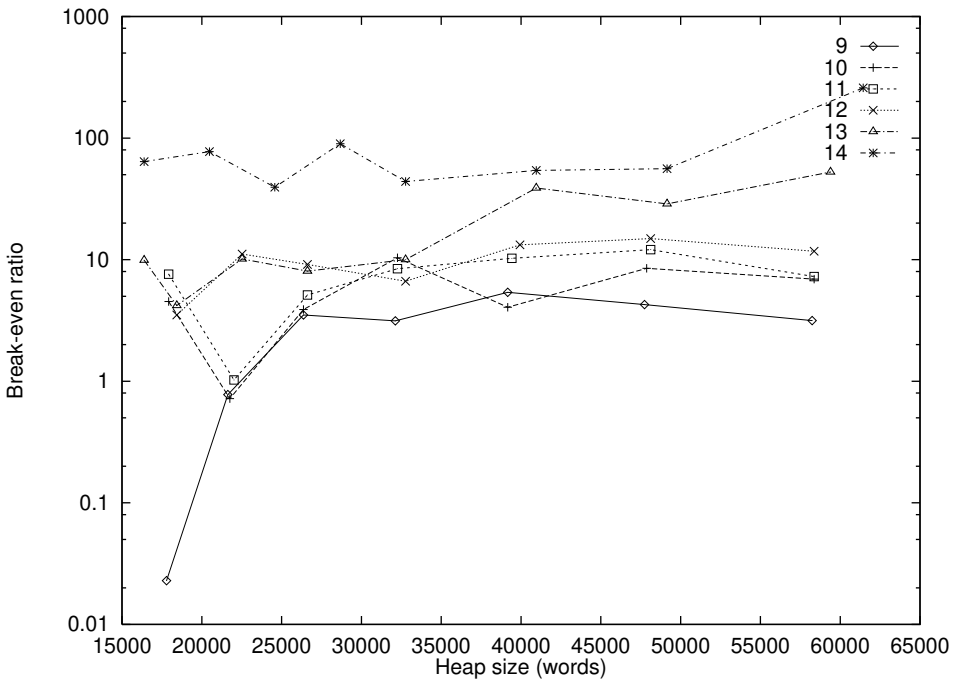


Figure 6.9. Copying and pointer cost tradeoff: Tree-Replace-Binary.

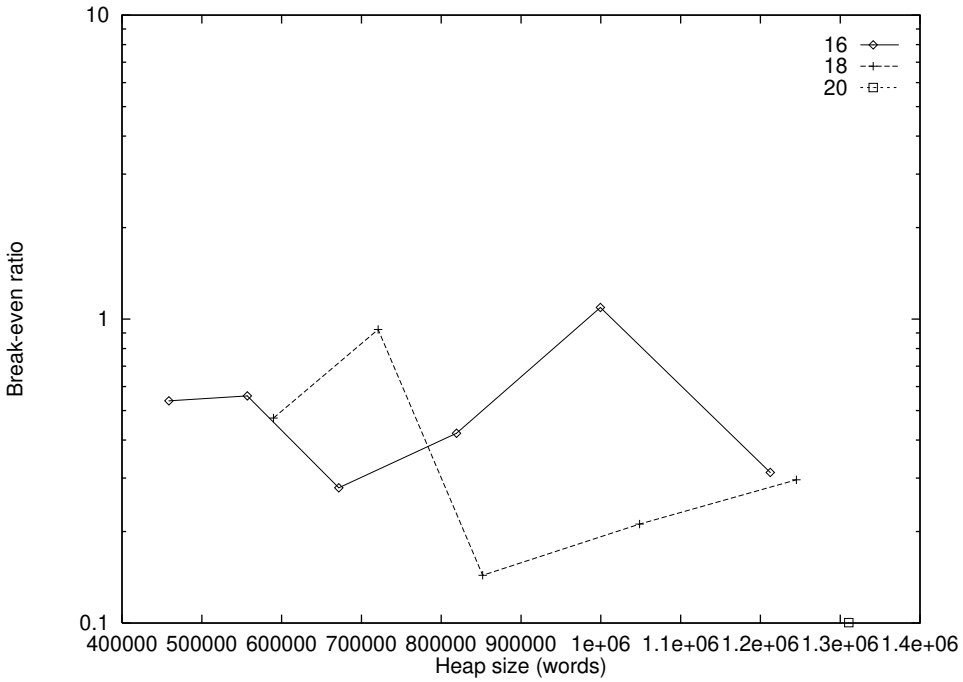


Figure 6.10. Copying and pointer cost tradeoff: Bloat-Bloat.

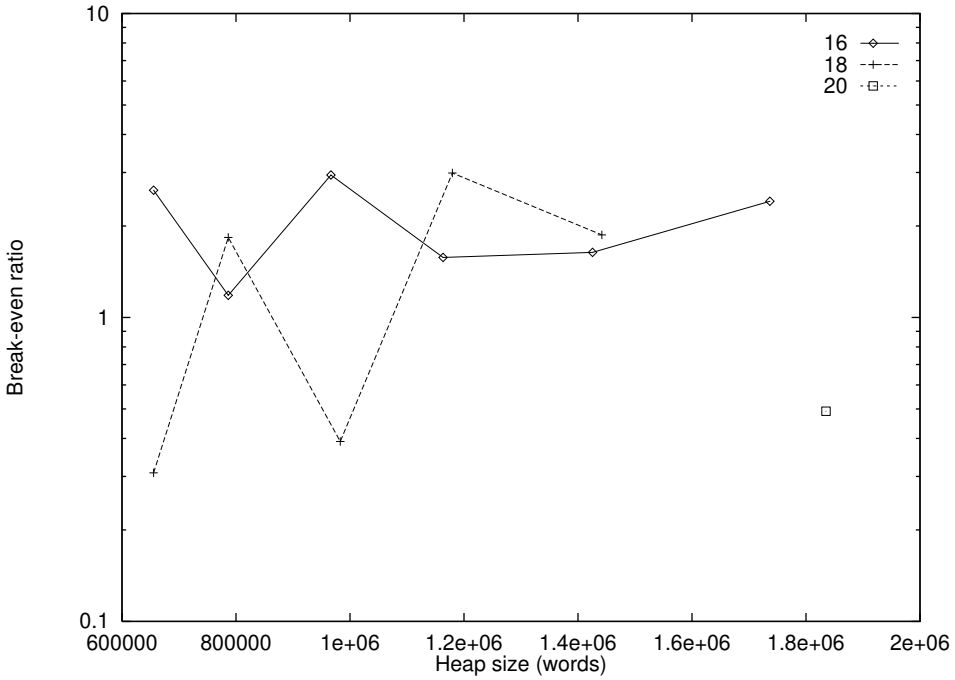
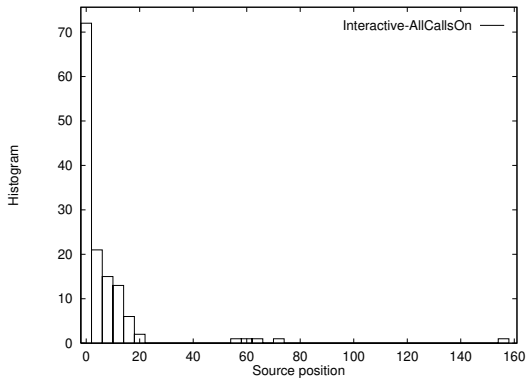
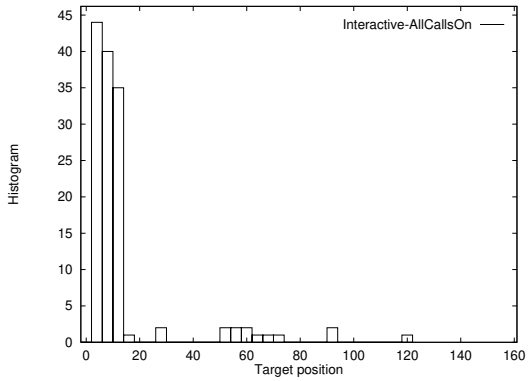


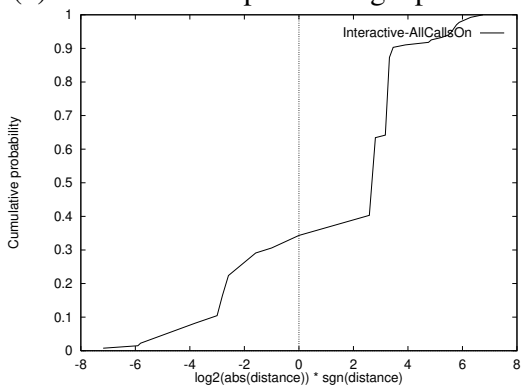
Figure 6.11. Copying and pointer cost tradeoff: Toba.



(a) Distribution of pointer source positions

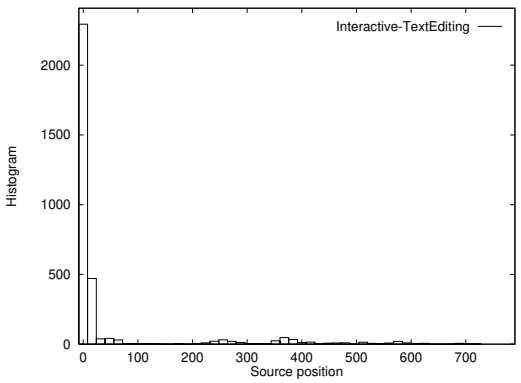


(b) Distribution of pointer target positions

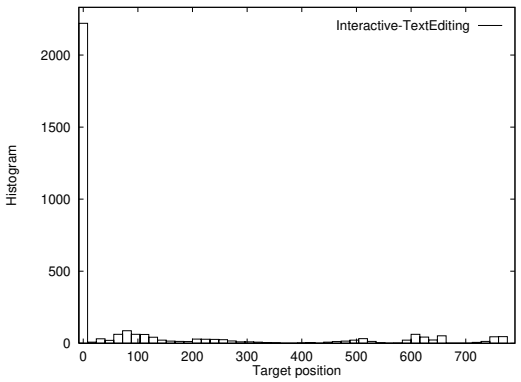


(c) Distribution of pointer distances

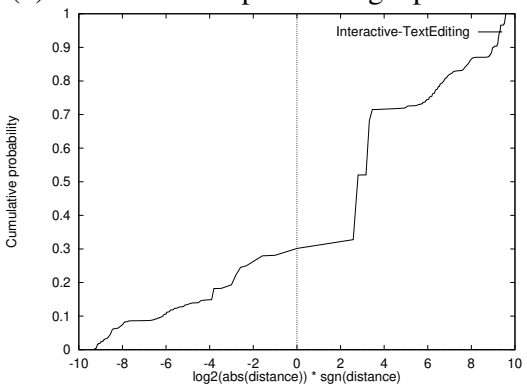
Figure 6.12. Pointer store heap position: Interactive-AllCallsOn.



(a) Distribution of pointer source positions

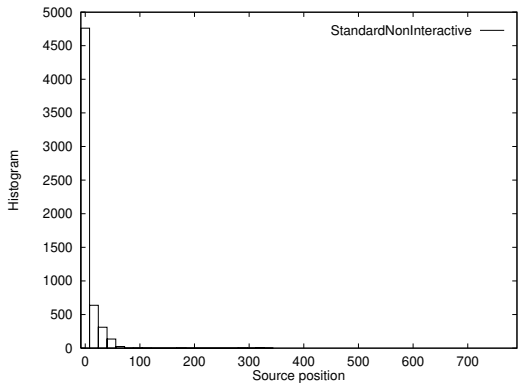


(b) Distribution of pointer target positions

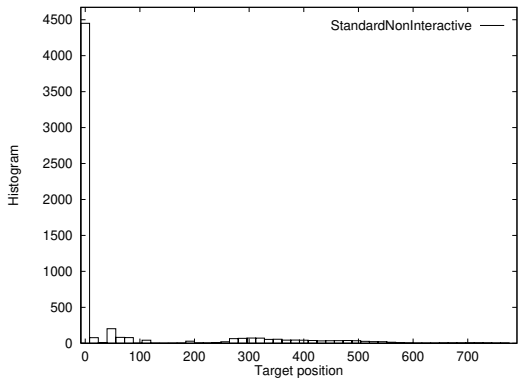


(c) Distribution of pointer distances

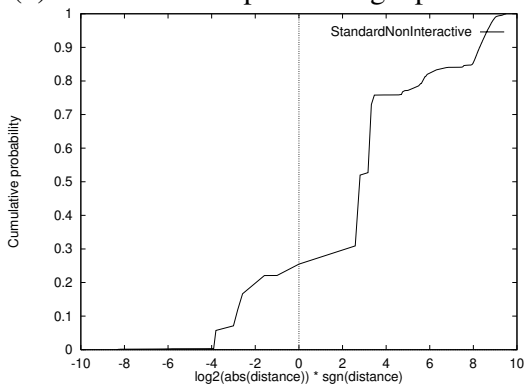
Figure 6.13. Pointer store heap position: Interactive-TextEditing.



(a) Distribution of pointer source positions

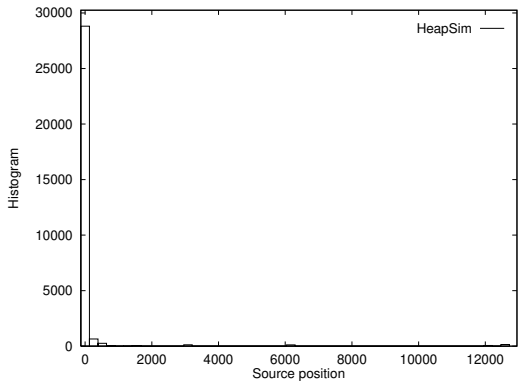


(b) Distribution of pointer target positions

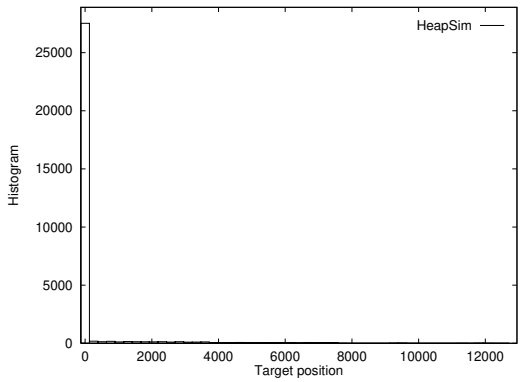


(c) Distribution of pointer distances

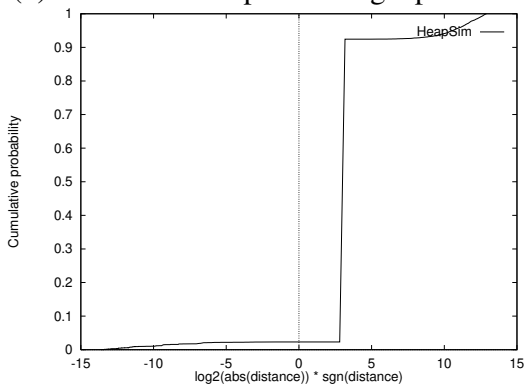
Figure 6.14. Pointer store heap position: StandardNonInteractive.



(a) Distribution of pointer source positions

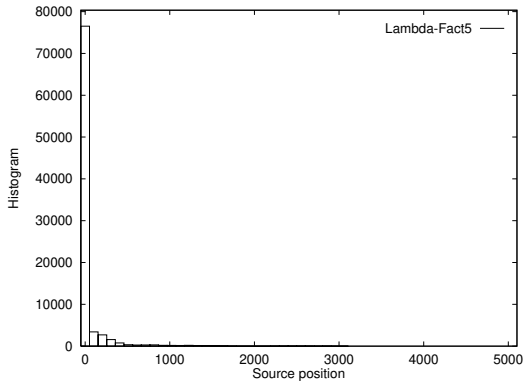


(b) Distribution of pointer target positions

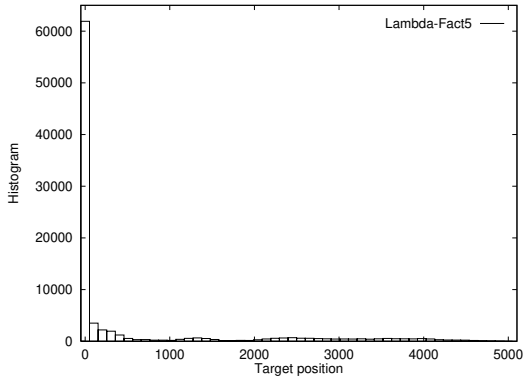


(c) Distribution of pointer distances

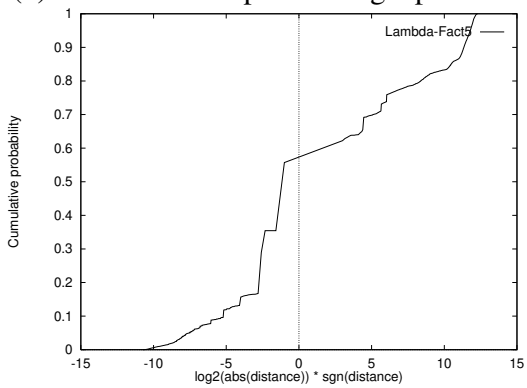
Figure 6.15. Pointer store heap position: HeapSim.



(a) Distribution of pointer source positions

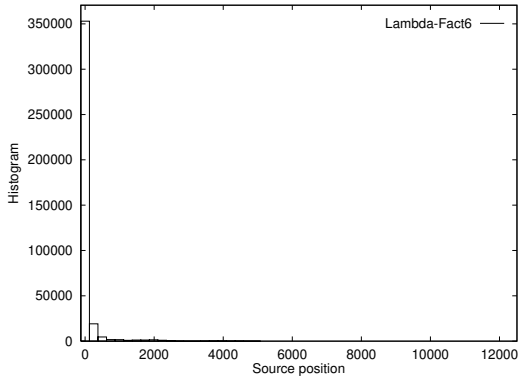


(b) Distribution of pointer target positions

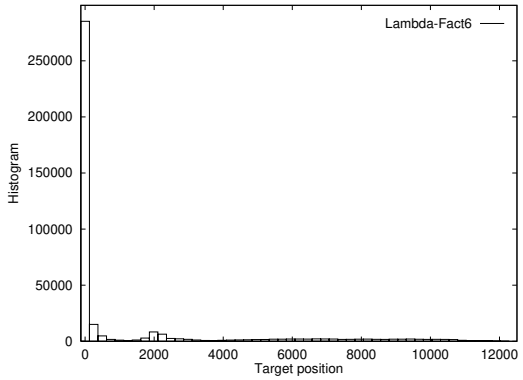


(c) Distribution of pointer distances

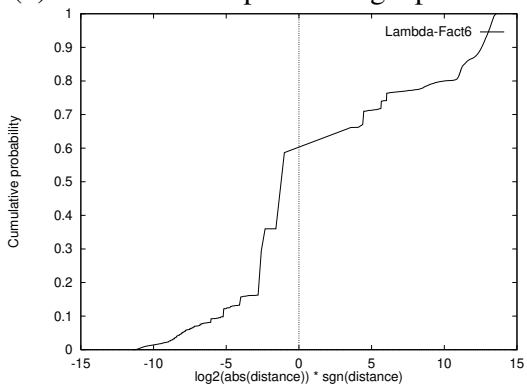
Figure 6.16. Pointer store heap position: Lambda-Fact5.



(a) Distribution of pointer source positions

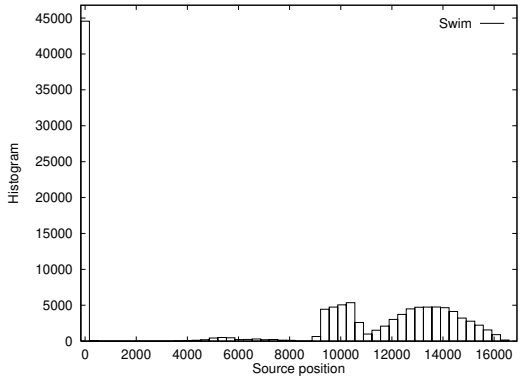


(b) Distribution of pointer target positions

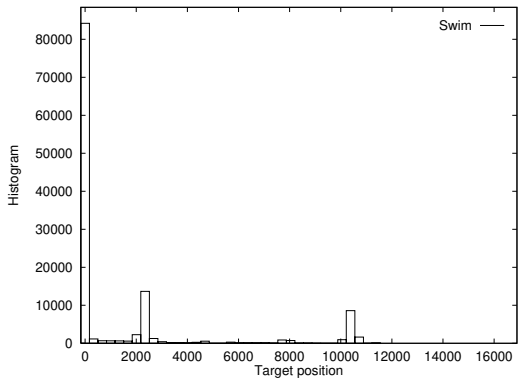


(c) Distribution of pointer distances

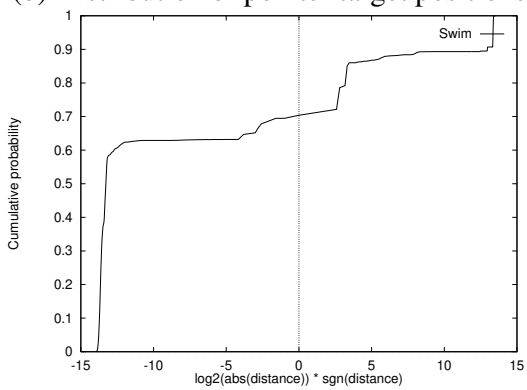
Figure 6.17. Pointer store heap position: Lambda-Fact6.



(a) Distribution of pointer source positions

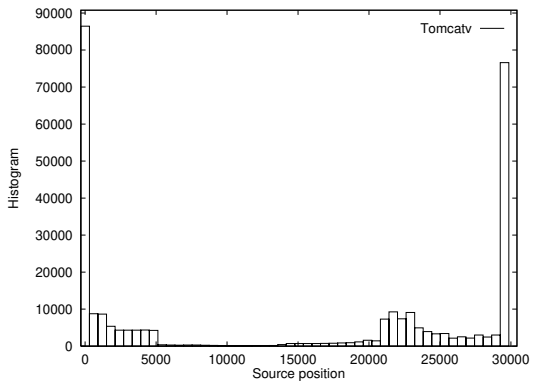


(b) Distribution of pointer target positions

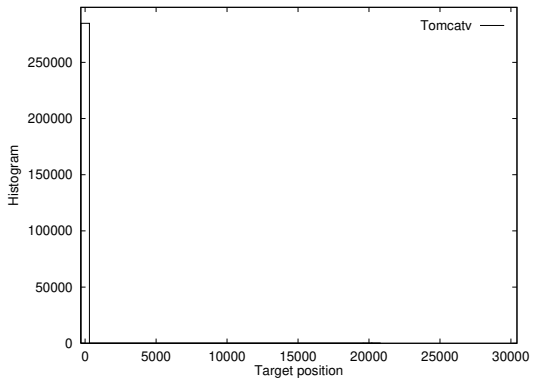


(c) Distribution of pointer distances

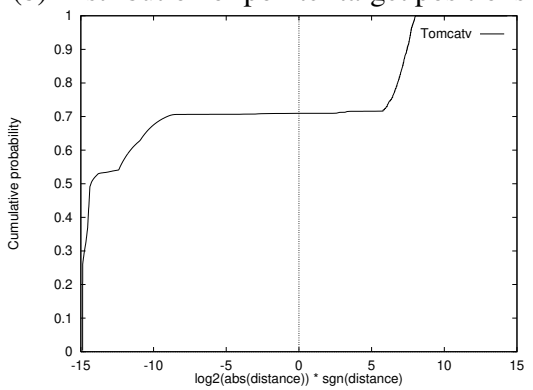
Figure 6.18. Pointer store heap position: Swim.



(a) Distribution of pointer source positions

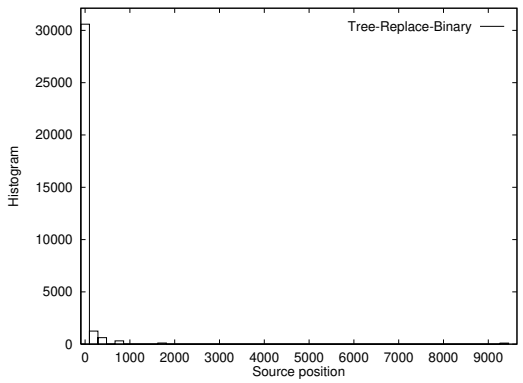


(b) Distribution of pointer target positions

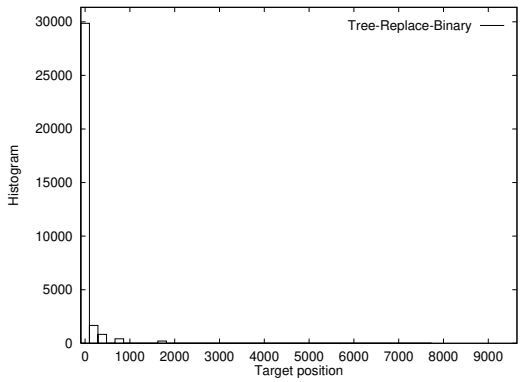


(c) Distribution of pointer distances

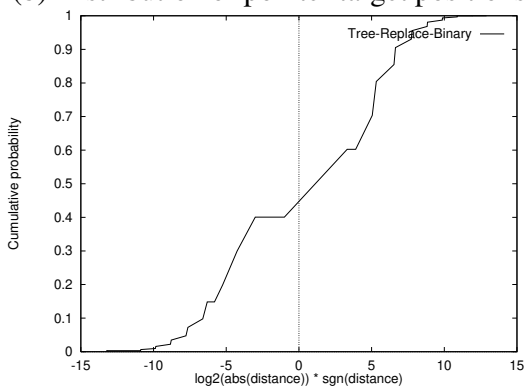
Figure 6.19. Pointer store heap position: Tomcatv.



(a) Distribution of pointer source positions

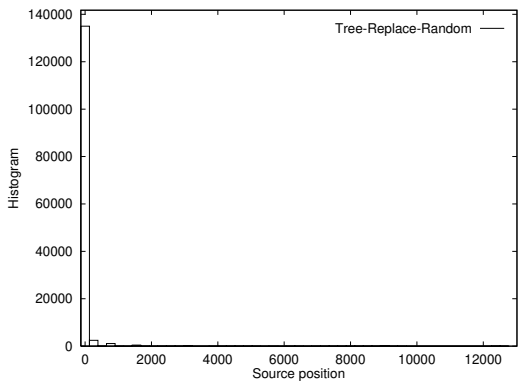


(b) Distribution of pointer target positions

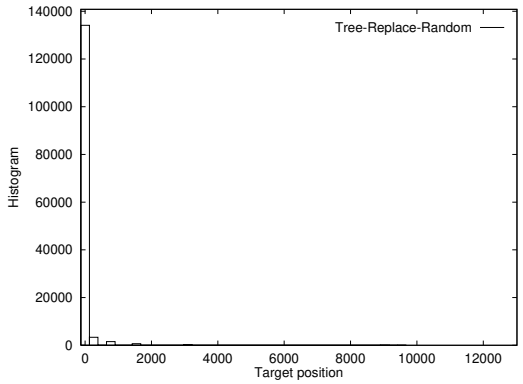


(c) Distribution of pointer distances

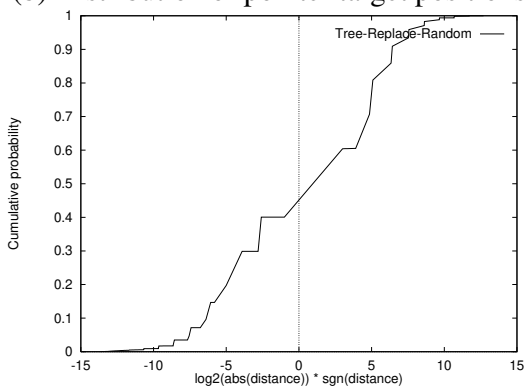
Figure 6.20. Pointer store heap position: Tree-Replace-Binary.



(a) Distribution of pointer source positions

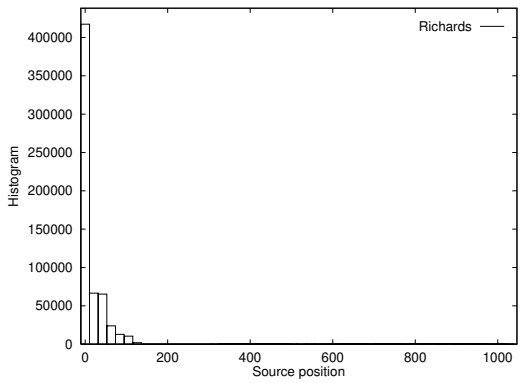


(b) Distribution of pointer target positions

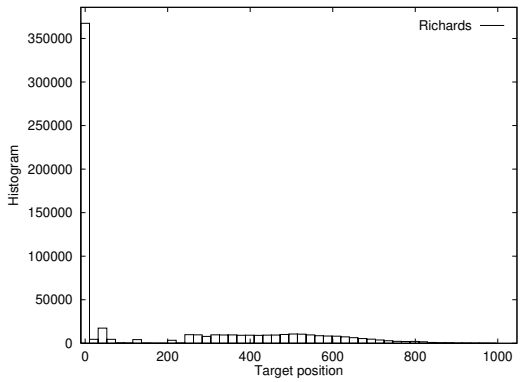


(c) Distribution of pointer distances

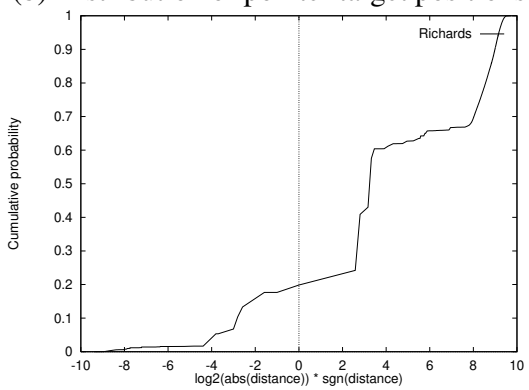
Figure 6.21. Pointer store heap position: Tree-Replace-Random.



(a) Distribution of pointer source positions

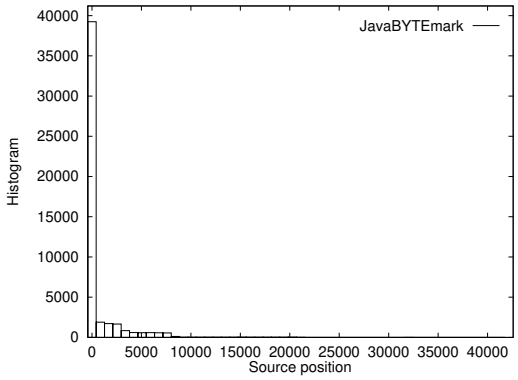


(b) Distribution of pointer target positions

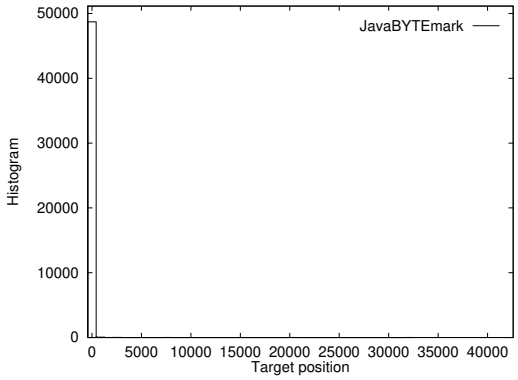


(c) Distribution of pointer distances

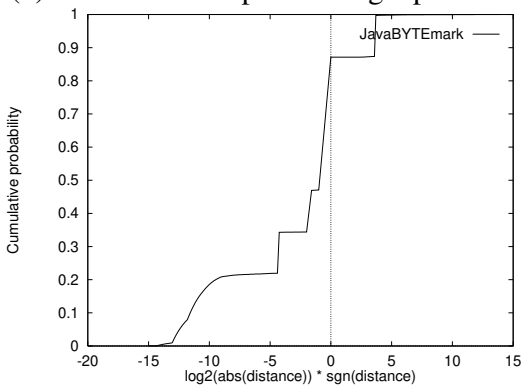
Figure 6.22. Pointer store heap position: Richards.



(a) Distribution of pointer source positions

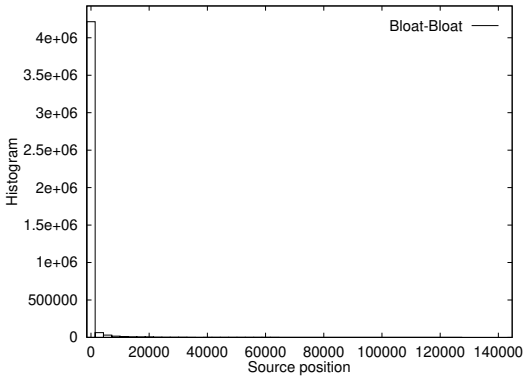


(b) Distribution of pointer target positions

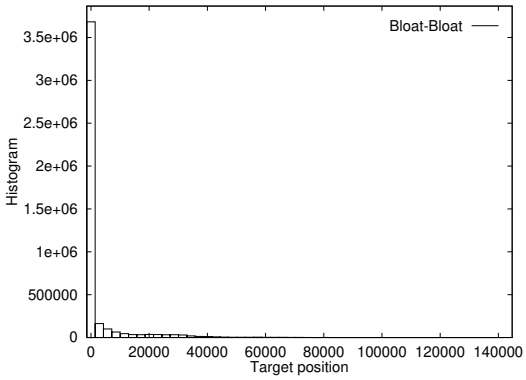


(c) Distribution of pointer distances

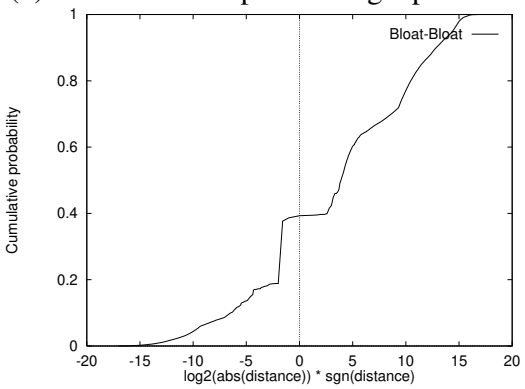
Figure 6.23. Pointer store heap position: JavaBYTEmark.



(a) Distribution of pointer source positions

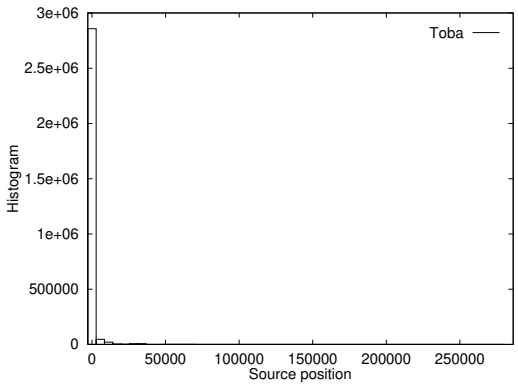


(b) Distribution of pointer target positions

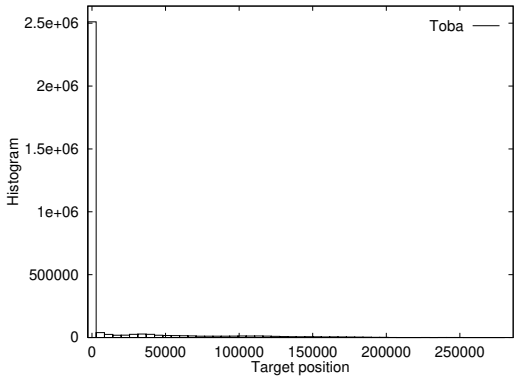


(c) Distribution of pointer distances

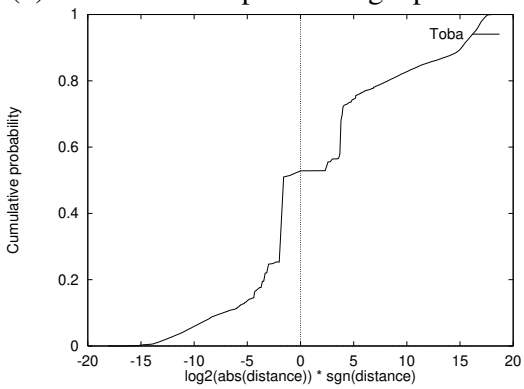
Figure 6.24. Pointer store heap position: Bloat-Bloat.



(a) Distribution of pointer source positions

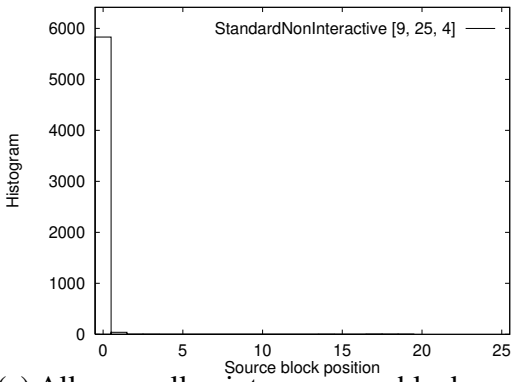


(b) Distribution of pointer target positions

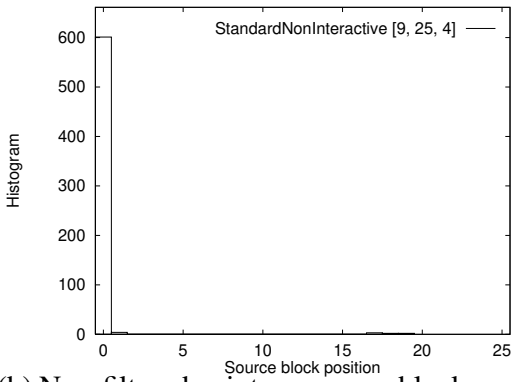


(c) Distribution of pointer distances

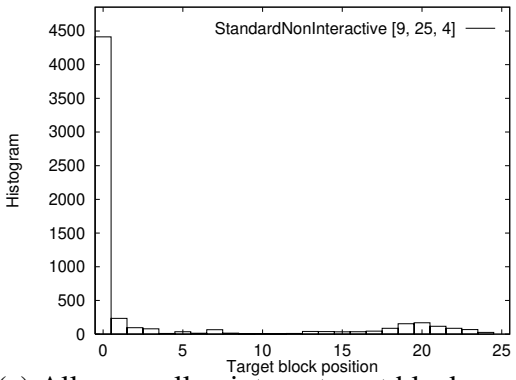
Figure 6.25. Pointer store heap position: Toba.



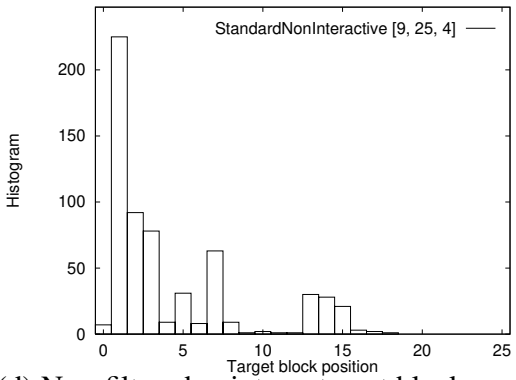
(a) All non-null pointers: source block position



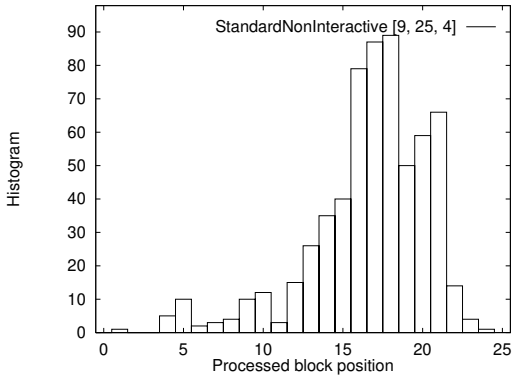
(b) Non-filtered pointers: source block position



(c) All non-null pointers: target block position

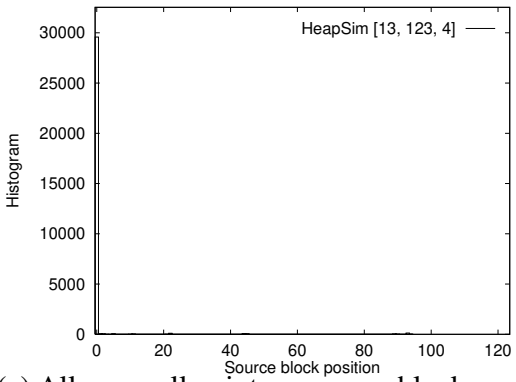


(d) Non-filtered pointers: target block position

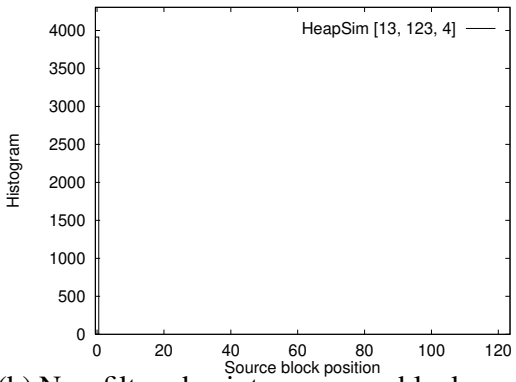


(e) Processed remembered set entry block position

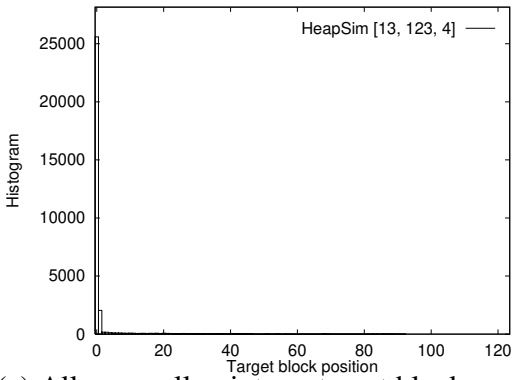
Figure 6.26. DOF pointer position distributions: StandardNonInteractive.



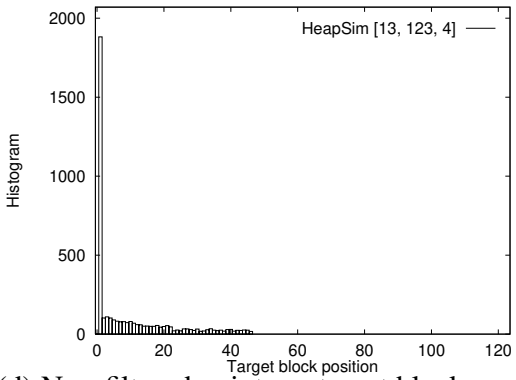
(a) All non-null pointers: source block position



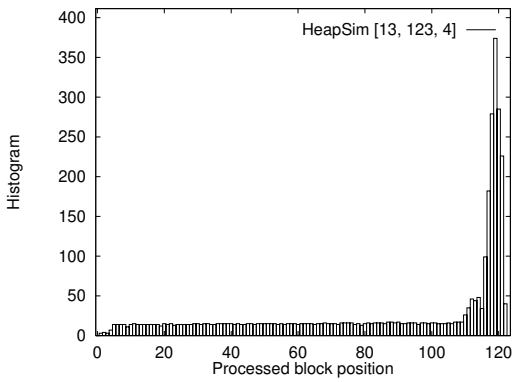
(b) Non-filtered pointers: source block position



(c) All non-null pointers: target block position

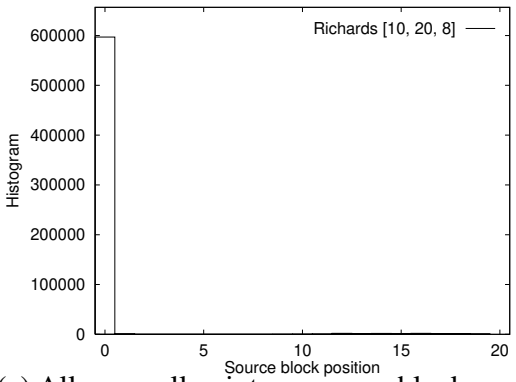


(d) Non-filtered pointers: target block position

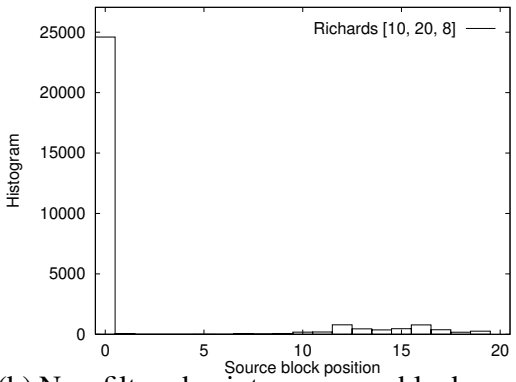


(e) Processed remembered set entry block position

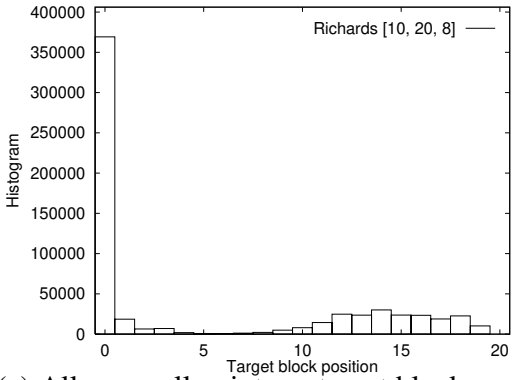
Figure 6.27. DOF pointer position distributions: HeapSim.



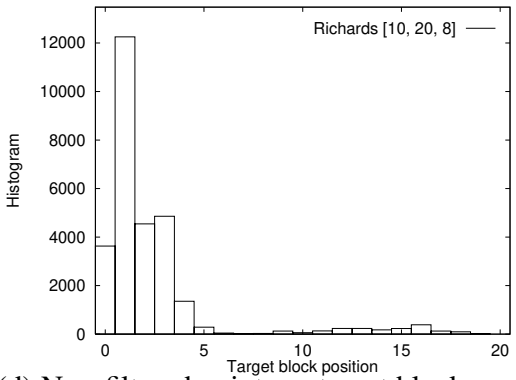
(a) All non-null pointers: source block position



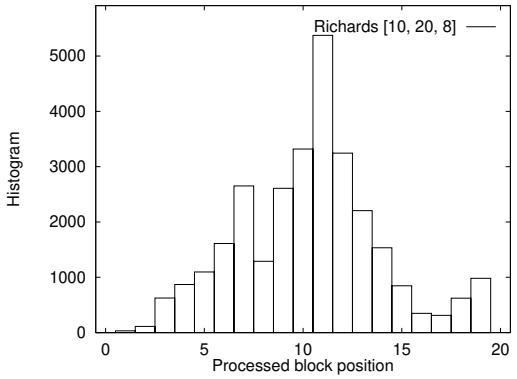
(b) Non-filtered pointers: source block position



(c) All non-null pointers: target block position

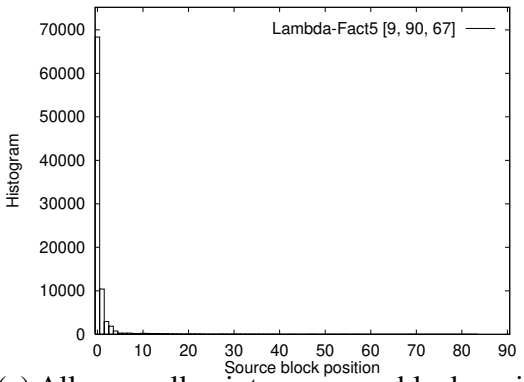


(d) Non-filtered pointers: target block position

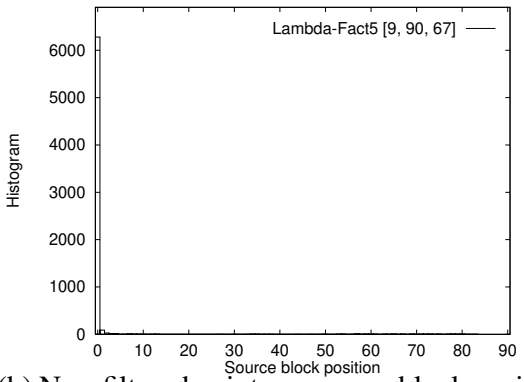


(e) Processed remembered set entry block position

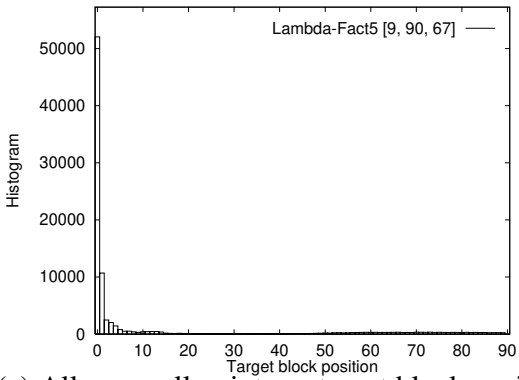
Figure 6.28. DOF pointer position distributions: Richards.



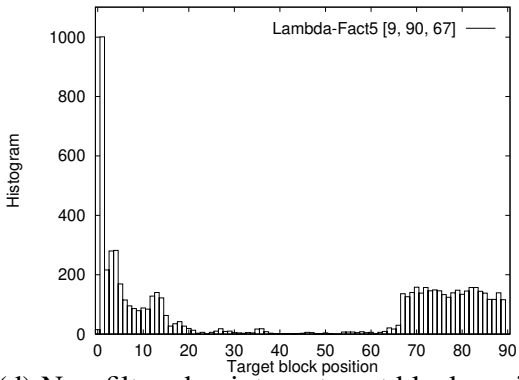
(a) All non-null pointers: source block position



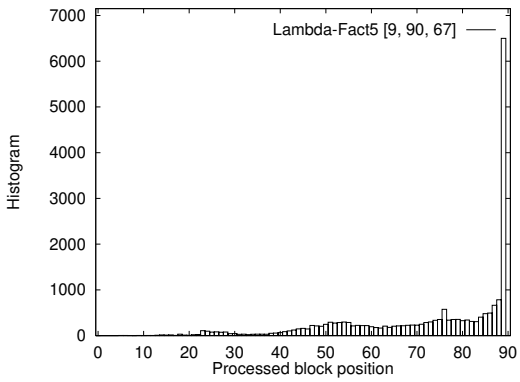
(b) Non-filtered pointers: source block position



(c) All non-null pointers: target block position

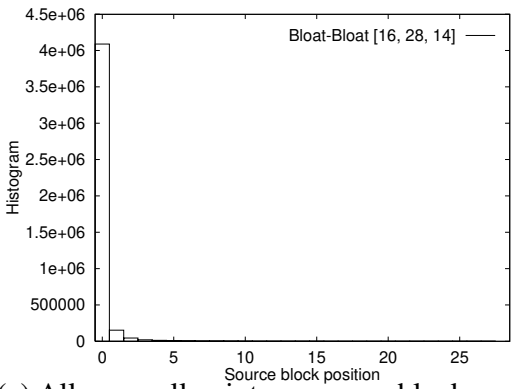


(d) Non-filtered pointers: target block position

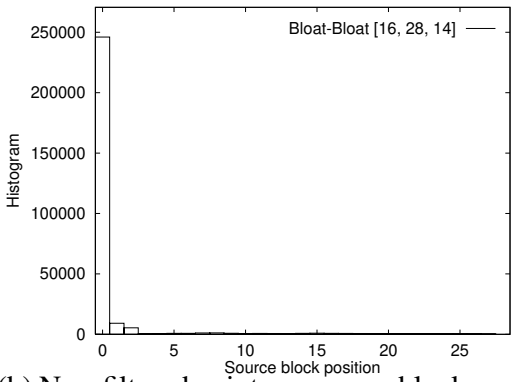


(e) Processed remembered set entry block position

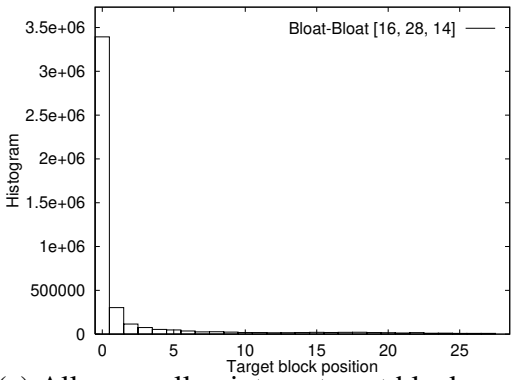
Figure 6.29. DOF pointer position distributions: Lambda-Fact5.



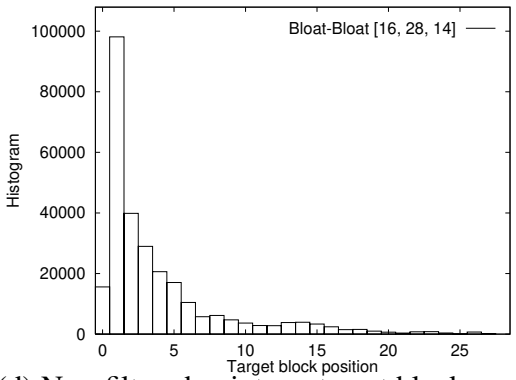
(a) All non-null pointers: source block position



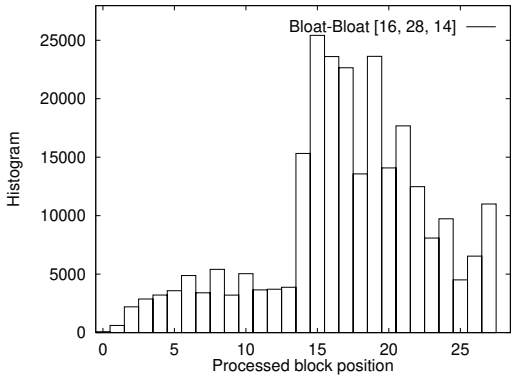
(b) Non-filtered pointers: source block position



(c) All non-null pointers: target block position

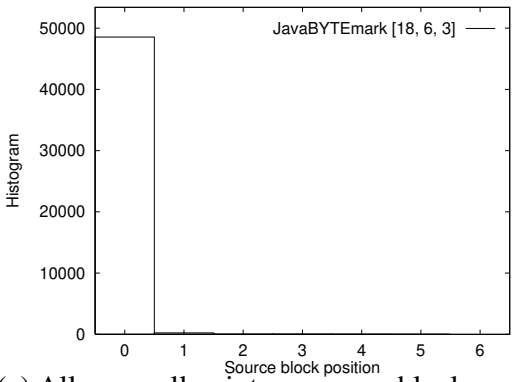


(d) Non-filtered pointers: target block position

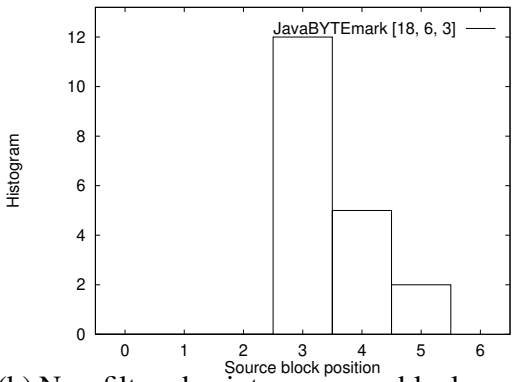


(e) Processed remembered set entry block position

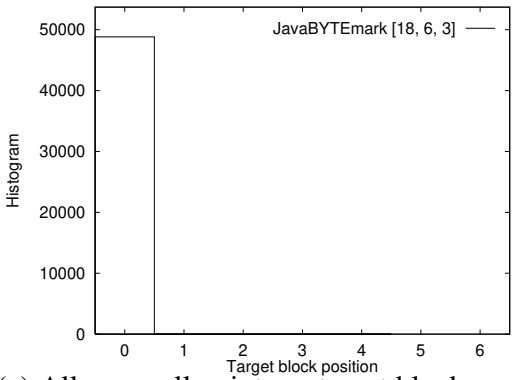
Figure 6.30. DOF pointer position distributions: Bloat-Bloat.



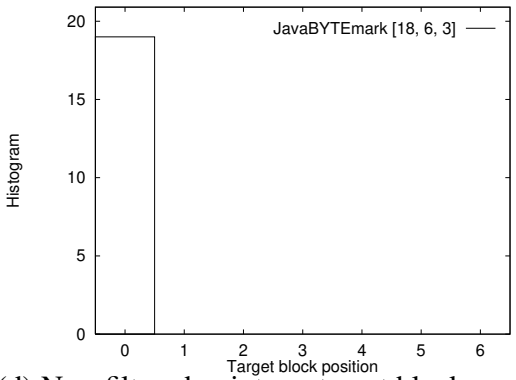
(a) All non-null pointers: source block position



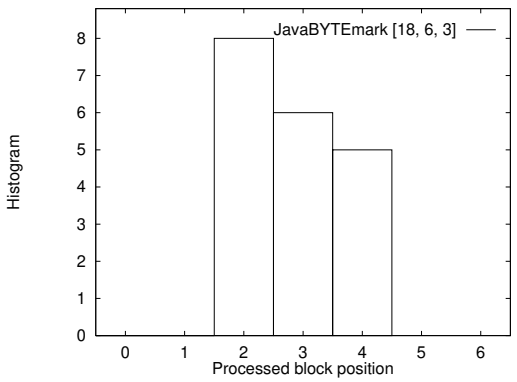
(b) Non-filtered pointers: source block position



(c) All non-null pointers: target block position

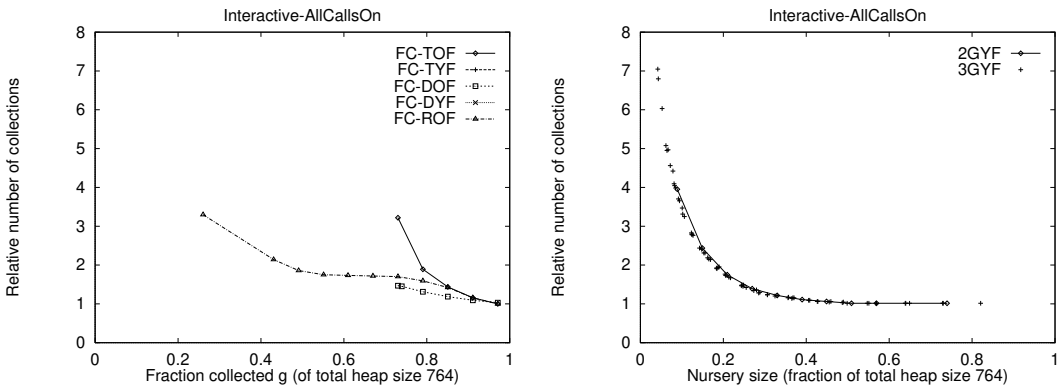


(d) Non-filtered pointers: target block position



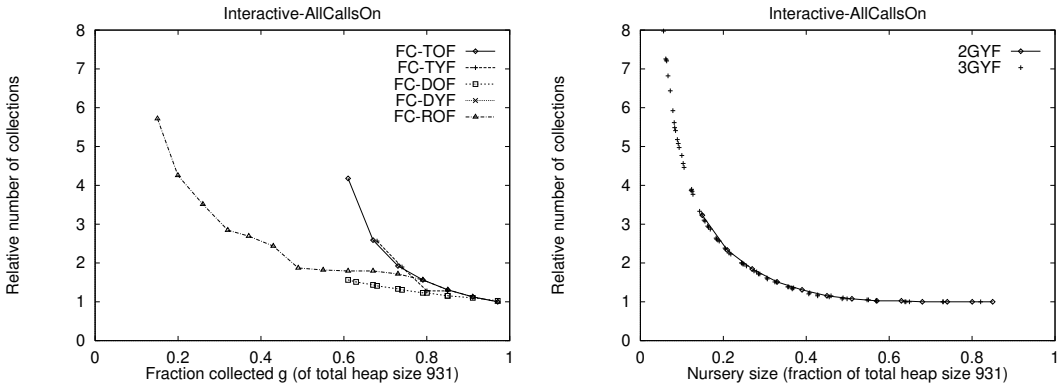
(e) Processed remembered set entry block position

Figure 6.31. DOF pointer position distributions: JavaBYTEmark.



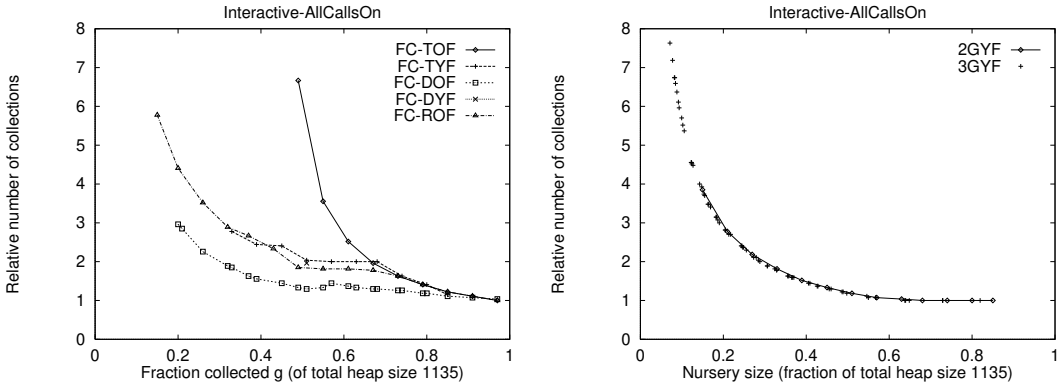
(a) FC schemes (b) GYF schemes

Figure 6.32. Relative invocation counts: Interactive-AllCallsOn, $V = 764$.



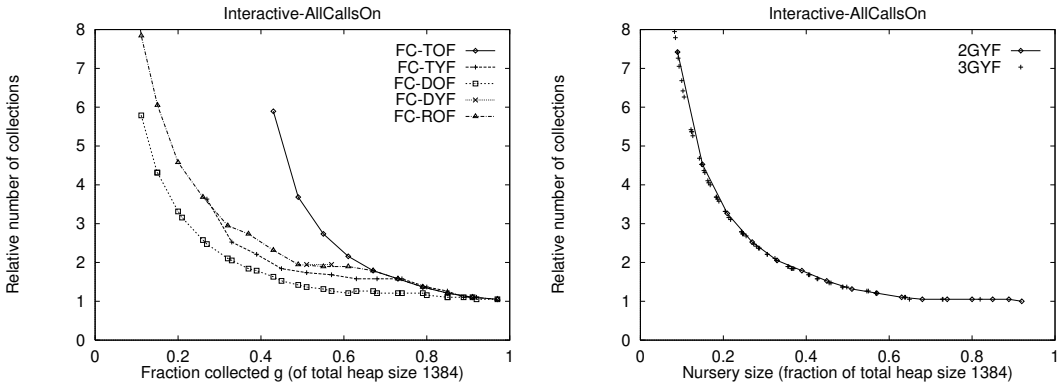
(a) FC schemes (b) GYF schemes

Figure 6.33. Relative invocation counts: Interactive-AllCallsOn, $V = 931$.



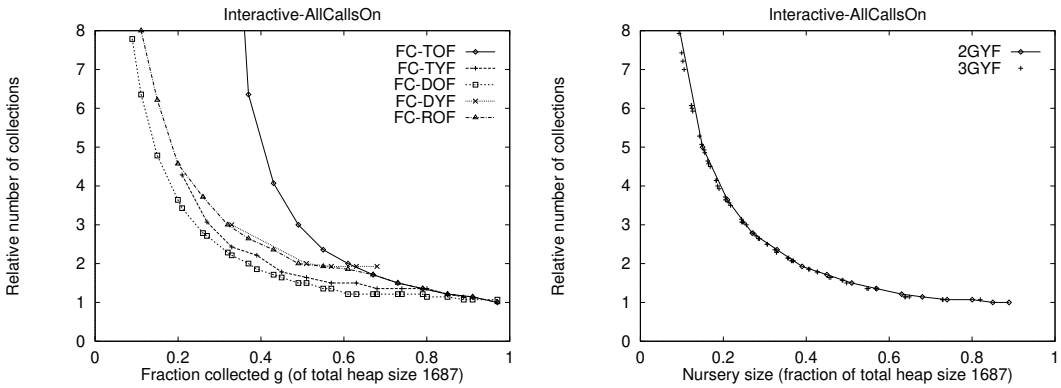
(a) FC schemes (b) GYF schemes

Figure 6.34. Relative invocation counts: Interactive-AllCallsOn, $V = 1135$.



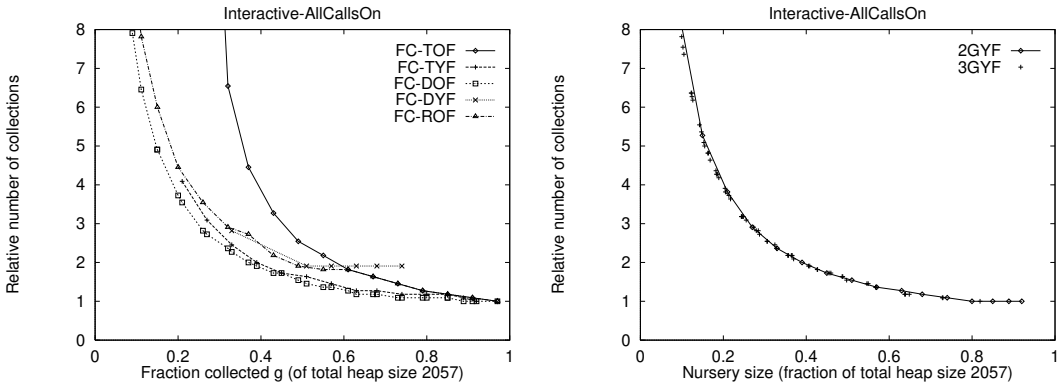
(a) FC schemes (b) GYF schemes

Figure 6.35. Relative invocation counts: Interactive-AllCallsOn, $V = 1384$.



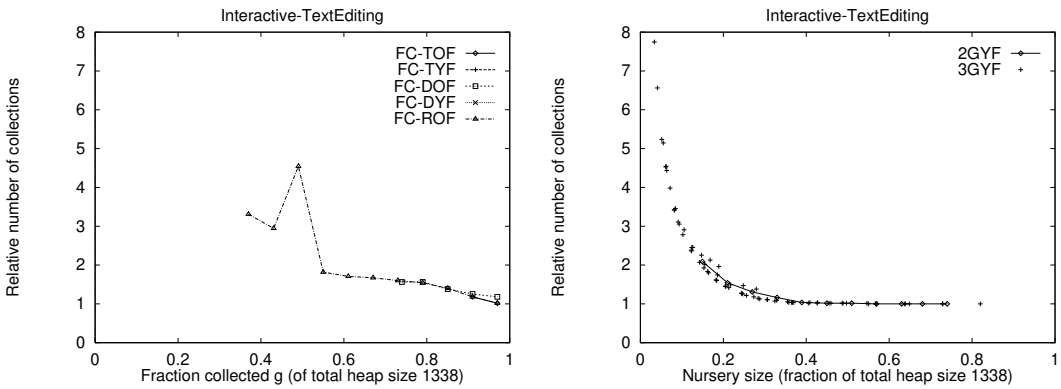
(a) FC schemes (b) GYF schemes

Figure 6.36. Relative invocation counts: Interactive-AllCallsOn, $V = 1687$.



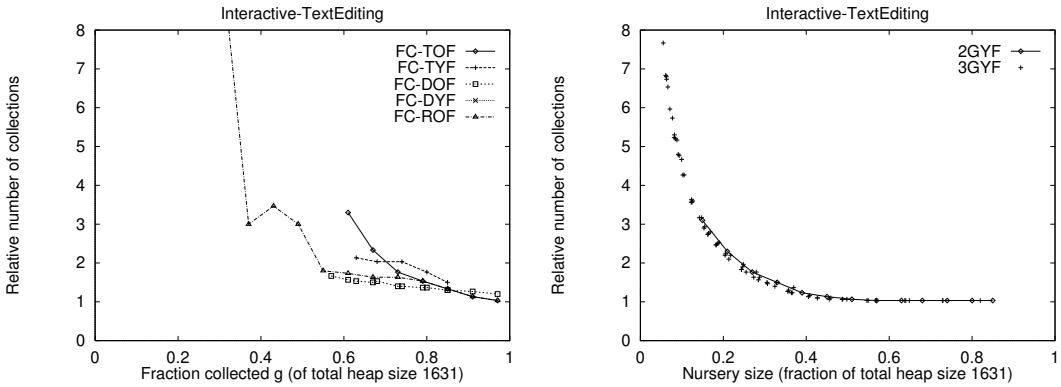
(a) FC schemes (b) GYF schemes

Figure 6.37. Relative invocation counts: Interactive-AllCallsOn, $V = 2057$.



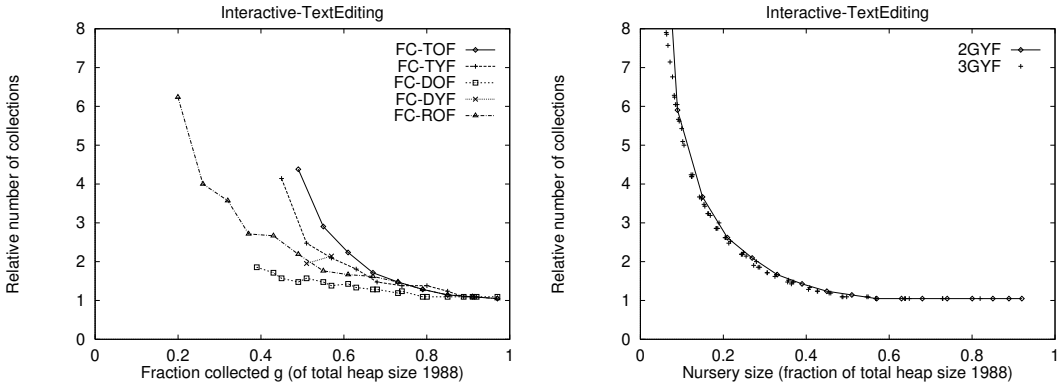
(a) FC schemes (b) GYF schemes

Figure 6.38. Relative invocation counts: Interactive-TextEditing, $V = 1338$.



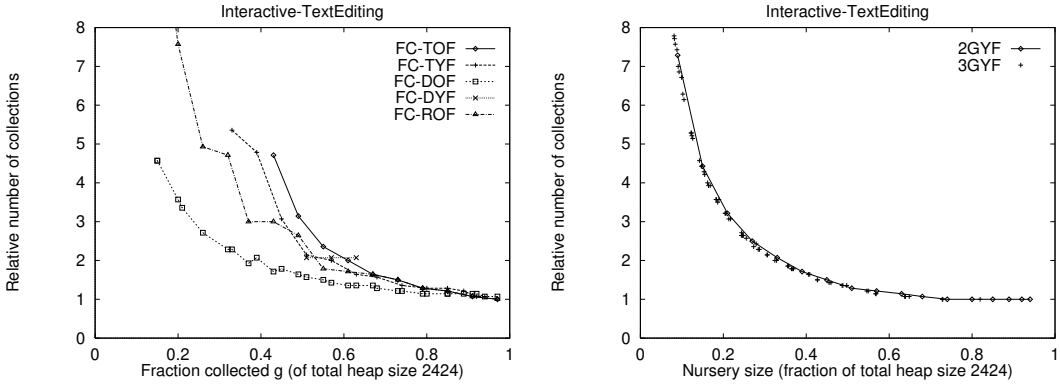
(a) FC schemes (b) GYF schemes

Figure 6.39. Relative invocation counts: Interactive-TextEditing, $V = 1631$.



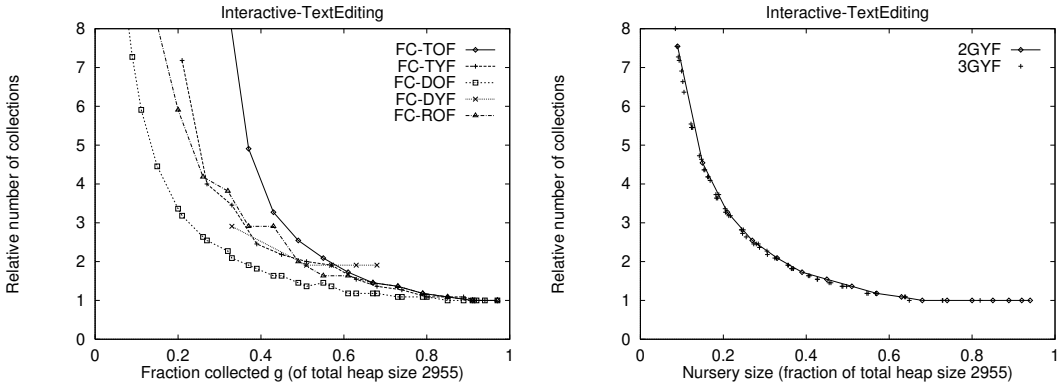
(a) FC schemes (b) GYF schemes

Figure 6.40. Relative invocation counts: Interactive-TextEditing, $V = 1988$.



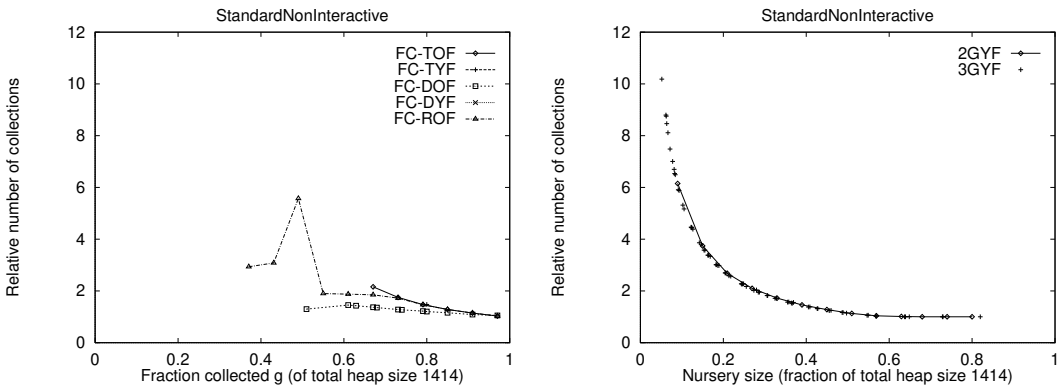
(a) FC schemes (b) GYF schemes

Figure 6.41. Relative invocation counts: Interactive-TextEditing, $V = 2424$.



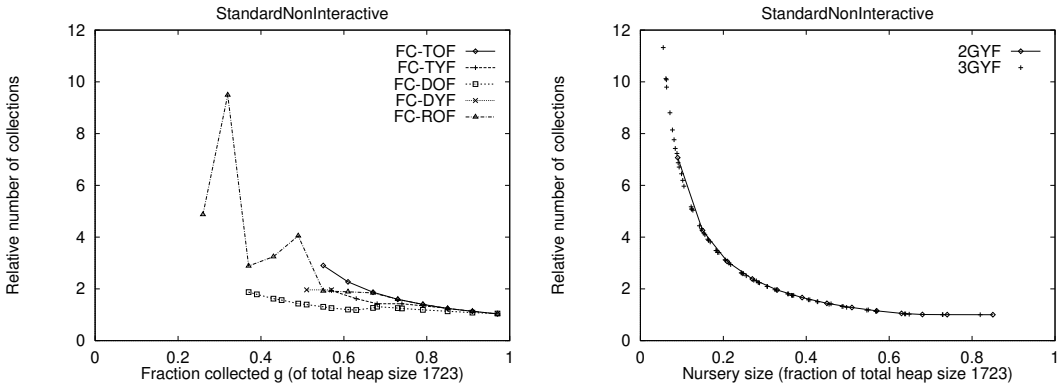
(a) FC schemes (b) GYF schemes

Figure 6.42. Relative invocation counts: Interactive-TextEditing, $V = 2955$.



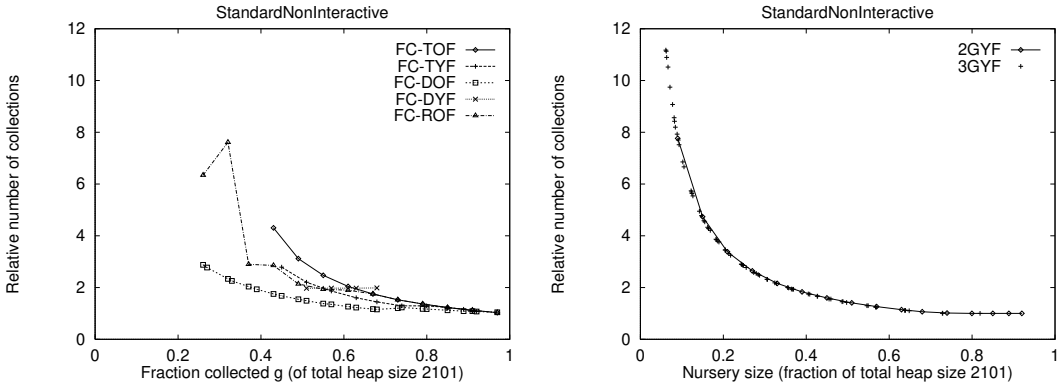
(a) FC schemes (b) GYF schemes

Figure 6.43. Relative invocation counts: StandardNonInteractive, $V = 1414$.



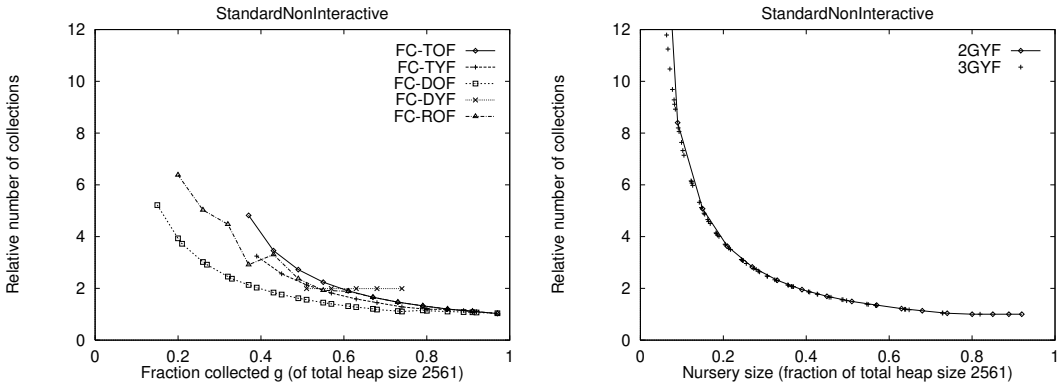
(a) FC schemes (b) GYF schemes

Figure 6.44. Relative invocation counts: StandardNonInteractive, $V = 1723$.



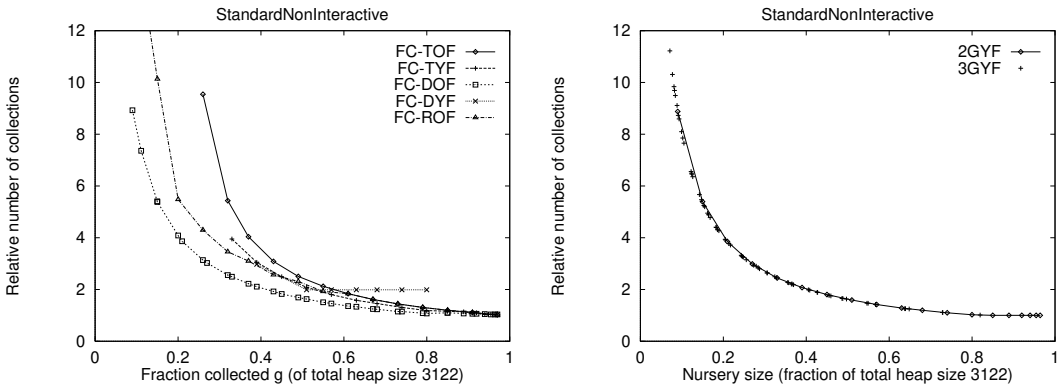
(a) FC schemes (b) GYF schemes

Figure 6.45. Relative invocation counts: StandardNonInteractive, $V = 2101$.



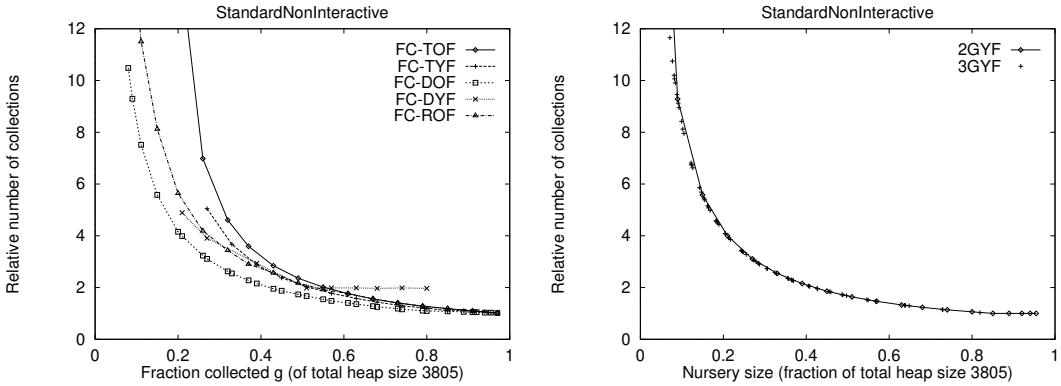
(a) FC schemes (b) GYF schemes

Figure 6.46. Relative invocation counts: StandardNonInteractive, $V = 2561$.



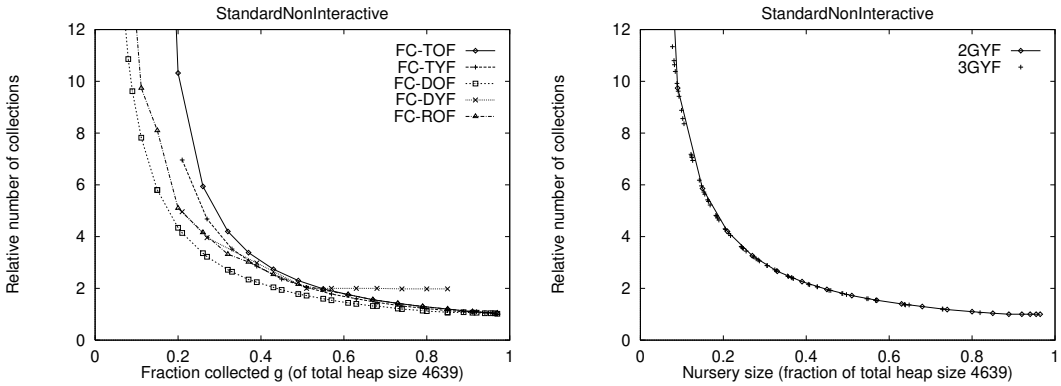
(a) FC schemes (b) GYF schemes

Figure 6.47. Relative invocation counts: StandardNonInteractive, $V = 3122$.



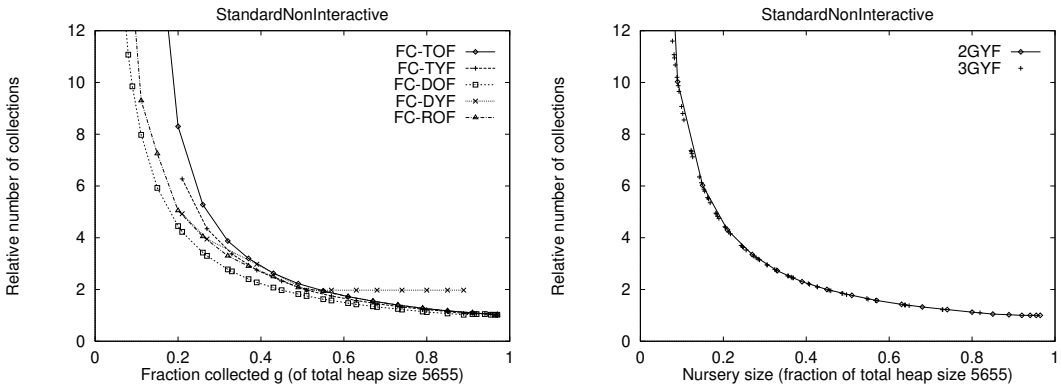
(a) FC schemes (b) GYF schemes

Figure 6.48. Relative invocation counts: StandardNonInteractive, $V = 3805$.



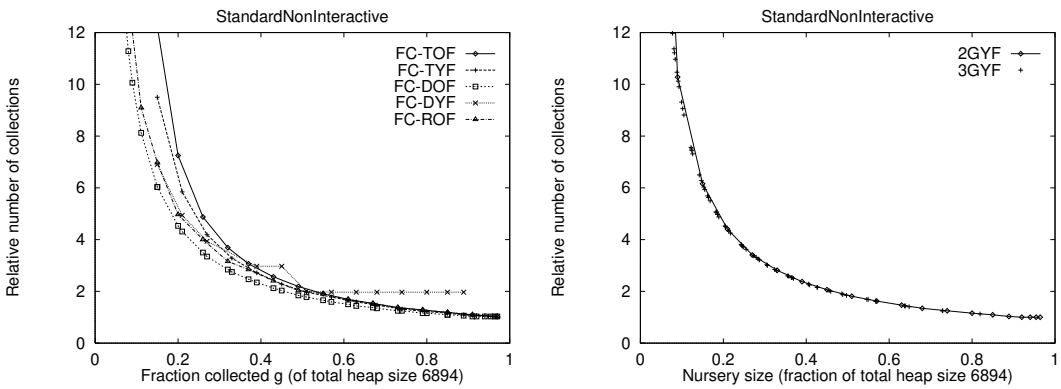
(a) FC schemes (b) GYF schemes

Figure 6.49. Relative invocation counts: StandardNonInteractive, $V = 4639$.



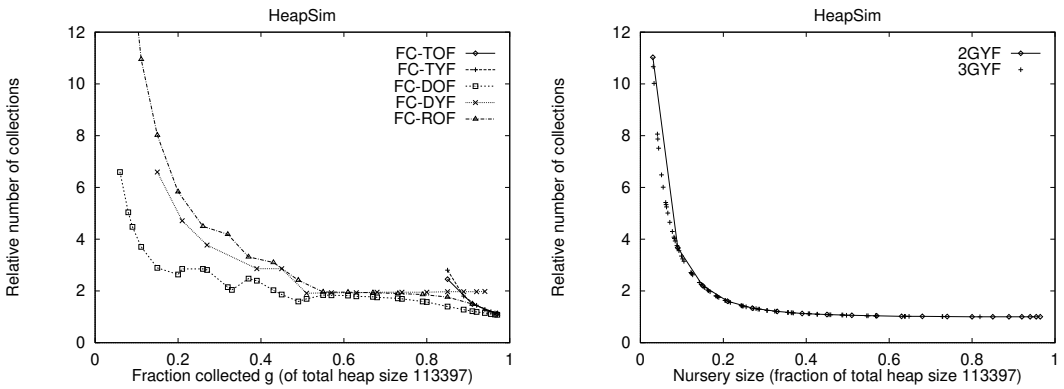
(a) FC schemes (b) GYF schemes

Figure 6.50. Relative invocation counts: StandardNonInteractive, $V = 5655$.



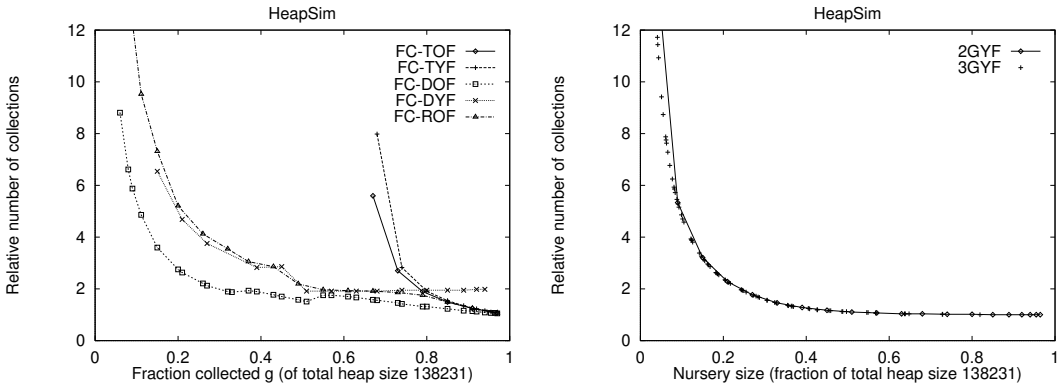
(a) FC schemes (b) GYF schemes

Figure 6.51. Relative invocation counts: StandardNonInteractive, $V = 6894$.



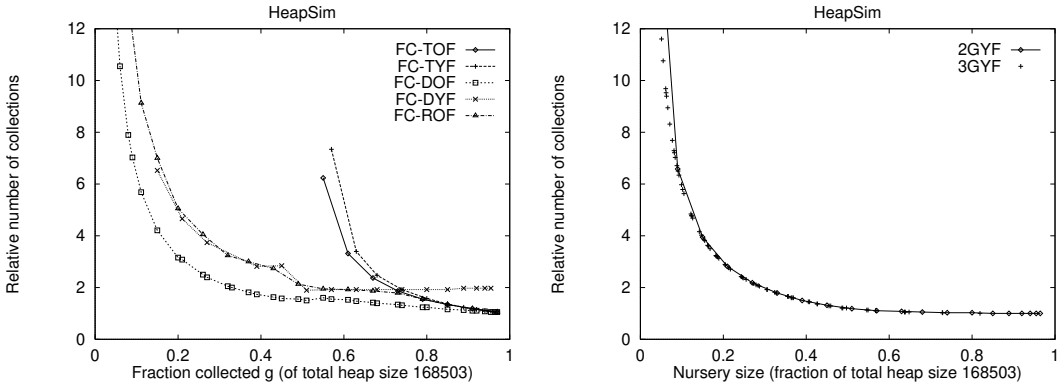
(a) FC schemes (b) GYF schemes

Figure 6.52. Relative invocation counts: HeapSim, $V = 113397$.



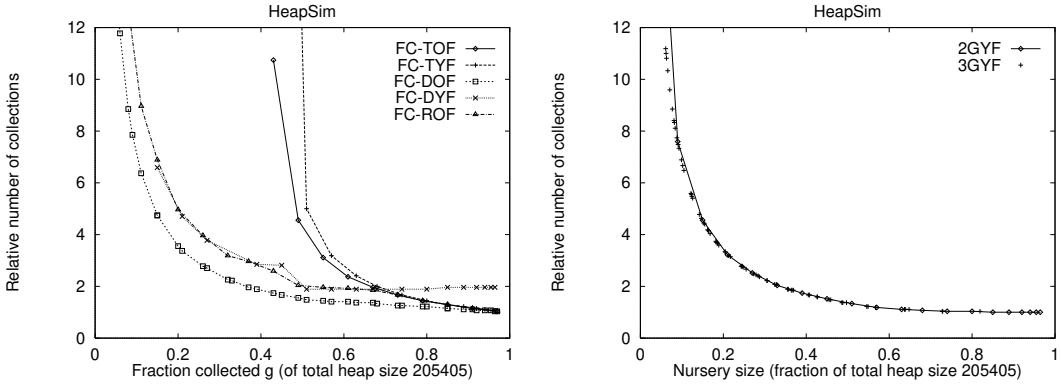
(a) FC schemes (b) GYF schemes

Figure 6.53. Relative invocation counts: HeapSim, $V = 138231$.



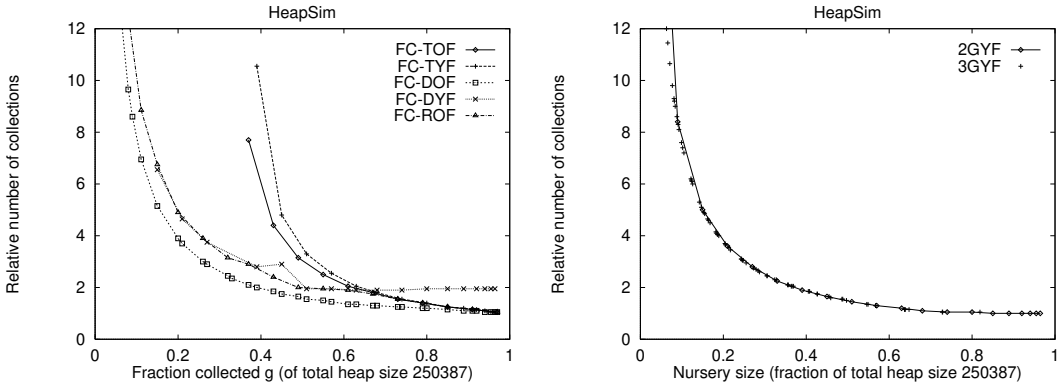
(a) FC schemes (b) GYF schemes

Figure 6.54. Relative invocation counts: HeapSim, $V = 168503$.



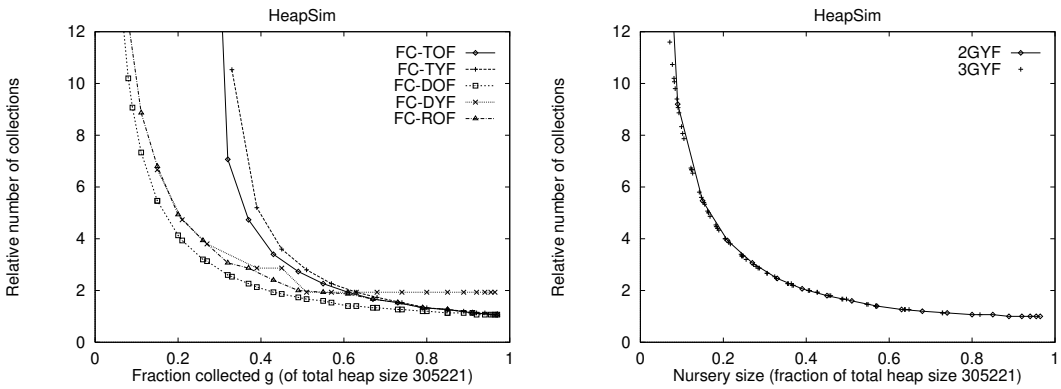
(a) FC schemes (b) GYF schemes

Figure 6.55. Relative invocation counts: HeapSim, $V = 205405$.



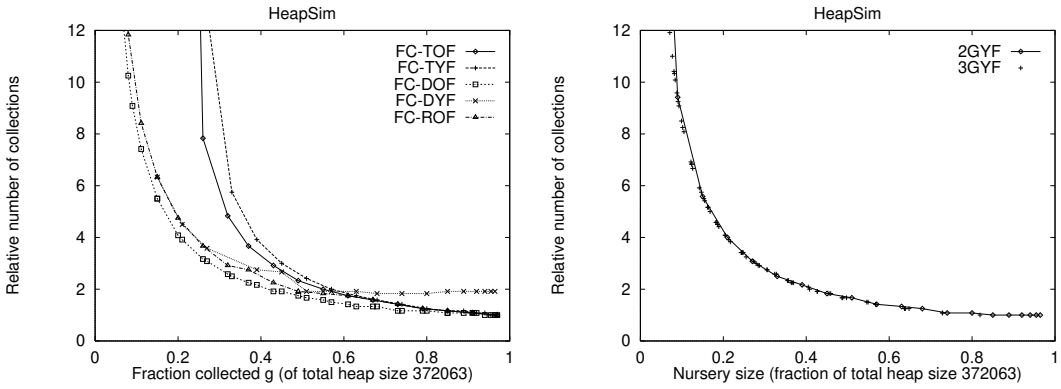
(a) FC schemes (b) GYF schemes

Figure 6.56. Relative invocation counts: HeapSim, $V = 250387$.



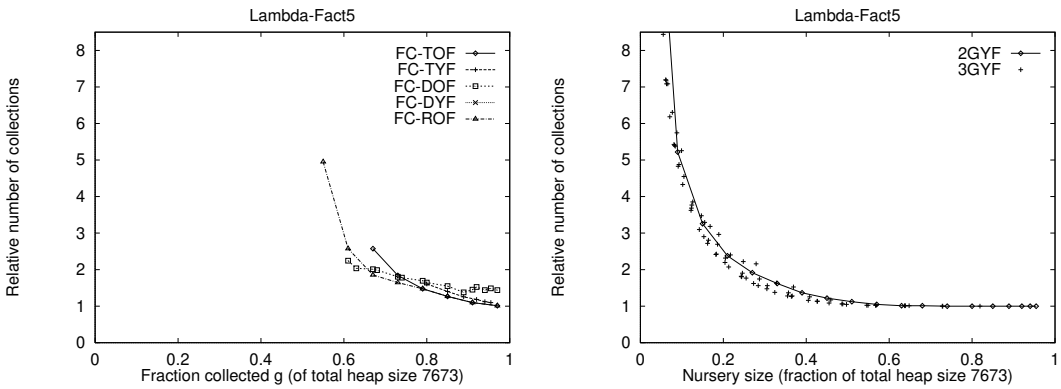
(a) FC schemes (b) GYF schemes

Figure 6.57. Relative invocation counts: HeapSim, $V = 305221$.



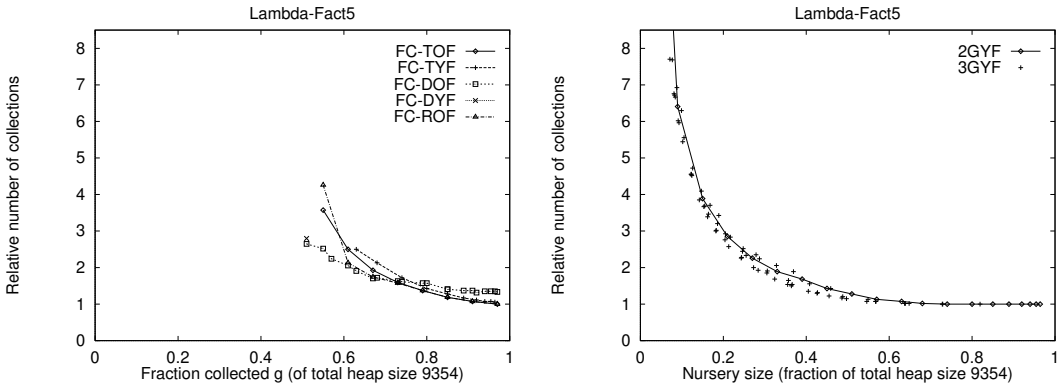
(a) FC schemes (b) GYF schemes

Figure 6.58. Relative invocation counts: HeapSim, $V = 372063$.



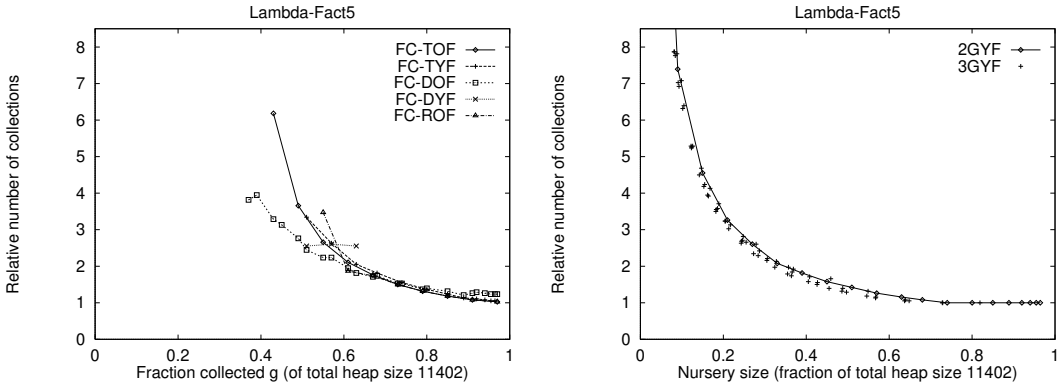
(a) FC schemes (b) GYF schemes

Figure 6.59. Relative invocation counts: Lambda-Fact5, $V = 7673$.



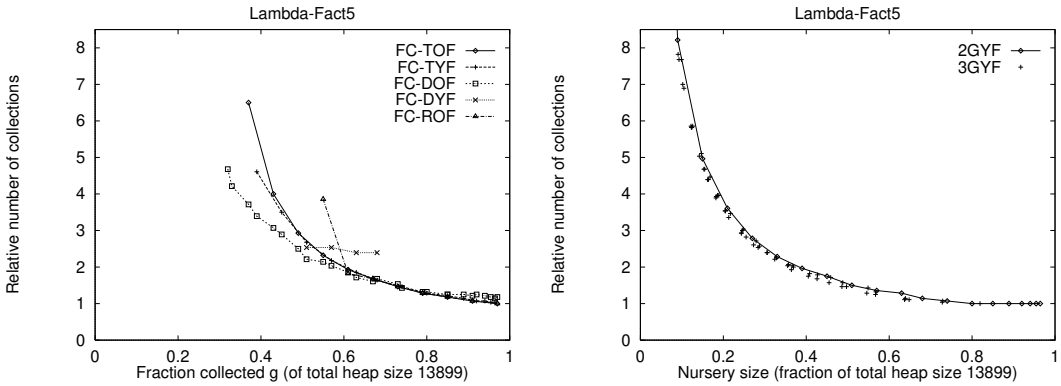
(a) FC schemes (b) GYF schemes

Figure 6.60. Relative invocation counts: Lambda-Fact5, $V = 9354$.



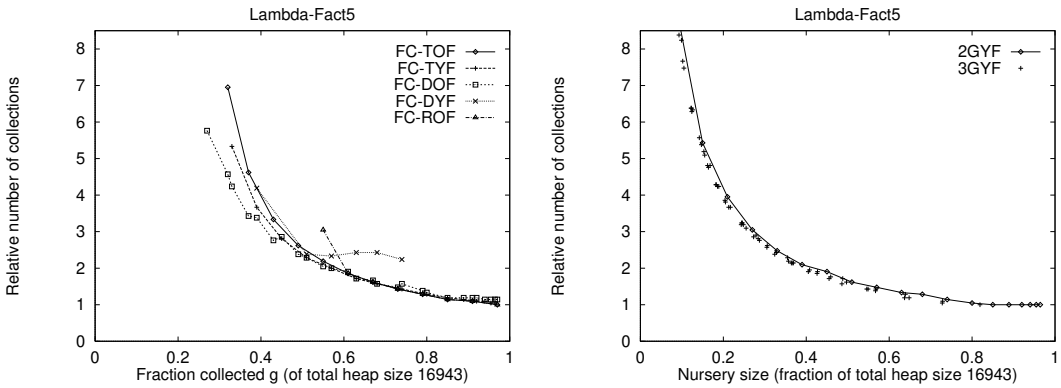
(a) FC schemes (b) GYF schemes

Figure 6.61. Relative invocation counts: Lambda-Fact5, $V = 11402$.



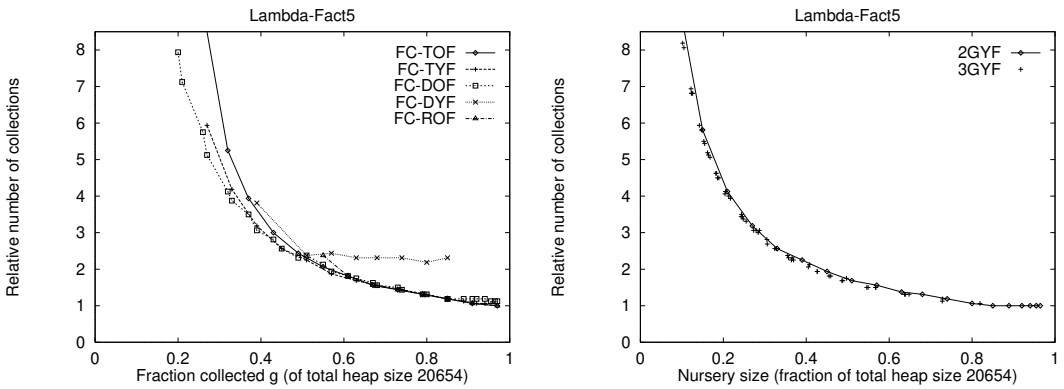
(a) FC schemes (b) GYF schemes

Figure 6.62. Relative invocation counts: Lambda-Fact5, $V = 13899$.



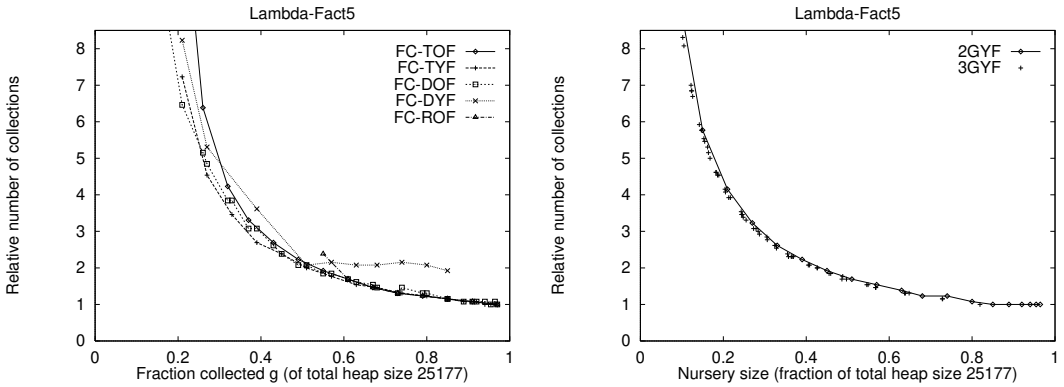
(a) FC schemes (b) GYF schemes

Figure 6.63. Relative invocation counts: Lambda-Fact5, $V = 16943$.



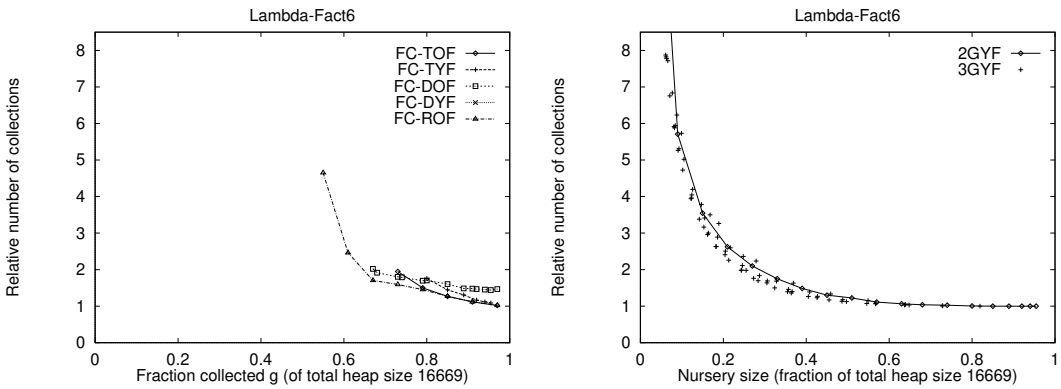
(a) FC schemes (b) GYF schemes

Figure 6.64. Relative invocation counts: Lambda-Fact5, $V = 20654$.



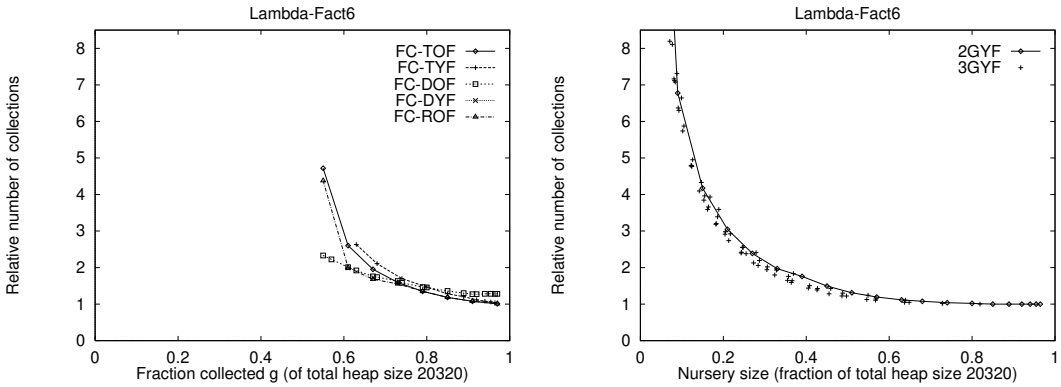
(a) FC schemes (b) GYF schemes

Figure 6.65. Relative invocation counts: Lambda-Fact5, $V = 25177$.



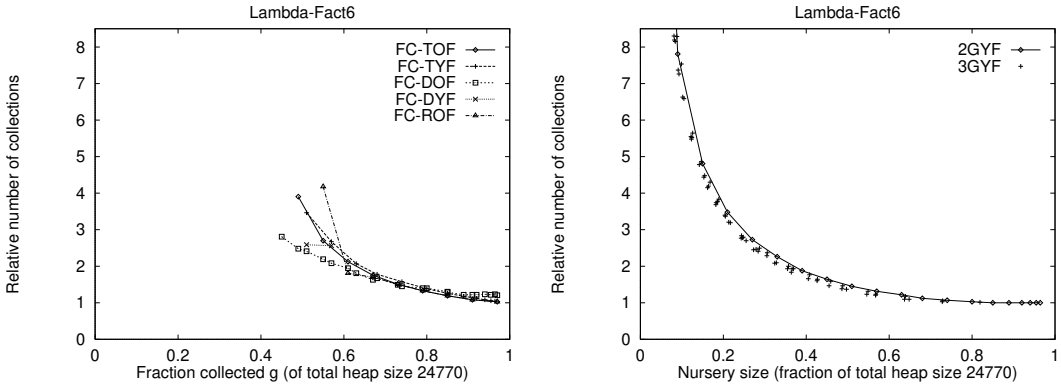
(a) FC schemes (b) GYF schemes

Figure 6.66. Relative invocation counts: Lambda-Fact6, $V = 16669$.



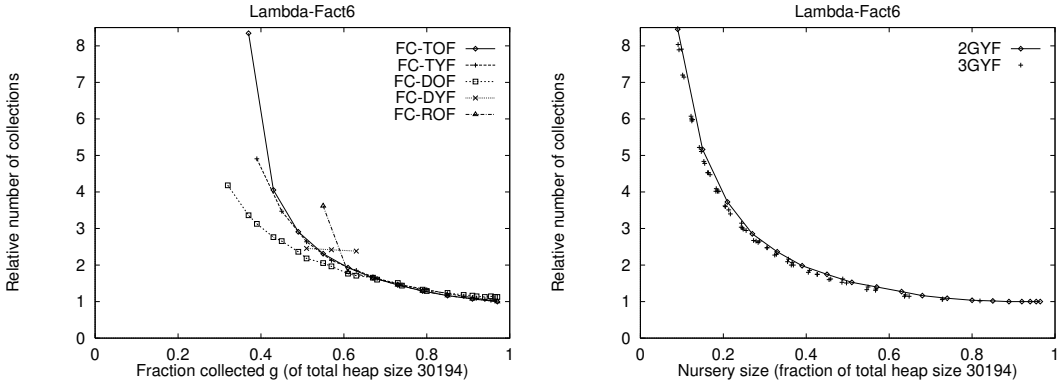
(a) FC schemes (b) GYF schemes

Figure 6.67. Relative invocation counts: Lambda-Fact6, $V = 20320$.



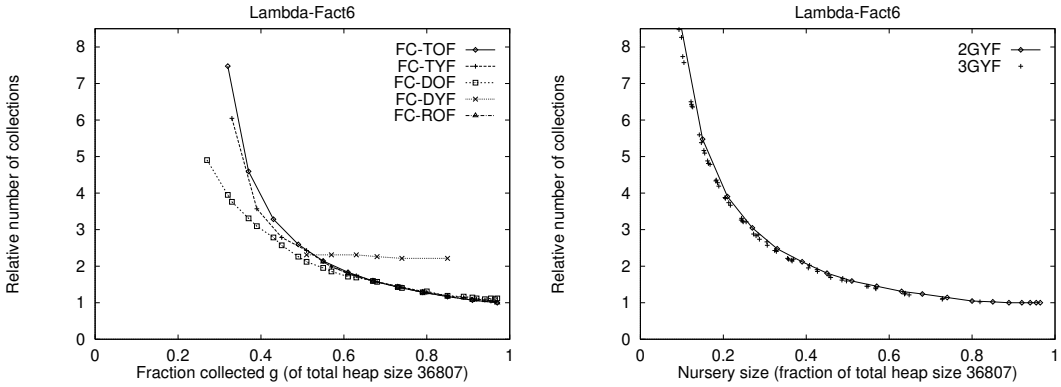
(a) FC schemes (b) GYF schemes

Figure 6.68. Relative invocation counts: Lambda-Fact6, $V = 24770$.



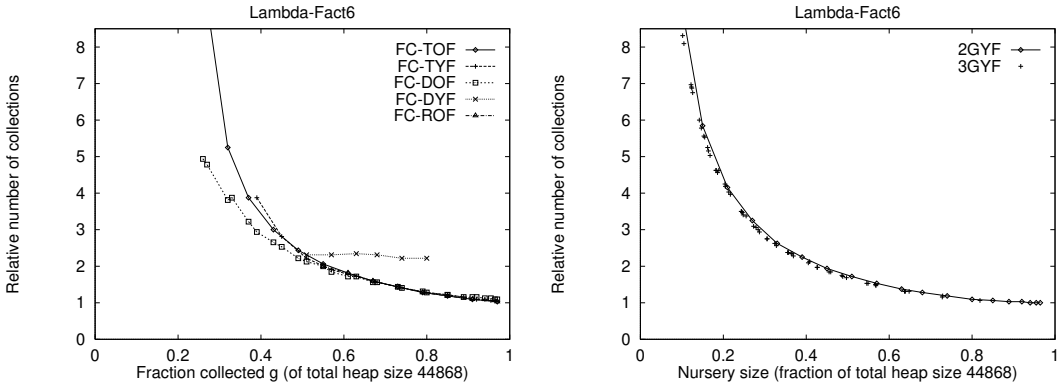
(a) FC schemes (b) GYF schemes

Figure 6.69. Relative invocation counts: Lambda-Fact6, $V = 30194$.



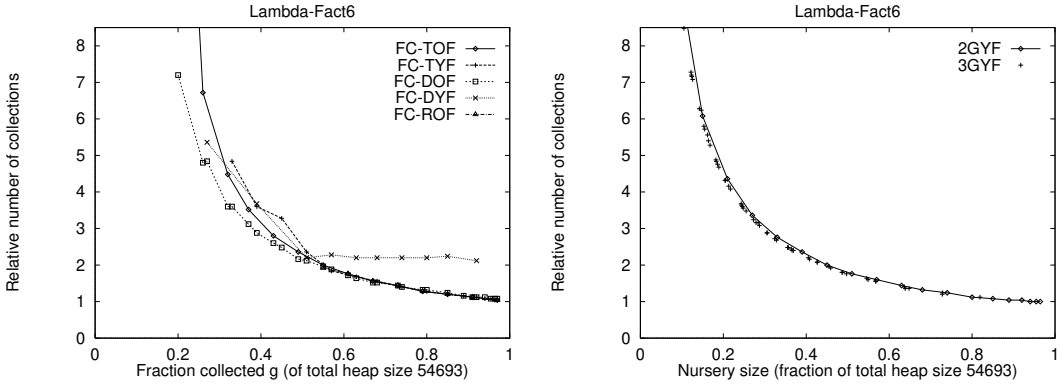
(a) FC schemes (b) GYF schemes

Figure 6.70. Relative invocation counts: Lambda-Fact6, $V = 36807$.



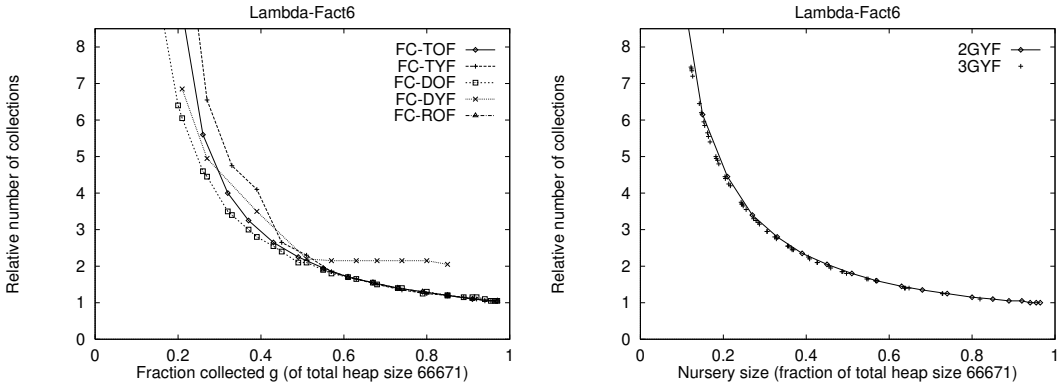
(a) FC schemes (b) GYF schemes

Figure 6.71. Relative invocation counts: Lambda-Fact6, $V = 44868$.



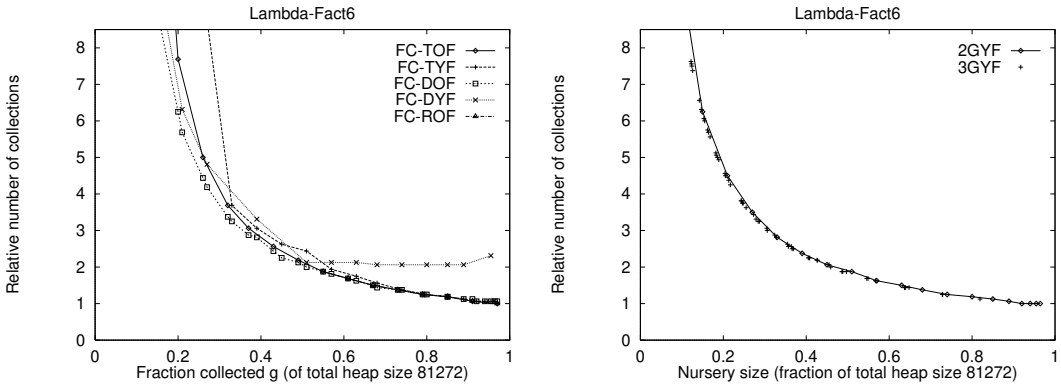
(a) FC schemes (b) GYF schemes

Figure 6.72. Relative invocation counts: Lambda-Fact6, $V = 54693$.



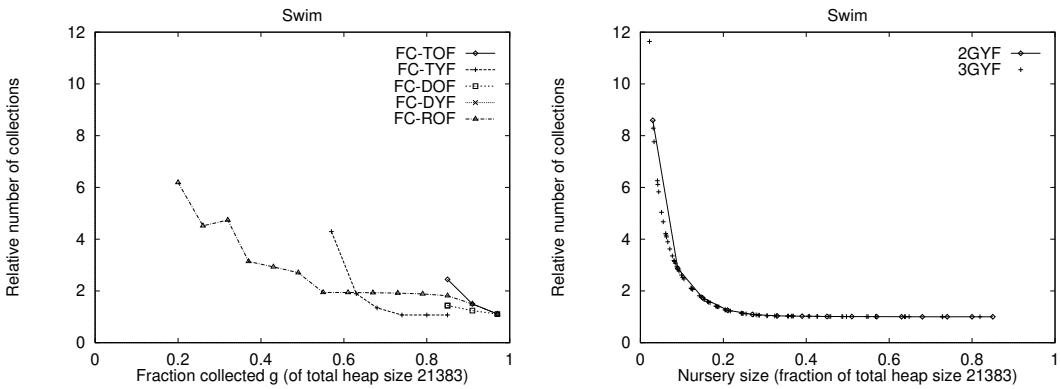
(a) FC schemes (b) GYF schemes

Figure 6.73. Relative invocation counts: Lambda-Fact6, $V = 66671$.



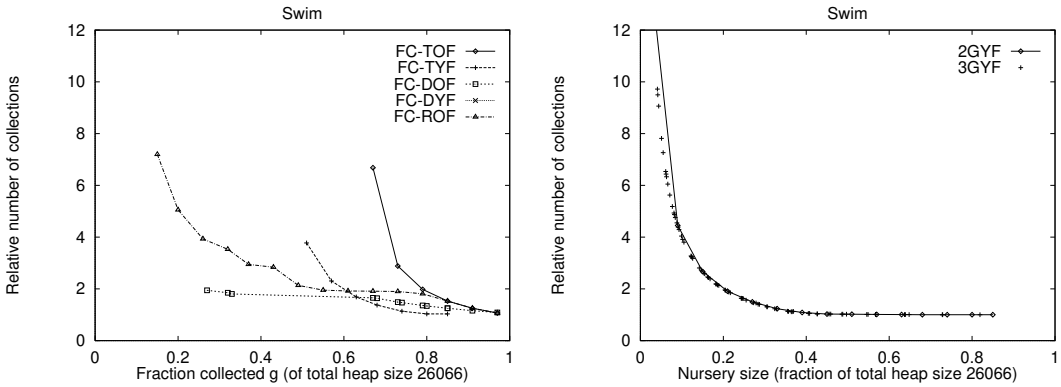
(a) FC schemes (b) GYF schemes

Figure 6.74. Relative invocation counts: Lambda-Fact6, $V = 81272$.



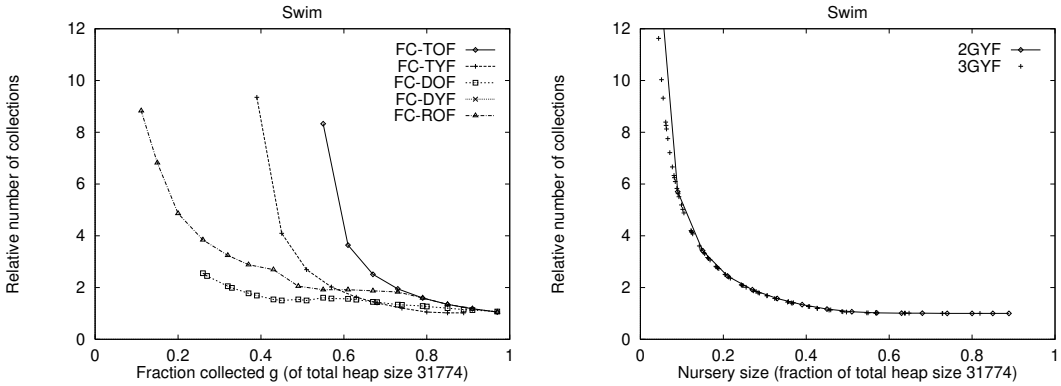
(a) FC schemes (b) GYF schemes

Figure 6.75. Relative invocation counts: Swim, $V = 21383$.



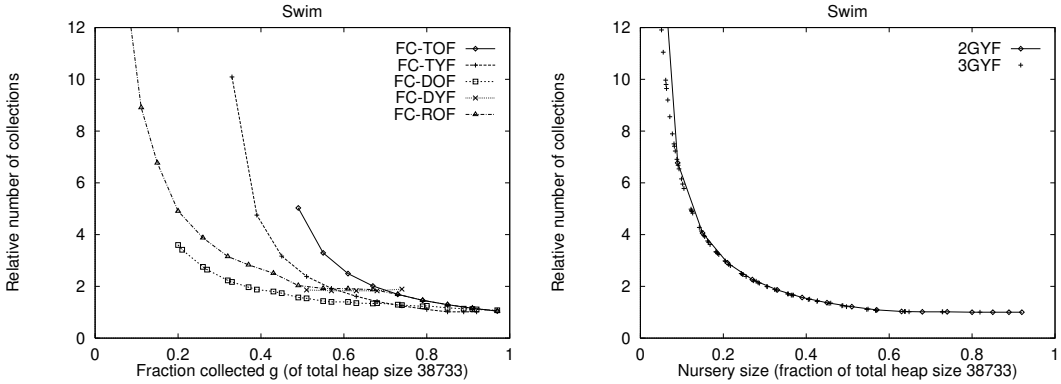
(a) FC schemes (b) GYF schemes

Figure 6.76. Relative invocation counts: Swim, $V = 26066$.



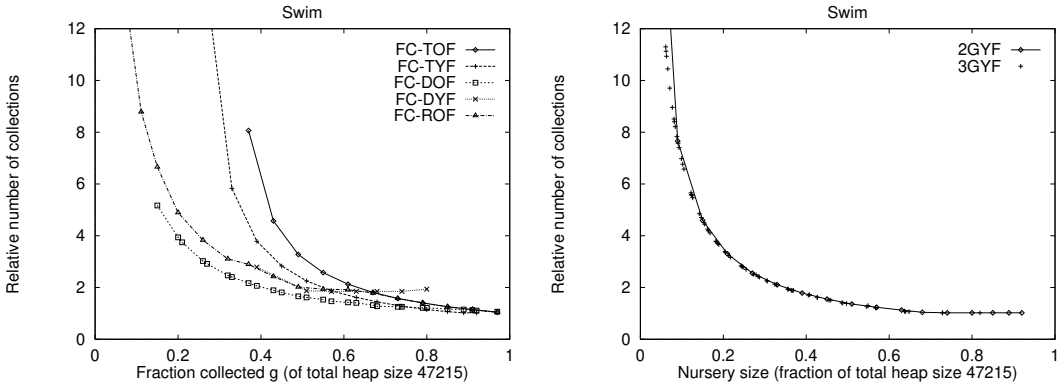
(a) FC schemes (b) GYF schemes

Figure 6.77. Relative invocation counts: Swim, $V = 31774$.



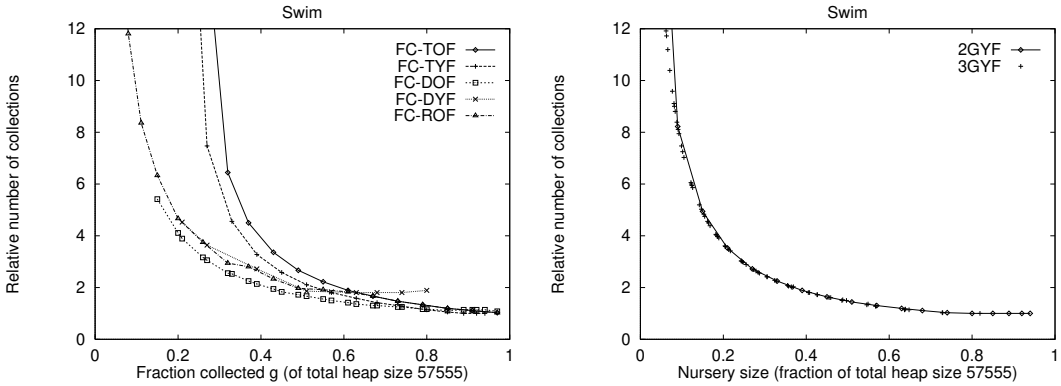
(a) FC schemes (b) GYF schemes

Figure 6.78. Relative invocation counts: Swim, $V = 38733$.



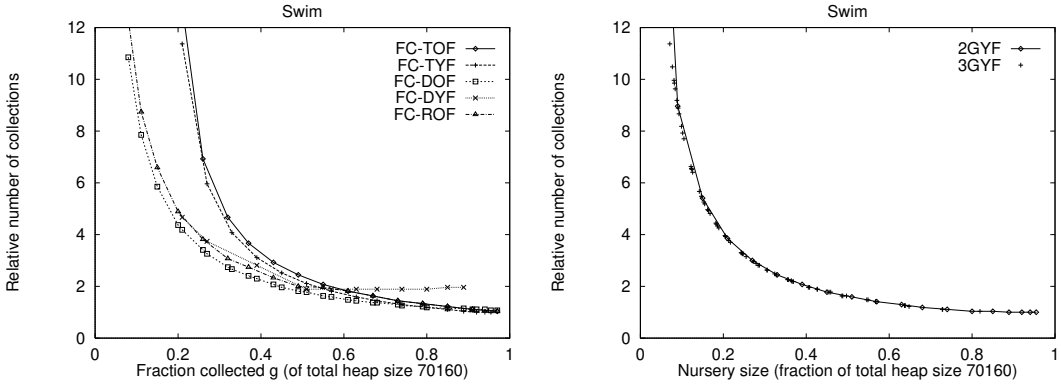
(a) FC schemes (b) GYF schemes

Figure 6.79. Relative invocation counts: Swim, $V = 47215$.



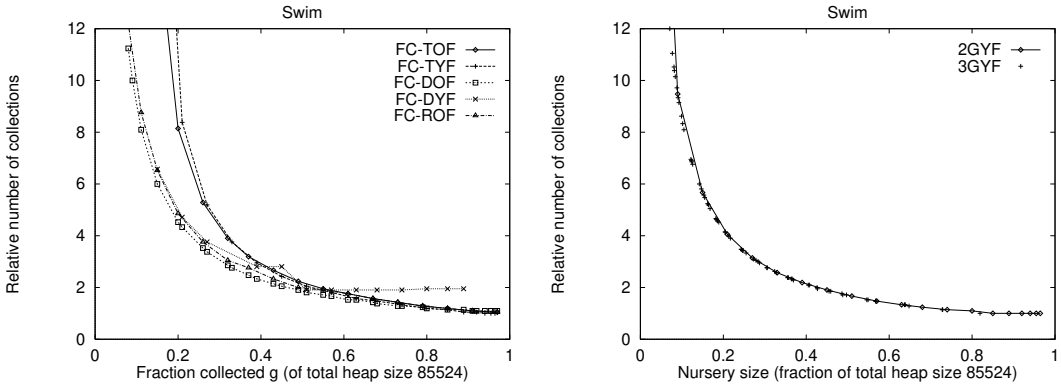
(a) FC schemes (b) GYF schemes

Figure 6.80. Relative invocation counts: Swim, $V = 57555$.



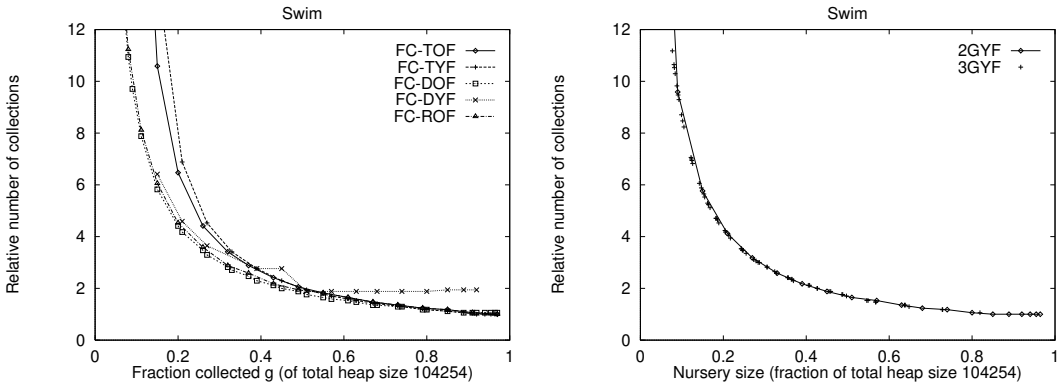
(a) FC schemes (b) GYF schemes

Figure 6.81. Relative invocation counts: Swim, $V = 70160$.



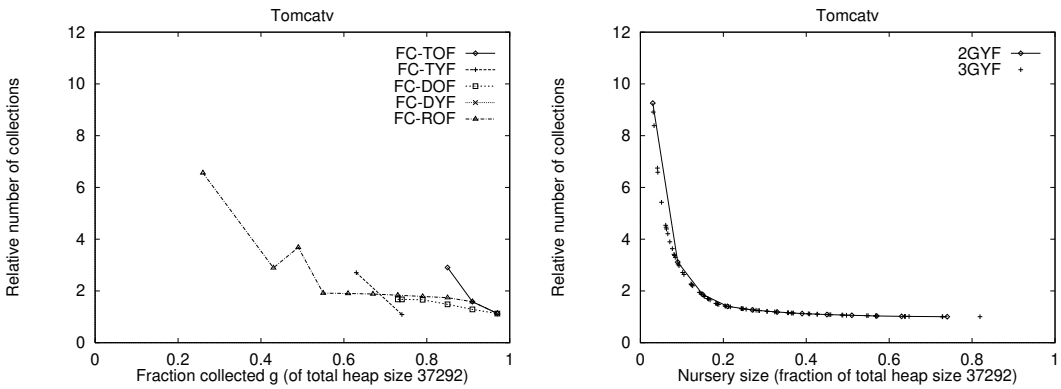
(a) FC schemes (b) GYF schemes

Figure 6.82. Relative invocation counts: Swim, $V = 85524$.



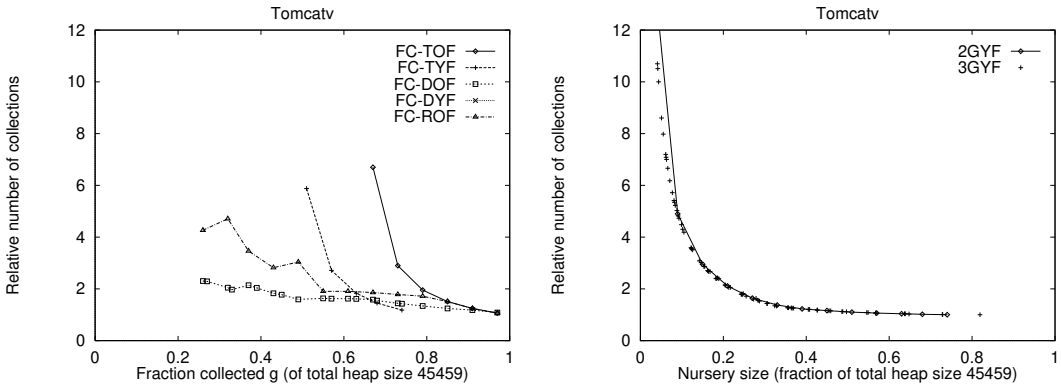
(a) FC schemes (b) GYF schemes

Figure 6.83. Relative invocation counts: Swim, $V = 104254$.



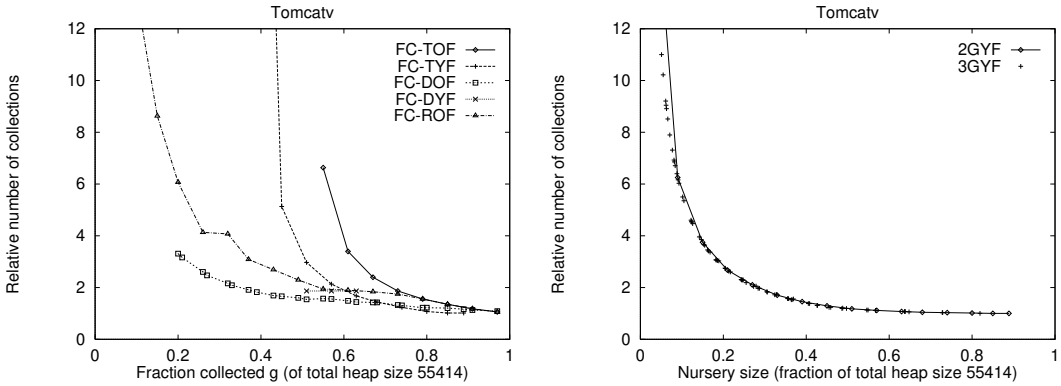
(a) FC schemes (b) GYF schemes

Figure 6.84. Relative invocation counts: Tomcatv, $V = 37292$.



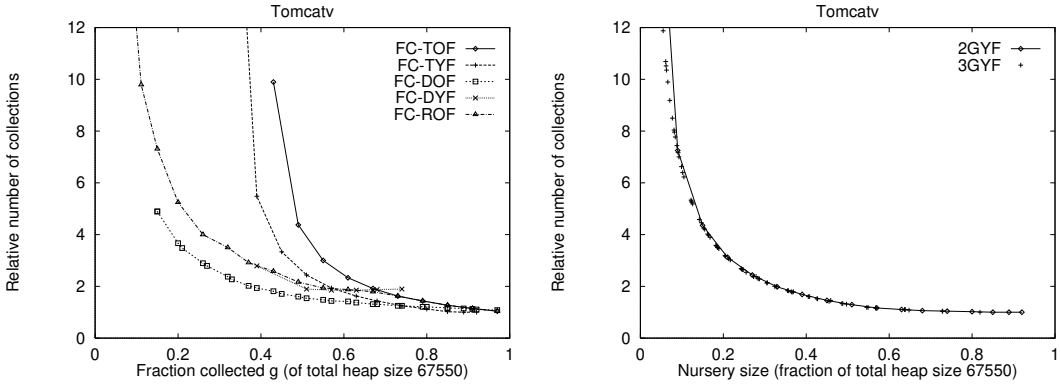
(a) FC schemes (b) GYF schemes

Figure 6.85. Relative invocation counts: Tomcatv, $V = 45459$.



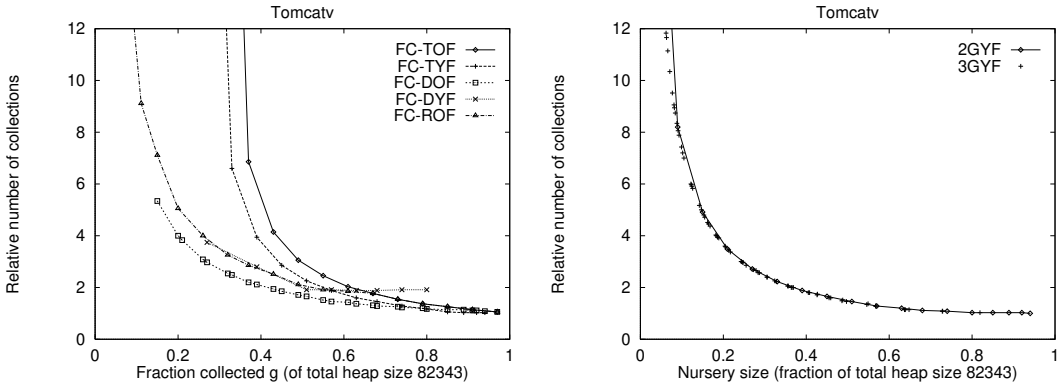
(a) FC schemes (b) GYF schemes

Figure 6.86. Relative invocation counts: Tomcatv, $V = 55414$.



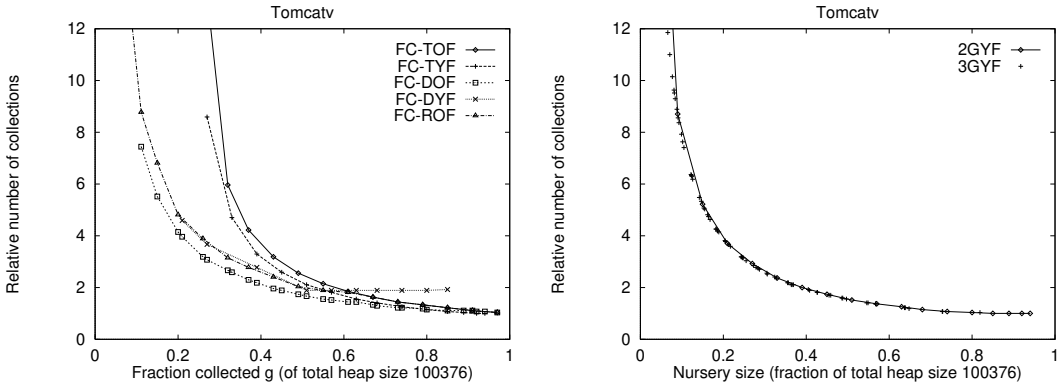
(a) FC schemes (b) GYF schemes

Figure 6.87. Relative invocation counts: Tomcatv, $V = 67550$.



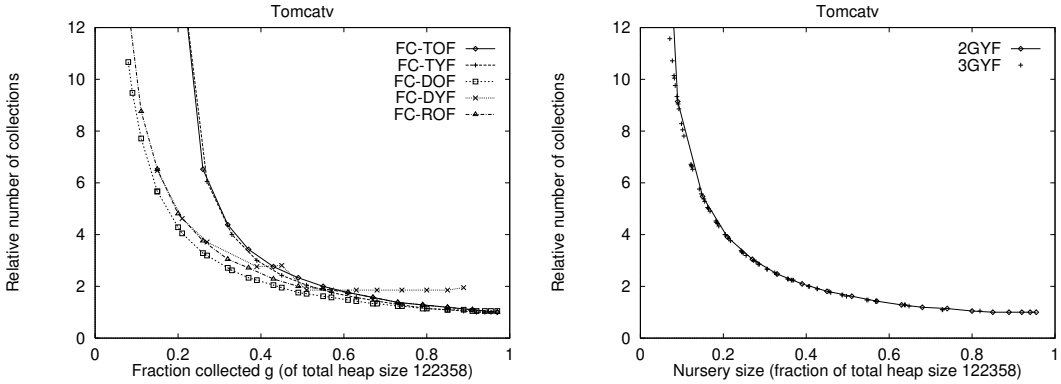
(a) FC schemes (b) GYF schemes

Figure 6.88. Relative invocation counts: Tomcatv, $V = 82343$.



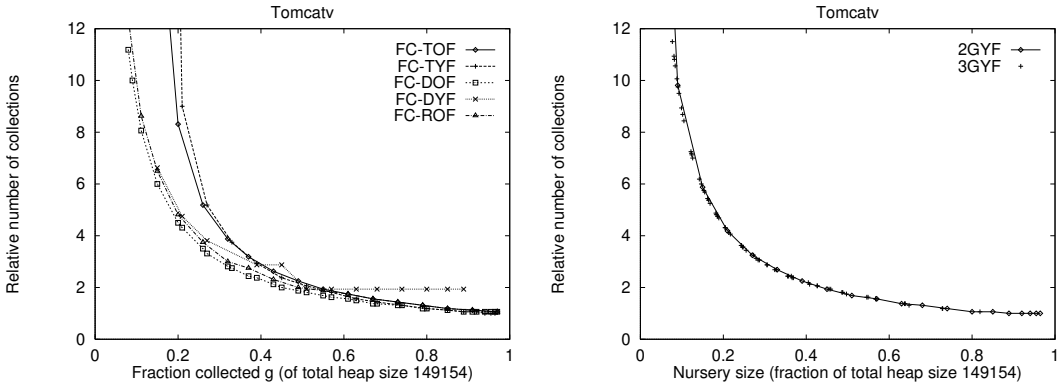
(a) FC schemes (b) GYF schemes

Figure 6.89. Relative invocation counts: Tomcatv, $V = 100376$.



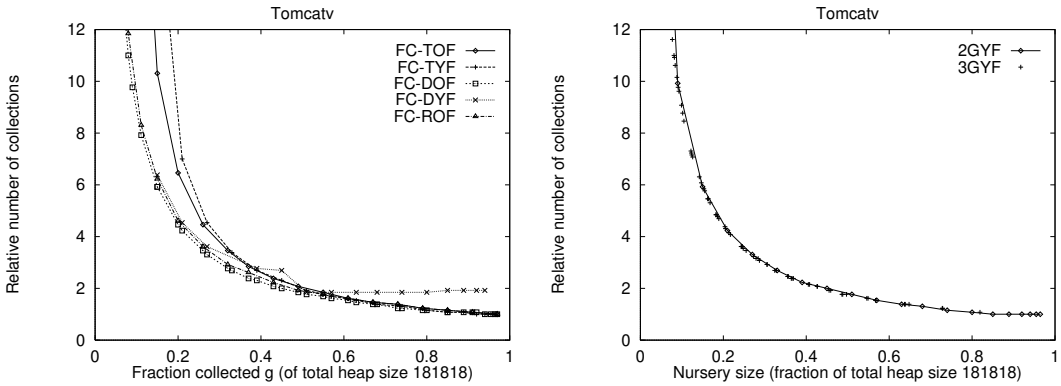
(a) FC schemes (b) GYF schemes

Figure 6.90. Relative invocation counts: Tomcatv, $V = 122358$.



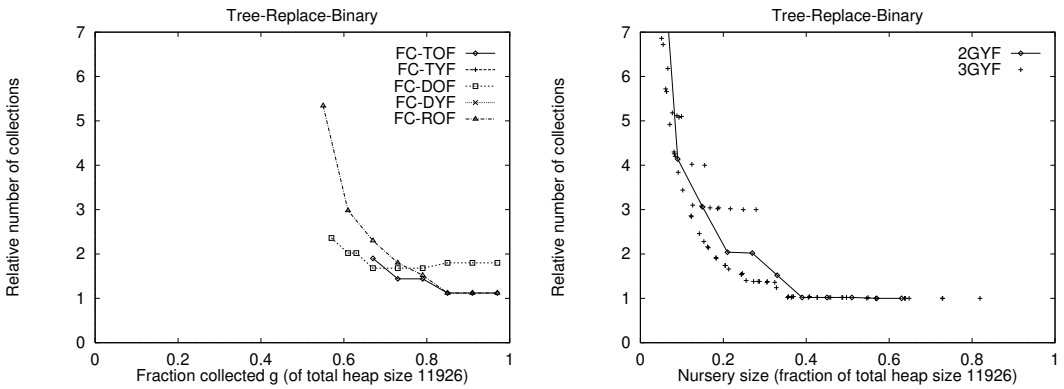
(a) FC schemes (b) GYF schemes

Figure 6.91. Relative invocation counts: Tomcatv, $V = 149154$.



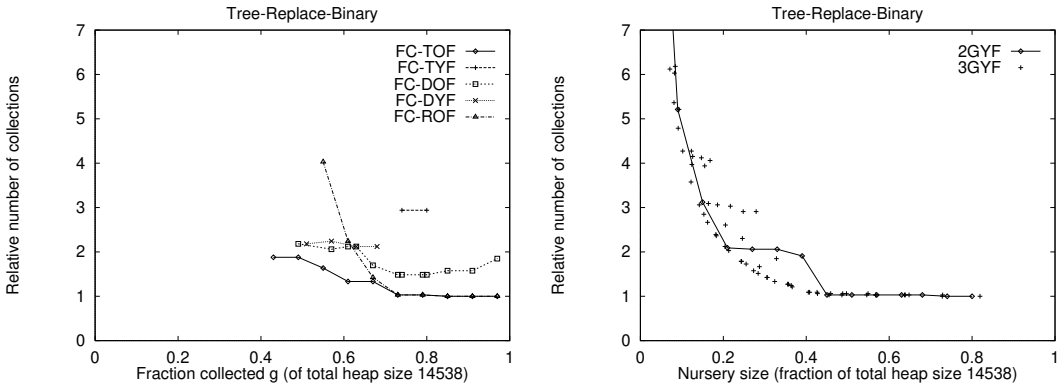
(a) FC schemes (b) GYF schemes

Figure 6.92. Relative invocation counts: Tomcatv, $V = 181818$.



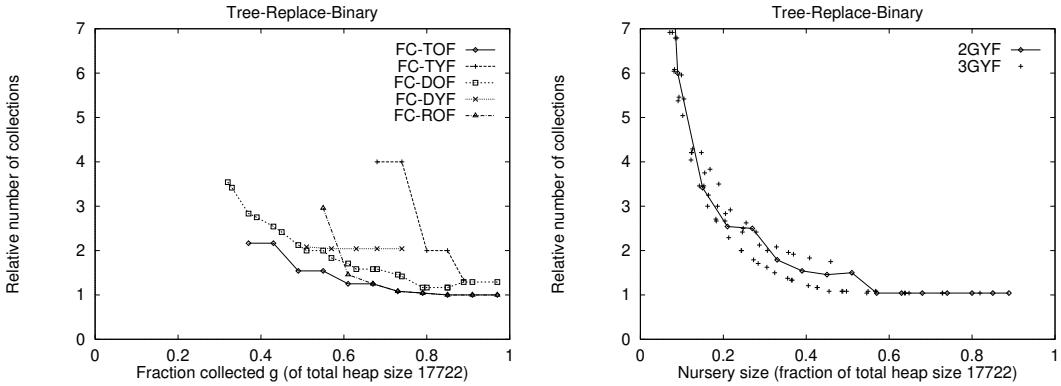
(a) FC schemes (b) GYF schemes

Figure 6.93. Relative invocation counts: Tree-Replace-Binary, $V = 11926$.



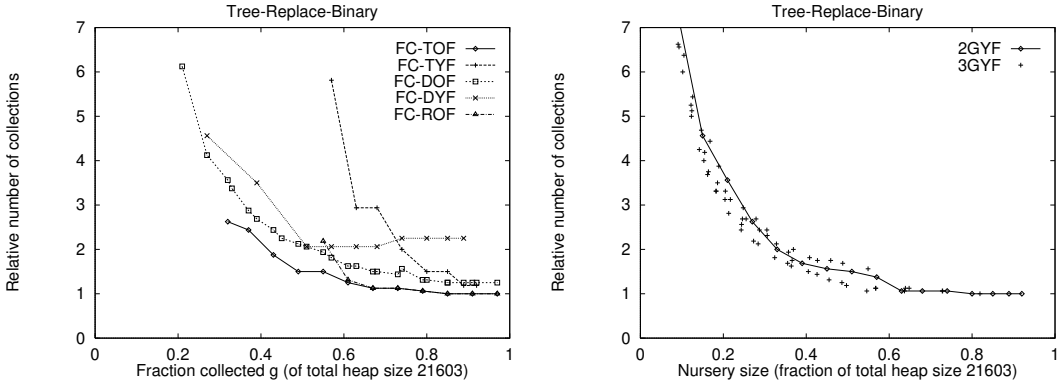
(a) FC schemes (b) GYF schemes

Figure 6.94. Relative invocation counts: Tree-Replace-Binary, $V = 14538$.



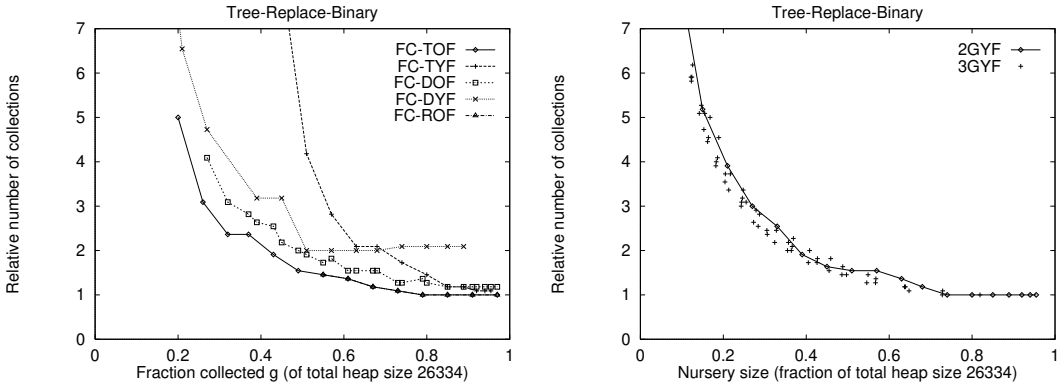
(a) FC schemes (b) GYF schemes

Figure 6.95. Relative invocation counts: Tree-Replace-Binary, $V = 17722$.



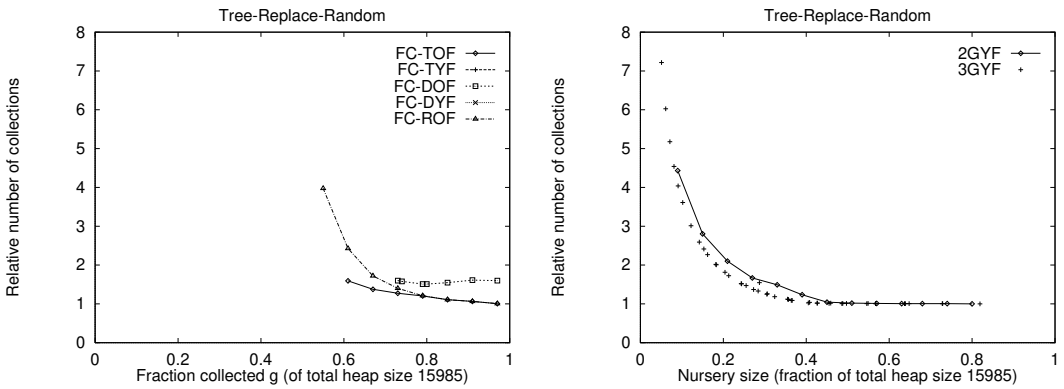
(a) FC schemes (b) GYF schemes

Figure 6.96. Relative invocation counts: Tree-Replace-Binary, $V = 21603$.



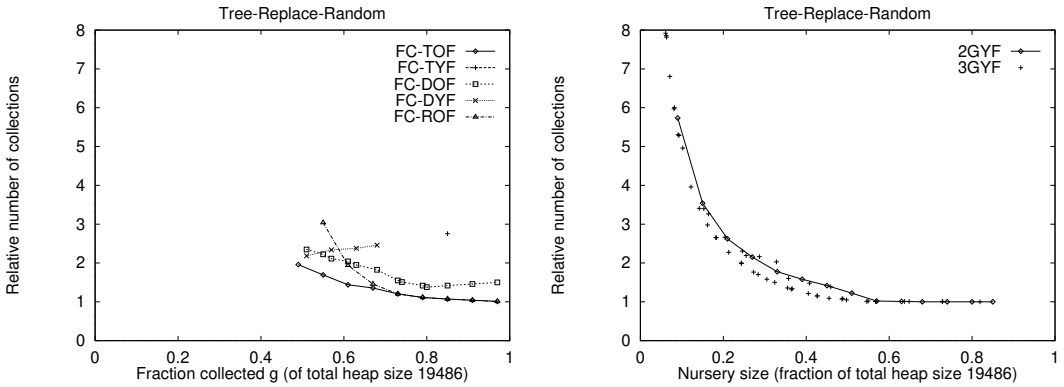
(a) FC schemes (b) GYF schemes

Figure 6.97. Relative invocation counts: Tree-Replace-Binary, $V = 26334$.



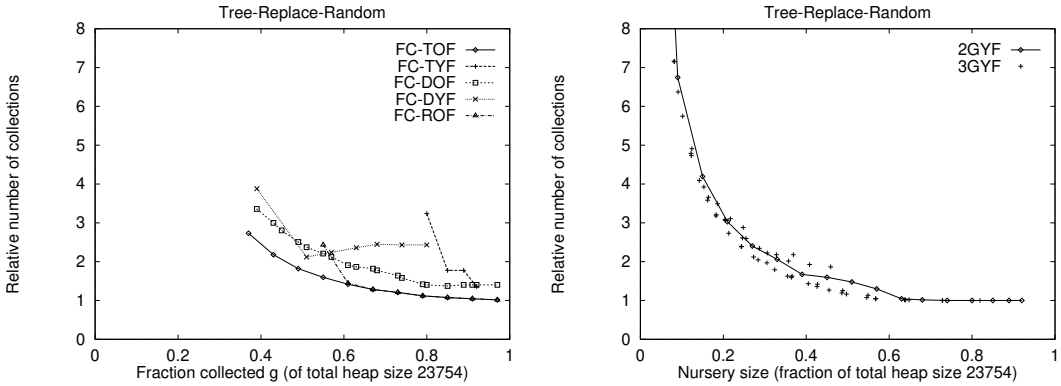
(a) FC schemes (b) GYF schemes

Figure 6.98. Relative invocation counts: Tree-Replace-Random, $V = 15985$.



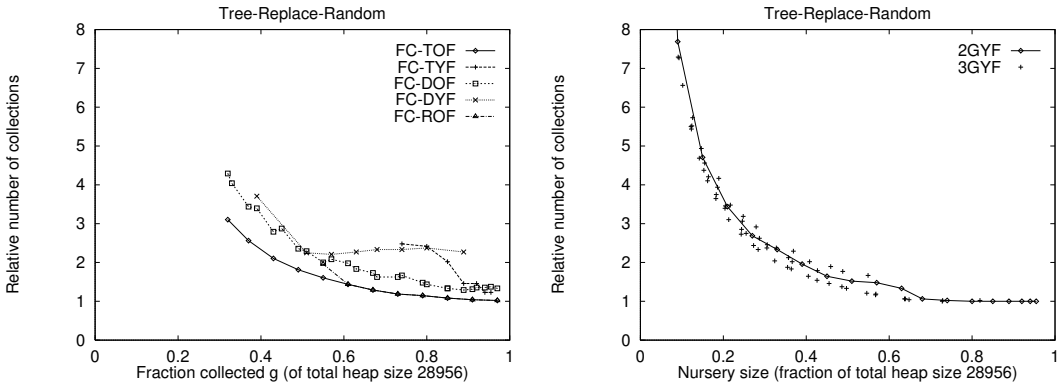
(a) FC schemes (b) GYF schemes

Figure 6.99. Relative invocation counts: Tree-Replace-Random, $V = 19486$.



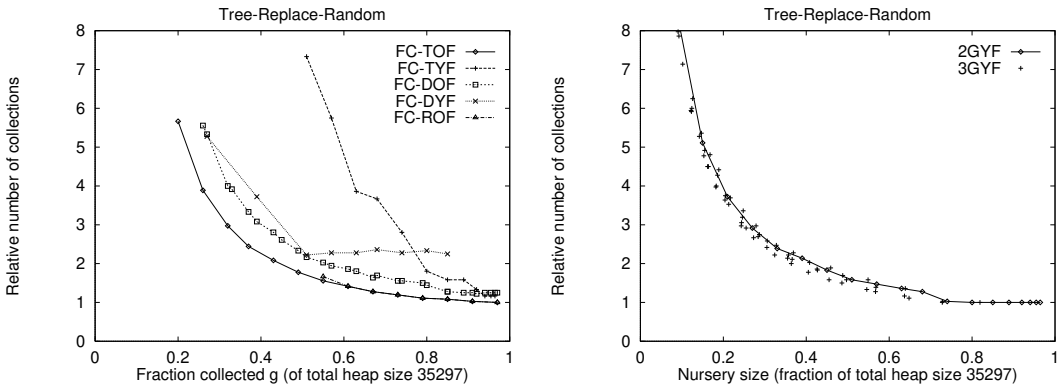
(a) FC schemes (b) GYF schemes

Figure 6.100. Relative invocation counts: Tree-Replace-Random, $V = 23754$.



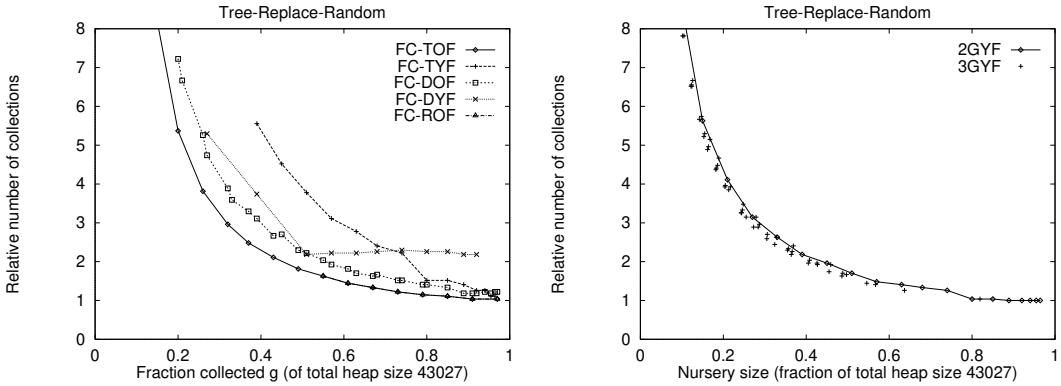
(a) FC schemes (b) GYF schemes

Figure 6.101. Relative invocation counts: Tree-Replace-Random, $V = 28956$.



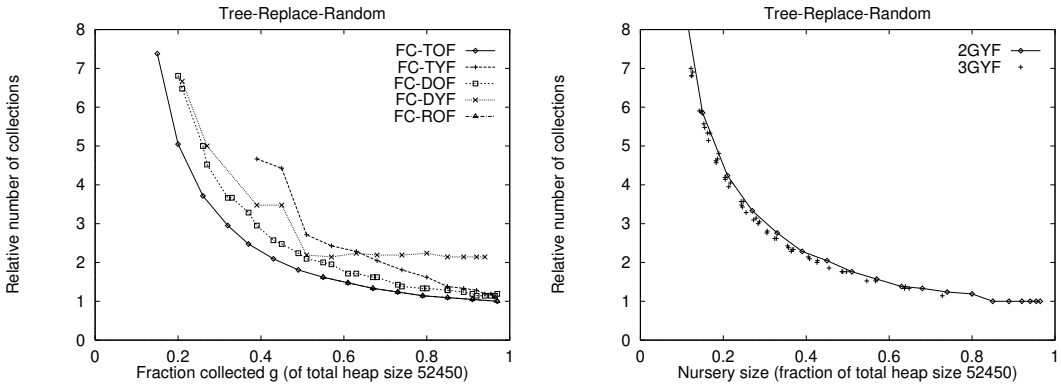
(a) FC schemes (b) GYF schemes

Figure 6.102. Relative invocation counts: Tree-Replace-Random, $V = 35297$.



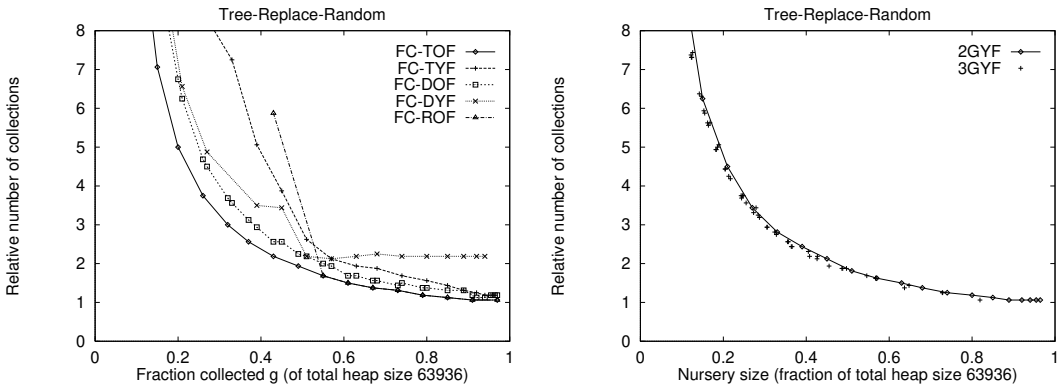
(a) FC schemes (b) GYF schemes

Figure 6.103. Relative invocation counts: Tree-Replace-Random, $V = 43027$.



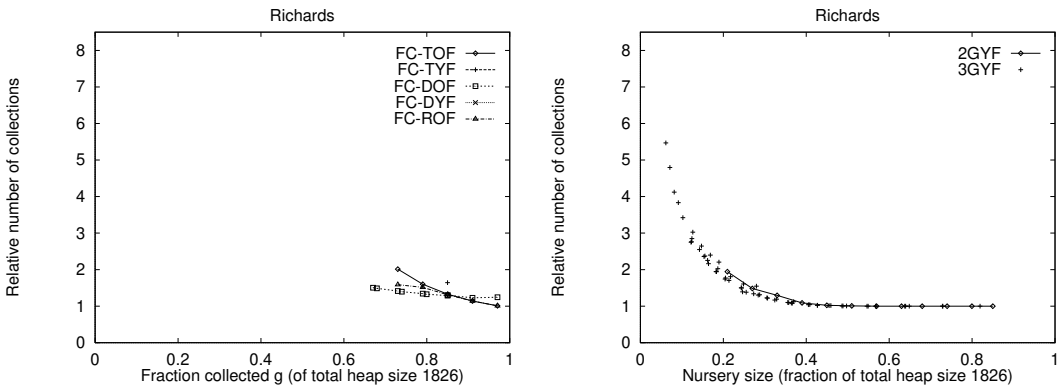
(a) FC schemes (b) GYF schemes

Figure 6.104. Relative invocation counts: Tree-Replace-Random, $V = 52450$.



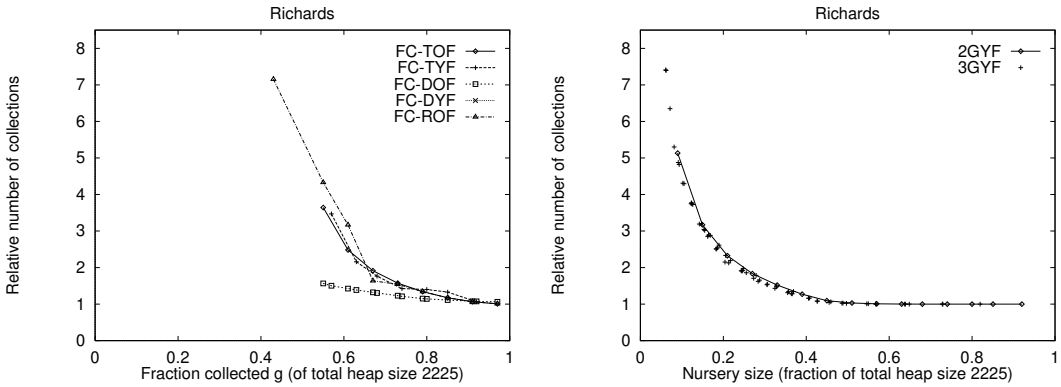
(a) FC schemes (b) GYF schemes

Figure 6.105. Relative invocation counts: Tree-Replace-Random, $V = 63936$.



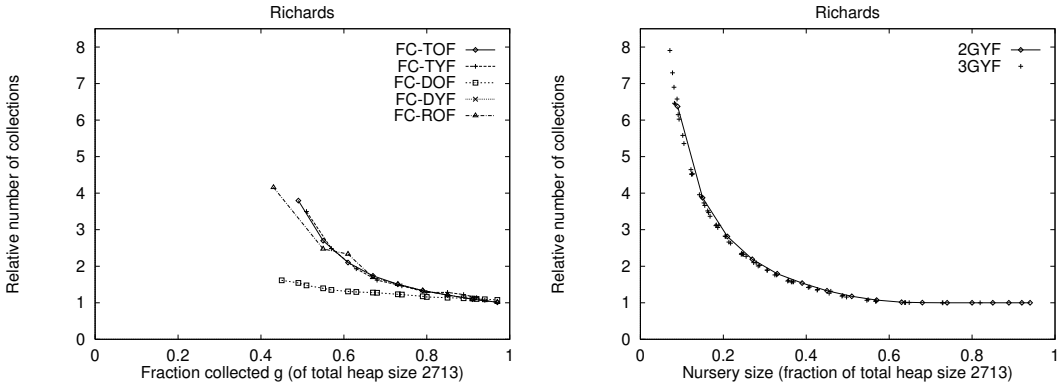
(a) FC schemes (b) GYF schemes

Figure 6.106. Relative invocation counts: Richards, $V = 1826$.



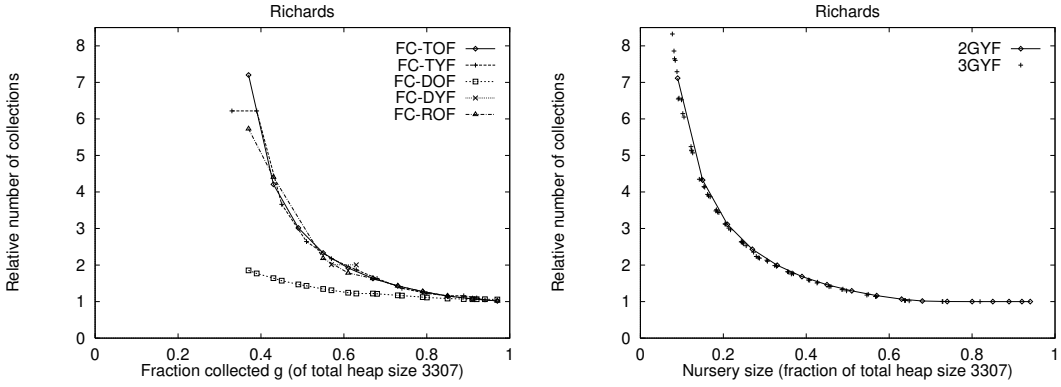
(a) FC schemes (b) GYF schemes

Figure 6.107. Relative invocation counts: Richards, $V = 2225$.



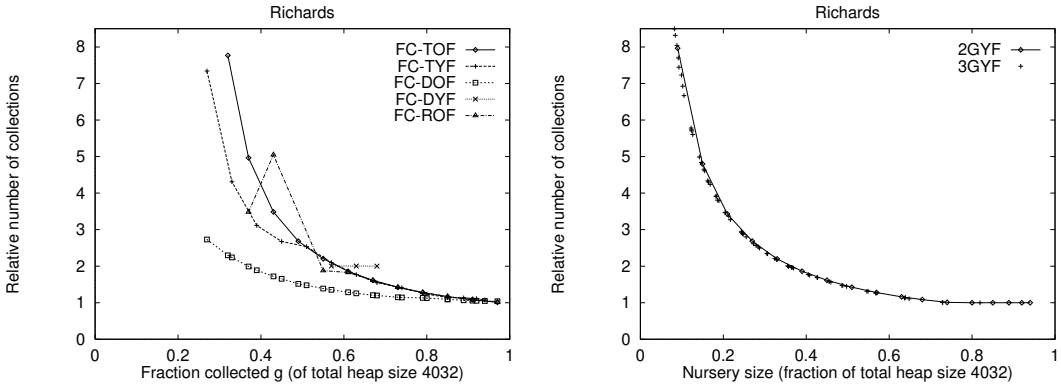
(a) FC schemes (b) GYF schemes

Figure 6.108. Relative invocation counts: Richards, $V = 2713$.



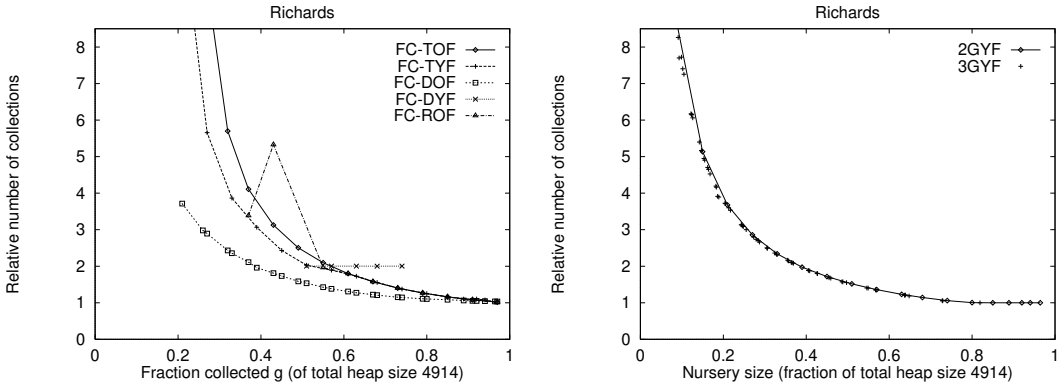
(a) FC schemes (b) GYF schemes

Figure 6.109. Relative invocation counts: Richards, $V = 3307$.



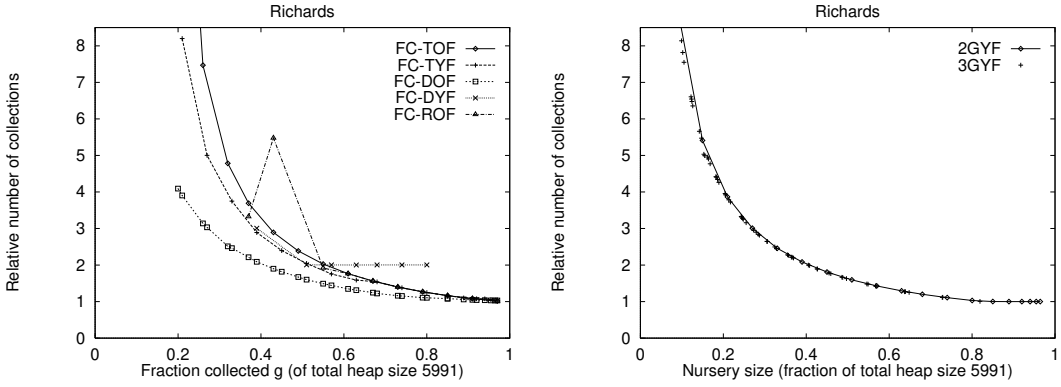
(a) FC schemes (b) GYF schemes

Figure 6.110. Relative invocation counts: Richards, $V = 4032$.



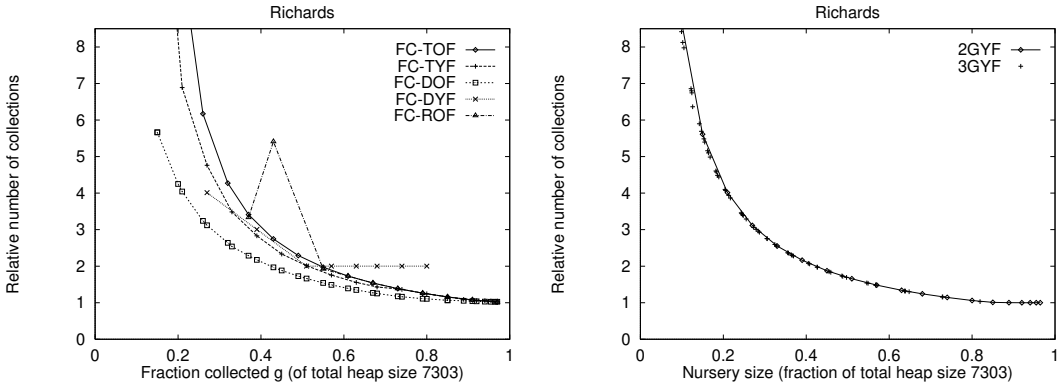
(a) FC schemes (b) GYF schemes

Figure 6.111. Relative invocation counts: Richards, $V = 4914$.



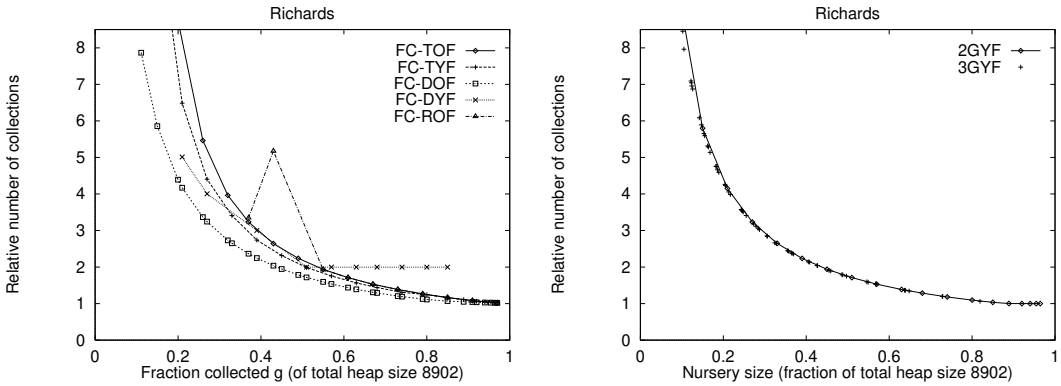
(a) FC schemes (b) GYF schemes

Figure 6.112. Relative invocation counts: Richards, $V = 5991$.



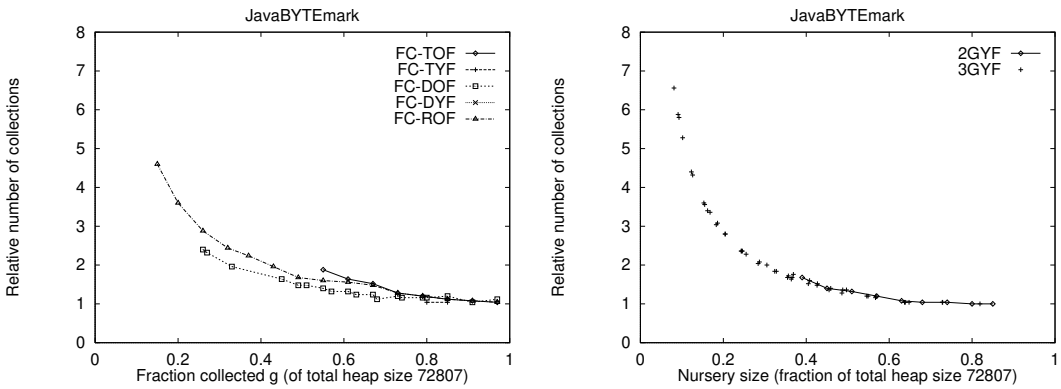
(a) FC schemes (b) GYF schemes

Figure 6.113. Relative invocation counts: Richards, $V = 7303$.



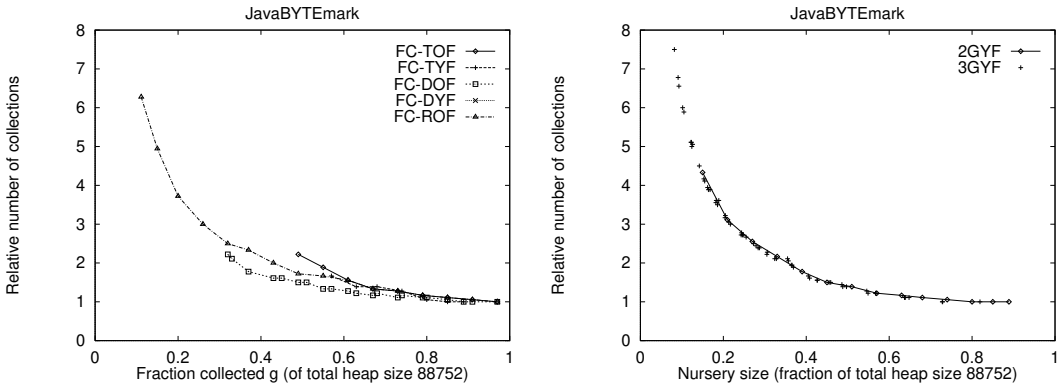
(a) FC schemes (b) GYF schemes

Figure 6.114. Relative invocation counts: Richards, $V = 8902$.



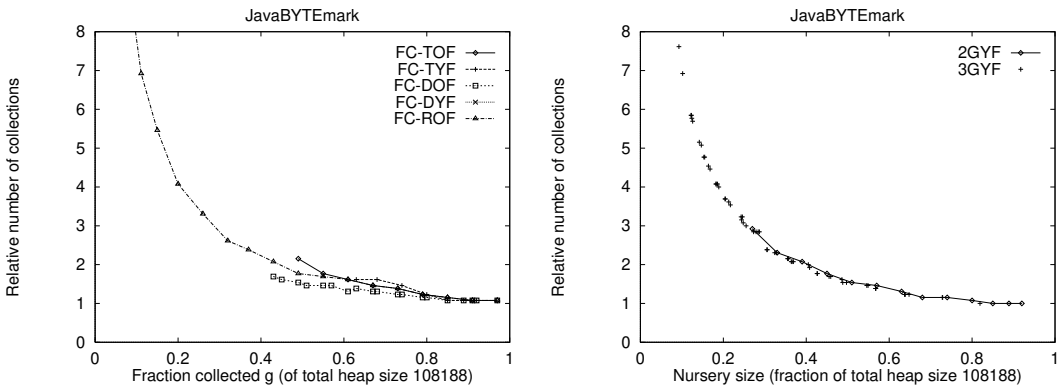
(a) FC schemes (b) GYF schemes

Figure 6.115. Relative invocation counts: JavaBYTEmark, $V = 72807$.



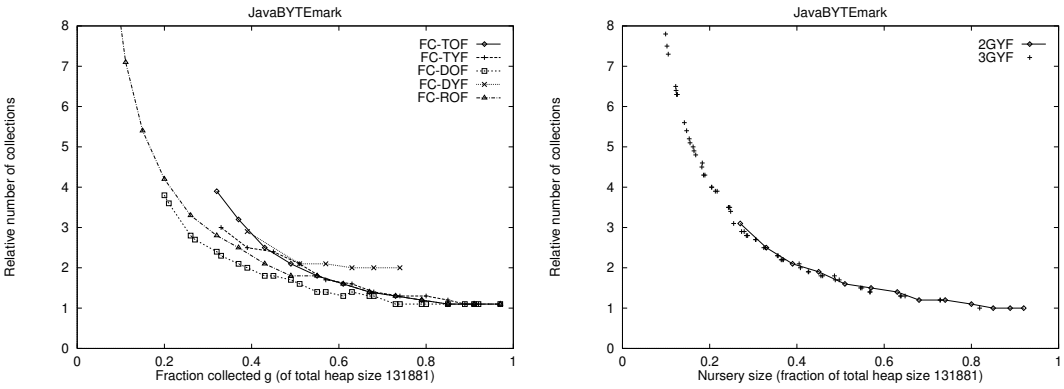
(a) FC schemes (b) GYF schemes

Figure 6.116. Relative invocation counts: JavaBYTEmark, $V = 88752$.



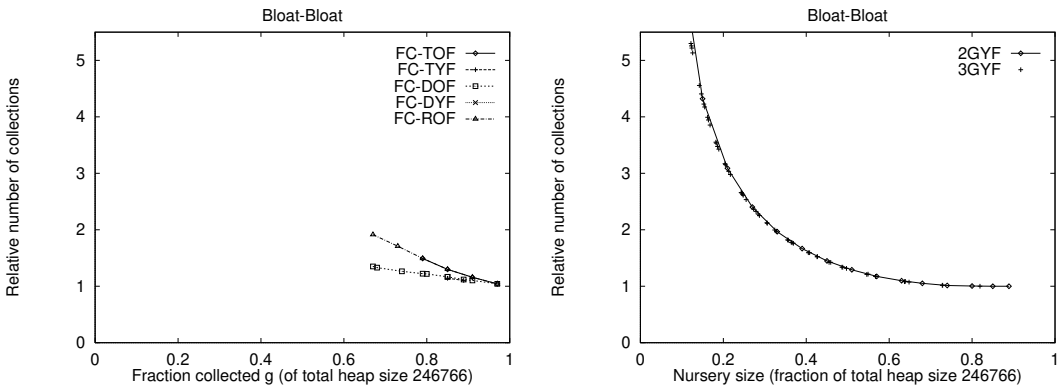
(a) FC schemes (b) GYF schemes

Figure 6.117. Relative invocation counts: JavaBYTEmark, $V = 108188$.



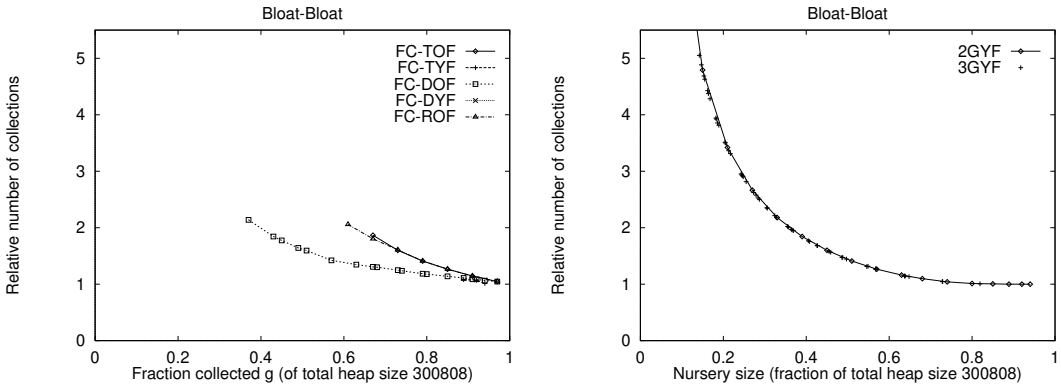
(a) FC schemes (b) GYF schemes

Figure 6.118. Relative invocation counts: JavaBYTEmark, $V = 131881$.



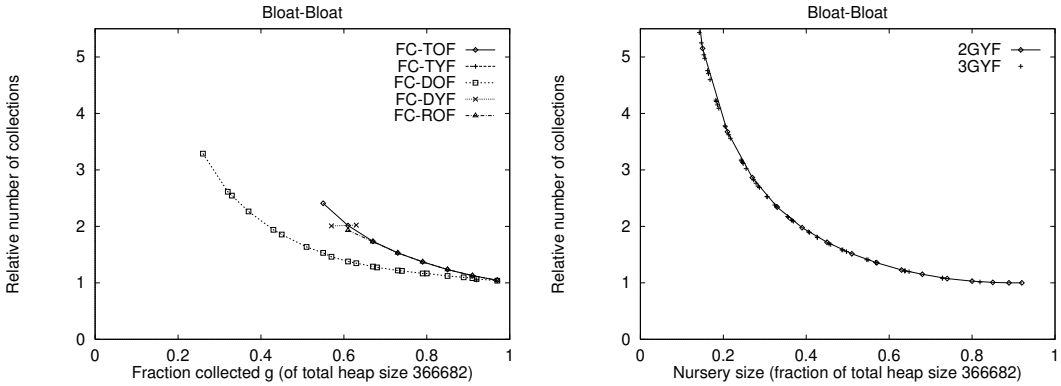
(a) FC schemes (b) GYF schemes

Figure 6.119. Relative invocation counts: Bloat-Bloat, $V = 246766$.



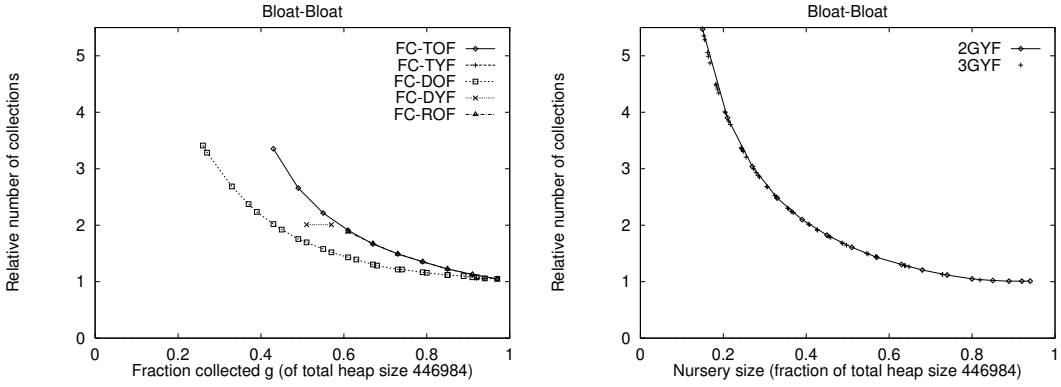
(a) FC schemes (b) GYF schemes

Figure 6.120. Relative invocation counts: Bloat-Bloat, $V = 300808$.



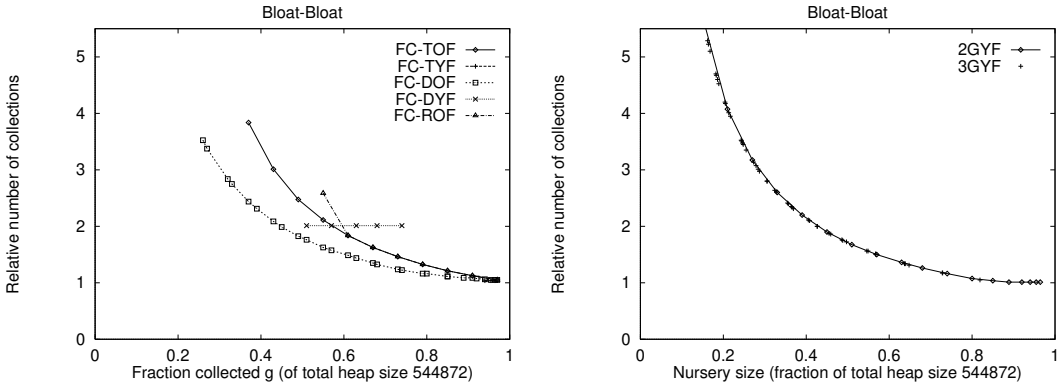
(a) FC schemes (b) GYF schemes

Figure 6.121. Relative invocation counts: Bloat-Bloat, $V = 366682$.



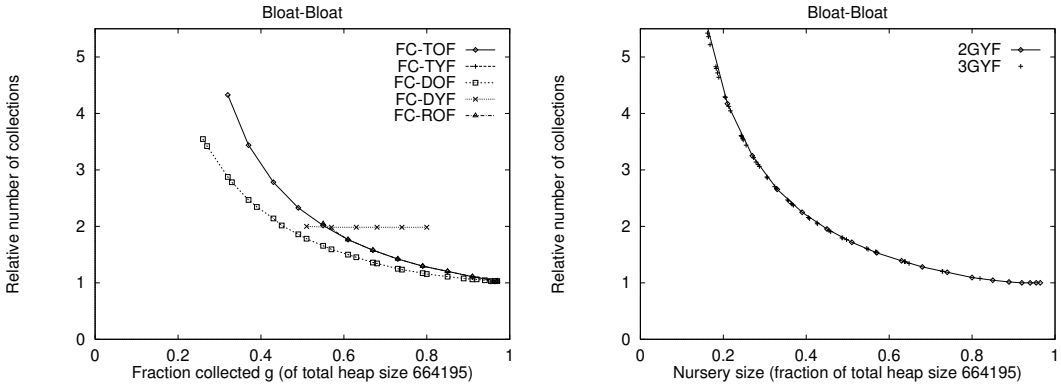
(a) FC schemes (b) GYF schemes

Figure 6.122. Relative invocation counts: Bloat-Bloat, $V = 446984$.



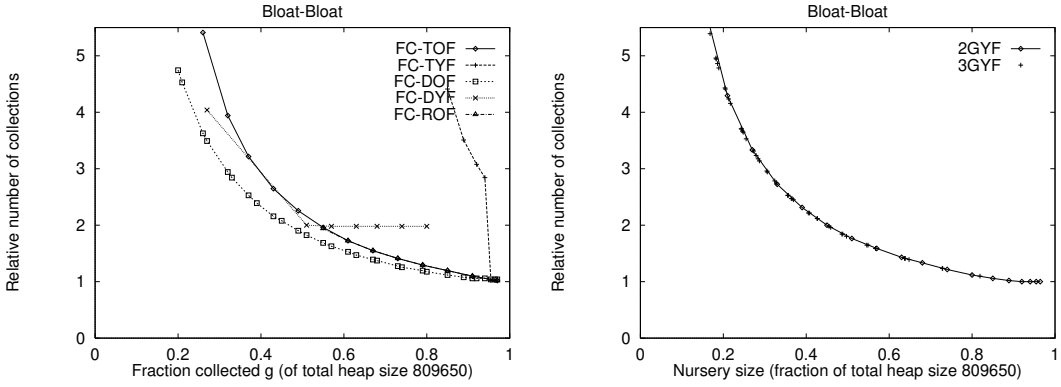
(a) FC schemes (b) GYF schemes

Figure 6.123. Relative invocation counts: Bloat-Bloat, $V = 544872$.



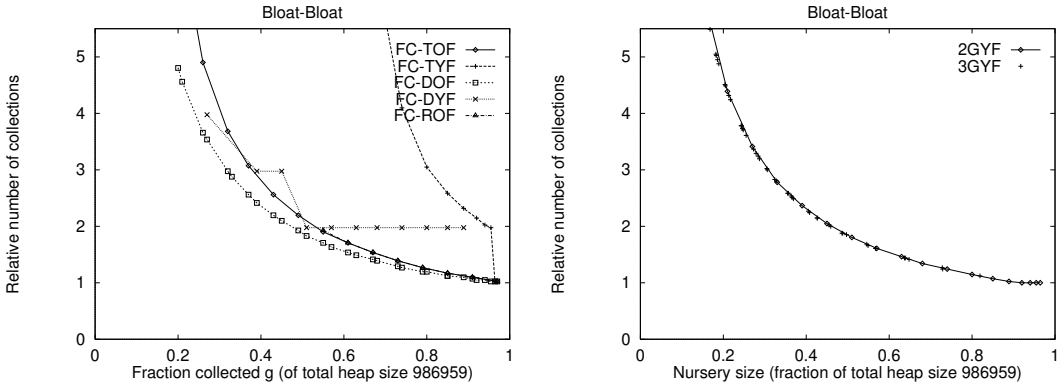
(a) FC schemes (b) GYF schemes

Figure 6.124. Relative invocation counts: Bloat-Bloat, $V = 664195$.



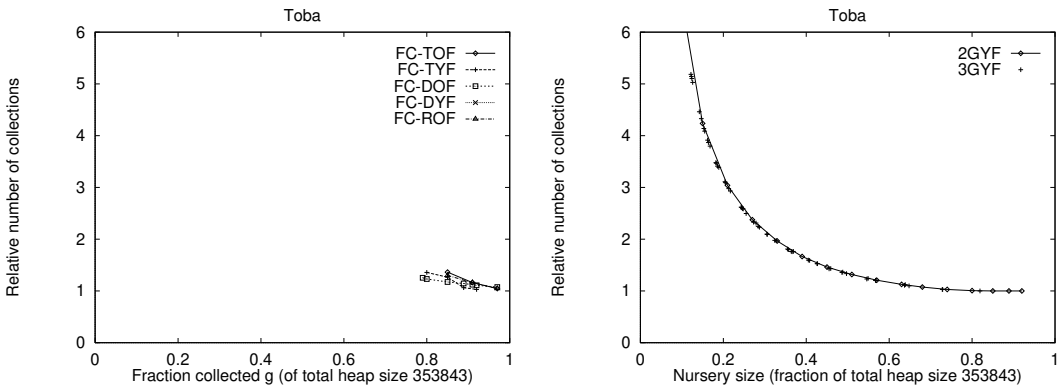
(a) FC schemes (b) GYF schemes

Figure 6.125. Relative invocation counts: Bloat-Bloat, $V = 809650$.



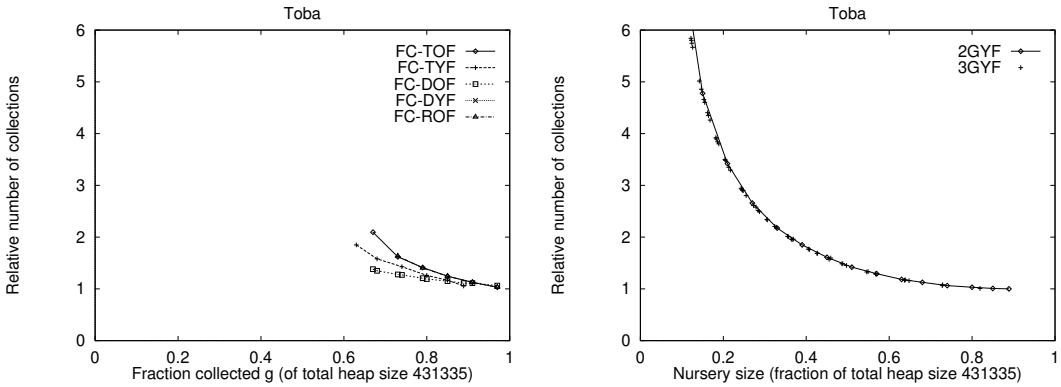
(a) FC schemes (b) GYF schemes

Figure 6.126. Relative invocation counts: Bloat-Bloat, $V = 986959$.



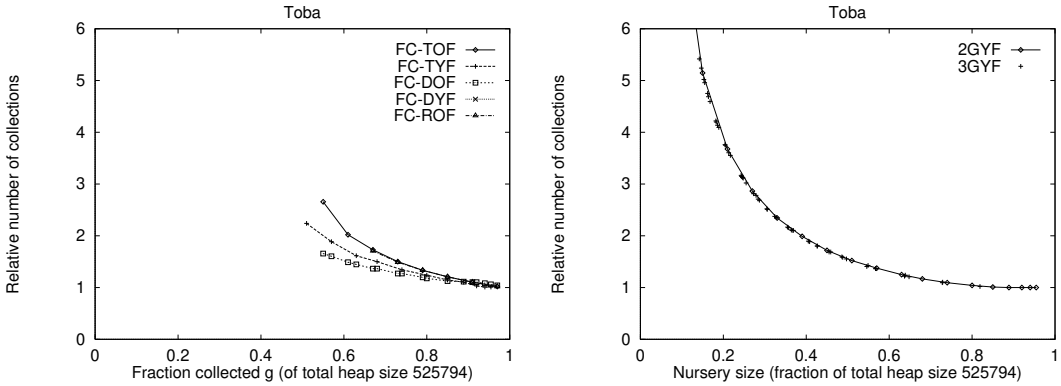
(a) FC schemes (b) GYF schemes

Figure 6.127. Relative invocation counts: Toba, $V = 353843$.



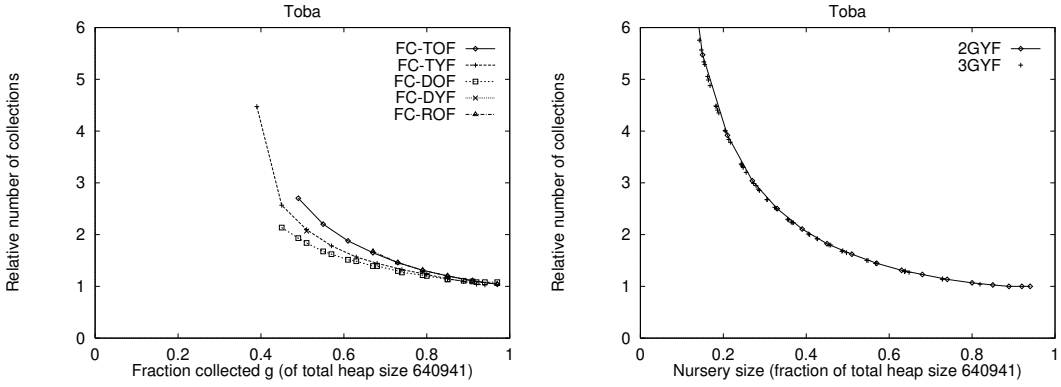
(a) FC schemes (b) GYF schemes

Figure 6.128. Relative invocation counts: Toba, $V = 431335$.



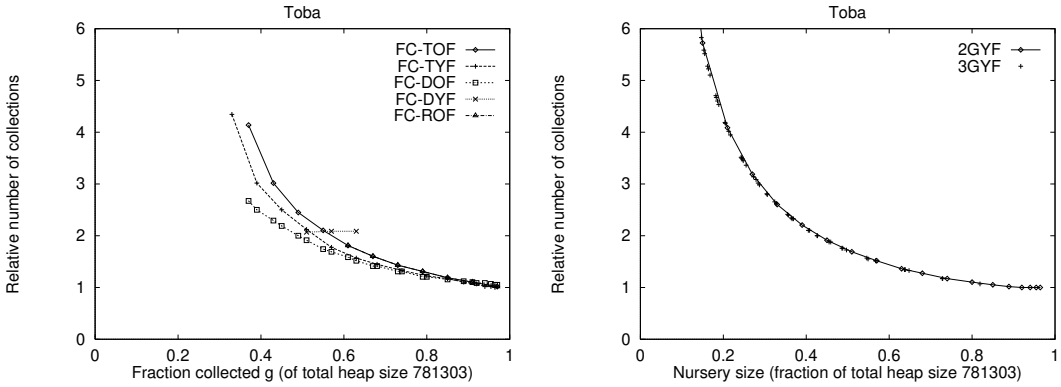
(a) FC schemes (b) GYF schemes

Figure 6.129. Relative invocation counts: Toba, $V = 525794$.



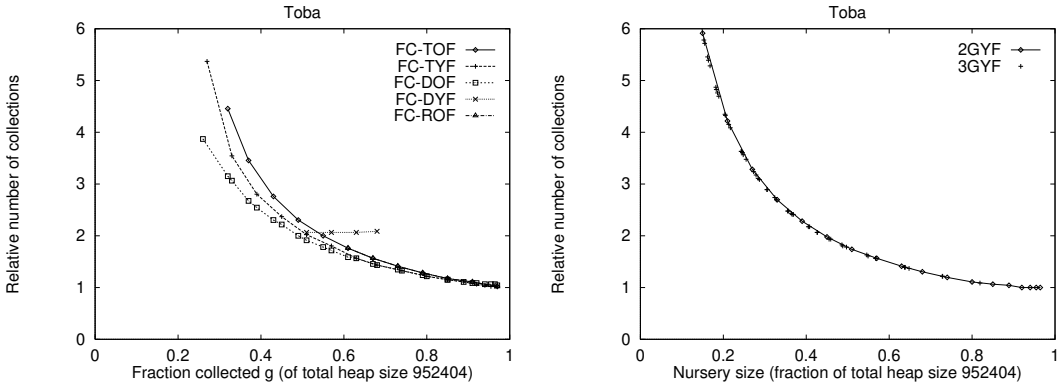
(a) FC schemes (b) GYF schemes

Figure 6.130. Relative invocation counts: Toba, $V = 640941$.



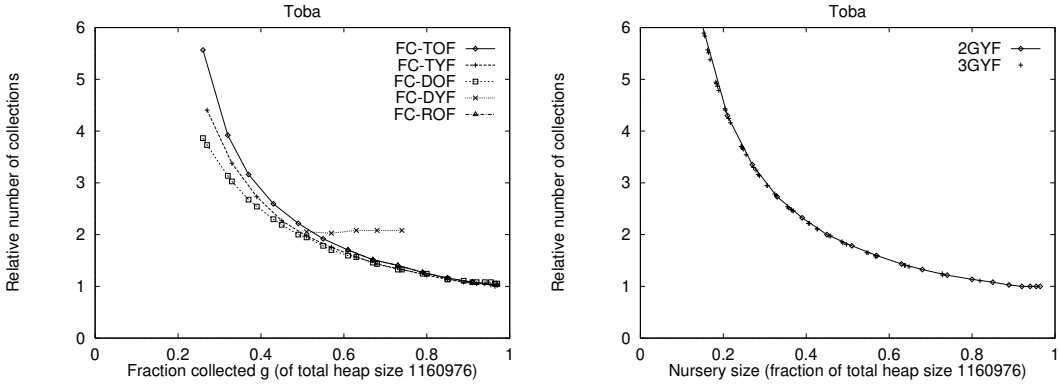
(a) FC schemes (b) GYF schemes

Figure 6.131. Relative invocation counts: Toba, $V = 781303$.



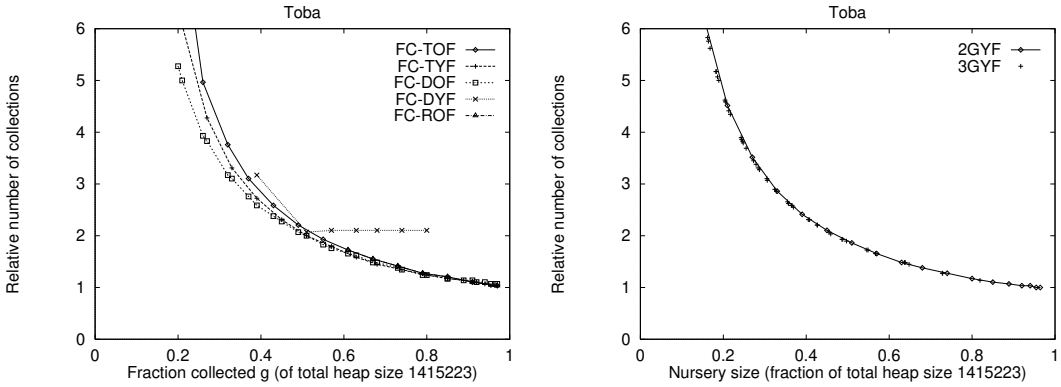
(a) FC schemes (b) GYF schemes

Figure 6.132. Relative invocation counts: Toba, $V = 952404$.



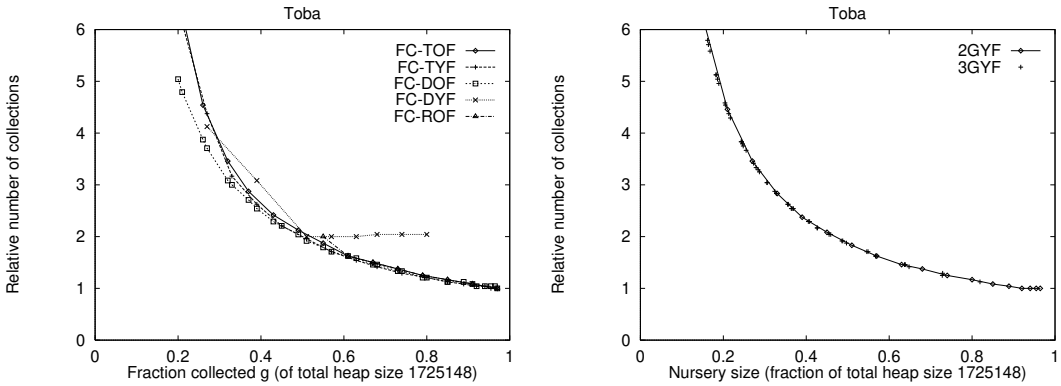
(a) FC schemes (b) GYF schemes

Figure 6.133. Relative invocation counts: Toba, $V = 1160976$.



(a) FC schemes (b) GYF schemes

Figure 6.134. Relative invocation counts: Toba, $V = 1415223$.



(a) FC schemes (b) GYF schemes

Figure 6.135. Relative invocation counts: Toba, $V = 1725148$.

CHAPTER 7

CONCLUDING REMARKS

In this study, we have explored a heretofore neglected alternative to generational garbage collection and found it to be competitive with generational algorithms, and in many cases superior to them. We introduced a simple new algorithm, called deferred older-first, within the classification of age-based algorithms. We described an efficient implementation of the new algorithm, its heap organization, copying mechanisms, and pointer-maintenance mechanisms (Chapter 3). We demonstrated, in a comparative trace-based simulation study, that the new algorithm achieves a significantly lower copying cost than generational collection on many programs, sometimes as much as 11 times lower (Chapter 5). We showed that, even though the pointer-maintenance cost of the new algorithm is usually higher than that of generational collection, the total cost is nevertheless lower on many programs, sometimes as much as 4 times (Chapter 6). These results were unexpected in light of the prior understanding of generational collector performance, which suggested that collecting among the youngest data achieves the best performance.

Using profiling of an age-ordered heap, our investigation provides insight into the copying performance of the new algorithm, and shows how it genuinely concentrates effort on those object ages, neither the youngest nor the oldest in the heap, which result in the lowest copying cost. Our new algorithm is very simple, and non-adaptive, yet its performance is good. While much work has been devoted to the design of adaptive generational algorithms, adaptive versions of our alternative, or indeed, other alternatives, are unexplored, but, according to our results, promising. Thus the field of garbage collection, even in the simplest setting of stop-and-copy uniprocessor algorithms, is reopened for study.

In our examination of the pointer-maintenance structures to support the demands of new algorithms, we found that precepts for the design of pointer maintenance from existing studies of generational collection do not carry over to alternative algorithms (for instance, the number of duplicate remembered set insertions is much greater in generational collection), and thus pointer maintenance design is also open for further study. Furthermore, the investigation of the positional distribution of pointers dispells the oft-cited notion that younger-to-older pointers dominate even in object-oriented languages.

This work has exposed the facts about the performance of copying garbage collection algorithms, the factors affecting it, and the limits of its possible improvement. However, the settings in which the experimental results were obtained were necessarily limited, and other settings remain to be explored.

We used trace-based simulation, albeit in conjunction with a prototype implementation of the collectors. However, traces do not capture stack processing costs nor the interactions between the collector and the mutator. A desirable follow-up study is to integrate the several collectors we described with a running mutator system, such as a Java virtual machine. This integration will permit the verification of the design we outlined for implementation, the validation of the cost models we assumed (Chapter 3), the measurement of memory locality effects, and the final verdict in the form of absolute timing of different algorithms.

In this study, we eschewed analytical modelling of garbage collection costs. Modelling the actions of a collector presupposes the availability of analytical models for the behavior of the mutator. Whereas some models have been attempted for object lifetimes, with unclear utility, models for pointer structure are entirely unexplored. The success of the deferred older-first collector as an approximation of the ideal queue collector (Chapter 1), the heap profiles (Chapter 5), and the pointer distributions (Chapter 6) indicate a high degree of spatio-temporal correlation of object lifetimes and of pointer structure, suggesting that first-order statistical models will be inadequate.

Our investigation focused on the application of collectors to the entire heap, from the most recently allocated objects to the oldest ones. As we discovered in the study of demise positions as well as of pointer directions, and entirely in accordance with intuitive expectations, an advantage is possible in large and long-running systems if an *a priori* division is imposed on the heap, in which the most recently allocated data are managed separately to lower the pointer maintenance overhead for stores into new objects, the oldest data are identified as permanent (by programmer declaration, or adaptively) to exclude them from consideration entirely, and the middle portion, representing an active mature space, is managed by a deferred-older-first algorithm. Indeed, this observation can be seen as the basis for defining policies on top of the mechanisms described for the Mature Object Space and its *train* algorithm [Hudson and Moss, 1992; Seligmann and Grarup, 1995].

The evaluation of different algorithms applied to a mature space could, in principle, proceed as in this study. The trace of mature space objects can be obtained either as the sequence of objects promoted out of a concrete younger-space collector, or, more abstractly, as the subsequence of objects with lifetimes above a fixed threshold extracted from a full sequence of allocated objects. Since the volume of objects presented to the mature space collector is, by design, much smaller than the total volume of allocation, then, in order for the mature space collector to be exercised sufficiently (Section 4.2.2.2) the total allocation must be considerably larger. We believe that it will prove infeasible to obtain accurate traces as for this study, and that direct implementation and measurement will be especially needed for the mature space.

BIBLIOGRAPHY

[Abadi and Cardelli, 1996] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[Aditya *et al.*, 1994] Shail Aditya, Christine Flood, and James Hicks. Garbage collection for strongly-typed languages using run-time type reconstruction. In *Proceedings of SIGPLAN'94 Conference on Programming Languages Design and Implementation*, volume 29 of *SIGPLAN Notices*, pages 12–23, Orlando, Florida, June 1994. ACM Press. Also Lisp Pointers VIII 3, July–September 1994.

[Agesen *et al.*, 1998] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java(TM) virtual machines. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, volume 33 of *SIGPLAN Notices*, pages 269–279, Montreal, Québec, Canada, June 1998. ACM Press.

[Appel and Shao, 1996] Andrew W. Appel and Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, January 1996.

[Appel, 1987] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[Appel, 1989a] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.

[Appel, 1989b] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[Appel, 1992] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, 1992.

[Appel, 1997] Andrew Appel. A better analytical model for the strong generational hypothesis. November 1997.

[Baker, 1978] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[Baker, 1990] Henry G. Baker. Unify and conquer (garbage, updating, aliasing, ...) in functional languages. In LFP [1990], pages 218–226.

[Baker, 1993] Henry G. Baker. 'Infant Mortality' and generational garbage collection. *SIGPLAN Notices*, 28(4):55–57, 1993.

[Barendregt, 1984] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier (North-Holland), Amsterdam, 1984. Second edition.

[Barrett and Zorn, 1995] David A. Barrett and Benjamin Zorn. Garbage collection using a dynamic threatening boundary. In *Proceedings of SIGPLAN'95 Conference on Programming Languages Design and Implementation*, volume 30 of *SIGPLAN Notices*, pages 301–314, La Jolla, CA, June 1995. ACM Press.

[Bekkers and Cohen, 1992] Yves Bekkers and Jacques Cohen, editors. *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, St. Malo, France, September 1992. Springer-Verlag.

[Bekkers *et al.*, 1986] Yves Bekkers, B. Canet, Olivier Ridoux, and L. Ungaro. MALI: A memory with a real-time garbage collector for implementing logic programming languages. In *3rd Symposium on Logic Programming*, pages 258–264. IEEE Press, 1986.

[Bishop, 1977] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1977.

[Boehm and Shao, 1993] Hans-Juergen Boehm and Zhong Shao. Inferring type maps during garbage collection. In Moss et al. [1993].

[Boehm and Weiser, 1988] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[Boehm, 1993] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *SIGPLAN Notices*, pages 197–206, Albuquerque, New Mexico, June 1993. ACM Press.

[Caudill and Wirfs-Brock, 1986] Patrick J. Caudill and Allen Wirfs-Brock. A third generation Smalltalk-80 implementation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 119–130, Portland, Oregon, September 1986. *SIGPLAN Notices 21*, 11 (November 1986).

[Chambers, 1992] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for an Objected-Oriented Programming Language*. PhD thesis, Stanford University, March 1992.

[Cheney, 1970] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[Clinger and Hansen, 1997] William D. Clinger and Lars T. Hansen. Generational garbage collection and the radioactive decay model. *SIGPLAN Notices*, 32(5):97–108, May 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*.

[Cohen, 1981] Jacques Cohen. Garbage collection of linked data structures. *Computing Surveys*, 13(3):341–367, September 1981.

[Courts, 1988] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, September 1988.

[Cox and Oakes, 1984] D. R. Cox and D. Oakes. *Analysis of Survival Data*. Chapman and Hall, London, 1984.

[Demers *et al.*, 1990] Alan Demers, Mark Weiser, Barry Hayes, Daniel G. Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, pages 261–269, San Francisco, CA, January 1990. ACM Press.

[DeTreville, 1990a] John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, DEC Systems Research Center, Palo Alto, CA, August 1990.

[DeTreville, 1990b] John DeTreville. Heap usage in the Topaz environment. Technical Report 63, DEC Systems Research Center, Palo Alto, CA, August 1990.

[Diwan *et al.*, 1992] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of SIGPLAN'92 Conference on Programming Languages Design and Implementation*, volume 27 of *SIGPLAN Notices*, pages 273–282, San Francisco, CA, June 1992. ACM Press.

[Diwan *et al.*, 1995] Amer Diwan, David Tarditi, and J. Eliot B. Moss. Memory system performance of programs with intensive heap allocation. *ACM Transactions on Computer Systems*, 13(3):244–273, August 1995.

[Elandt-Johnson and Johnson, 1980] Regina C. Elandt-Johnson and Norman Lloyd Johnson. *Survival Models and Data Analysis*. Wiley, New York, 1980.

[Fenichel and Yochelson, 1969] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[Fradet, 1994] Pascal Fradet. Collecting more garbage. In LFP [1994].

[Goldberg and Gloger, 1992] Benjamin Goldberg and Michael Gloger. Polymorphic type reconstruction for garbage collection without tags. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming*, pages 53–65, San Francisco, CA, June 1992. ACM Press.

[Goldberg and Robson, 1983] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Gonçalves, 1995] Marcelo J. R. Gonçalves. *Cache Performance of Programs with Intensive Heap Allocation and Generational Garbage Collection*. PhD thesis, Princeton University, May 1995.

[Gosling *et al.*, 1996] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading, Mass., 1996.

[Grarup and Seligmann, 1993] Steffen Grarup and Jacob Seligmann. Incremental mature garbage collection. Technical report, Computer Science Department, Aarhus University, Aarhus, Denmark, August 1993. M.Sc. Thesis.

[Harrison, 1989] Williams Ludwell Harrison, III. The Interprocedural Analysis and Automatic Parallelization of Scheme Programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, October 1989.

[Hayes, 1991] Barry Hayes. Using key object opportunism to collect old objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 33–46, Phoenix, Arizona, October 1991. *SIGPLAN Notices 26*, 11 (November 1991).

[Hayes, 1993] Barry Hayes. *Key Objects in Garbage Collection*. PhD thesis, Stanford University, Stanford, California, March 1993.

[Hicks *et al.*, 1998] Michael Hicks, Luke Hornof, Jonathan T. Moore, and Scott M. Nettles. A study of large object spaces. In *International Symposium on Memory Management*, pages 138–145, Vancouver, British Columbia, Canada, October 1998. ACM Press.

[Hosking and Hudson, 1993] Antony L. Hosking and Richard L. Hudson. Remembered sets can also play cards. In Moss et al. [1993].

[Hosking *et al.*, 1992] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada, October 1992. *SIGPLAN Notices 27*, 10 (October 1992).

[Hudson and Moss, 1992] Richard L. Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In Bekkers and Cohen [1992], pages 388–403.

[Hudson *et al.*, 1991] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. COINS Technical Report 91-47, University of Massachusetts, Amherst, September 1991.

[Jones and Lins, 1996] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley, Chichester, 1996.

[Kolmogorov and Uspenskii, 1958] A. N. Kolmogorov and V. A. Uspenskii. On the definition of an algorithm. *Uspekhi Mat. Nauk*, 13:3–28, 1958. English translation in: *Russian Math. Surveys* **30** (1963) 217–245.

[Lang and Dupont, 1987] Bernard Lang and Francis Dupont. Incremental incrementally compacting garbage collection. In *SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, volume 22(7) of *SIGPLAN Notices*, pages 253–263, Inst. Rech. Informat. Automat. Lab., France, 1987. ACM Press.

[LFP, 1990] *Conference Record of the 1990 ACM Symposium on Lisp and Functional Programming*, Nice, France, June 1990. ACM Press.

[LFP, 1994] *1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, June 1994. ACM Press.

[Lieberman and Hewitt, 1983] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[McCarthy, 1960] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[Minsky, 1963] Marvin L. Minsky. A Lisp garbage collector algorithm using serial secondary storage. Technical Report Memo 58 (rev.), Project MAC, MIT, Cambridge, MA, December 1963.

[Mohnen, 1995] Markus Mohnen. Efficient compile-time garbage collection for arbitrary data structures. Technical Report 95–08, University of Aachen, May 1995. Also in Seventh International Symposium on Programming Languages, Implementations, Logics and Programs, PLILP95.

[Moon, 1984] David Moon. Garbage collection in a large Lisp system. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 235–246, Austin, TX, August 1984.

[Morrisett *et al.*, 1995] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Functional Programming and Computer Architecture*, June 1995.

[Moss *et al.*, 1993] Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors. *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[Moss, 1987] J. Eliot B. Moss. Managing stack frames in Smalltalk. In *Proceedings of the ACM SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 229–240, St. Paul, Minnesota, July 1987. *SIGPLAN Notices 22*, 7 (July 1987).

[Nystrom, 1998] Nathaniel Nystrom. Bytecode-level analysis and optimization of Java class files. Master's thesis, Purdue University, West Lafayette, IN, May 1998.

[Odersky and Wadler, 1997] Martin Odersky and Philip Wadler. Pizza into Java: translating theory into practice. In *POPL '97. Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming*, pages 146–159, New York, 1997. ACM Press.

[Proebsting *et al.*, 1998] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications, a way ahead of time (WAT) compiler. (at http://www.cs.arizona.edu/sumatra/toba/), 1998.

[Reinhold, 1993] Mark B. Reinhold. *Cache performance of garbage-collected programming languages*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, September 1993.

[Röjemo and Runciman, 1996] Niklas Röjemo and Colin Runciman. Lag, drag, void, and use: heap profiling and space-efficient compilation revisited. In *Proceedings of SIGPLAN'96 Conference on Programming Languages Design and Implementation*, volume 31 of *SIGPLAN Notices*, pages 34–41. ACM Press, 1996.

[Runciman and Röjemo, 1995] Colin Runciman and Niklas Röjemo. Lag, drag and post-mortem heap profiling. In *Implementation of Functional Languages Workshop*, Båstad, Sweden, September 1995.

[Runciman and Wakeling, 1992] Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. Technical Report 172, University of York, Dept. of Computer Science, Heslington, York Y01 5DD, United Kingdom, April 1992.

[Sansom and Peyton Jones, 1994] Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. Technical Report Research Report FP-1994-10, Dept. of Computing Science, University of Glasgow, Glasgow, Scotland, 1994.

[Sansom, 1994] Patrick M. Sansom. *Execution Profiling for Non-strict Functional Languages*. PhD thesis, University of Glasgow, Glasgow, Scotland, April 1994.

[Schönhage, 1980] A. Schönhage. Storage modification machines. *SIAM Journal on Computing*, 9:490–508, 1980.

[Seligmann and Grarup, 1995] Jacob Seligmann and Steffen Grarup. Incremental mature garbage collection using the train algorithm. In O. Nierstras, editor, *Proceedings of 1995 European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pages 235–252, University of Aarhus, August 1995. Springer-Verlag.

[Shaw, 1988] Robert A. Shaw. *Empirical Analysis of a LISP System*. PhD thesis, Stanford University, February 1988. Available as Technical Report CSL-TR-88-351.

[Sleator and Tarjan, 1983] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. In *Proceedings of the ACM SIGACT Symposium on Theory*, pages 235–245, Boston, Massachusetts, April 1983.

[Sobalvarro, 1988] Patrick G. Sobalvarro. A lifetime-based garbage collector for LISP systems on general-purpose computers, 1988. B.S. Thesis, Dept. of EECS, Massachusetts Institute of Technology, Cambridge.

[Stefanović and Moss, 1994] Darko Stefanović and J. Eliot B. Moss. Characterisation of object behaviour in Standard ML of New Jersey. In LFP [1994].

[Stefanović *et al.*, 1998a] Darko Stefanović, J. Eliot B. Moss, and Kathryn S. McKinley. Oldest-first garbage collection. Computer Science Technical Report 98-81, University of Massachusetts, Amherst, April 1998.

[Stefanović *et al.*, 1998b] Darko Stefanović, J. Eliot B. Moss, and Kathryn S. McKinley. On models for object lifetimes. February 1998.

[Stefanović, 1993a] Darko Stefanović. The garbage collection toolkit as an experimentation tool. In Moss et al. [1993].

[Stefanović, 1993b] Darko Stefanović. Generational copying garbage collection for Standard ML: a quantitative study. Master's thesis, Department of Computer Science, University of Massachusetts, 1993.

[Sullivan and Chillarege, 1991] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability – a study of field failures in operating systems. In *Digest of the 21st International Symposium on Fault Tolerant Computing*, pages 2–9, June 1991.

[Ungar and Jackson, 1988] David M. Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. *SIGPLAN Notices*, 23(11):1–17, 1988.

[Ungar and Jackson, 1992] David M. Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, 1992.

[Ungar, 1984] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, Pennsylvania, April 1984. *SIGPLAN Notices 19*, 5 (May 1984).

[Ungar, 1986] David M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. ACM distinguished dissertation 1986. MIT Press, 1986.

[van Emde Boas, 1990] Peter van Emde Boas. *Machine Models and Simulations*, volume A: Algorithms and Complexity. Elsevier and MIT Press, 1990.

[Wilson and Moher, 1989] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 23–35, New Orleans, Louisiana, October 1989. *SIGPLAN Notices 24*, 10 (October 1989).

[Wilson *et al.*, 1991] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 177–191, Toronto, Canada, June 1991. *SIGPLAN Notices 26*, 6 (June 1991).

[Wilson *et al.*, 1995] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, Lecture Notes in Computer Science, pages 1–116, Kinross, Scotland, September 1995. Springer-Verlag.

[Wilson, 1989] Paul R. Wilson. A simple bucket-brigade advancement mechanism for generation-based garbage collection. *SIGPLAN Notices*, 24(5):38–46, May 1989.

[Wilson, 1990] Paul R. Wilson. Some issues and strategies in heap management and memory hierarchies. In Eric Jul and Niels-Christian Juul, editors, *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, October 1990. Also in SIGPLAN Notices 23(1):45–52, January 1991.

[Wilson, 1992] Paul R. Wilson. Uniprocessor garbage collection techniques. In Bekkers and Cohen [1992], pages 1–42.

[Yi and Harrison, 1992] Kwangkeun Yi and Williams Ludwell Harrison, III. Interprocedural data flow analysis for compile-time memory management. Technical Report 1244, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, Illinois, August 1992.

[Zorn, 1989] Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, December 1989. Available as Technical Report UCB/CSD 89/544.

[Zorn, 1990a] Benjamin Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado at Boulder, November 1990.

[Zorn, 1990b] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In LFP [1990].

[Zorn, 1991] Benjamin G. Zorn. The effect of garbage collection on cache performance. Technical Report CU-CS-528-91, University of Colorado at Boulder, May 1991.