A TOOL FOR MONITORING AND RECORDING HEAP-ALLOCATED OBJECT BEHAVIOR

by

QINGFENG DUAN

B.S., Wuhan University of Technology, 1995

M.S., Mechanical Engineering, University of New Mexico, 2000

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico
Albuquerque, New Mexico

May, 2003

Acknowledgments

First, I would like to thank my advisor, Professor Darko Stefanović, for his constant encouragement and support. I am impressed with his rigorous research and teaching styles. He is always able to point me to the right direction of my research. I have to say, I have learned a lot from working with him. I would also like to thank other committee members, Professor Arthur Maccabe and Professor Patrick Bridges, for reviewing this document and attending my oral defense.

Secondly, I would like to thank Zhenlin Wang at the University of Massachusetts at Amherst, who has done great work on this topic. Numerous ideas from his work have been applied in this thesis. I would also like to thank Xianglong Huang at the University of Texas at Austin, who developed the Dynamic SimpleScalar tool which is used in this thesis.

Thirdly, I would like to thank my coworkers at the Object Architecture Lab. Rotem helped me learn how to build Jikes RVM. Ben, Trek, and of course, other members in the lab, have been helpful.

Last, but not least, I would like to thank my wife, Huajun, and our son, Luke, for their constant love and support.

A TOOL FOR MONITORING AND RECORDING HEAP-ALLOCATED OBJECT BEHAVIOR

by

QINGFENG DUAN

ABSTRACT OF THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico
Albuquerque, New Mexico

May, 2003

A TOOL FOR MONITORING AND RECORDING HEAP-ALLOCATED OBJECT BEHAVIOR

by

QINGFENG DUAN

B.S., Wuhan University of Technology, 1995
M.S., Mechanical Engineering, University of New Mexico, 2000

M.S., Computer Science, University of New Mexico, 2003

Abstract

A tool, to be used for monitoring and recording the heap-allocated object behavior, is designed, implemented, evaluated, and documented. Object-oriented programming languages, such as Java, require the support of automatic memory management (garbage collection), because of their intensive heap allocation. Modern garbage collection techniques rely on exploiting the object behaviors. These behaviors include the ages, the types, and the sizes of objects, and the references among objects. For example, the Appel generational copying collector and the Older-First collector are built on the basis of the distribution of object ages. To understand these object behaviors and thus be able to improve garbage collection techniques, we need a simulation tool. The tool described here correlates the low-level read/write data accesses with the high-level object characteristics. When an object is allocated, modified, moved, or deallocated, the tool monitors and records this information. By further analyzing this information, one obtains the relevant data to understand the desired object behaviors. The tool consists of three components: IBM's Jikes RVM, Dynamic SimpleScalar, and an off-line Analyzer. Jikes RVM is a Java

virtual machine which itself is written in Java; Dynamic SimpleScalar is a machine-level emulator to simulate a PowerPC model running the AIX operating system; and the Analyzer is used to postprocess the results generated by the first two components. To be running, the entire tool maintains a layering of structures: the Java program, Jikes RVM, Dynamic SimpleScalar, and the host machine, in which the Java program resides at the first layer. We evaluate our tool using three SPECjvm98 Java benchmarks. We also illustrate how the tool can be used to analyze the *write* statistics to the fields of an object with certain class type. In general, this tool could have great significance in garbage collection research, by being able to provide accurate and flexible analyses at the heap-allocated object level.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Object-oriented programming languages, such as Java, typically allocate a substantial amount of objects on the heap. Many of these languages support automatic memory management, that is, garbage collection. Garbage collection reclaims heap objects when they are no longer used. Most modern garbage collectors are devised on the basis of observations of object behaviors, including ages, sizes, types, and references and so forth. This chapter reviews two garbage collection algorithms which both rely on object age behavior. While garbage collection relieves the programmer from painful debugging of memory errors, it also raises issues like *poor* data locality of references. One thus needs to understand the data memory reference behaviors of objects. These issues motivate our present work. We would like to build a tool to understand some of these object behaviors.

1.1 Overview

Current technology leads to an increasingly widening gap between processor speed and memory speed. Wulf and McKee [WM95] further explore this. Owing to the exponential growth of this gap, average access time to memory will increase and cost more and more

processor cycles in the foreseeable future. At the same time, the size of application programs is getting larger when compared to the limited memory sizes. The memory system becomes *expensive* in some sense. It dominates the system performance and becomes a bottleneck in the improvement of the overall performance of programs. Memory storage management, therefore, has been an active research subject for decades.

With advances in compiler technology, many high level programming languages allow programmers to dynamically allocate storage to the heap. For example, in C++ and C, programmers can allocate data to the heap by using *new* (in C++) or *malloc* (in C). These languages, however, require that programmers deallocate the heap storage manually when the storage is no longer used. One can use *delete* (in C++) or *free* (in C) to deallocate the heap storage, for instance. It is clear that doing memory management this way is painful and error-prone, even for experienced programmers. On the other hand, object-oriented programming languages have become the mainstream in the commercial software development. In such languages, objects are more intensively allocated to the heap than those in traditional languages. A recent study has shown that there are ten times as many heap allocations in C++ than in C [CGZ94]. Java, another popular object-oriented language, requires that all objects be allocated to the heap. One can expect that there are even more heap allocations in Java than in C++. Based on a combination of these reasons, it is essential for these languages to provide some sort of automatic memory management mechanism for heap allocations.

Many garbage collection algorithms have been designed to serve this purpose [JL96]. Three classical algorithms are Reference counting, Mark-Sweep, and Copying. Reference counting counts the references to heap objects during the execution of user programs. Once the reference count to a heap object drops to zero, the heap object will be reclaimed. This algorithm is simple and it does not need to suspend the running programs. The major problem with reference counting is that it cannot reclaim cyclic data structures like doubly-linked lists. Reference counts will never be zero because of the existence of internal back

references in these structures. The Mark-Sweep algorithm works in a way that traverses all of the heap objects and marks the ones reachable from the *roots*. Unmarked heap objects are then swept back into a free list, to be used in the future allocations. One of the advantages of this algorithm over reference counting is that it can handle cyclic data structures. However, it has to stop the user program. Put another way, the user program has to wait until the mark-sweep garbage collector completes. This influences the response time for large user programs. The Copying algorithm divides the heap into two semi-spaces, *fromspace* and *tospace*. The copying collector starts by traversing all heap objects in the *fromspace* and copying all reachable objects into *tospace*. As a result, *tospace* contains a replica of all reachable heap objects from the roots. Then, the roles of *fromspace* and *tospace* are flipped. The old *tospace* becomes *fromspace* into which new data will be allocated. Similar to mark-sweep algorithm, it has to suspend the user program. The benefit is that it leads to smaller pause time since it only needs to traverse half of the heap. Another benefit is the elimination of the fragmentation since it compacts the reachable heap objects into the bottom of *tospace* during copying.

It should be noted that in both mark-sweep and copying collectors, all of the reachable heap objects are marked or copied, including relatively long-lived objects. That is to say, a certain amount of unnecessary time is spent on recycling of these long-lived objects. To solve this issue, researchers proposed the generational garbage collection technique in which objects are segregated into generations by their ages. This brought about another series of garbage collection algorithms, i.e., relying on the object demographics, such as ages, sizes, and types. The details on these techniques will be discussed in the following section.

1.2 Modern Garbage Collection Algorithms

In this section, we discuss two garbage collection techniques, the Appel generational copying collector and the older-first collector, both of which are designed in terms of object ages.

1.2.1 Appel Generational Copying Algorithm

The basis of generational garbage collection is the *Weak generational hypothesis*, that is, that most objects die young [Ung84, JL96, App89]. The entire heap space in a generational collector is divided into two or more *generations* in terms of object ages. The youngest generation is called the *nursery*. The application always allocates new data into the *nursery*. Once a collection is triggered, the *nursery* will be collected first since most of the objects it contains are supposed to be dead (according to the *Weak generational hypothesis*). Objects that survive are then promoted to older generations. Every *generation* is collected at different frequencies. The younger generations are collected more frequently than older generations. In short, a generational collector does not need to collect the entire heap, but a small region of the heap, *one generation*. This leads to relatively short pause times.

Appel's generational copying collector [App89] organizes the heap into two generations. The size of the *nursery* changes during garbage collections. Figure 1.1 and Figure 1.2 illustrate how an Appel generational copying collector works.

Figure 1.1(a) sketches a typical organization of logical address space for a process. During the execution of program, the heap grows down to higher addresses and the stack grows up to lower addresses. There is a pointer pointing to the upper end of the heap, which is called "break" in Unix. With the growth of the heap, the "break" is changed correspondingly. Once this "break" hits the top of the stack, a segmentation fault signal

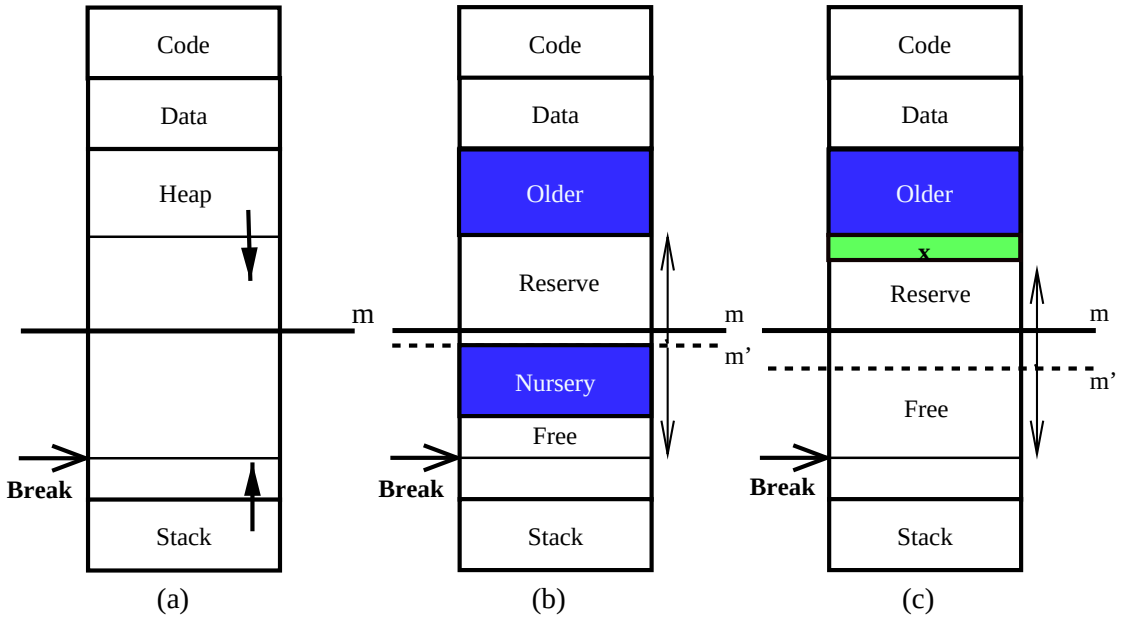

Figure 1.1: Appel generational copying collector, part I.

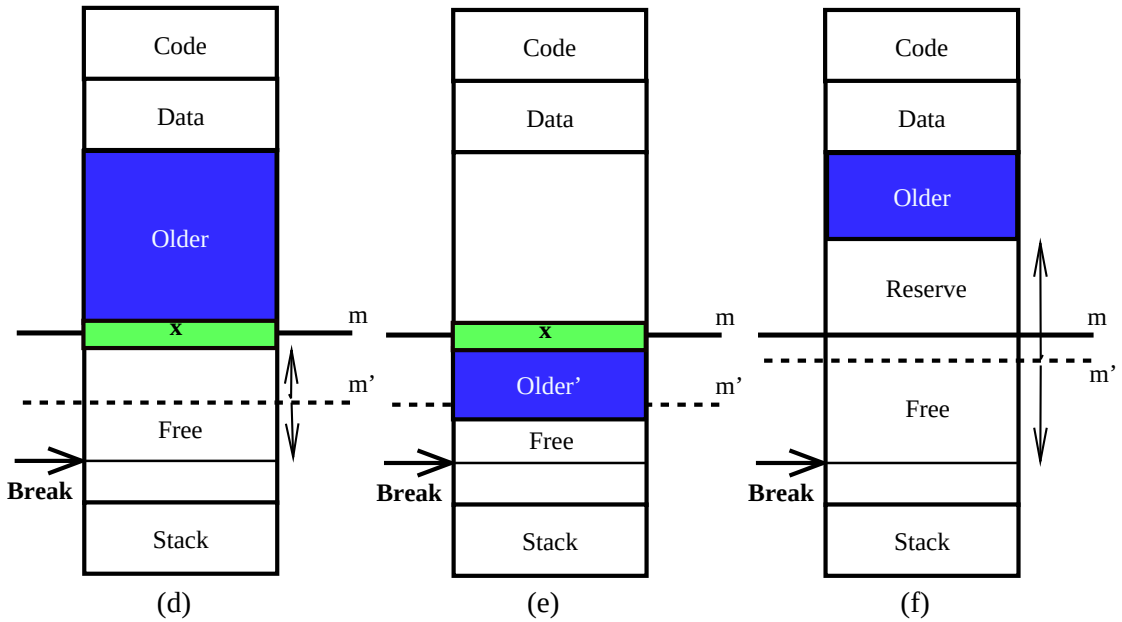

Figure 1.2: Appel generational copying collector, part II.

is raised in Unix. This triggers the garbage collector. At this point, the heap is organized as shown in Figure 1.1(b). The solid line *m* indicates the middle of the entire heap, while the dashed line *m'* shows the middle of the *reserve* space and the *nursery* space. The new data is allocated to the *free* portion of the *nursery*. When the allocation hits the "break", a *minor* collection is triggered and the *nursery* is collected. The surviving objects are promoted to the *reserve* space, marked as *x* seen in Figure 1.1(c). The *reserve* and *nursery* space are then reorganized. After a couple of cycles of *minor* collections, the entire *older* generation will be filled up. A *major* collection is triggered once the promoted objects *x* span over the middle line *m* (Figure 1.2(d). The *older* generation is collected and the collector promotes the survived objects into the region immediately right of the *x*, *older'*, as shown in Figure 1.2(e). Finally, as indicated in Figure 1.2(f), the heap is reorganized by moving objects in *older'* to *older*. The next cycle of collection will start with this organization.

1.2.2 Older-First Algorithm

Stefanović *et al.* [SMM99, SHB+02] observe that *very youngest objects* are repeatedly collected in a generational copying collector. In fact, every object needs certain time to die. These youngest (most newly-allocated) objects are still *alive*, while they are being collected. Based on this observation, they invented older-first algorithm which avoids collecting the very youngest objects. In their algorithm, the heap is logically organized as an ordered list where objects are arranged from left to right in terms of the ages. As shown in Figure 1.3, the oldest objects are at the leftmost of the heap. The data is allocated to the right end of the heap. Thus, the youngest object is always arranged at the rightmost of the heap.

An older-first collector works by collecting a window at one cycle. A window is a fixed-size region which contains a group of objects from older to younger. Upon the

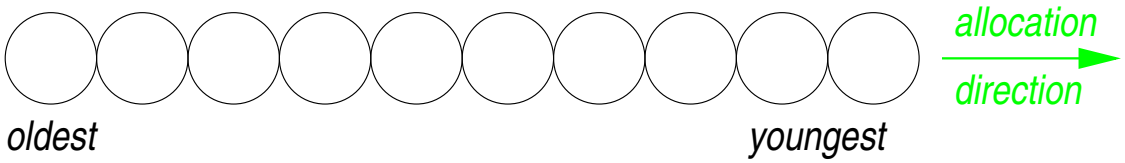

Figure 1.3: Organization of the heap in older-first algorithm, as courtesy of Stefanović *et al.* .

completion of a collection, the window is moved into a new position immediately to the right of survived objects. Figure 1.4 illustrates how the survived objects from collected

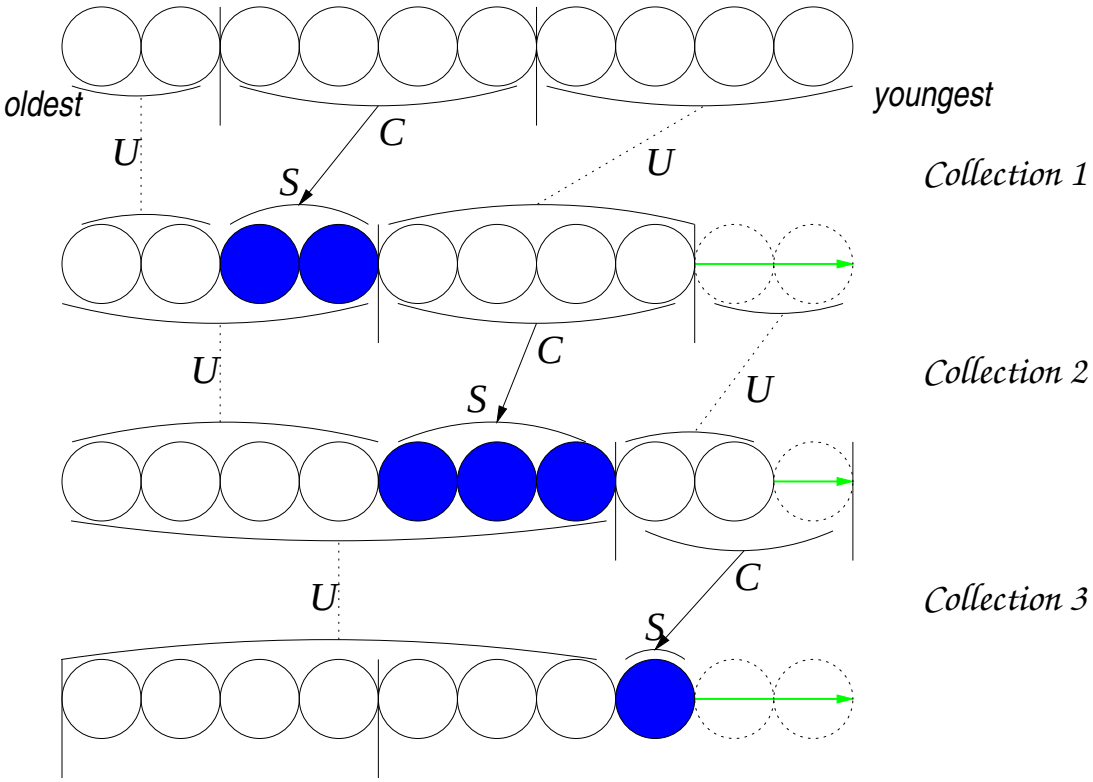

Figure 1.4: Older-first algorithm, as courtesy of Stefanović *et al.* .

region *C* (i.e. current window) are copied (in region *S*) and how the freed memory blocks are moved into the *nursery*, through a couple of cycles of collections. *U* represents an uncollected region wherein objects are assumed to be alive. Figure 1.5 shows how the algorithm works from an angle of window sliding. In the meantime, this diagram gives a situation where the collector gains the best performance. In collections 4 through 8, the window moves very slowly since there is only a very small set of survived objects at each collection. In other words, most part of window *C* can be freed and used for further allocations. This implies that the copying cost of survived objects is minimized.

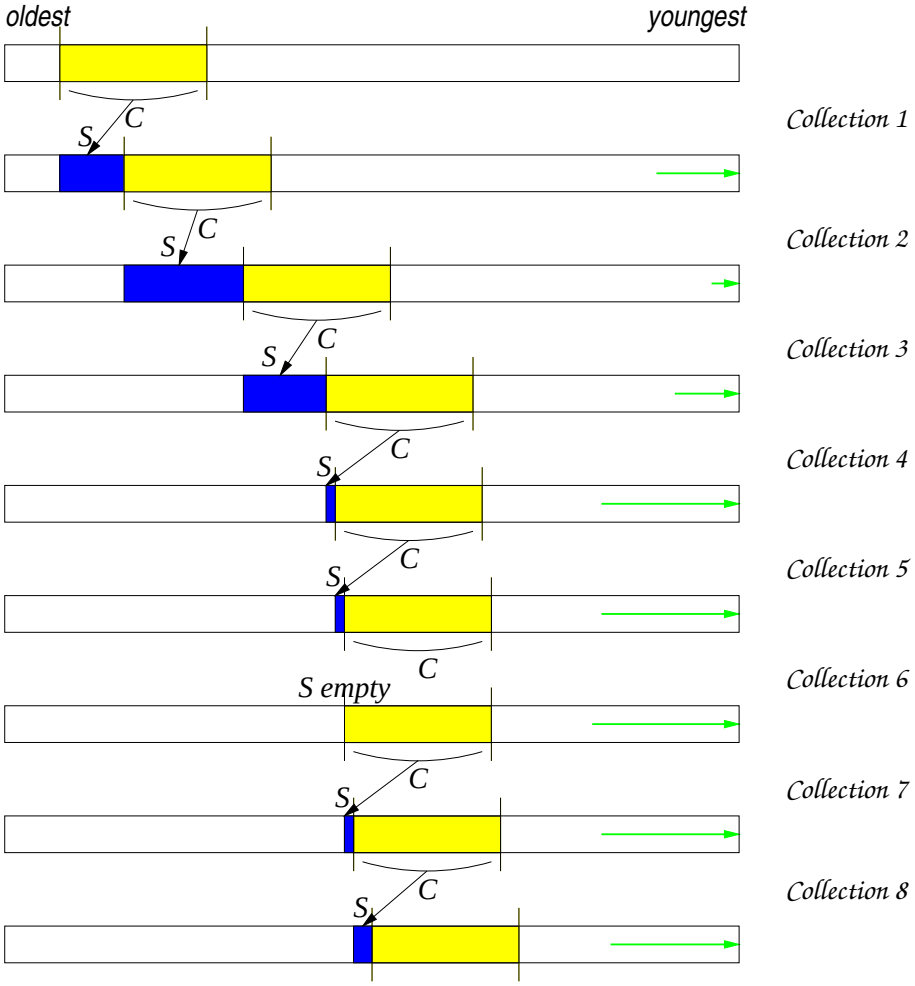

Figure 1.5: Older-first algorithm with window motion, as courtesy of Stefanović *et al.* .

1.3 Heap-allocated Object Behaviors

Upon being allocated, an object tends to live for some time. By *live* we mean that this object can be reachable from the *roots* (the run-time stack, registers, and global variables). It is dead when it becomes unreachable and then needs to be garbage-collected. The object thus has a lifetime, or an age. We say that a newly-allocated object is younger than those already allocated for a while. Researchers observe that most young objects have a short lifetime. This is the basis of generational garbage collection algorithms, as we have already seen in Section 1.2.

The size and the type of an object are determined during allocation. A recent study by Shuf *et al.* [SGBS02] observes that some types of objects are *prolific*, while others are *non-prolific*, according to the number of the instantiated instances on those types. Then they propose a prolific hypothesis, which states that objects of *prolific* types die younger than objects of *non-prolific* types. Based on this hypothesis, they invent *type-based* garbage collection algorithm.

Objects do not live alone and they have *references* (i.e. pointers) to one another. Two objects may connect, either directly or transitively through other intermediary objects. Some researchers [HHDH02] further explore the "connectivity" among the heap objects and claim that the connected objects die together.

Java programs generally have poor data locality due to the influence of garbage collection. This is especially serious in copying garbage collection since objects have to be copied around in such collectors. On the other hand, locality serves as the basis for caches and virtual memory system. Poor locality degrades the performance of entire system. Several researchers have studied the memory system behaviors of Java programs at the object level. Kim and Hsu [KH00] investigate the temporal locality by using the memory reference traces on several SPECjvm98 benchmarks [Sta99] executed with a Just-In-Time(JIT) compiler. Shuf et al. [SSGS01] further correlate the cache and TLB performance with the

accesses to fields and methods of objects, using traces generated by instrumenting the run-time optimizing compiler in a virtual machine.

1.4 Objective of This Work

As we discussed in Section 1.3, modern garbage collectors are designed based on the object behaviors. To devise better garbage collectors, one needs to have a good understanding of these object behaviors. A simulation tool, which traces the object activities during the execution of programs, can help us to reach this goal. In this work, we describe such a tool. Using this tool, we expect to correlate the machine-level accessing activities (reads/whites) with the high-level object characteristics. Figure 1.6 illustrates how our tool is used to "picture" a typical Java object.

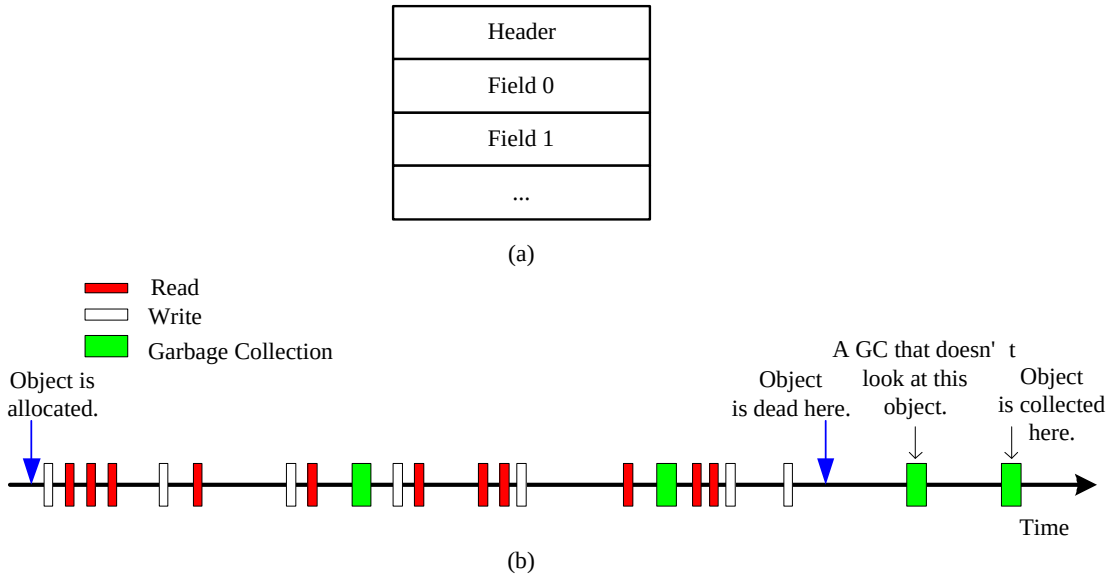

Figure 1.6: Data memory reference behavior of a single scalar object.

Figure 1.6(a) gives a simple object layout usually exploited in a Java run-time system. Our tool monitors and records each data access, either a read or a write, to a particular field

of an object. We can know whether this access was a hit or a miss in the data cache. Together with the information of high-level objects (type, start address, and offsets of fields, etc.), we can know which field of the object was accessed. As such, with the execution of programs, we record a sequence of accesses on an object, as shown in Figure 1.6(b). It should be mentioned that, unlike previous work [KH00], our tool allows the movement of objects. For example, after a cycle of garbage collection, an object survives and is copied to another heap space. Also, to generate traces, we do not need to instrument run-time compiler as they did in [SSGS01]. More powerful usage of this tool will be discussed in Chapter 4.

1.5 Organization of the Thesis

The remainder of this document is organized as follows: Chapter 2 gives the methodology we exploit to develop this tool; Chapter 3 describes the experiences learned during developing the tool in detail; Chapter 4 discusses how to run the tool; and Chapter 5 concludes with a discussion of future work.

Chapter 2

Methodology

This chapter gives an overview of the method we used to build this tool. The tool is composed of IBM's Jikes Research Virtual Machine (Jikes RVM), Dynamic SimpleScalar, and an off-line Analyzer. In Section 2.1, we introduce an organization of the tool and outline the mechanisms how the tool works. Section 2.2 describes the three components of the tool in detail. The last section discusses the previous work and our contribution to the tool.

2.1 Overview

Figure 2.1 gives the hierarchical organization of the tool. Layer 1 is the Java program. Layer 2 is a Java Virtual Machine (JVM), IBM's Jikes RVM. Layer 3 is Dynamic SimpleScalar, which simulates a PowerPC machine. Layer 4 is the host machine on top of which Dynamic SimpleScalar runs. Generally, higher layers run on top of lower layers. The third component, Analyzer, also runs on top of host machine.

During the execution of Java programs, Jikes RVM's allocator and collector trig-

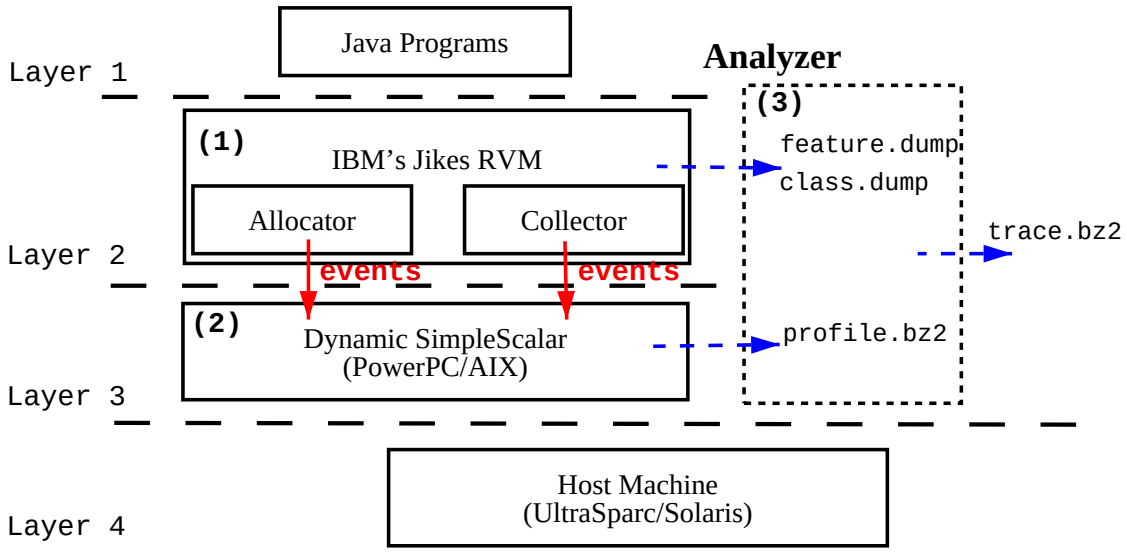

Figure 2.1: Organization of the tool.

ger some events. Semantics of these events are shown in Table 2.1. Upon the occurrence of an event, Jikes RVM stores certain information in some featured memory locations (see Table 2.2), which are known to both Jikes RVM and Dynamic SimpleScalar. This information is conveyed to Dynamic SimpleScalar when it makes access to those locations. For example, NEW_SCALAR event indicates that a new Java scalar object is allocated. Once this event is triggered, it stores information regarding this object, such as start address and type, to featured addresses, FEATURE_ADDRESS_NEXT and FEATURE_ADDRESS_EXTRA, respectively. FEATURE_ADDRESS is used to store the event ID. FEATURE_END is a mark, and when it is accessed, Dynamic SimpleScalar knows something happened and then extracts information stored in other featured addresses. This information (eventID, start address, and type), along with other access results (access address, read/write, hit/miss, and values), is saved to a disk file, *profile.bz2*.

The choice of featured addresses is in terms of the convention of virtual memory address space in Dynamic SimpleScalar[1]. The base of *stack* segment is at the address

[1]See `memory.c` in the distribution of Dynamic SimpleScalar.

0x7fffc000 and the *stack* grows towards lower memory as needed. The four featured addresses we have chosen are higher than this stack base address, which are supposed to not be used by any user program.

Table 2.1: Event semantics generated by Jikes RVM.

Event ID	*Events*	*Semantics*
1	NEW_SCALAR	A new scalar object is allocated.
2	NEW_ARRAY	A new array object is allocated.
3	GC_START	Garbage collector starts.
4	GC_END	Garbage collector ends.
5	COPY_SCALAR	A scalar object is copied in GC.
6	COPY_ARRAY	An array object is copied in GC.
7	CLASS_DUMP_START	Saving class information starts.
8	CLASS_DUMP_END	Saving class information is done.
9	MEM_UNMAP	Cleaning up dead/copied objects starts.

Table 2.2: Featured addresses known to Jikes RVM and Dynamic SimpleScalar.

No.	*Name*	*Featured Addresses*
1	FEATURE_ADDRESS	0x7fffe000
2	FEATURE_ADDRESS_NEXT	0x7fffe004
3	FEATURE_ADDRESS_EXTRA	0x7fffe008
4	FEATURE_END	0x7fffe00c

Finally, an off-line Analyzer comes into play. It takes three files, *profile.bz2*[2], *feature.dump* (which stores information about Jikes RVM's boot image objects and is created during the building time of Jikes RVM), and *class.dump* (which stores information on classes and is created by Jikes RVM at run-time), as inputs, and generates the data memory reference traces into a file, *trace.bz2*. The detailed information saved in these files will be further discussed in Chapter 3.

[2]The extension *bz2* means that this file is compressed using bzip2 C routine in `bzlib.h`.

2.2 Components of the Tool

This section describes the three components of the tool: IBM's Jikes RVM, Dynamic SimpleScalar, and Analyzer. We first present the basic structure and implementation of Jikes RVM. Then we describe the specific features of Dynamic SimpleScalar which make it possible to simulate Java programs running on top of Jikes RVM.

2.2.1 IBM's Jikes RVM

Jikes RVM, previously known as Jalapeño, is a virtual machine for servers, developed at IBM's T. J. Watson Research Center [AAB+99, AAB+00]. It includes the latest virtual machine technologies for dynamic compilation, adaptive optimization, garbage collection, thread scheduling, and synchronization. It itself is written in the Java programming language. It takes a compile-only approach to program execution in which bytecodes are always compiled to machine code before execution. There is no interpreter at all. Currently, Jikes RVM runs on PowerPC/AIX, PowerPC/Linux, and IA32/Linux platforms.

Structure of Jikes RVM

There are five key components of Jikes RVM: an object model; the run-time subsystem, the thread and synchronization subsystem, the memory management subsystem, and the compiler subsystem [AAB+00].

Object Model and Memory Layout. Jikes RVM operates on two kinds of types: *primitive types* and *reference types*. Correspondingly, there are two kinds of values: *primitive values* and *reference values*. The primitive types include `int`, `char`, `boolean`, etc., and the reference types include *array* types and *class* types. Objects with reference types are allocated on the heap. These objects are *array* objects and *class* (scalar) objects. Figure 2.2

shows the object model and memory layout exploited in Jikes RVM. The *object reference* always points to three words away of the top of headers (TIB field). In an array object, it points to the first component. Scalar objects grow down from the object reference with all fields at a negative offset, whereas array objects grow up from the object reference with the *length* field at a fixed negative offset (-4 bytes).

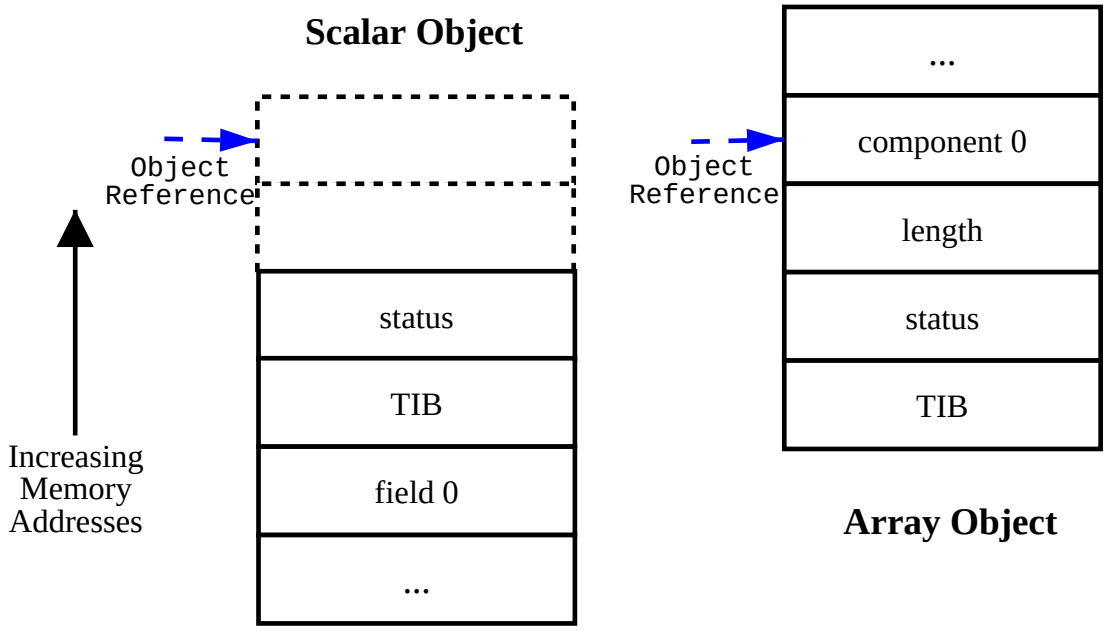

Figure 2.2: Object model employed in Jikes RVM.

On PowerPC/AIX, this object model and memory layout allows fast access to fields of a scalar object and components of an array object. Using a single instruction with base-plus-displacement addressing (with *object reference* being held in a *base* register), it can access any field of a scalar object. To access an array, it first needs to use a single trap instruction to check if the index out-bounds the array by loading the *length* field. Then a shift instruction is used to shift the index to get byte index. Finally, a single instruction with base-plus-index addressing (again, with *object reference* being held in a *base* register) can be used to access the components of the array. This layout also allows hardware support of null pointer checking. In Jikes RVM, a null pointer is represented by Address 0x0. If

the object reference is null, this means loading a word at high memory address. The AIX operating system does not allow that and generates a hardware interrupt. Recall that to access to either a scalar object or an array object, it has to access to *negative* offsets (*fields* of scalars or *length* field of arrays).

There is a two-word header for each object: one word for *status* and the other for *TIB*. The *status* header is divided into three *bit fields*: the first bit field is used for locking; the second one holds the default hash value of hashed objects; and the third one is used by the memory management subsystem. The *TIB* header is a reference to the *Type Information Block (TIB)* for the object's class. A TIB is an array of Java object references. Its first component describes the object's class, including its superclass, the interfaces it implements, offsets of any object reference fields, etc. The remaining components are compiled method bodies, for the virtual methods of the class.

All static data, including static fields, references to static method bodies, constants, and the TIB for each class, are stored into a single array called the Jalapeño Table of Contents (JTOC). All of the objects in Jikes RVM are reachable from this array.

The Run-Time Subsystem. This subsystem provides support for run-time services of a virtual machine, exception handling, dynamic type checking, dynamic class loading, interface invocation, input and output, reflection, etc. All these services are written in Java.

The Thread And Synchronization Subsystem. Jikes RVM multiplexes Java threads on *virtual processors* that are implemented as Posix kernel-level threads (pthreads), rather than mapping Java threads to operating system threads directly. Jikes RVM's thread scheduling mechanism uses simple time slicing within each pthread to schedule the Java threads. To support the synchronization, it uses three kinds of locks: *processor* locks, *thin* locks, and *thick* locks. More detailed discussions on this part can be found elsewhere [AAB+00].

The Memory Management Subsystem. The role of memory management is object allocation and garbage collection. Jikes RVM is designed to support a family of interchangeable memory managers, each of which consists of a concurrent object allocator and a stop-the-world, parallel, type-accurate garbage collector. The four major types of managers supported are: copying, noncopying, generational copying, and generational noncopying. It should be noted that in our tool, we are using a modified version of Jikes RVM v2.0.3, incorporated with UMass Garbage Collector Toolkit (GCTk) [HMDW91]. The GCTk is also written in Java language and incorporates many state-of-the-art garbage collectors, such as the Appel generational copying collector and the older-first collector.

The Compiler Subsystem. Jikes RVM contains three compilers: *a baseline compiler*, *an optimizing compiler*, and *an adaptive compiler*. The *baseline* compiler mimics the stack machine behavior of the JVM specification [LY99]. It translates bytecodes to machine code quickly, but the resultant machine code typically performs poorly. The *optimizing* compiler applies traditional static compiler optimizations as well as a number of optimizations specific to object-oriented features and the dynamic Java context. It produces high-quality machine code. The *adaptive* compiler first performs the initial compilation of a method using the *baseline* compiler and then, identifying the methods either frequently executed or computationally intensive, applies optimizations on those methods.

Implementation of Jikes RVM

The run-time services, such as exception handling, dynamic type checking, dynamic class loading, and interface invocation and alike, are primarily written in Java. This strategy is quite different from some conventional virtual machines wherein all these services are implemented by native methods written in C, C++, or assembly. This leads to more opportunities for optimizations. A running Java program normally involves four layers of functionality [AAB+99]: the user code, the virtual machine, the operating system, and the underlying hardware. By layering down Java/Non-Java interface below the virtual

machine, an optimizing compiler could be finely-designed, to reduce the semantic gap between the high level language like Java and the underlying machine.

To get Jikes RVM started, an executable *boot image* of a working snapshot of Jikes RVM is built during the building time of Jikes RVM. The boot image contains some key initial services (Java objects), such as class loader, object allocator, and compiler, to be used to start Jikes RVM. After loading the boot image objects, Jikes RVM starts all other remaining services. A Java program called *boot-image writer* creates the boot image. Since it is written in Java, it can run on any host JVM. When running, it works like a cross compiler and linker: it compiles bytecodes to machine code and rewrites machine addresses to bind program components into a runnable image. Finally, the boot image is written to a disk file, which can be loaded later by a C *boot loader*.

There is a small portion of code that is written in C. Jikes RVM is designed to run as a user-level process. As such, it must use the host operating system to access the underlying file system, network, and processor resources. Instead of using low-level system-calling conventions to access these resources, Jikes RVM chooses to use the standard C library. Thus this part of code is written in C. In addition, the *boot loader* is also written in C. The boot loader allocates memory for the Jikes RVM *boot image*, reads the image from disk into memory, and branches to the image startup code. Finally, the C code also contains two signal handlers. One is used to capture hardware traps (generated by null pointer dereferences on the PowerPC/AIX) and trap instructions (generated for array bounds and divide-by-zero checks), and relays these into the virtual machine, along with the current state of register values; the other is used to pass timer interrupts to the running system, by which it implements thread scheduling.

2.2.2 Dynamic SimpleScalar

The SimpleScalar tool set is an *execution-driven* architecture-level simulator, which can perform functional simulations of a target machine architecture (*sim-fast* and *sim-safe*), functional cache simulations of cache memory systems (*sim-cache*), and complicated timing simulations of superscalar machines which support out-of-order issue and execution, associated with the memory system (*sim-outorder*) [BA97]. Existing versions of SimpleScalar support the PISA, the Alpha, and the PowerPC and a few other architectures [SNKB01]. The normal SimpleScalar, however, can only simulate statically compiled binaries. To simulate Java programs, in which code is dynamically generated, we need to use Dynamic SimpleScalar, an extended tool set specifically for simulating Java Virtual Machines, like Jikes RVM [HMM+03].

The following summarizes the major changes and the extended features of Dynamic SimpleScalar over normal SimpleScalar. These extensions aim to provide support for common run-time services of Java Virtual Machines, such as dynamic code generation, exception handling, and thread scheduling and synchronization. More detailed discussions can be found elsewhere [HMM+03].

- *Dynamic code generation.* In normal SimpleScalar, the simulated program is predecoded after the program is loaded into the simulated memory and before the simulation starts. For every instruction, there is a corresponding function that simulates the instruction's opcode. SimpleScalar predecodes an instruction by looking up its handling function, then replacing the instruction in the simulated memory with a pointer pointing to that function. Unlike normal SimpleScalar, Dynamic SimpleScalar decodes the instructions on the fly, to support the dynamic compilations. Unavoidably, dynamic code generation causes cache coherence issue of instruction cache since the code is dynamically generated, moved, and modified in the simulated memory. Dynamic SimpleScalar thus implements some special PowerPC instructions to handle

this issue.

- *Signals*. The normal SimpleScalar does not support Unix signals, whereas Dynamic SimpleScalar has to do so. The Java Virtual Machines, particularly Jikes RVM, run as a general user process on the host machine, more accurately, the simulated machine by Dynamic SimpleScalar. Jikes RVM needs some Unix signals to support run-time services, such as Java exception handling and multithreading. Dynamic SimpleScalar implements a set of signal generation, delivery, handling, and recovery mechanisms. For example, Dynamic SimpleScalar generates a *SIGTRAP* signal and delivers it to Jikes RVM, indicating that an exception of outbounds or divide-by-zero occurs. Jikes RVM detects this signal and throws a ArrayIndexOutOfBoundsException or ArithmeticException. Then a proper signal handler is called. This is exactly how the Java exceptions (try/throw/catch) in Jikes RVM are handled.

- *Thread scheduling and synchronization*. Multithreading is also supported by signal handling mechanism in Dynamic SimpleScalar. Jikes RVM uses an internal timer to handle the thread scheduling. It sets the timer by making the system calls *gettimerid* and *incinterval*. Dynamic SimpleScalar updates this timer and when the timer expires it generates a *SIGALRM* signal and delivers it to Jikes RVM. Then Jikes RVM performs thread scheduling. To support synchronization, Dynamic SimpleScalar implements two PowerPC instructions *lwarx* and *stwcx*, which in turn are used to build *locks* in Jikes RVM.

2.2.3 Analyzer

The off-line Analyzer is written in C and performs the following tasks. First, it reads the class information of both Jikes RVM and Java programs from *class.dump* into a local buffer, containing the class *typeID, class name, offsets and typeID's of fields* and *sizes of class instances*, etc. Then, it processes *feature.dump* and *profile.bz2*. It inserts heap

objects into a sorted buffer in the increasing order of *start address* of objects. For every access, it does a binary search over this object buffer to determine which object is accessed, by communicating with the local buffer generated in the first step. Finally, it saves the analytical results, i.e., the data memory access traces, to *trace.bz2*.

2.3 Previous Work and Our Contribution

This tool has been originally implemented by Wang[3] at University of Massachusetts to summarize a class-level statistics of accesses/misses to the fields. For a specific class type, say `java.lang.String`, it summarizes the number of misses for each individual fields over the total allocated objects for this class. In his version of the tool:

- The information of boot image objects is stored into *feature.dump* when building Jikes RVM.
- The class information is stored into *class.dump* at run time.
- In Dynamic SimpleScalar, the events (event, type, and start address) and regular accesses (access address and hit/miss) are stored into two separate files.
- The Analyzer analyzes those two files, together with *feature.dump* and *class.dump*.
- The MEM_UNMAP event is handled in Analyzer to clean up the dead/copied objects in its object buffer.
- Array objects are not supported.

To extend this tool for our purpose, i.e., to monitor and record heap-allocated object behaviors, as stated in Section 1.4 of Chapter 1, we made the following modifications:

[3]Personal communications through zlwang@cs.umass.edu

- To be a completely useful tool, array objects must be supported. We added code to treat the array objects.

- The regular access information includes access address, read/write, hit/miss, and the value at that address.

- To maintain an order of events and regular accesses, we generate both event information and regular access information into a single file. Noticeably, this is a huge file ($>$ 2GBytes) for some intermediate Java benchmarks. It cannot be correctly created in typical 32-bit systems. We solve this problem by taking advantage of 64-bit file implementation in Solaris.

- The Analyzer analyzes this file, together with *feature.dump* and *class.dump*.

- Most interestingly, our tool can perform various analyses on object behaviors based on the results produced by Analyzer. Chapter 4, in its entirety, contributes to this discussion.

Chapter 3

Building the Tool

This chapter describes how we implement the tool. Different than previous chapters, we examine the three components in one more level of depth, at the code level. First, Section 3.1 gives an overview of the directory structure of the tool, which involves with the layout of the sources of three components and the benchmarks. This structure will be frequently referred to in the rest of chapter when we discuss the instrumentation of Jikes RVM (Section 3.2), the modification of Dynamic SimpleScalar (Section 3.3), and the building of Analyzer (Section 3.4).

3.1 Overview

Figure 3.1 details the directory tree of the tool. The root is DynamicSimpleScalar-TOOL/, which contains four subdirectories: JikesRVM-2.0.3/, dssppc/, benchmarks/, and analyzer/. The JikesRVM-2.0.3/ directory contains the sources of a special version of Jikes RVM we utilized, v2.0.3, which is incorporated with the GCTk [HMDW91] (in the directory JikesRVM-2.0.3/rvm/src/vm/memoryManagers/GCTk/). This directory has a subdirectory, Booter/, which contains a precompiled C binary, the *boot loader* `JikesRVM`.

It should be clear that this is a successfully built version of Jikes RVM since there is a subdirectory build/, in which the *boot image* file RVM.image is saved. In the second subdirectory of the root, dssppc/, it contains the sources of Dynamic SimpleScalar. In benchmarks/, it stores the sources of Java benchmarks from SPECjvm98 [Sta99]. Finally, the directory analyzer/ contains the C source of the Analyzer. It should be mentioned here that, in the directory tree, we only list the relevant source files we added or modified, while ignoring other less relevant files.

3.2 Instrumentation of Jikes RVM

The role of Jikes RVM in our tool is twofold: to generate the events (see Table 2.1); and to store the information into the featured memory addresses (see Table 2.2). The stored information includes, *eventID*, *start address*, and *typeID*, where the *start address* is the object reference of an object (see Figure 2.2) and the *typeID* denotes the class type of the object. Table 3.1 shows how these information is stored for each individual events. Note that for the event of NEW_ARRAY, the size of array object is stored along with the *eventID*, through the operation of *eventID* $|$ *(size* $<<$ *5)*.

Table 3.1: Illustration of how the information is stored for each individual events.

ID	*Events*	*Featured Addresses*			
		0x7fffe000	0x7fffe004	0x7fffe008	0x7fffe00c
1	NEW_SCALAR	1	start addr	typeID	0
2	NEW_ARRAY	2 + size	start addr	typeID	0
3	GC_START	3	0	0	0
4	GC_END	4	0	0	0
5	COPY_SCALAR	5	src addr	dest addr	0
6	COPY_ARRAY	6	src addr	dest addr	0
7	CLASS_DUMP_START	7	0	0	0
8	CLASS_DUMP_END	8	0	0	0

3.2.1 Generating *feature.dump* at Building Time

As we discussed in Chapter 2, some core objects of Jikes RVM virtual machine are saved into a *boot image* file at the building time of Jikes RVM. These objects are also allocated to the heap, along with the data of Java programs. It is possible that these objects will be *referenced* by the objects from Java programs. We thus create a file, *feature.dump*, to store the information of these "live" objects, during the building of Jikes RVM. The information will be directly written into the file, rather than stored into the featured addresses for the obvious reason. As the following piece of code indicates, the `event` method is declared and defined in `GCTk_BuildFeatureDump.java`. This method will be called in `copyToBootImage` of `BootImageWriter2.java`, when copying allocated objects into boot image.

```
************************************************
// ./JikesRVM-2.0.3/rvm/src/vm/memoryManagers/GCTk/util/
// GCTk_BuildFeatureDump.java
************************************************
class GCTk_BuildFeatureDump implements
      GCTk_Constants, VM_Uninterruptible {
  final static void buildInit() {
    ... // open file ‘‘/tmp/feature.dump’’
    ... // called by VM.java in ./JikesRVM-2.0.3/rvm/src/vm/
  }

  final static void event(int eventId, int start, int typeId) {
    ... // output (eventID, start address, typeID)
  }

  final static void event(int eventId, int start,
                          int typeId, int size) {
    event(eventId | (size << 5), start, typeId);
  }

  final static void postBuild() {
    ... // close file ‘‘/tmp/feature.dump’’
  }
}
```

```
************************************************
// ./JikesRVM-2.0.3/rvm/src/tools/bootImageWriter/
// BootImageWriter2.java
************************************************
/**
 * Construct a RVM virtual machine bootimage.
 *
 ...
 */
public class BootImageWriter2 extends BootImageWriterMessages
implements BootImageWriterConstants {
  ...

  public static void main(String args[]) {
  ...
  }

  private static int copyToBootImage
              (Object jdkObject, boolean copyTIB) {
    ... // other stuff

    // copy object to image
    if (jdkType.isArray()) {
      ...
      if (VM.interleavedProfile) {
        GCTk_BuildFeatureDump.event(GCTk_Constants.NEW_ARRAY,
        bootImageAddress+arrayImageOffset,
        rvmArrayType.getDictionaryId(), arraySize);
      }
      ...
    } else {
      ...
      if (VM.interleavedProfile) {
        GCTk_BuildFeatureDump.event(GCTk_Constants.NEW_SCALAR,
        bootImageAddress+scalarImageOffset,
        rvmScalarType.getDictionaryId());
      }
    }
  }

  ... // other stuff
}
```

3.2.2 Generating Events and *class.dump* at Run Time

Generating events and storing information into featured addresses

As shown in Table 3.1, Jikes RVM is responsible for generating those events and storing the relevant information into the featured addresses, at run-time. This task is completed by the Allocator and the Collector in Jikes RVM (as we have already seen in Figure 2.1). In code, three files are involved: `GCTk_InterleavedProfiler.java`, `GCTk_Allocator.java`, and `GCTk_Collector.java`. The first file defines and declares an `event` method, which is called in the latter two files. Upon being called, this method stores the information into featured addresses.

```
**************************************************
// ./rvm/src/vm/memoryManagers/GCTk/experimental/util/
// GCTk_InterleavedProfiler.java
**************************************************
class GCTk_InterleavedProfiler implements
GCTk_Constants, VM_Uninterruptible {
  final static int featureIdAddress = 0x7fffe000;
  final static int featureInfoAddress = 0x7fffe004;
  final static int featureInfoExtraAddress = 0x7fffe008;
  final static int featureInfoEnd = 0x7fffe00c;

  final static void event(int eventId, int start, int typeId) {
    VM_Magic.pragmaNoInline();
    setEventId(eventId);
    VM_Magic.setMemoryWord(featureInfoAddress, start);
    VM_Magic.setMemoryWord(featureInfoExtraAddress, typeId);
    VM_Magic.setMemoryWord(featureInfoEnd, 0);
  }

  // for Array
  final static void event(int eventId, int start,
                          int typeId, int size) {
    // Note: size of array object is stored within eventId
    event(eventId | (size << 5), start, typeId);
  }

  final static void setEventId(int eventId) {
```

```
    // store eventId into featureIdAddress
    VM_Magic.setMemoryWord(featureIdAddress, eventId);
  }
}

************************************************
// ./JikesRVM-2.0.3/rvm/src/vm/memoryManagers/GCTk/allocators/
// GCTk_Allocator.java
************************************************
abstract class GCTk_Allocator implements
GCTk_Constants, VM_Uninterruptible {

  // NEW_SCALAR event
  public static final Object allocateScalar
  (int size, Object[] tib, int allocator) {
    ... // other stuff

    //-#if RVM_WITH_GCTk_EXP_INTERLEAVED_PROFILE
    VM_Type type =
    VM_Magic.objectAsType(VM_Magic.getObjectAtOffset(tib,0));
    GCTk_InterleavedProfiler.event(NEW_SCALAR,
    VM_Magic.objectAsAddress(rtn), type.getDictionaryId());
    //-#endif
    ... // other stuff
  }

  // NEW_ARRAY event
  public static final Object allocateArray(
  int numElements, int size, Object[] tib, int allocator) {
    ... // other stuff

    //-#if RVM_WITH_GCTk_EXP_INTERLEAVED_PROFILE
    VM_Type type =
    VM_Magic.objectAsType(VM_Magic.getObjectAtOffset(tib,0));
    GCTk_InterleavedProfiler.event(NEW_ARRAY,
    VM_Magic.objectAsAddress(rtn), type.getDictionaryId(), size);
    //-#endif
    ... // other stuff
   }

  // COPY_SCALAR event
  private static final Object quickCloneScalar
  (Object fromObj, int allocator) {
```

```
    ... // other stuff

    //-#if RVM_WITH_GCTk_EXP_INTERLEAVED_PROFILE
    GCTk_InterleavedProfiler.event(COPY_SCALAR,
    VM_Magic.objectAsAddress(fromObj), dstRef );
    //-#endif
    ... // other stuff
  }

  // COPY_ARRAY event
  private static final Object quickCloneArray
  (Object fromObj, int allocator) {
    ... // other stuff

    //-#if RVM_WITH_GCTk_EXP_INTERLEAVED_PROFILE
    GCTk_InterleavedProfiler.event(COPY_ARRAY,
    VM_Magic.objectAsAddress(fromObj), dstRef);
    //-#endif
    ... // other stuff
  }

  ... // other stuff
}

************************************************
// ./JikesRVM-2.0.3/rvm/src/vm/memoryManagers/GCTk/collectors/
// GCTk_Collector.java
************************************************
abstract class GCTk_Collector extends VM_CollectorThread
implements GCTk_Constants,
VM_Uninterruptible,VM_Callbacks.ExitMonitor {

  ... // other stuff

  // CLASS_DUMP_START and CLASS_DUMP_END events
  public void notifyExit(int value) {
    ... // other stuff
    //-#if RVM_WITH_GCTk_EXP_INTERLEAVED_PROFILE
    GCTk_InterleavedProfiler.event(CLASS_DUMP_START);
    GCTk_ClassDump.dump();
    GCTk_InterleavedProfiler.event(CLASS_DUMP_END);
    //-#endif
  }
```

```
  // GC_START and GC_END events
  public static void gc(int allocator, int request) {
    ... // other stuff
    //-#if RVM_WITH_GCTk_EXP_INTERLEAVED_PROFILE
    GCTk_InterleavedProfiler.event(GC_START);
    //-#endif
    ... // other stuff
    //-#if RVM_WITH_GCTk_EXP_INTERLEAVED_PROFILE
    GCTk_InterleavedProfiler.event(GC_END);
    //-#endif
  }
  ... // other stuff
}
```

Generating *class.dump*

Thus far, we have seen that a numeric typeID has been saved for an object when it is allocated or copied. In the future, we need to map typeID to its real class name. So we generate the file, *class.dump*, which is used to store the mappings between classes and their numeric typeID's, along with other information as shown in Table 3.2. Some of these information, such as `Field offset`, is useful when we determine which field is accessed during a read/write. We will see further discussions in Section 3.4.

Table 3.2: Data saved in *class.dump*.

Data	*Descriptions*
type ID	a numeric value of a class. For example, 1 is mapped to void.
Class Name	a "real" name of a class, for example, `java.lang.String`.
Size of instance	size (in bytes) of an instance for a scalar class.
No. of fields	number of fields of a scalar class.
Field offset	offset of each field for a scalar class.
Field typeID	typeID of each field for a scalar class.
Component typeID	typeID of the components for an array class.
SuperClass Name	a "real" name of its super class.

```
************************************************
// ./rvm/src/vm/memoryManagers/GCTk/experimental/util/
// GCTk_ClassDump.java
************************************************
class GCTk_ClassDump implements
GCTk_Constants, VM_Uninterruptible {
  final static void dump() {
    try {
      ...
      VM_Type[] typeList = VM_TypeDictionary.getValues();
      for (int i=1; i < typeList.length; i++) {
        VM_Type vt = typeList[i];

        // Class Type
        if (vt.isClassType()) {
          ...
        }

        // Array Type
        if (vt.isArrayType()) {
          ...
        }
      }
    }
  }
}
```

3.3 Modification of Dynamic SimpleScalar

On the side of Dynamic SimpleScalar, it is responsible for capturing the events generated by Jikes RVM and extracting the saved information from the featured addresses during data accesses. Other than that, it also needs to save the regular access information (as shown in Table 3.3). It works by repeatedly storing these information into a local buffer, and then flushing the buffer to a compressed file, *profile.bz2*, as the buffer becomes full. This can be seen from source files *profile.h* and *profile.c*. The method `profiling_access` is called as long as there is an access. If a data access is to the address FEATURE_END, then it calls

`add_feature` to add *(eventID, start address, and typeID)* into the buffer; Otherwise (a regular access), it calls `add_access` to add *(timestamp, access address, is_hit, is_read, and value)* into the buffer. Note that, to save the disk space, we only consider the case that the *value* is a *reference* to other objects. In other words, we ignore primitive types.

Table 3.3: Data saved in *profile.bz2*.

Data	*Descriptions*
is_feature	a flag indicating if this record is a feature or a regular access.
eventID	a numeric ID for an event generated by Jikes RVM.
start address	the object reference pointer of an object.
typeID	the type of an object, class type or array type.
timestamp	the time (in instructions) when a read/write access takes place.
access address	the access address.
is_hit	a flag indicating if this access is a hit or miss.
is_read	a flag indicating if this access is a read or write.
value	the value is read/written at the access address.

```
************************************************
./dssppc/
profile.[hc]
************************************************
... /* other method */

void profiling_access(md_addr_t addr,  /* address to access */
                      char is_hit,     /* is a hit or miss */
                      enum mem_cmd cmd,/* Read or Write */
                      int nbytes)      /* # of bytes to access */
{
  if (addr >= FEATURE_ADDRESS && addr <= FEATURE_ADDRESS_EXTRA) {
    return; /* skip, waiting for next access */
  } else if (addr == FEATURE_END) {
    /* read the values stored at featured addresses */
    unsigned int event_id = read_mem_word(FEATURE_ADDRESS);
    unsigned int s_addr = read_mem_word(FEATURE_ADDRESS_NEXT);
    unsigned int type_id = read_mem_word(FEATURE_ADDRESS_EXTRA);

    ... /* other stuff */
```

```
    add_feature(event_id, s_addr, type_id, sim_num_insn);
  } else { /* regular access */

    unsigned int value = 0;
    if (nbytes == 4)
      value = read_mem_word(addr);
    char is_read = (cmd == Read) ? 1 : 0;
    add_access(sim_num_insn, addr, is_hit, is_read, value);
  }
}

/* see syscall.c */
void add_munmap_feature(word_t start_address, size_t len) {
   add_feature(MEM_UNMAP, start_address, len, sim_num_insn);
}

... /* other method */
```

We choose to use `sim-cache`, a functional cache simulator in SimpleScalar, to simulate the data cache accesses. As such, we need to modify `sim-cache.c` and `cache.c`. The method `profiling_access` in `profile.c` gets called in the method `cache_access` in `cache.c`. The following piece of code shows the modifications of the method `cache_access`.

```
************************************************
./dssppc/
cache.c
************************************************
unsigned int cache_access
(struct cache_t *cp,    /* cache to access */
 enum mem_cmd cmd,      /* access type, Read or Write */
 md_addr_t addr,        /* address of access */
 void *vp,              /* ptr to buffer for input/output */
 int nbytes,            /* number of bytes to access */
 tick_t now,            /* time of access */
 byte_t **udata,        /* for return of user data ptr */
 md_addr_t *repl_addr)  /* for address of replaced block */
{
  ... /* other stuff */
```

```
  cp->misses++;

  #ifdef JVM_PROFILING
  if(!strcmp(cp->name,"dl1"))
    profiling_access(original_addr, 0, cmd, nbytes);
  #endif

  ... /* other stuff */

  cache_hit: /* slow hit handler */
    cp->hits++;
    #ifdef JVM_PROFILING
    if(!strcmp(cp->name,"dl1"))
      profiling_access(original_addr, 1, cmd, nbytes);
    #endif

  ... /* other stuff */

  cache_fast_hit: /* fast hit handler */
    cp->hits++;
    #ifdef JVM_PROFILING
    if(!strcmp(cp->name,"dl1"))
      profiling_access(original_addr, 1, cmd, nbytes);
    #endif

  ... /* other stuff */
}
```

One of the most important modifications should be emphasized here. In Jikes RVM, once a cycle of garbage collection finishes, it makes a AIX system call *munmap* through a PowerPC instruction *sc*. The *munmap* system call in AIX is used to zero out some portion of memory locations. By calling *munmap*, the garbage collector in Jikes RVM gets rid of the dead, or copied objects, which was occupying that portion of memory. We need to record this event (See MUNMAP in Table 2.1). We use a method called `add_munmap_feature` in `profile.c` to do so. This method gets called in *munmap* system call handler, `syscall_munmap`, which can be found in `syscall.c`. See below for details.

```
************************************************
./dssppc/
syscall.c
************************************************
... /* other stuff */

#ifdef JVM_PROFILING
extern void add_munmap_feature();
#endif

... /* other stuff */
void syscall_munmap
(struct regs_t *regs, struct mem_t *mem) {
  void *addr;
  size_t len;

  addr = (void *)regs->regs_R[3];
  len = (size_t)regs->regs_R[4];
  mem_munmap(mem,(md_addr_t)addr, len);
  regs->regs_R[3] = 0;

  #ifdef JVM_PROFILING
    add_munmap_feature(addr, len);
  #endif
}
```

3.4 Building the Analyzer

The Analyzer processes the files generated by Jikes RVM (*feature.dump* and *class.dump*) and Dynamic SimpleScalar (*profile.bz2*), and generates the analytical results into a file, *trace.bz2*, as shown in Figure 2.1 in Chapter 2.

First of all, the class information saved in *class.dump* is read into an array buffer `class_table`, every entry of which is a structure `VM_Class_Layout` as declared in `analyzer.h`.

Secondly, it reads the information of Jikes RVM objects from *feature.dump* into an

array buffer, `mem_layout`, with every entry being a structure `Memory_Layout` holding the object data, such as *start address*, *instance ID*, and *type ID*, etc. This is done by calling a method `update_memory_layout` in `analyzer.c`. This method takes separate actions in terms of different *events* it read from *feature.dump*. For a NEW_SCALAR or a NEW_ARRAY, it inserts this object into `mem_layout`. For a COPY_SCALAR, or a COPY_ARRAY, it first finds out the object from current `mem_layout` using *source address* and then copies (inserts) this object into a new place in `mem_layout` according to *destination address*. Both *source address* and *destination address* are read from *feature.dump* (See Table 3.1). Finally, for a MEM_UNMAP event, it calls a method `clear_memory_layout` to clear up the dead or copied objects in `mem_layout` (Reflecting the fact that a garbage collection removes these objects from certain area of memory locations).

Thirdly, it reads *profile.bz2* by calling the method `read_profile`. Refer back to Table 3.3 to see the data saved in *profile.bz2*. In `read_profile`, it checks the first field `is_feature`. If this field is true, it grabs the feature information (*eventID, start address*, and *typeID*) and calls `update_memory_layout`; Otherwise it extracts the access information (*timestamp*, *access address*, *is_hit*, *is_read*, and *value*) and calls another method `register_access` to find out which object (and then which fields or which components) was accessed, by doing a binary search over `mem_layout`. The results are buffered into `access_trace` declared in `analyzer.c`, whose entry is a structure `Access_Trace`, as seen in `analyzer.h`.

Finally, the `access_buffer` is flushed into a file, *trace.bz2*, while it fills up.

Further analyses can be carried out on the basis of the results in *trace.bz2*. For instance, given a particular field, of a specific object with a specific class type, we can obtain the *write* statistics on this field, which include: 1) total number of *write* that this field is written to; 2) total number of reads after each *write*; and 3) the time passed between *write*. This analysis will be further discussed in Chapter 4.

To summarize, Table 3.4 shows the data which is stored in *trace.bz2*.

Table 3.4: Data saved in *trace.bz2*.

Data	*Descriptions*
access_time	timestamp in instructions when an access occurs.
is_read	Is this access a read or write?
is_hit	Is this access a hit or miss?
copied	How many times this object copied by GC
instance_id	Which instance object is accessed?
type_id	class type of this instance object.
field_id	Which field of this instance is accessed
ref_obj_id	Which object is referenced by this field?
ref_obj_type_id	What is the type of referenced object

```
************************************************
./analyzer
analyzer.h
************************************************
... /* other stuff */
typedef struct{
  unsigned long long access_time;
  char is_read;
  char is_hit;
  unsigned int instance_id;
  short copied;
  unsigned int type_id;
  unsigned int field_id;
  unsigned int ref_obj_id;
  unsigned int ref_obj_type_id;
} Access_Trace;

/* Field of an Class Instance */
typedef struct {
  short  offset;         /* offset from the ref point */
  unsigned int type_id;  /* the type of this field */
} Field;

typedef struct {
  char *className;
```

```
  char *superClassName;
  unsigned int type_id; /* 0 id of this class*/
  int num_fields;       /* number of 0 of this class */
  Field *fields;        /* fields of this class */
  int size;             /* size of class instance */
  int total_objects;    /* # of objects created for this class */
} VM_Class_Layout;

/* Object info */
typedef struct {
  char is_scalar;           /* is a scalar or an array object */
  unsigned start_address;   /* stating address of an object */
  unsigned int instance_id; /* unique id of this instance */
  unsigned int type_id;     /* type id of the class */
  char  dead;               /* dead or alive */
  unsigned int size;        /* instance size */
  short copied;             /* number of times copied by GC */
} Memory_Layout;

... /* other stuff */


************************************************
./analyzer
analyzer.c
************************************************
Access_Trace access_trace[TRACE_BUFFER_SIZE];
VM_Class_Layout class_table[MAX_NUM_CLASSES];
Memory_Layout mem_layout[MAX_NUM_OBJECTS];

... /* other stuff */

void update_memory_layout(Feature_Access feature) {
  switch (feature.event) {
  case NEW_SCALAR:
    insert_new_object(1, feature.type_id,
                      feature.start_addr, 0, 0, 0, 0);
    break;
  case GC_START:
    ...
  case GC_END:
    ...
  case COPY_SCALAR:
```

```
    ...
  case COPY_ARRAY:
    ...
  case MEM_UNMAP:
    clear_memory_layout(feature.start_addr, feature.type_id);
    break;
  case 0:
    break;
  default: /* NEW_ARRAY */
    insert_new_object(0, feature.type_id, feature.start_addr,
    0, 0, ((feature.event - NEW_ARRAY) >> 5), 0);
    break;
  }
}

void insert_new_object(
char is_scalar,          /* 1 - scalar object; 0 - array */
word_t type_id,          /* class type */
word_t start_address,    /* object pointer */
short status,            /* dead? or live? */
unsigned int instance_id,/* which object it is */
unsigned int size,       /* object size (w/ header) */
short copied)            /* number of times copied */
{
  unsigned int new_start_addr;
  /* tem_mem_layout gets full, merge objs to mem_layout */
  if (tmp_obj_num == MAX_TMP_OBJ_NUM)
    merge_memory_objs();

  /* start address,
     ...|tib|status|   |   | for scalar object
          ^            ^ <-------- Objref
       start addr

        |tib|status|len|  |... for array object
          ^            ^ <-------- Objref
       start addr
 */
  new_start_addr = start_address + OBJECT_HEADER_OFFSET;

  if (status == 0) { /* new allocated object */
    class_table[type_id].total_objects++;
    if (is_scalar) { /* new scalar object */
```

```
      assert(is_scalar == 1);
      insert_tmp_object(is_scalar, type_id, new_start_addr,
      status, class_table[type_id].total_objects,
      class_table[type_id].size, copied);
    } else {
      assert(is_scalar == 0);
      insert_tmp_object(is_scalar, type_id, new_start_addr,
      status, class_table[type_id].total_objects,
      size, copied);
    }
  } else if (status == -1) { /* insert copied object */
    insert_tmp_object(is_scalar, type_id, new_start_addr,
                      status, instance_id, size, copied);
  }
}

... /* other stuff */
```

```
+ ./
   + DynamicSimpleScalar-TOOL/
      + JikesRVM-2.0.3/
         + build/
            + PowerPC32-AIX/
               + GCTkAppelOLWBOptOptFastTimingDSSEXPINTERLEAVED/
                  + RVM.classes
                  - RVM.image
                  - ...
         + Booter/
            - JikesRVM
         + rvm/
            + config/
               + build/
                  - GCTkAppelOLWBOptOptFastTimingDSSEXPINTERLEAVED/
            + src/
               + tools/
                  + bootImageWriter/
                     - BootImageWriter2.java
                  + bootImageRunner/
                     - RunBootImage.C
               + vm/
                  - VM.Constants.java
                  - VM.java
                  - VM_ObjectLayoutConstants.java
                  + memoryManagers/
                     + GCTk/
                        + allocators/
                           - GCTk_Allocator.java
                        + collectors/
                           - GCTk_Collector.java
                        + experimental/
                           + util/
                              - GCTk_ClassDump.java
                              - GCTk_InterleavedProfiler.java
                        + util/
                           - GCTk_BuildFeatureDump.java
      + dssppc/
         - cache.[hc]
         - main.c
         - memory.[hc]
         - profile.[hc]
         - sim-cache.c
         - syscall.c
      + benchmarks/
         + spec/
            + jvm98/
            + pseudojbb/
      + analyzer/
         - analyzer.[hc]
         - Makefile
         - readtrace.c
         - utils.c
```

Figure 3.1: Directory tree of the tool.

Chapter 4

Running the Tool

This chapter focuses on a sample usage of the tool. After having talked about the methodologies and implementations of the tool, this is a good place to illustrate how we run the tool, utilizing some Java benchmarks.

4.1 Benchmarks

We choose to run Java benchmarks from SPECjvm98 suite [Sta99]. Most of the benchmarks in the suite are either real applications or derived ones from real applications that are commercially available. The SPECjvm98 suite benchmarks are used to evaluate the performance of both software and hardware aspects of Java platforms. We use three Java benchmarks: _202_jess, _213_javac, and _228_jack, as described in the following:

_202_jess *JESS* is the Java Expert Shell System based on NASA'S CLIPS expert shell system. In simplest terms, an expert shell system continuously applies a set of `if-then` statements, called rules, to a set of data, called the fact list. The benchmark workload solves a set of puzzles commonly used with CLIPS. To increase run

time the benchmark problem iteratively asserts a new set of facts representing the same puzzle but with different literals. The older sets of facts are not retracted. Thus the inference engine must search through progressively larger rule sets as execution proceeds.

_213_javac *JAVAC* is a Java compiler from the JDK 1.0.2.

_228_jack *JACK* is a Java parser generator that is based on the Purdue Compiler Construction Tool Set (PCCTS). The workload consists of a file named jack.jack, which contains instructions for the generation of *jack* itself. This is fed to *jack* so that the parser generates itself multiple times.

In our experiments, we run the three benchmarks as the stand-alone applications, rather than as applets, using the smallest input data set size by specifying `-s1` at the command line. We run these benchmarks to completion. The heap size is set to be equal to 200MB through a command line argument, `-X:h=200`. Using this heap size, we observe that there are two cycles of garbage collections invoked, for all three benchmarks.

4.2 Experiments

We build and run our tool set on SUN's UltraSPARC model running the Solaris 5.9 operating system. Actually, building the tool involves several stages, to build each component of the tool. See Appendix A for the detailed instructions on how to build and run the tool. The Java program runs on top of Jikes RVM, which runs on top of Dynamic SimpleScalar. Dynamic SimpleScalar finally runs on UltraSPARC/Solaris platform. We invoke the tool in a single command line, which contains particular commands to invoke each component in certain order.

Jikes RVM is built with the configuration of `GCTkAppelOLWBOptOptFast`

`TimingDSSEXPINTERLEAVED`[1]. `GCTkAppelOLWB` means that we use the Appel generational copying collector in GCTk, with the *out-of-line write barriers* implementation. `OptOpt` says that both boot image compiler and run-time compiler are *optimizing* compiler. `Fast` identifies that this is a configuration where all Jikes RVM classes are included in the boot image, but all assertion checkings have been turned off. `Timing` means that there will be no detailed statistics and `DSS` indicates that this is a special configuration to build Jikes RVM such that it can run on top of Dynamic SimpleScalar. Finally, `EXPINTERLEAVED` says that it needs to build the sources found in directory ./GCTk/experimental, where all our newly-added source files reside.

We use `sim-cache` simulator of Dynamic SimpleScalar. Table 4.1 lists the parameters for a two-level cache system we used.

Table 4.1: `sim-cache` cache configuration parameters

Name	*Descriptions*
L1 Data Cache	32KB, 8-way set associative LRU, 32B cache lines
L1 Instruction Cache	32KB, 8-way set associative LRU, 32B cache lines
L2 Caches	Unified, 256KB, 8-way set associative Random, 64B lines
Data TLB	128 entries, 2-way set associative LRU, 4KB page size
Instruction TLB	128 entries, 2-way set associative LRU, 4KB page size

4.3 Results and Discussion

In this section, we first present some direct results over the entire run of the benchmarks that we can obtain through the tool. These results include, for example, total number of objects allocated, total bytes allocated, and total number and total bytes of dead/copied

[1]This configuration file can be found in the directory ./JikesRVM-2.0.3/rvm/config/build/, along with other generic configurations it includes, such as `GCTkTiming`, `GCTkOptOptFast`, `GCTkAppelOLWB`, and `DSS`.

objects during garbage collections. we also discuss the *write* statistics on a particular field of an object with certain type. This is an example illustrating how our tool can be used to do such *read/write* analyses on some interesting objects at the *field* level. Then, we present the memory hierarchy behaviors of the three Java benchmarks.

4.3.1 Results from the Analyzer and Further Analysis

The off-line Analyzer generates the access traces by collecting and analyzing the results produced by Jikes RVM and Dynamic SimpleScalar. Recall that these traces include the access information (access time, hit or miss, and read or write) and object information (object id, type id, and field id).

Some Direct Results

We observe that there are two cycles of garbage collections invoked, for all three benchmarks. Figure 4.1 sketches the interleaving of *mutators* (user programs) and *garbage collections* during the execution of benchmarks. The mutator executes for some time. At some point, the garbage collector is triggered to collect the dead objects and to copy the surviving objects into new heap space. The mutator is suspended during garbage collection. After the collector ends, the mutator then continues.

Table 4.2 shows the timestamps (in *instructions*) when the garbage collections start and end, and the total instructions executed for each benchmark. Three benchmarks bear some similarities with the time when the first cycle of garbage collection starts, that is, around $40 \sim 50\%$ of total instructions (see *% of total* column in the table). Table 4.3 shows the total number of objects and total bytes allocated during the execution of benchmarks. As can be seen in the table, the first cycle of garbage collection is triggered when about 133 MBytes are allocated and the second cycle is triggered when about another 101 MBytes

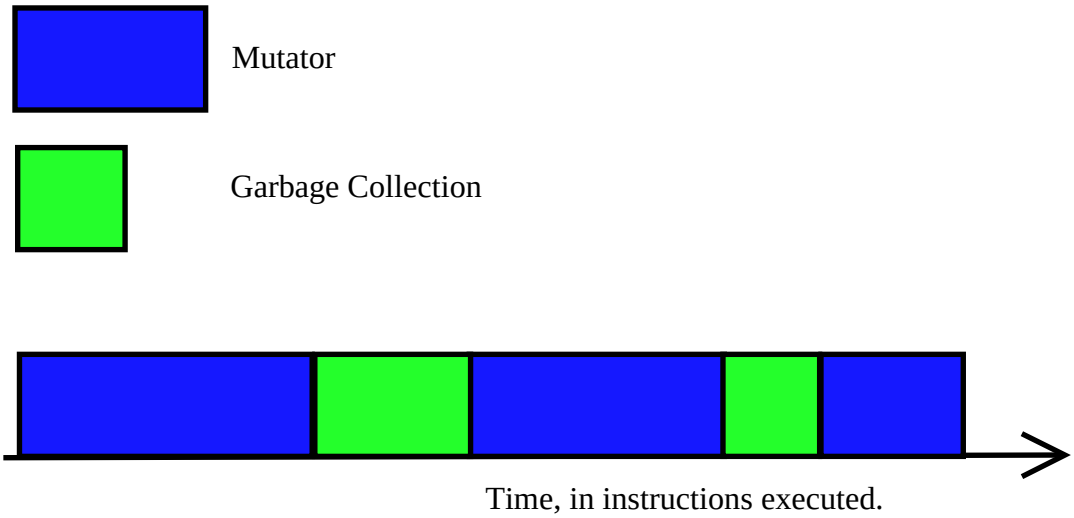

Figure 4.1: Sketches of the interleavings of mutators and garbage collections during the execution of benchmarks.

are allocated.

Table 4.2: Timestamps in instructions when a gc starts and ends.

Benchmark	Total *Insts* ($\times 10^6$)	*Garbage Collection Timestamp* ($\times 10^6$)					
		start	*% of total*	*end*	*start*	*% of total*	*end*
_202_jess	4,100	1,660	40.5	1,732	3,815	93.1	3,879
_213_javac	4,340	2,087	48.1	2,177	3,776	87.0	3,821
_228_jack	4,648	2,116	45.5	2,221	3,967	85.3	4,058

Table 4.3: Total objects and bytes allocated.

	_202_jess		*_213_javac*		*_228_jack*	
	Objects	*Bytes*	*Objects*	*Bytes*	*Objects*	*Bytes*
Up to the 1st GC	3061866	133467437	3113702	133467537	3036959	133467353
Up to the 2nd GC	5880936	234916201	5942117	234654181	5839946	234260777
Total	6182478	246160897	6765036	264619837	6747806	267382605

Table 4.4 and Table 4.5 show the total number and total bytes of dead and copied objects within each cycle of garbage collection, respectively. It is evident that during each cycle of garbage collection, most objects are dead and much less objects survive. For instance, in _202_jess, about 98% objects are dead during the first cycle of garbage collection. This observation somehow verifies that the hypothesis of the Appel garbage collection algorithms, that is, that most objects are short-lived. One may notice that within each garbage collection, the sum of the number of dead objects (In Table 4.4) and the number of copied objects (In Table 4.5) is not equal to (actually, less than) the number of total objects (In Table 4.3). This is because we also count the number of boot image objects in Table 4.3. There are about 264316 boot image objects, and totally around 29 MBytes.

Table 4.4: Total *dead* objects and bytes within each GC.

	_202_jess		*_213_javac*		*_228_jack*	
	Deads	*Bytes*	*Deads*	*Bytes*	*Deads*	*Bytes*
Within the 1st GC	2765687	101578628	2808251	101299964	2722924	100997988
Within the 2nd GC	2785977	99956740	2807151	99984348	2759000	98667580
Total	5551664	201535368	5615402	201284312	5481924	199665568

Table 4.5: Total *copied* objects and bytes within each GC.

	_202_jess		*_213_javac*		*_228_jack*	
	Copied	*Bytes*	*Copied*	*Bytes*	*Copied*	*Bytes*
Within the 1st GC	31863	2882068	41135	3160832	49719	3462624
Within the 2nd GC	33093	1492024	21264	1202296	43987	2125844
Total	64956	4374092	62399	4363128	93706	5588468

***Write* Statistics**

In this section, we present some results obtained by applying further analysis on the tracing results generated by the Analyzer, that is, *read/write* statistics on some field of objects. The motivation, as illustrated in Figure 4.2, is that we want to record the information on the read/write accesses on a particular field of an object, such as total number of *writes*, total number of *reads* after the field is written to, and the interval between two subsequent *writes*.

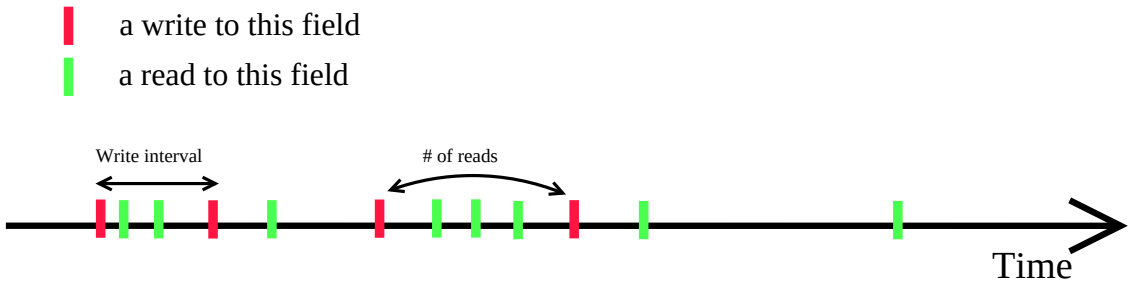

Figure 4.2: Illustration of read/write analysis on a particular field of an object with certain type.

We report the results for some scalar objects and some array objects, which have survived from the first cycle of garbage collection – by a preliminary analysis, we notice that these objects are copied by the collector during the first cycle of collection. The class type for scalar objects is `java.lang.String` and it has three fields. Field #1 has the type of `char` array. Both field #2 and field #3 have `int` type. The class type for array objects is `[Ljava.lang.Object`, which means that each component is with the type of `java.lang.Object`. We choose these objects in one of the benchmarks, _228_jack.

Tables 4.6 and 4.7 show the results for some different `String` objects. There is typically one write, after which two reads follow. The interval between the first write access to the last read is about 53 instructions (57 in Table 4.7). Note that we only show the results for accessing the field #1 since the results for the other two fields are the same.

Table 4.6: Read/Write analysis results on a String object.

Class type: `java.lang.String`; Type ID: 12; Object ID: 50830; Field ID: 1		
No. of Writes	*No. of reads*	*Intervals (in instructions)*
1	2	53

Table 4.7: Read/Write analysis results on a String object.

Class type: `java.lang.String`; Type ID: 12; Object ID: 128990; Field ID: 1		
No. of Writes	*No. of reads*	*Intervals (in instructions)*
1	2	57

Table 4.8 and Table 4.9 list the results for two different array objects with the same type. As we can see, after a write, there are normally a few reads (mostly $1 \sim 5$). Following the last write, there are relatively large number of reads. For example, there are 12 reads following the last write for Object 6525 (69 reads for Object 9481). There is no observable pattern on the interval between two subsequent writes.

Table 4.8: Read/Write analysis results on an array object.

Class type: `[java.lang.Object`; Type ID: 95; Object ID: 6525; Element ID: 1		
No. of Writes	*No. of reads*	*Intervals (in instructions)*
1	1	6
2	2	121
3	2	220
4	1	94
5	1	168
6	12	4327

Table 4.9: Read/Write analysis results on an array object.

Class type: `[java.lang.Object`; Type ID: 95; Object ID: 9481; Element ID: 1		
No. of Writes	*No. of reads*	*Intervals (in instructions)*
1	1	6
2	1	6
3	1	6
4	1	6
5	5	202
6	3	307
7	1	6
8	1	6
9	3	148
10	116	18418
11	1	94
12	234	35463
13	1	6
14	2	121
15	2	220
16	1	94
17	1	168
18	69	10315

4.3.2 Memory Hierarchy Behaviors

This section reports the results from `sim-cache` simulations. We list the miss ratios (of all levels of caches), total executed instructions, and loads/stores instructions, as shown in Table 4.10. In the last column of this table, We also show the percentage of data references.

We roughly compare our results with those data collected by Shuf et al. [SSGS01] in their experiments where they used Jikes RVM, SPECjvm98 Java benchmarks, and similar cache system parameters with ours[2]. We observe that there are relatively low miss ratios

[2]The cache parameters they used: 1) L1 data cache: 32KB, 4-way set associative LRU, 32B lines; 2) L2 data cache: 4MB, direct-mapped, 32B lines; and 3) data TLB: 128 entries, 2-way associative LRU, 4K pages.

Table 4.10: Results from `sim-cache` simulations.

	Miss Ratios					*Instructions* ($\times 10^6$)		
Benchmark	I–L1	D–L1	L2	I–TLB	D–TLB	Total	Load/Store	%
_202_jess	.0061	.0209	.2556	.0003	.0034	4,100	1,828	44.60
_213_javac	.0060	.0207	.2446	.0003	.0031	4,340	1,944	44.80
_228_jack	.0052	.0187	.2506	.0003	.0031	4,649	2,093	45.03

for both instruction L1 ($\sim$ 0.6% for all three benchmarks) and instruction TLB caches ($\sim$ 0.3% for all three benchmarks). These are consistent with their results. It is interesting to see that the miss ratio for data TLB is about 0.3%, but as high as 2% in their observations. Note that they use the same TLB configurations as ours. For data L1 cache, we also get lower miss ratio, $\sim$ 2% than $\sim$ 4% in their case. Surprisingly enough, we observe very high miss ratios in L2 cache, about 25%. This result is not good, compared to $\sim$ 1.5% in their data. As they noted, even $\sim$ 1.5% is also a very high miss ratio considering the very large L2 cache size (4MB) they used. The good indication, however, is that such high miss ratios in L2 cache do not effect too much on the performance of L1 caches. This may imply that there is no need of the L2 cache at all, for Java programs. In this work, we do not seek to investigate further on the cache behaviors of these benchmarks. It is beyond the scope of this work.

Chapter 5

Conclusions and Future Work

We have described the methodologies and implementations we used to develop the tool, which is used for monitoring and recording the heap-allocated object behaviors.

Our main contributions are:

- We present a fully understandable description on the design, implementation, and evaluation of the tool. We show how the three components, Jikes RVM, Dynamic SimpleScalar, and Analyzer, can be organized to run and test the Java programs. To our knowledge, this tool, as the first attempt, combines the machine-level simulator with state-of-the-art Java virtual machine to analyze data accesses at the object level.

- We present a successful run of Java benchmarks. We illustrate how to build, run, and use the tool.

- We present a sample analysis on the *write* statistics to some particular fields of objects with some class type, which include the number of *writes*, the interval between *writes*, and the *read* pattern among the *intra-writes*. Such analyses allow us to understand the low-level activities on some "hot" types.

This work leads to some interesting topics we can do in the future:

- To deal with the accesses to the object headers, particularly TLB field. This field contains a reference to the TIB (*Type Information Block* of this object's class. In TLB, an array of object references, it contains the references to the method bodies for the virtual methods of this object's class. We could know the access information to the *methods* of the class, other than *fields*.
- To deal with the accesses to each component of the array. Actually, in the current version of the code (Analyzer), we have option (by 0 on the switch ARRAY when compiling the code) to treat every access to the components. But this operation "slows down" the Analyzer.

Appendices

Appendix A

Technical Instructions

A.1 Directory Tree

Refer to Figure 3.1 in Section 3.1 of Chapter 3.

A.2 Quick Guide to Build and Run the Tool

Unless otherwise noted, $HOME_TOOL is an environment variable that represents the directory where the tool resides in. Further details on the contents of this part can be found in my thesis notes[1].

[1]It notes the experiences I went through when I was building Jikes RVM and Dynamic SimpleScalar on the machines at the Object Architecture Lab. It also contains the information on the SPEC2000, SPECjvm98, and pseudojbb.
See `http://www.cs.unm.edu/~qfduan/cs/thesisnotes.txt`

A.2.1 Build the Tool

Build Jikes RVM v2.0.3

This is a cross-build. The *boot image* RVM.image is built on x86/Linux (epsilon machine at the Object Architecture Lab). The *boot loader* Jikes.RVM is precompiled on a PowerPC/AIX machine, and distributed together with the sources of Jikes RVM. This file can be found in $RVM_ROOT/Booter.

1. Set up the environment variables.
   setenv RVM_ROOT $HOME_TOOL/DynamicSimpleScalar-TOOL/JikesRVM-2.0.3
   setenv PATH $RVM_ROOT/rvm/bin:$PATH
   setenv RVM_HOST_CONFIG $RVM_ROOT/rvm/config/i686-pc-linux-gnu.ibmjdk
   setenv RVM_TARGET_CONFIG $RVM_ROOT/rvm/config/powerpc-ibm-aix4.3.3.0.static
   setenv CONFIGURATIONNAME GCTkAppelOLWBOptOptFastTimingDSSEXPINTERLEAVED
   setenv RVM_BUILD $RVM_ROOT/build/PowerPC32-AIX/$CONFIGURATIONNAME

2. Edit the configuration scripts.
   Two configuration scripts are needed in the subsequent installing process.
   `i686-pc-linux-gnu.ibmjdk`
   `powerpc-ibm-aix4.3.3.0.static`.

3. Run `jconfigure`.
   Run `jconfigure` script (which resides in $RVM_ROOT/rvm/bin) and populate the build directory $RVM_BUILD, using a particular configuration.

   ```
   $ jconfigure $CONFIGURATIONNAME
   ```

   The configuration, `GCTkAppelOLWBOptOptFastTimingDSSEXPINTERLEAVED` can be found in $RVM_ROOT/rvm/config/build.

4. Build.

   Run `jbuild` script (which resides in $RVM_ROOT/rvm/bin) and build Jikes RVM.

   ```
   $ cd $RVM_BUILD
   $ jbuild
   ```

   After the `jbuild` completes successfully, you will get `Please run me on Aix`, indicating that Jikes RVM has been successfully built and ready to run.

NOTE: *feature.dump* is stored into tmp/ and you need to move it to your directory which saves the results, say /nfs/sampi/dss-jikes/javac/.

Build Dynamic SimpleScalar

In `Makefile`, there are two switches should be turned on:

```
-DJVM_PROFILING
-D_FILE_OFFSET_BITS=64
```

This version of Dynamic SimpleScalar can be built on the UltraSPARC/Solaris platform (sampi machine at the Object Architecture Lab).

```
$ make config-ppc
$ rm -f machine.def
$ ln -s target-ppc/powerpc-nonnative.def ./machine.def
$ make sim-cache
```

Build Analyzer

Analyzer is also installed on UltraSPARC/Solaris machine (sampi machine at the Object Architecture Lab).

There is a `Makefile`, so just type:

```
$ make
```

A.2.2 Run the Tool

The entire tool runs on UltraSPARC/Solaris platform. There are two steps: 1) Run the benchmarks and save the results; and 2) Run the Analyzer.

Step 1: Run the benchmarks and save the results

All Java benchmarks can be found in directory $HOME_TOOL/DynamicSimpleScalar-TOOL/benchmarks, which is stored in an environment variable $BENCHMARKS.

For example, to run a SPECjvm98 benchmark _213_javac:

```
$ cd $BENCHMARKS/spec/jvm98
$ ./runme.sh
```

The `runme.sh` is a script file contains the following commands to run the whole thing:

1. `$HOME_TOOL/DynamicSimpleScalar-TOOL/dssppc/sim-cache`
2. `$PPC_SS_CACHE`
3. `-output /nfs/sampi/dss-jikes/javac/`
4. `-redir:sim /nfs/sampi/dss-jikes/jack/sim.javac.out`
5. `$RVM_ROOT/Booter/JikesRVM -X:i=$RVM_BUILD/RVM.image`
6. `-X:vmClasses=$RVM_BUILD/RVM.classes -X:h=200`
7. `-classpath .  SpecApplication -s1 _213_javac`

It should be noted that all of the commands should be put in a single command line. We add numbers there clearly for the ease of explanations.

Line 1 is for the Dynamic SimpleScalar command `sim-cache`.
Line 2, $PPC_SS_CACHE, stands for the cache organization we used for `sim-cache`:

```
-cache:dl1 dl1:128:32:8:l
-cache:il1 il1:128:32:8:l
-cache:dl2 ul2:512:64:8:r
-cache:il2 dl2
-tlb:dtlb dtlb:64:4096:2:l
-tlb:itlb itlb:64:4096:2:l
```

Line 3 specifies the directory that we use to store the output file, *profile.bz2*.

Line 4 is used to redirect the `sim-cache` output file.

Line 5 states that we use the boot loader JikesRVM to load image file RVM.image and virtual machine classes in RVM.classes (in Line 6).

Line 6 uses `-X:h=200` to indicate that 200 MBytes heap size will be used.

Line 7 specifies the arguments to run the benchmark.

NOTE: You need to move *class.dump* to the directory where you save the results.

Step 2: Run the Analyzer

Suppose the results from step 1 (*feature.dump, class.dump, and profile.bz2*) have been collected in a single directory, /nfs/sampi/dss-jikes/javac/, for example. We can invoke Analyzer in this directory:

```
$ $HOME_TOOL/DynamicSimpleScalar-TOOL/analyzer/analyzer
```

The Analyzer can be invoked as follows:

```
$ analyzer profile.bz2 feature.dump class.dump
```

Finally, it generates a file, *trace.bz2*, and other files as necessary, into the same directory where you invoked it.

References

[AAB+99] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *ACM Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, 1999.

[AAB+00] B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russel, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1):211–237, 2000.

[App89] A. W. Appel. Simple Generational Garbage Collection and Fast Allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[BA97] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News*, 25(3):13–25, 1997.

[CGZ94] B. Calder, D. Grunwald, and B. Zorn. Quantifying Behavioral Differences between C and C++ Programs. *Journal of Programming Languages*, 2(4):313–351, 1994.

[HHDH02] M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the Connectivity of Heap Objects. In *International Symposium on Memory Management*, Berlin, Germany, 2002.

[HMDW91] R. L. Hudson, J. E. B. Moss, A. Diwan, and C. F. Weight. A Language-independent Garbage Collector Toolkit. Technical report 91-47, Department of Computer Science, University of Massachusetts at Amherst, 1991.

[HMM+03] X. Huang, J. E. B. Moss, K. S. McKinley, S. Blackburn, and D. Burger. Dynamic SimpleScalar: Simulating Java Virtual Machines. Technical report

tr-03-03, Department of Computer Science, University of Texas at Austin, 2003.

[JL96] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[KH00] J.-S. Kim and Y. Hsu. Memory System Behavior of Java Programs: Methodology and Analysis. In *2000 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Santa Clara, California, 2000.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[SGBS02] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting Prolific Types for Memory Management and Optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Portland, Oregon, 2002.

[SHB+02] D. Stefanovic, M. Hertz, S. M. Blackburn, K. S. McKinley, and J. E. B. Moss. Older-first Garbage Collection in Practice: Evaluation in a Java Virtual Machine. In *ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, 2002.

[SMM99] D. Stefanovic, K. S. McKinley, and J. E. B. Moss. Age-Based Garbage Collection. In *Proceedings of SIGPLAN 1999 Conference on Object-Oriented Programming, Systems, Languages, & Applications*, volume 34, pages 370–381, 1999.

[SNKB01] K. Sankaralingam, R. Nagarajan, S. Keckler, and D. Burger. SimpleScalar Simulation of the PowerPC Instruction Set Architecture. Technical report tr-00-04, Department of Computer Science, University of Texas at Austin, 2001.

[SSGS01] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the Memory Behavior of Java Workloads: A Structured View and Opportunities for Optimizations. In *SIGMETRICS 2001 / Performance 2001 Joint International Conference on Measurement and Modeling of Computer Systems*, Cambridge, Massachusetts, 2001.

[Sta99] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, March 1999. release 1.03d.

[Ung84] D. M. Ungar. A Non-disruptive High Performance Storage Reclamation Algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, 1984.

References

[WM95] Wm. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM Computer Architecture News*, 23(1):20–24, 1995.