Object Lifetime Prediction in Java

Hajime Inoue Darko Stefanović Stephanie Forrest
Department of Computer Science
University of New Mexico
Albuquerque, NM 87106
{hinoue, darko, forrest}@cs.unm.edu

Technical Report TR-CS-2003-28
May 2003

Abstract

Accurately predicting the lifetimes of objects in object-oriented and functional languages is important because such predictions can be used to improve memory management performance at run-time. A typical approach to this prediction problem is first to observe object lifetimes by tracing a sample program execution, then to construct a predictor function based on these observations, and finally to test the predictions on reference program executions. Four quantitative measures characterize this approach: *coverage*, the proportion of objects in the reference program execution for which the predictor function is defined; *accuracy*, the fraction of predicted lifetimes that are correct; *precision*, the granularity of the predictions; and *size*, the size of the predictor itself. These four properties are not independent; for example, increased precision often leads to less coverage and accuracy.

We describe a fully precise prediction method and report experimental results on its performance. By "fully precise" we mean that the granularity of predictions is equal to the smallest unit of allocation. We show that for a number of benchmark programs in the Java programming language, fully precise prediction can be achieved, together with high coverage and accuracy. Our results also show that a significant proportion of objects have a measured lifetime of zero, a fact which a dynamic compiler could use to avoid explicit allocation. The method described here is the first to combine high-precision and efficiency in a single lifetime predictor.

1 Introduction

Garbage-collected languages, such as C# and Java are increasingly important. Garbage collection (GC) improves developers' productivity by removing the need for explicit memory reclamation, thereby eliminating a significant source of programming errors. However, garbage-collected languages incur increased overhead, and consequently, improvement in their performance is essential to continuing adoption of these languages. Many algorithms have been proposed over the several decades since GC became available, but their performance has been heavily application-dependent. For example, Fitzgerald and Tarditi showed that a garbage collector must be "tuned" to fit a program (Fitzgerald and Tarditi, 2000). Such application sensitivity limits the usefulness of garbage-collection methods. Another approach relies on larger heap sizes and simply runs the collection algorithms less frequently. However, this does not always result in better performance (Brecht et al., 2001). GC algorithms typically make certain assumptions about the lifetimes of the application's objects, and tailor the collection algorithm to these assumptions. If the assumptions are not borne out, poor performance is the outcome. What is needed is the ability to make good predictions about object lifetimes and to incorporate these predictions into a general GC method which works on a wide range of applications.

The overhead of GC, compared to explicit deallocation, arises from the cost of identifying which objects are still active (*live*) and which are no longer needed (*dead*). GC algorithms, therefore, go to some lengths to collect regions of memory that are mostly dead. The "ideal" garbage collector would collect regions where all the objects died recently—so that heap space isn't wasted by dead objects and living objects are not processed unnecessarily. To do this, the allocator would need to know the exact death time of an object at the time it was allocated, and then it could allocate it to a region occupied by objects with the same death time. To date, this has been accomplished only in a limited way by a process called "pre-tenuring". Pre-tenuring algorithms make coarse predictions of the lifetimes of objects, predicting which allocations will result in long-lived objects and then allocating them to regions that are not frequently collected. For example, in Blackburn's pre-tenuring scheme (Blackburn et al., 2001), objects are allocated into short-lived, long-lived, and eternal regions.

Modern language runtimes provide a wealth of profiling information, which we believe can be used to improve object lifetime prediction. In virtual machine (VM) environments, such as C# and Java, profiling is an important part of the just-in-time (JIT) compilation process; here we show how information available to the VM can be leveraged to improve object lifetime prediction.

In this paper we demonstrate, using profiling information, that there is a significant correlation between the state of the stack at an allocation point and the allocated object's lifetime. Next, we describe how this information can be used to predict object lifetimes at the time they are allocated. Using this technique, we then show that a significant proportion of objects have zero lifetime. We discuss potential applications of this technique, both of memory management and dynamic compilation systems. Finally, we speculate on how these results can be used for anomaly intrusion detection for computer security.

2 Object Lifetime Prediction

As stated above, one goal of object lifetime prediction is to aid in optimization, by providing run-time advice to the memory allocation subsystem about the likely lifetime of an object at the time it is allocated. To accomplish this, we construct a *predictor*, which bases its object lifetime predictions on information available at allocation time. This includes the context of the allocation request, namely the dynamic sequence of method calls that led to the request, and the actual type of the object being allocated. This information is referred to as an *allocation site*; if the observed lifetimes of all objects allocated at a site are the same, then the predictor should predict that value at run-time for all objects allocated at the site. We do not yet have predictors built into the memory allocation subsystem, so our testing is trace-driven and not performed at run-time.

We consider two circumstances for prediction—self prediction and true prediction. *Self prediction* uses the same program trace for training (predictor construction) and for testing (predictor use). Self-prediction allows us to measure how useful the information in an allocation site is with respect to object lifetimes. *True prediction* uses a different (and smaller) training trace for predictor construction, and a larger one from the same program (but with different inputs) for testing. If self-prediction performance is poor, true prediction will also be poor. Yet if self prediction works well, it is still possible that true prediction will be poor. This largely depends on how different inputs affect the behavior of the program. If a program is completely data-driven, it is possible that true prediction will not perform well, regardless of the success of self prediction.

We evaluate predictor performance using four quantities: precision, coverage, accuracy, and size.

- *Precision* is the granularity of the prediction in bytes.[1] A predictor with exact granularity predicts with a precision of one byte; e.g., it may predict that a certain allocation site always yields objects with a lifetime of 10304 bytes. A less precise predictor might predict a range such as 10000–10999, or, more commonly, a range from a set of geometrically proportioned bins, such as 8192–16383. Our aim is to achieve high precision (narrow ranges); in practice, the desired precision will depend on the manner in which the memory allocation subsystem exploits the predictions.

- *Coverage* is the percentage of predicted objects for which the system has a prediction. In other words, we are prepared to construct a predictor that, at run-time, does not always make a prediction. For certain allocation sites (presumably those which we cannot predict with confidence) we make no prediction, rather than making one that is wrong too often. Clearly, the memory allocation subsystem will need to have a fallback allocation strategy for these cases. Note that, although the decision whether to predict is made per allocation site, the natural measure of coverage is not the percentage of sites at which we predict (a static count) but the percentage of object allocation events predicted (a dynamic count). We desire the coverage to be as high as possible.

- *Accuracy* is the percentage of predicted objects for which a predictor produces a correct lifetime. Among all objects allocated at run-time for which a prediction is made, some will have a true lifetime that falls in the same range as the predicted lifetime; the range being defined by the precision parameter. We desire the accuracy to be as high as possible.

- *Size* is the number of entries the predictor contains; each entry is a pair consisting of a descriptor of an allocation site and a prediction for the lifetimes for that site. Since the predictor will incur space and time overhead at run-time, we desire the size to be

[1]The load on the memory management subsystem is determined by the heap allocation and death events, and is independent of other computational effects of the program. Therefore, the lifetime of an object in garbage collection studies is defined as the sum of the sizes of other objects allocated between the given object's allocation and death, and is expressed in bytes or words.

small.

There are tradeoffs between the characteristics of precision, coverage, accuracy, and size. A highly precise predictor may cover less or be less accurate. Its size may also be greater. Alternatively, a predictor with a lower precision may be able to cover a larger proportion of objects with greater accuracy despite smaller size. Intuitively, decreasing the precision creates a larger, easier to hit target (coverage and accuracy). Similarly, a smaller target (a precise predictor), is harder to hit, and it seems reasonable that a predictor would need more information, resulting in a larger size.

The predictors are constructed as follows. For each benchmark, we collect a trace. This trace includes, among other things, accurate records for object allocation and object death events. At each allocation event, we record the object identifier, its type, and the execution context. The execution context consists of the identifiers of the methods on the stack. This is referred to as the stack string; later in the construction of predictors, we reduce the amount of information by using only a prefix of the stack string and explore the effects of varying the length of the prefix. We record each death accurately, to a precision of 1 byte. This precision of granularity is unusual in object lifetime traces. Object lifetimes traces reported in the literature usually have coarse granularity in garbage-collected languages because objects are known to be dead only at the point of collection, and collections are relatively infrequent events. We are able fully precisely to determine the lifetimes of our objects by using an implementation of the Merlin trace algorithm (Hertz et al., 2002) within the JikesRVM open-source Java virtual machine (Alpern et al., 2000).

The trace is used to construct a predictor for the corresponding program. For each allocation for which that all objects allocated at the site have the same lifetime up to the desired precision, we include in the predictor an entry that predicts that objects allocated at the site will have that lifetime. Note that if any two objects allocated at an allocation site have different lifetimes, or *collide*, the predictor refuses to make a prediction for that allocation site.

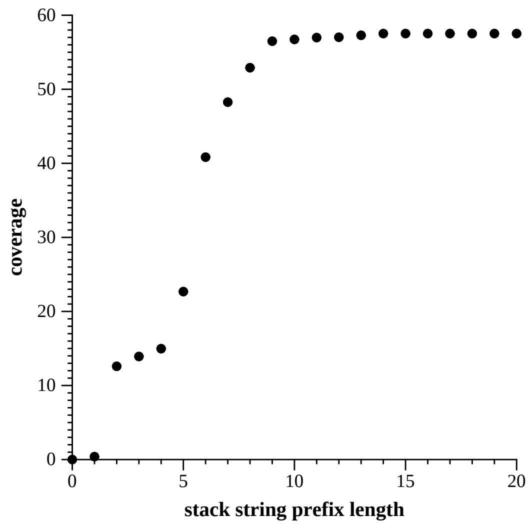
This type of predictor is computationally efficient and tunable. For example, the number of entries in these predictors is often large, and mostly populated by *singletons*. These are entries in predictor that saw only one object allocated, so no collisions could knock them out. These entries might be removed to form a smaller predictor without greatly reducing coverage. Beyond this, it is possible to examine each entry in the predictor to best trade off coverage and size.

We look at three aspects of lifetime prediction:

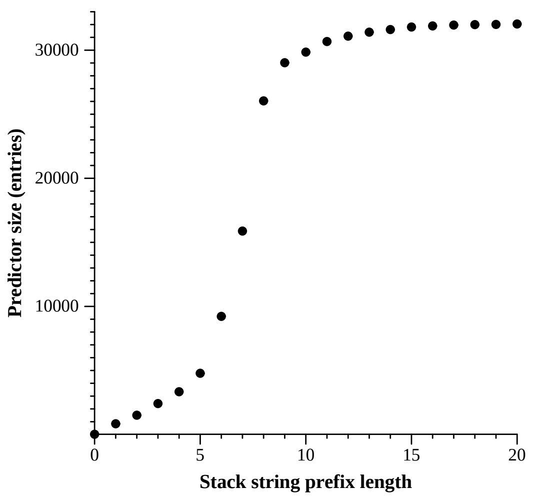
- **Exact Lifetime Prediction:** The predictor attempts to predict the lifetime of an object to the exact byte.
- **Granulated Lifetime Prediction:** The predictor has lower precision: it bins lifetimes using the scheme $bin = \log_2(lifetime + 1)$. Note that since most objects die young, the granularity is still very fine.
- **Zero Lifetime Prediction:** We discovered that some benchmarks generate a large number of objects that die before the next object allocation. Predicting zero lifetime objects is an interesting subproblem of exact prediction.

We illustrate these concepts on the example of the benchmark *pseudojbb*.[2] In Figure 1 we show the dependence of achieved predictor coverage and of required predictor size, as the stack string prefix (SSP) length is varied from 0 to 20; we used exact lifetime prediction, and we kept the singletons in the predictor. In Figure 1(a), we have plotted the SSP length along the horizontal axis. For each SSP length, we examined the trace and synthesized a predictor. The coverage of the predictor, i.e., the percentage of the

[2]See Section 3 for a description of the benchmarks we use.

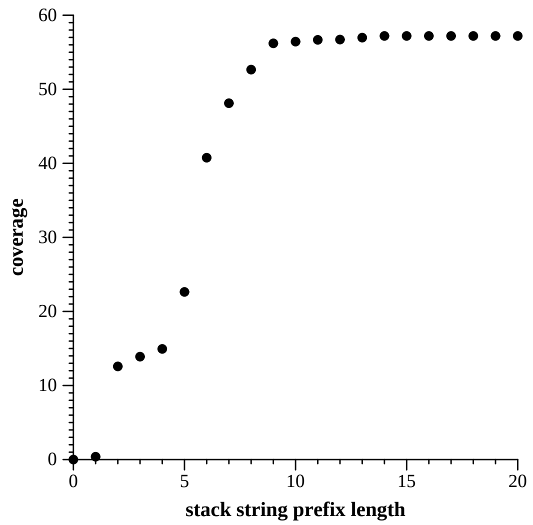


(a) Predictor coverage

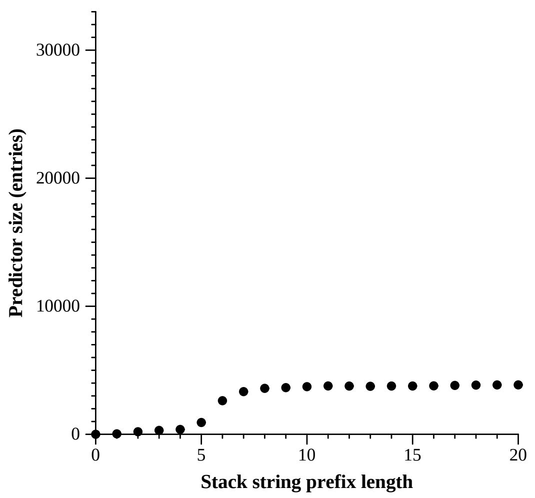


(b) Predictor size

Figure 1: Benchmark *pseudojbb*: prediction including singletons.

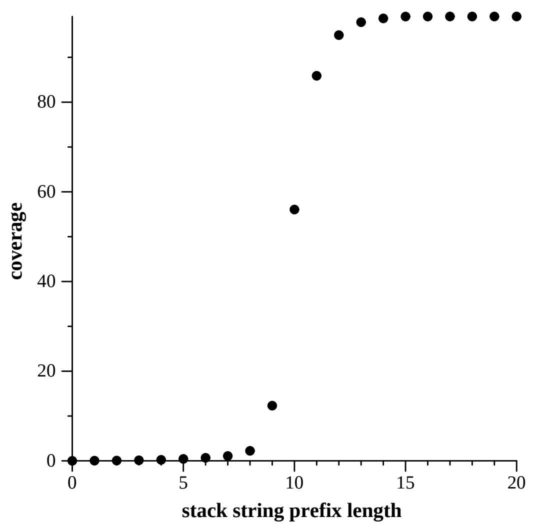


(a) Predictor coverage

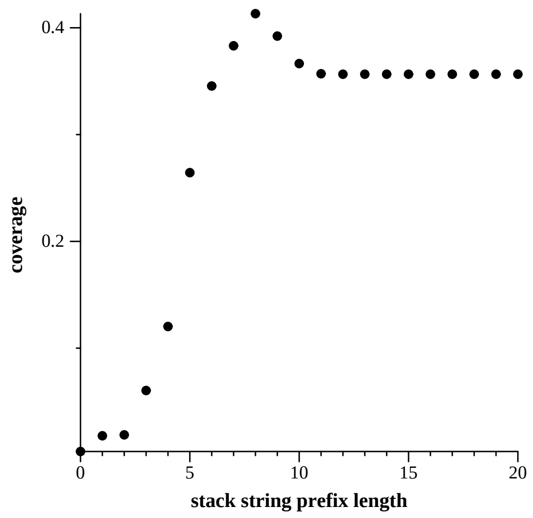


(b) Predictor size

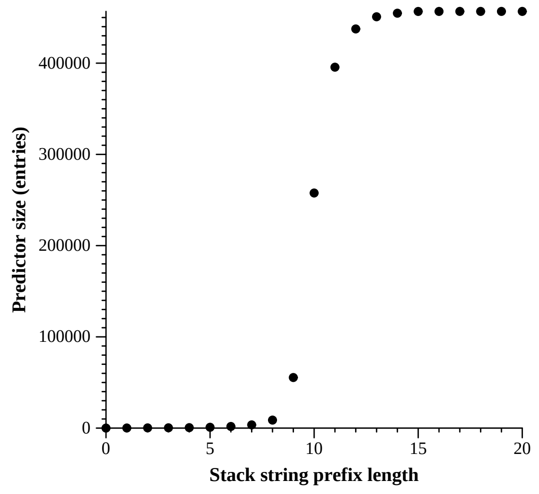
Figure 2: Benchmark *pseudojbb*: prediction excluding singletons.



(a) Predictor coverage

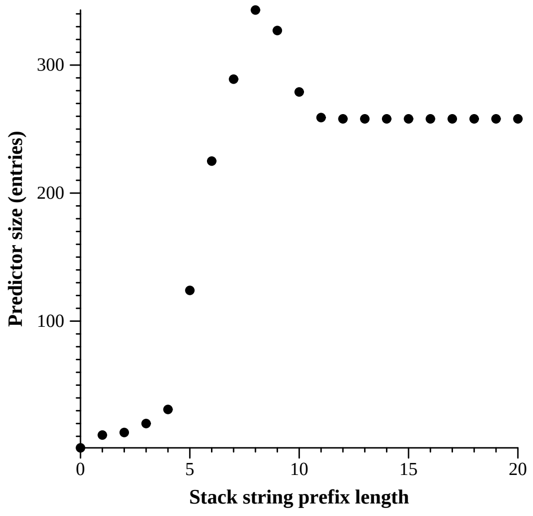


(a) Predictor coverage



(b) Predictor size

Figure 3: Benchmark *perimeter*: prediction including singletons.



(b) Predictor size

Figure 4: Benchmark *perimeter*: prediction excluding singletons.

dynamic count of objects allocated for which the predictor makes a prediction, is plotted along the vertical axis. The predictor coverage improves with increasing SSP length, as more information is available to disambiguate allocation sites. However, a plateau is already reached for an SSP of length about 10, indicating that that is a sufficient SSP length. In Figure 1(b), we show the growth of predictor size, i.e., the number of entries, with increasing SSP length.

In Figure 2 we show the plots for the same metrics for a predictor that excludes singletons. Note that the predictor coverage is almost unchanged, but the size requirement for the predictor is drastically reduced.

In general, when singletons are excluded, predictor size and coverage are closely linked by SSP length. There is a tradeoff between collisions and singletons—if the SSP is too short, too many objects will collide; too long and the SSP will convert entries into singletons. We can see this on the example of the *perimeter* benchmark, in Figure 4(a), which has a maximum at around SSP length 8, and then decays to a plateau. This effect is more dramatic in Figure 4(b). A sharp peak defines the area between the regime of collisions and singletons.[3]

Our results are reported in terms of object counts, rather than bytes. Bytes are often used in the garbage collector literature, and we collected these data as well. The results are similar. In other words, object size is not signifcantly correlated with the predictability of object lifetime.

3 Benchmarks

We use two sets of benchmarks: the Olden benchmarks (Carlisle and Rogers, 1995; Rogers et al., 1995), ported to Java (Cahoon and McKinley, 1999), and the SPECjvm98 and SPECjbb2000 benchmarks (SPEC, 1999). The Olden benchmarks emphasize pointer manipulation but do so with algorithms consistent with useful computation. They consist of traditional problems such as traveling salesman and Voronoi triangle decomposition, as well as simulations such as the Colombian health care bureaucracy. The SPEC benchmarks are much larger and more realistic than the Olden set. They consist of useful "real world" programs (with the exception of *db*) and are intended to be representative of real applications run on Java virtual machines. Table 1 describes the individual benchmarks and Table 2 gives some general run-time characteristics of the benchmarks.

4 Self-Prediction

We now consider the results of self-prediction experiments for exact granularity as well as for logarithmic granularity, and consider the effects of including or excluding singletons from the predictors.

4.1 Exact Granularity

4.1.1 Predictors including singletons

On the Olden benchmarks the exact granularity predictor that includes singletons achieves high coverage (about 90%) on *bh*, *perimeter*, *power*, *bisort*, and *tsp*. We originally used traces that recorded only a 16-long prefix of the stack string, but we noticed that for *bisort*, *treeadd*, and *voronoi*, a longer SSP length was required, because as the SSP reaches 16, the size and coverage of the predictors are still increasing. Barret and Zorn (Barrett and Zorn, 1993)found that a short string, on the order of length four, was acceptable. It is interesting that a much longer string is required for our benchmarks, even for those cases that can be predicted well. The increase may be due to the greater distance between an object's allocation and its use in object-oriented languages. *Em3d* and *mst* show some level of predictability (16 and 28%, respectively), but for one benchmark, *health*, it is not possible to predict well even though the size

[3]Figure 3 is included for completeness.

Benchmark	Description
bh	N-body problem solved using hierarchical methods
bisort	Bitonic sort both forwards and backwards
em3d	Simulates electromagnetic waves propagation in 3d
health	Simulates the Colombian health care system
mst	Computes minimum spanning tree of a graph
perimeter	Computes perimeter of quad-tree encoded image
power	Solves the Power-System-Optimization problem
treeadd	Sums a tree by recursively walking it
tsp	The traveling salesman problem
voronoi	Voronoi triangle decomposition
compress	Uses Lempel-Ziv to compress and decompress some strings
jess	Expert shell based on NASA's CLIPS system
db	Emulates database operations on a memory resident database
javac	The Java compiler from jdk 1.0.2
mpegaudio	Decodes mpeg layer3 (mp3) files
mtrt	Multi-threaded raytracer draws a scene with a dinosaur
jack	Java parser generator is a lex and yacc equivalent
pseudojbb	Java Business Benchmark altered for GC research

Table 1: The benchmarks and a short description.

and coverage data show that we have a sufficient SSP length.

All of the SPEC benchmarks show some level of predictability. The predictor does particularly well on the synthetic benchmark *db*. It achieves greater than 50% coverage on *compress*, *mpegaudio*, *mtrt*, *pseudojbb*, and *jack*, and more than 20% on the remaining two, *jess* and *javac*. The greater realism of the SPEC programs suggests that exact prediction will have some level of success on most programs. Unfortunately, the predictor achieved greater than 90% coverage only on the synthetic benchmark. Perhaps the type of performance seen in *perimeter*, *power*, and *db* would not be typical of "real" programs.

4.1.2 Predictors excluding singletons

If the predictor is constructed to exclude singleton entries, coverage drops. On the Olden benchmarks the predictor works well only for *bh* and *power*, with *tsp* falling by about 30% to 60% coverage and *perimeter* to less than 1%. This reveals that a large proportion of the capability of our predictor is due to singletons. For some benchmarks, the predictor can be vastly reduced in size without a corresponding loss of coverage. The size of the predictor for *power* drops by 85% compared to a loss of less than 1% coverage, and *bh* drops 94% in size with a corresponding loss of less than a tenth percent coverage.

For *bisort*, *treeadd*, and *voronoi*, plots of SSP length versus predictor size excluding singletons are similarly shaped to those including them, but have smaller maxima. Coverage behaves similarly. For *em3d*, *health*, and *mst*, predictor size shows a maximum between SSPs of length 7 and 10 before falling off slightly, with coverage following a similar distribution as with singletons, but reduced. Benchmarks *perimeter* and *tsp* show what are best described as phase transitions in predictor size, and *voronoi* shows a phase transition in coverage. Interestingly, the maximum coverage does not correspond to maximum predictor size. In each, the maximum coverage is achieved with an SSP shorter than the SSP that corresponds to the maximum predictor size. This is useful, since it means that smaller predictors sometimes cover more than larger predictors.

In general, coverage with predictors excluding singletons is much better on the SPEC benchmarks than on the Olden benchmarks. As

Testing and self-prediction				
Benchmark	Command line	Objects allocated	Bytes allocated	Static sites
bh	-b 2048 -m	6728211	162842400	530
bisort	-s 250000 -m	137896	3383612	418
em3d	-n 2000 -d 100 -m	23863	7582636	439
health	-l 5 -t 500 -s 1 -m	1205975	22990756	457
mst	-v 1024 -m	10578	7074760	437
perimeter	-l 16 -m	462220	15388636	441
power	-m	792772	22972476	475
treeadd	-l 20 -m	1055004	21710584	412
tsp	-c 10000 -m	57837	2667280	452
voronoi	-n 20000 -m	1442667	36670008	488
compress	-s100	24206	111807704	707
jess	-s100	5971861	203732580	1162
db	-s100	3234616	79433536	719
mpegaudio	-s100	39763	3579980	1867
mtrt	-s100	6538354	147199132	897
javac	-s100	7287024	228438896	1816
raytrace	-s100	6399963	142288572	992
jack	-s100	7150752	288865804	1184
pseudojbb	70000 transactions	8729665	259005968	1634

Training				
Benchmark	Command line	Objects allocated	Bytes allocated	Static sites
bh	-b 512 -m	1132767	28214644	530
jess	-s1	125955	6978524	1143
javac	-s1	20457	2641064	1188
mtrt	-s1	328758	11300528	989
jack	-s1	512401	21911316	1183
pseudojbb	10000 transactions	2778857	103683020	1634

Table 2: Trace statistics. For each trace, the number of objects allocated and the total size of all allocated objects are given. Also given is the number of allocation sites; each site is counted only once, even if executed more than once, but sites that are not executed in these particular runs are not counted. The top part of the table lists the traces used for the self-prediction study (Section 4). The bottom part of the table lists the training traces used in the true prediction study (Section 5); for testing the true prediction we reuse the traces from the top part.

Benchmark	Exact					Logarithmic				
	incl. singletons		excluding singletons			incl. singletons		excluding singletons		
	size	coverage	size	coverage	SSP	size	coverage	size	coverage	SSP
bh	4626	92.09	65	91.99	4	5139	92.67	955	92.61	11
bisort	133367	97.11	55	0.43	9	133800	99.73	518	3.09	10
em3d	2864	16.00	137	4.57	8	3298	64.53	601	53.35	10
health	3225	0.43	138	0.17	7	3701	0.96	637	0.71	9
mst	2748	27.55	104	6.54	8	3180	56.12	565	35.57	10
perimeter	456707	99.11	343	0.41	8	457159	99.90	76108	11.55	9
power	4038	96.47	988	96.08	29	5036	99.88	11986	99.49	29
treeadd	1050673	99.63	50	0.04	9	1051102	99.97	509	0.39	10
tsp	51943	92.67	281	59.65	7	52384	98.94	1133	65.71	9
voronoi	318676	22.28	9099	1.72	14	329215	30.46	24336	11.06	14
compress	9133	67.78	792	33.33	10	9692	85.66	1381	84.84	10
jess	24999	23.40	3326	23.03	24	25873	34.87	4197	34.51	26
db	8847	90.36	73	89.91	3	9474	90.56	348	90.09	4
raytrace	14761	41.61	325	41.27	4	15509	42.36	514	41.97	4
javac	109357	28.63	75571	28.17	32	144844	45.72	111058	45.25	32
mpegaudio	17550	78.42	1931	39.67	8	18260	89.68	2704	51.12	9
mtrt	12490	50.11	1566	49.79	3	13167	50.92	306	50.53	3
jack	29542	61.25	14126	61.04	20	31659	66.30	14600	65.90	17
pseudojbb	32044	57.52	3861	57.20	14	33158	63.65	4861	63.23	14

Table 3: Self-prediction behavior of the benchmarks. For the columns including singletons, the size refers to the number of entries in the predictor using an SSP length of 20, except where the SSP of the predictors excluding singletons is greater than 20. In these cases, the SSP length used for predictors including singletons is the same as that for the predictor excluding them.

in the full predictor case, all the benchmarks show some level of self-prediction, with *jess* having the smallest level of coverage at 23%. In fact, *jess*, *db*, *javac*, and *mtrt* show little drop-off in coverage, even though the predictor size decreases by an average of 64%. The remaining programs show a sizable drop-off in coverage, but none shows the negligible coverage that some of the Olden predictors excluding singletons showed. Again, this may be due to the larger size and greater realism of the SPEC benchmarks.

For no benchmarks was type alone enough (an SSP of length zero). Benchmarks *jess* and *mtrt* needed at least the method containing the allocation site (an SSP of length one) to have significant coverage, and the rest needed more.[4]

In summary, the exact predictors were able to cover significant fractions of objects in both sets of benchmarks. With singletons excluded, the predictors still have significant coverage for three of the Olden benchmarks and all of SPEC. Although self-prediction is not really "predicting," it does indicate the degree of correlation the stack has with the lifetime of the object allocated at that point, which is evidence that true prediction is possible.

4.2 Logarithmic Granularity

The behavior of the logarithmic predictors was better than the exact predictors'. Looking at predictors with logarithmic granularity, coverage remained high for *perimeter*, *power*, and *tsp* among the Olden benchmarks, but did not improve significantly. Benchmarks *em3d* and *mst*

[4]Is type required at all? Type is probably required in our predictors because type also disambiguates allocations made in the same method. Note that we are using a flow-insensitive notion of the stack, recording only the method and not the bytecode within the method that makes the next call.

showed large coverage increases with the predictor covering over 50% of objects. The predictor still had no ability to cover *health*. Coverage improved for all SPEC benchmarks, with coverage ranging from about 35% for *jess* to about 90% for *db* and *mpegaudio*.

Coverage excluding singletons decreased less than in the exact case, though the behavior was qualitatively similar. There was some ability to cover in all benchmarks except for *bisort*, *health* and *treeadd*. Compared to the exact case, most of the gains came in benchmarks where even if the factor of improvement was large, the absolute coverage was still modest, as in *bisort*, *health*, and *perimeter*. However, substantial increases were seen in *em3d* and *mst*.

The behavior of the granulated predictors is qualitatively similar to the exact predictors. As expected, their coverage is significantly better. Granulated prediction is much more likely to be useful than exact prediction since tracing can, in principle, be done much faster than in the exact case. Because of our logarithmic bin size, however, the predictors are still very precise for the large number of short-lived objects.

5 True Prediction

Barrett and Zorn found that accuracy in true prediction is high in benchmarks with high coverage in self-prediction that are not data-driven. We tested true prediction against a subset of the benchmarks to see if this correlation held true even with the extreme change of precision. We used *bh*, *jess*, *javac*, *mtrt*, *jack*, and *pseudojbb*. We used SSP lengths as in Table 3 and included singletons. This is not meant as an exhaustive study, but a demonstration that true prediction performs well, with similar results to the Barrett and Zorn study.

Results for the six examples are shown in Table 4. For both exact and logarithmic granularity, all the predictors are highly accurate. For three of the benchmarks, the high accuracy comes at the price of coverage. Coverage is insignificant for *jess*, *javac*, and *mtrt*. The other benchmark predictors show considerable coverage. The difference in coverage is probably due to the degree the program is data-driven. For example, the training run of *jess* is quite different from its text run. In *pseudojbb*, the only difference is in the length of the run. Although these are not exhaustive, these examples are evidence that highly precise, true prediction is possible for some applications.

6 Zero-Lifetime Objects

Table 5 shows the fraction of zero-lifetime objects generated by each benchmark and the fraction self-predicted. Many of the benchmarks allocate large numbers of zero-lifetime objects. A zero-lifetime object is one that is allocated and then dies before the next object is allocated. Of the Olden benchmarks, *bh*, *health*, *power*, *tsp*, and *voronoi* allocate large numbers of zero-lifetime objects. All the SPEC benchmarks allocate a significant percentage of them, with *jack* allocating the least at 13%. This peculiar result could only be obtained through the use of exact traces!

On Olden, the predictor does well at predicting zero-lifetime objects with the exception of *health*. This is because *health* is really a micro-benchmark of linked-list traversals (Zilles, 2001), with the lifetime of objects being data-driven by random insertions and removals from the list. On the SPEC benchmarks, coverage (% predicted of possible) of zero-lifetime objects is greater than 50% on all but *jack*. On the whole, the predictor is able to cover a large fraction of zero-lifetime objects.

7 Prediction and Object Types

We undertook a simple classification of allocated objects according to their type. We divide them into application types, library types, and virtual machine types (since the virtual machine we use is written in Java itself). Library types

Benchmark	Exact		Logarithmic	
	Coverage	Accuracy	Coverage	Accuracy
bh	47.31	99.99	48.45	98.35
jess	0.05	99.54	0.11	100.00
javac	0.04	99.53	0.09	100.00
mtrt	0.04	99.33	0.10	99.97
jack	19.20	99.89	20.33	99.74
pseudojbb	56.87	99.99	63.00	99.85

Table 4: True prediction.

Benchmark	% of all objects	% predicted	% predicted of possible
bh	46.01	45.14	98.11
bisort	0.33	0.32	96.32
em3d	3.30	3.23	97.84
health	51.99	0.15	0.28
mst	13.08	4.84	37.02
perimeter	0.26	0.26	97.82
power	24.14	24.12	99.91
treeadd	0.03	0.03	96.95
tsp	58.67	58.54	99.78
voronoi	50.13	50.12	95.94
compress	21.72	20.86	96.03
jess	39.63	19.96	50.36
db	45.06	45.01	99.97
mpegaudio	25.98	25.29	97.34
mtrt	40.01	33.37	83.39
javac	12.95	10.48	80.93
raytrace	41.30	29.57	71.60
jack	43.44	0.22	0.49
pseudojbb	20.82	18.75	90.04

Table 5: Frequency of zero-lifetime objects, the percentage predicted, and success of their prediction using exact granularity with the SSP lengths used for prediction in Table 3.

are those classes belonging to the `java` hierarchy. VM classes are easily identified by their `VM` prefix. Application classes are all others.

As Table 6 shows, global coverage (greater than 90%) is usually associated with high coverage of application types. This makes sense, since for most benchmarks, application types dominate. The exceptions, *tsp* and *db*, allocate a lot of library types, which also have high coverage. A predictor's ability to cover is a result of predicting types resulting from application behavior, rather than the underlying mechanisms of the compiler or VM.

8 Related Work

Using simulation, Hayes (Hayes, 1991; Hayes, 1993) examined which objects were "entry" objects into clusters of objects that die when the "keyed" object dies. For choosing *automatically* what objects are "keyed", he suggested random selection, monitoring the stack for when pointers are popped, creating key objects, and doing processing during promotion in generational garbage collection. Cohn and Singh (Cohn and Singh, 1997) revisited the results of Barrett and Zorn, using decision trees based on the top *n* words of the stack to classify short-lived and long-lived objects. They did better than Barret and Zorn; it is not clear, however, that the system described could be implemented other than off-line. Seidl and Zorn (Seidl and Zorn, 1997; Seidl and Zorn, 1998) sought to predict objects according to the categories *highly referenced*, *short-lived*, *low referenced*, and *other*. Their prediction scheme was based on the stack, but not ordered. Their goal was to improve paging rather than cache performance. They emphasized that during profiling, it was important to choose the right depth of the stack predictor: too shallow is not predictive enough and too deep results in overspecialization. The results were mixed, with larger programs giving better results. Cheng et al. (Cheng et al., 1998) describe pretenuring using a simple algorithm that looks at allocation sites by profiling beforehand and using generational stack collection—that is, caching portions of the root set so that not the entire stack needs to be scanned each time. Dieckmann and Hölzle (Dieckman and Hölzle, 1998; Dieckmann and Hölzle, 2001) studied the allocation behavior of the SPECjvm98 benchmarks by creating a heap simulator. They found that more than 50% of the heap was used by non-references (primitives), and that alignment and extra header words expanded heap size significantly, since objects tended to be small. They confirm the weak generational hypothesis for Java, though not as firmly as in other languages (up to 21% of all objects were still alive after 1 MB of allocation). Cannarozzi et al. (Cannarozzi et al., 2000) use a single-threaded program model and keep track of the lowest stack frame that referenced an object or other objects. In many ways this is similar to our use of Merlin, since they use roots to keep track of when objects are "touched". Fitzgerald and Tarditi (Fitzgerald and Tarditi, 2000) demonstrated that memory allocation behavior differs dramatically over a variety of Java benchmarks. They pointed out that performance would improve by at least 15% if they had chosen the appropriate collector. The biggest choice is whether to use a generational collector (and pay the penalty for write barriers). Harris (Harris, 2000) attempted pretenuring using only the current method signature and bytecode offset. He studied call-chains, but decided they provided little information unless recursion is removed. He found that detecting phase changes (from long-lived to short-lived objects at allocation sites) helped in some benchmarks. He speculated that using the class hierarchy might be an easier and less expensive way to predict lifetimes. The approach was implemented in Sun's Research VM. Blackburn (Blackburn et al., 2001) applied coarse-grain prediction in Java to construct pretenuring advice for garbage collectors and found they were able to reduce garbage collection times for

Benchmark	VM		Library		Application		% Predicted
	% Alloc.	% Pred.	% Alloc.	% Pred.	% Alloc.	% Total Pred.	
bh	0.08	51.26	0.02	53.56	99.90	92.14	92.09
bisort	2.27	30.05	0.45	38.59	97.28	98.94	97.11
em3d	15.00	36.68	3.01	43.53	81.98	11.20	16.00
health	0.31	38.66	0.07	48.07	99.62	0.28	0.43
mst	25.85	34.02	38.22	6.39	34.93	45.72	27.55
perimeter	0.90	28.33	0.20	51.76	98.90	99.72	99.11
power	0.54	44.94	0.11	48.94	99.34	96.80	99.47
treeadd	0.28	27.16	0.05	37.87	99.67	99.87	99.64
tsp	6.96	41.19	58.07	98.67	34.97	92.96	92.67
voronoi	0.34	49.62	0.07	51.45	99.58	22.16	22.28
compress	37.97	63.63	11.89	53.41	50.14	74.33	67.78
jess	0.57	78.90	19.69	99.01	79.74	5.25	23.40
db	0.29	61.47	94.83	94.78	4.88	94.78	90.36
mpegaudio	42.48	70.51	10.16	66.57	47.37	88.05	78.42
mtrt	0.19	68.40	1.94	67.01	97.87	49.74	50.11
javac	15.91	5.86	26.54	32.21	57.54	33.28	28.63
raytrace	0.22	67.97	1.03	66.68	98.76	41.29	41.61
jack	3.22	96.94	48.26	42.99	48.52	77.05	61.25
pseudojbb	0.53	73.35	33.19	33.81	66.27	88.43	57.52

Table 6: Prediction for three categories of objects according to object type. For each of the three categories of types (virtual machine, library, application), the percentage of total allocated objects that fall in the category is given, together with the percentage of objects in the category that are predicted. Rightmost column is the overall percentage of objects predicted.

several types of garbage collection algorithms. Hirzel (Hirzel et al., 2002) looked at connectivity in the heap to discover correlations among object lifetimes. They found that objects accessible from the stack have short lifetimes, objects accessible from globals are immortal or very long-lived, and objects connected via pointers usually die at about the same time, as might be expected.

9 Discussion and Conclusions

Most GC algorithms are effective when their assumptions about lifetimes match the actual behavior of the applications—beyond crude predictions like pretenuring, they do little to "tune" themselves to their applications. The "ideal" garbage collector would know the lifetime of every object at their birth. We move toward this goal by showing that for some applications, we can predict the lifetime of many objects to the byte.

It is remarkable that *exact* prediction works at all. Previous attempts at prediction used a much larger granularity, in the thousands of bytes. In particular, Barrett and Zorn used a two-class predictor with a division at the age of 32KB. It is not surprising that the predictor they described worked well, given that 75% of all objects lived to less than that age. Cohn and Singh's decision trees (Cohn and Singh, 1997) worked very well at the cost of much greater computational complexity. Blackburn's pretenuring scheme (Blackburn et al., 2001), used a coarse granularity. The method described here is the first to attempt high-precision and efficient lifetime prediction together.

Our results show that a significant percentage of all objects live for zero bytes, a result that required the use of exact traces. Because our predictors are able to cover zero-lifetime allocation sites, the zero-lifetime results have clear applications in code optimization. Zero-lifetime object prediction could be used to guide stack escape analysis so that objects are allocated on the stack instead of on the heap.

Object lifetime prediction could also be used as a hinting system, both for *where* an allocator should place an object and *when* the garbage collector should try to collect it. This would be a more general procedure than pretenuring, and it would support more sophisticated garbage collection algorithms, such as multiple-generation collectors and the Beltway collector (Blackburn et al., 2002).

Beyond optimization, object-lifetime prediction could have applications to computer security. Anomaly intrusion-detection systems use a variety of techniques to model normal (secure) behavior of a system empirically]. Anomaly-detection systems have previously used many different observables to define a model of normal behavior, including patterns of system calls (Forrest et al., 1996), network traffic (Heberlein et al., 1990; Mukherjee et al., 1994), and method invocations in a JVM (Inoue and Forrest, 2002). Few, if any, of these systems have analyzed the behavior of a memory management system. Accurate object prediction, together with a carefully chosen precision, might allow us to construct a new type of anomaly intrusion-detection system, which responds to anomalous objects, rather than anomalous code. We are currently investigating this possibility.

More generally, we are interested in building compact models of program behavior, independent of any direct application. Our earlier work in intrusion detection has convinced us that programs behave much more regularly at execution time than is commonly believed, and that it is possible to build reasonably accurate and very compact models of program behavior. Such models might be useful for predicting the aggregate behavior of large collections of interacting programs. Our results on object-lifetime prediction provide a new substrate from which to construct such models.

10 Acknowledgments

The authors gratefully acknowledge the support of the National Science Foundation (grants ANIR-9986555, CCR-0219587, CCR-0085792, and EIA-0218262), the Office of Naval Research (grant N00014-99-1-0417), Defense Advanced Projects Agency (grant AGR F30602-00-2-0584), Sandia National Laboratories, Microsoft Research, Intel Corporation, Hewlett-Packard, and the Santa Fe Institute. We are grateful to Matthew Hertz for creating and providing to us the Merlin algorithm implementation. We thank Kathryn McKinley, Amer Diwan, and Dennis Chao for valuable comments on the text of the paper.

References

Alpern, B., Attanasio, D., Barton, J., Burke, M., Cheng, P., Choi, J., Cocchi, A., Fink, S., Grove, D., Hind, M., Hummel, S., Lieber, D., Litvinov, V., Ngo, T., Mergen, M., Sarkar, V., Serrano, M., Shepherd, J., Smith, S., Sreedhar, V., Srinivasan, H., and Whaley, J. (2000). The Jalapeño virtual machine. *IBM Systems Journal*, 39(1).

Barrett, D. A. and Zorn, B. G. (1993). Using lifetime predictors to improve memory allocation performance. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 187–196.

Blackburn, S. M., Jones, R., McKinley, K. S., and Moss, J. E. B. (2002). Beltway: Getting around garbage collection gridlock. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ACM SIGPLAN Notices. ACM Press.

Blackburn, S. M., Singhai, S., Hertz, M., McKinley, K. S., and Moss, J. E. B. (2001). Pretenuring for Java. In *Proceedings of SIGPLAN 2001 Conference on Object-Oriented Programming,* *Languages, & Applications*, volume 36(10) of *ACM SIGPLAN Notices*, pages 342–352, Tampa, FL. ACM Press.

Brecht, T., Arjomandi, E., Li, C., and Pham, H. (2001). Controlling garbage collection and heap growth to reduce the execution time of java applications. In *OOPSLA*, pages 353–366.

Cahoon, B. and McKinley, K. (1999). Tolerating latency by prefetching Java objects. In *Workshop on Hardware Support for Objects and Microarchitectures for Java*.

Cannarozzi, D. J., Plezbert, M. P., and Cytron, R. K. (2000). Contaminated garbage collection. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 264–273.

Carlisle, M. C. and Rogers, A. (1995). Software caching and computation migration in Olden. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 29–38, Santa Barbara, CA.

Cheng, P., Harper, R., and Lee, P. (1998). Generational stack collection and profile-driven pretenuring. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, volume 33 of *SIGPLAN Notices*, pages 162–173, Montreal, Québec, Canada. ACM Press.

Cohn, D. A. and Singh, S. (1997). Predicting lifetimes in dynamically allocated memory. In Mozer, M. C., Jordan, M. I., and Petsche, T., editors, *Advances in Neural Information Processing Systems*, volume 9, page 939. The MIT Press.

Dieckman, S. and Hölzle, U. (1998). A study of the allocation behavior of the SPECjvm98 Java benchmarks. In Jul, E., editor, *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings*, volume 1445 of *Lecture Notes in Computer Science*, pages 92–115. Springer-Verlag.

Dieckmann, S. and Hölzle, U. (2001). The allocation behavior of the SPECjvm98 Java benchmarks. In Eigenman, R., editor, *Performance Evaluation and Benchmarking with Realistic Applications*. The MIT Press.

Fitzgerald, R. P. and Tarditi, D. (2000). The case for profile-directed selection of garbage collectors. In *Proceedings of the Second International Symposium on Memory Management (ISMM)*, pages 111–120.

Forrest, S., Hofmeyr, S., Somayaji, A., and Longstaff, T. (1996). A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*. IEEE Press.

Harris, T. L. (2000). Dynamic adaptive pretenuring. In *Proceedings of the Second International Symposium on Memory Management (ISMM)*, pages 127–136.

Hayes, B. (1991). Using key object opportunism to collect old objects. In *Proceedings of SIGPLAN 1991 Conference on Object-Oriented Programming, Languages, & Applications*, volume 26(11) of *ACM SIGPLAN Notices*, pages 33–40, Phoenix, AZ. ACM Press.

Hayes, B. (1993). *Key Objects in Garbage Collection*. PhD thesis, Stanford University, Stanford, California.

Heberlein, L. T., Dias, G. V., Levitte, K. N., Mukherjee, B., Wood, J., and Wolber, D. (1990). A network security monitor. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEE Press.

Hertz, M., Blackburn, S. M., Moss, J. E. B., M^{C}Kinley, K. S., and Stefanović, D. (2002).

Error-free garbage collection traces: How to cheat and not get caught. In *SIGMETRICS 2002 International Conference on Measurement and Modeling of Computer Systems*, volume 30(1) of *ACM Performance Evaluation Review*, pages 140–151, Marina Del Rey, CA. ACM Press.

Hirzel, M., Henkel, J., Diwan, A., and Hind, M. (2002). Understanding the connectivity of heap objects. In *Proceedings of the Third International Symposium on Memory Management (ISMM)*, pages 36–49.

Inoue, H. and Forrest, S. (2002). Anomaly intrusion detection in dynamic execution environments. In *Proceedings of the New Security Paradigms Workshop 2002*. ACM Press.

Mukherjee, B., Heberlein, L. T., and Levitt, K. N. (1994). Network intrusion detection. *IEEE Network*, pages 26–41.

Rogers, A., Carlisle, M., Reppy, J. H., and Hendren, L. J. (1995). Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263.

Seidl, M. L. and Zorn, B. (1997). Predicting references to dynamically allocated objects. Technical Report CU-CS-826-97, University of Colorado.

Seidl, M. L. and Zorn, B. G. (1998). Segregating heap objects by reference behavior and lifetime. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23.

SPEC (1999). *SPECjvm98 Documentation*. Standard Performance Evaluation Corporation, release 1.03 edition.

Zilles, C. B. (2001). Benchmark health considered harmful. *ACM Computer Architecture News*, 29(3):4–5.