McGill University
School of Computer Science
Sable Research Group

Assessing the Impact of Optimization in Java Virtual Machines

Sable Technical Report No. 2005-4

Dayong Gu, Clark Verbrugge and Etienne M. Gagnon
{dgu1, clump}@cs.mcgill.ca, egagnon@sablevm.org

November 2, 2005

Abstract

Many new Java runtime optimizations report relatively small, single-digit performance improvements. On modern virtual and actual hardware, however, the performance impact of an optimization can be influenced by a variety of factors in the underlying systems. Using a case study of a new garbage collection optimization in two different Java virtual machines, we show the various issues that must be taken into consideration when claiming an improvement. We examine the specific and overall performance changes due to our optimization and show how unintended side-effects can contribute to, and distort the final assessment. Our experience shows that VM and hardware concerns can generate variances of up to 9.5% in whole program execution time. Consideration of these confounding effects is critical to a good understanding of Java performance and optimization.

1 Introduction

Compiler and runtime optimizations are typically deemed successful if a measurable, reasonably stable performance improvement can be shown over a selection of benchmarks, even if the effect is relatively small or not uniformly positive. In the case of Java virtual machine (VM) or software level optimizations, low-level hardware and VM effects are often presumed amortized through the complexity of interaction, or by considering average case behaviour.

In fact, inadvertent changes in low-level behaviour can significantly affect overall program execution, and seemingly stable results can vary by surprisingly large amounts for functionally identical programs. Using an in-depth study of a new garbage collection optimization, we describe a variety of confounding, low-level factors that are not always deeply considered when analyzing optimization behaviour, and give experimental evidence of their relative impact. In our case we find that a combination of instruction cache changes due to trivial code modifications, and subtle, consequent data layout and usage differences can induce almost a 10% whole program performance variation. We are able to show significant variations in both interpreter and JIT environments. As the largest contributors to variance are not unique to our case study, other optimizations achieving single-digit performance improvements (or degradations) may thus be affected by the same issues.

Previous studies on the complexity of measuring performance in modern VMs have argued for the importance of a holistic view of program performance [20], or have pointed out some of the factors that can directly affect and distort the measurement of specific optimizations, such as garbage collection [4]. In this paper we extend these concerns showing the wide range of issues that must be addressed to ensure a well-informed interpretation of performance change due to optimization, and the surprisingly large potential impact of low-level concerns on high level performance.

Specific contributions of our work include:

- We present and analyze a new garbage collection optimization that can improve GC performance in both JIT (Jikes RVM) and interpreter (SableVM) environments. Our detailed, multi-level experimental analysis provides further guidance on when to apply our optimization and which applications will benefit most.
- We experimentally show that a relatively high variation in performance is possible in a JVM due to unintended side-effects of code modifications. This can greatly distort the evaluation of optimization behaviour, yet it is not always fully considered in the literature.
- Using both VM and low-level hardware counter information, we provide an analysis and unique characterization of the SPECjvm98 and the DaCapo benchmarks with respect to their instruction and data cache

sensitivity. Understanding such benchmark characteristics is of course crucial to choosing appropriate optimizations.

The remainder of this paper is organized as follows. In Section 2, we discuss related work on GC and Java performance measurement and analysis. Section 3 then describes our garbage collection optimization, and Section 4 gives initial data on its performance. In Section 5, critical, low-level and unintended factors that affect the overall performance are shown through an analysis of anomalies and inconsistencies in the base measurements, particularly with respect to whole program behaviour. Section 6 provides directions for future work in this area and presents our conclusions.

2 Related work

Our investigations are motivated through a case study of a garbage collection (GC) optimization. GC has been a target of optimization for decades, and from a variety of directions. Ungar's *generational scavenging* [31] technique and more recent works on *Age-based* GC [29] *Older-first* GC [28] and *Beltway GC* [6], for instance, all aim to improve performance by adjusting collection time according to object lifetimes. Alternatively, live objects can be aggregated into regions in the heap based on a selection of object attributes. This either aims to improve data locality in the program [18, 21], or to reduce the memory access overhead of the collector [24]. Optimizations on data prefetching and lazy sweeping [7, 9] aim to improve data cache performance. Our approach tries to reduce the GC workload, although the implementation design is also helpful in reducing data cache misses.

Measuring the performance and understanding the behavior of Java programs is challenging. Many factors, from program characteristics, VM techniques and implementations, and OS strategies, to hardware platform performance, can affect to the final measured performance. B. Dufour *et al.* provides a set of concise and precisely defined dynamic metrics for Java programs [12]. At a lower level, L. Eeckhout *et al.* [13] analyze the interaction between Java virtual machine and microarchitectural platform factors by using principal component analysis to reduce the data dimensionality. M. Hauswirth *et al.* suggest an approach named *vertical profiling* [20] to understand how Java programs interact with the underlying abstraction levels, from application, virtual machine, and OS, to hardware. To understand the behaviour of a particular program execution, they first obtain profiling data from different levels, then visualize the data and discover the correlations between the anomalous performance and the profiling data visually or using statistical metrics. Our examination here is in the same spirit of a multi-levelled view of performance, though focusing on the variance due to optimizations rather than for program development.

Some other works specifically study GC performance. S. Blackburn *et al.* [4] discuss performance *myths* of canonical GC algorithms on widely used Java benchmarks. They compare the performance of classic GC and memory allocation algorithms in different configurations and environments. The impact of special implementation factors, such as "write barriers" and the size of nursery space of generational collectors, on mutator and GC performance are carefully studied. In this paper we extend their results to a further range of factors and influences, particularly unintended cache effects. The large impact of instruction cache changes has been noticed in other contexts as well [19], although our treatment is more in depth.

In order to fully analyze our benchmarks we have correlated instruction cache, data cache, and other low level events with program behaviour. Similar analyses have been done for C benchmarks [11, 23]; we of course aim at Java benchmarks, and concentrate on the relation between instruction and data cache sensitivity. Our data is gathered using the *hardware performance counters* found in modern processors, and used in numerous low-level performance studies [2, 17, 25, 30, 32]. In our case we used the PCL [3] and PAPI [8]

libraries for this low-level access.

3 Case study: GC optimization

Most Java virtual machines use mark-sweep or copying GC [22], or some variant (often generational) of these two algorithms. Both of these algorithms are *tracing* collectors. Starting from a set of root references (static variables, stack references), they visit each *reachable* object seeking references to other reachable objects. Finally, the memory storage of non-reachable objects is reclaimed. Gagnon and Hendren proposed a *bi-directional* object layout [16] aiming to improve the performance of GC tracing. In this section, we present a *reference section* tracing strategy that improves on it. This optimization and its implementations in SableVM and Jikes RVM form the basic case study that motivates our analysis of measurement concerns.

3.1 Bi-directional layout and reference section scanning

Bi-directional layout is an alternative way of physically representing objects in memory. Traditionally, all the fields of an object are located after the object header. The left graph in Figure 1 shows the traditional layout of an object of type **C** extending type **B** extending type **A**. The right graph in Figure 1 shows the bi-directional layout of the same object. The basic idea of bi-directional layout is to relocate reference fields before the object header and group them together in a contiguous section; we denote these sections as *reference section*. The main advantage of the bi-directional layout is the simplicity of locating all references in an object during GC. References are contiguous, and only a single count of reference section size must be stored (usually in the object header). There is no need to access a table of offsets in the object's type information block to identify references, as must be done with the traditional layout.

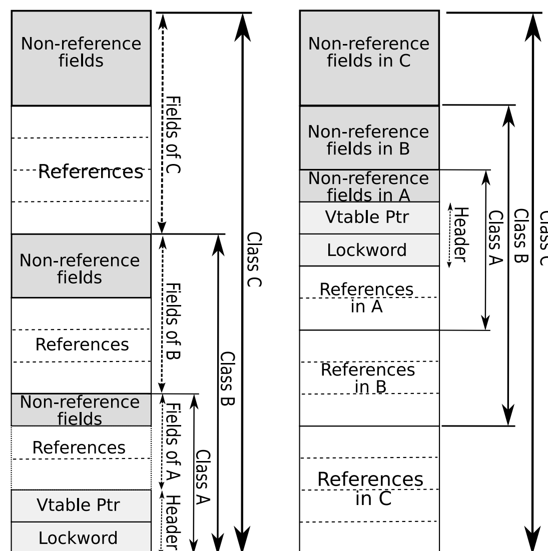

Figure 1: An instance of type **C** extending type **B** extending type **A** in both traditional and bi-directional object layouts

Based on the bi-directional layout, we developed a new reference section based (RS) scanning strategy to further reduce the required work for tracing from *per object* to *per reference section*: When a new reachable

object is found, the location of its reference section (if it does have one) is stored in a work list. The collector then uses this work list, which only contains relevant information, to copy or mark referents.

Compared to normal bi-directional layout tracing, our solution has the following advantages:

- The collector skips tracing of all reachable objects that have no references.
- The compactness of the work list may help improve cache locality.
- In coping collectors, using a work list allows for depth-first tracing instead of default breadth-first tracing. This usually leads to better cache locality [22].

3.2 Implementing RS scanning

We implemented RS scanning in two representative Java virtual machines: SableVM [14] and Jikes RVM [1]. SableVM is an efficient interpreter-based Java runtime which has a simple, yet efficient copying GC, and already implements the bi-directional object layout [15]. Jikes RVM is an adaptive compiler-based runtime for Java written in Java; new GC algorithms can be easily implemented using its *Memory Management Toolkit* (MMTk) [5].

3.2.1 SableVM

SableVM has a semi-space copying GC which uses a two-pointer scanning algorithm [22]. In the *to-space*, the *scan pointer* re-visits all copied objects to detect other reachable objects until it catches the *copying pointer*.

In our RS scanning implementation, the location of reference sections is saved in 512-entry blocks organized in a work list. We use the higher address end of the *to-space* to store these 512-entry blocks. Unused blocks are maintained in a free list, ready to be reused[1]. Compared to the total size of the heap, the space required by these 512-entry blocks is very small. For SPECjvm98 [27] benchmarks, we needed at most five blocks (in *javac*), or 20K at the end of *to-space* (and another 1K for headers) to perform GC on a two 16M semi-space heap.

Since our RS scanning strategy can reduce GC workload and improve data cache locality, we expect significant GC performance improvement in SableVM.

3.2.2 Jikes RVM

We also implemented the bi-directional layout and the RS scanning strategy in Jikes RVM version 2.3.4 by modifying both the RVM and the MMTk.

RVM

To implement the bi-directional object layout, we modified the object model component and the routines that compute the offset of fields. In the type information block, we replaced the array storing the offset of references with a single integer indicating the number of the references. We also moved the hash code from

[1] Actually, a separate block header is allocated for managing work/free lists.

before the object header to the end of the object in order to avoid changing the location of references when the object is hashed.

MMTk

The Memory Management Toolkit (MMTk) provides a full set of strategies to implement GC. It also provides a series of common compound data structures such as queue and dequeue which is quite useful in various GC implementations. We chose the address-pair dequeue data structure to maintain our reference section information. Then, we modified the scan utility to perform RS scanning. The sub-component of MMTk used to access the object type information was also changed to support the new object model. Furthermore, some basic GC plans, such as *basePlan* and *stopTheWorldGC*, were extended to support the new scan utility. In our current RS scanning implementation, we largely reuse existing MMTk routines—this is not always optimal for pure reference section scanning, but is sufficient for an initial implementation.

As Jikes RVM already used work lists for tracing, we do not expect as much improvement in Jikes RVM as in SableVM.

4 Initial experimental results

To study the effect of using the bi-directional layout and the RS scanning strategy, we collected performance data on the SPECjvm98 benchmarks [27] except *mpegaudio* and on five benchmarks, *antlr, bloat, fop, pms,* and *ps* of the DaCapo suite [10]. We excluded *mpegaudio* as it does not trigger garbage collection in SableVM's default settings. We excluded the *batik, chart, jython* and *xalan* DaCapo benchmarks as they either required unsupported graphical bindings or had reflection issues in the version of SableVM used for testing. We also excluded *hsqldb* as its execution time is mostly dependent on the thread scheduler of the underlying operating system. Experiments were run on an Athlon 1.4G workstation with 1G memory, with some earlier results from a Pentium III 733MHz workstation with 512M memory.

4.1 SableVM results

SableVM uses a simple semi-space copying GC. Yet, it delivers good GC performance due to the implementation of a number of efficient memory access techniques and an efficient algorithm for computing and retrieving GC maps [15].

Figure 2 shows the GC speedup obtained in SableVM by using RS scanning on our benchmarks with a 32MB initialize heap. A significant speedup, 16% in average, is obtained with a maximum of about 30% improvement on *db*.

We also measured the impact of RS whole program execution time, shown in figure 3. Although, the overall performance speedup is still positive in general, we notice a mysterious performance decline in some benchmarks, most obviously *raytrace*. Equally puzzling are the $> 2\%$ performance improvements shown by *compress* and *db*. GC usually takes less than 1% of execution time in the SableVM interpreter environment, and so this indicates a significant, unintentional impact on the mutator.

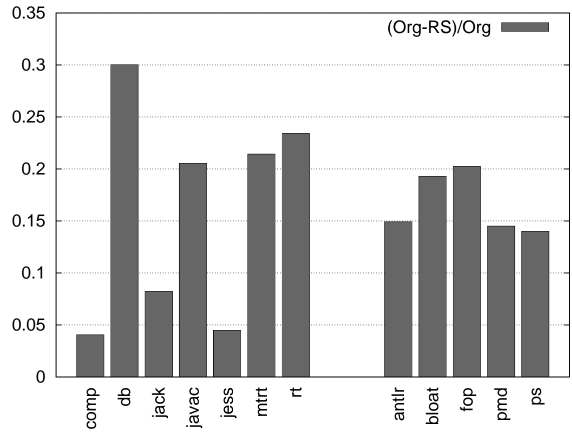

Figure 2: GC speedup in SableVM. Time is measured as cycles spent during GC execution. The vertical axis shows speedup, measured as: $\frac{(ExecutionTime_{Original} - ExecutionTime_{Optimized})}{ExecutionTime_{Original}}$

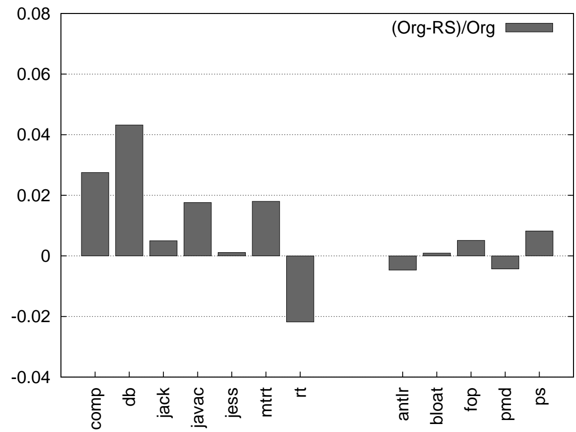

Figure 3: Whole program execution speedup in SableVM

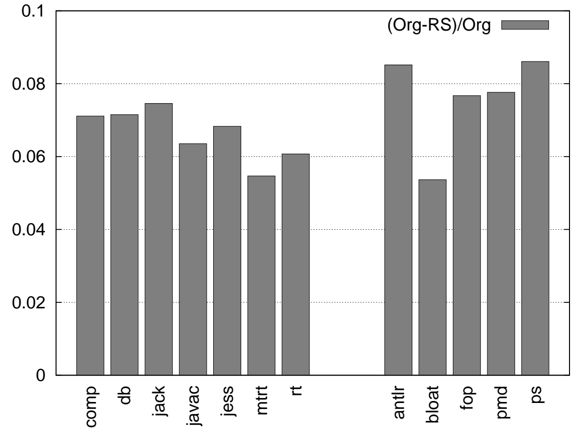

Figure 4: SS GC speedup in Jikes RVM

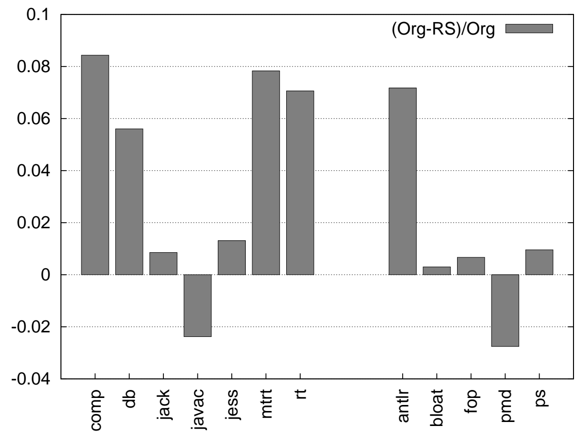

Figure 5: GenMS GC speedup in Jikes RVM

4.2 Jikes RVM results

We tested the RS scanning strategy in two types of GC in Jikes RVM: SemiSpace (SS) copying and Generational-copying-Marksweep hybrid GC (GenMS). We chose these two because they are representative GC configurations. The former is the most classic tracing GC which can give better performance for some benchmarks when the heap size is large enough [26]. The latter is the best choice for most benchmarks in most heap configurations of Jikes RVM.

We show the GC performance speedup for both collectors in Figures 4 and 5, and the results for whole program execution in Figures 6 and 7. The heap size was set to 32MB when testing SPECjvm98 benchmarks, as we did for SableVM. For the DaCapo benchmarks, the heap size was set to 80MB, due to a larger data workload.

For semi-space copying (SS), we obtained a stable improvement on the speed of GC for all benchmarks, similar to SableVM. At the same time we also show an overall positive performance for whole program execution time. We note that when using SS GC in Jikes RVM, GC takes a large portion of execution time (up to 40%). Whole program execution performance is therefore highly dependent on the collector's performance.

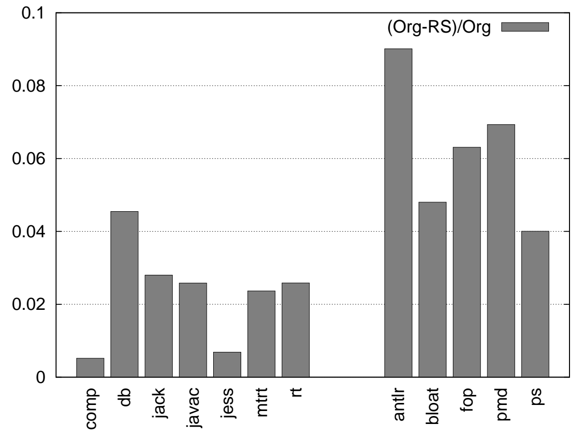

Figure 6: Whole program execution speedup when using SS GC in Jikes RVM

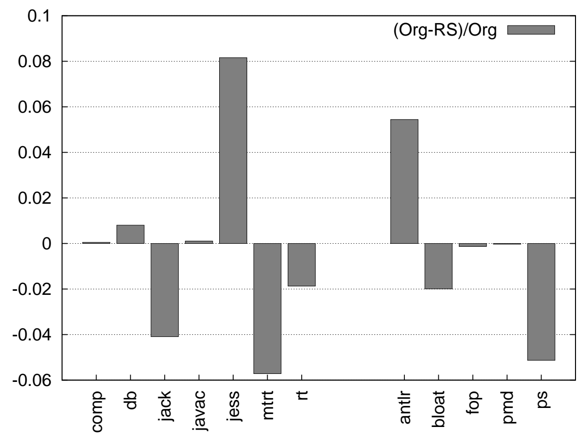

Figure 7: Whole program execution speedup when using GenMS GC in Jikes RVM

In the case of GenMS garbage collection performance results for both GC and whole program execution are less consistent. In the SPECjvm98 suite, the RS strategy still delivered overall GC improvement on most benchmarks (except *javac*), but in the DaCapo benchmarks we only see an improvement for the *antlr* benchmark. For other benchmarks GC performance is either similar to the original version or worse. Whole program execution time shows no obvious stable trend, positive or negative.

4.3 Summary

Viewed in isolation, and even overall in some cases, our RS scanning improves performance in both interpreter and adaptive JIT compiler environments. These results, however, are not well reflected in a general sense and anomalous measurements suggests significant variation in the performance of the mutator. A more detailed examination is thus the subject of the next section.

5 Detailed performance analysis

The inconsistent results found in Section 4 are puzzling from algorithmic and virtual machine implementation points of view, as nearly all of the VM structure beyond the necessary GC code was kept unmodified. Below we further investigate a series of low-level and benchmark-specific factors that contribute to the unexpected performance differences.

5.1 General factors

We integrated the PCL [3] and PAPI [8] libraries to SableVM and Jikes RVM respectively in order to retrieve hardware counter data during benchmark execution. We then identified and investigated each of the following factors as a potential source of performance irregularity.

5.1.1 Hardware instruction workload

As the source code of a virtual machines is compiled, an obvious source of performance difference is in the generated code. Even *improved* source code can generate an increase in hardware workload due to code generation patterns or optimizations.

We used hardware performance counter data to investigate the changes due to our implementation of RS. The final version of RS (used in our measurements) actually reduces the number of instructions executed during GC instructions for most benchmarks on both virtual machines. Furthermore, there is no noticeable difference in the executed instruction count for the mutator (variations were about 0.03% in average). In general, the RS strategy reduces the workload of GC and does not increase the workload of the mutator, and so is not a significant contributor to the performance differences.

5.1.2 Scan order

Changing the position of fields in the object layout has a potential impact on the data cache. Within the mutator these changes are expected to be both minor and amortized throughout execution. Within a semi-space copying collector, however, the scan order of references has a direct relation to the new location

of reachable objects in the heap after collection. Minor changes to scan ordering can result in a widely different distribution of objects in the heap, and can thus affect data locality in the mutator and in later collection cycles.

As the bi-directional layout changes the *natural* scan order of references, we define two scan orders:

- **Original favourite order (*OFO*):** This is the natural reference scan order in the traditional layout, where references of super classes are scanned first.
- **Bi-directional favourite order (*BFO*):** This is natural reference scan order in the bi-directional layout, where references of super classes are scanned last (after those of subclasses).

Figure 8 shows data cache miss comparison between *BFO* and *OFO* RS implementations in Jikes RVM. Switching the scan order leads to a new heap layout that changes data locality in the mutator. However, there is no obvious winner between the two orders. Most changes in data cache misses are lower than the variance in the execution time. Table 1 shows the average number of cycles between two consecutive L1 data or instruction cache misses. Given the low data cache density in the mutator part, it is safe to assert that data locality is not the key issue for the performance anomalies observed in Section 4.

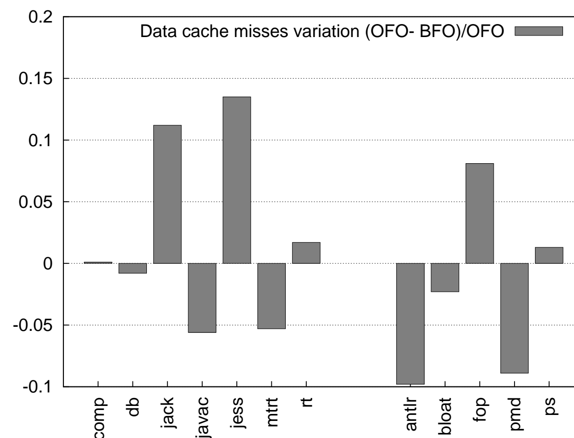

Figure 8: The effect of scan order on data cache performance in Jikes RVM

5.1.3 Hash code location

The position and value of the hash code is another implementation difference between our RS/bi-directional implementation and the original Jikes RVM implementation. However, our profiling results indicate that the number of objects that actually use a hash code is quite small for these benchmarks. For example, in the SPECjvm98 benchmarks (measured on SableVM, which uses a similar lazy hash code creation approach), most objects are not hashed. Even the *javac* benchmark, which exhibits the largest number of hashed objects, no more than 0.5% of copied objects are hashed. Therefore, we can rule out the hash code factor as the main source of anomalies.

Benchmark	In Mutator		In GC	
	Inst.	Data	Inst.	Data
compress	239	871	128K	77
db	725	400	341K	152
jack	145	244	38K	123
javac	201	259	264K	138
jess	176	376	80K	146
mtrt	534	312	264K	159
raytrace	475	311	242K	161
antlr	183	316	44K	150
bloat	150	205	132K	175
fop	203	269	200K	163
pmd	196	191	124K	148
ps	191	291	288K	200
Average	285	337	179K	150

Table 1: Benchmark characteristics: Cache Density (cycles per cache miss) in SableVM on a Pentium III workstation.

5.1.4 Code positioning

Finally, any change to the source code of the mutator or the collector is likely to change the precise location of parts of compiled code, possibly affecting the instruction cache success rate.

Table 1 shows that during GC very few instruction cache misses occur. In fact, in the GC phase the collector mostly works by iterating over a small set of instructions; it is thus unlikely for code position differences to cause any significant impact on GC performance.

On the other hand, Table 1 also shows that instruction cache misses are more frequent in the mutator. To gain additional insight on the issue, we performed two experiments.

The second column of Table 2 shows the largest performance changes we found in SPECjvm98 benchmark on a series of *code shifted* versions of SableVM. The only difference between these versions is the length of some extra useless space, varying from 0 bytes to double the size of a cache line, reserved in the string table section of the executable binary. This causes later binary executable code to be shifted, without actually changing the binary code. Surprisingly, such a trivial modification triggered significant performance differences, up to 6.09%.

As a second experiment, we changed the position of some code in Jikes RVM by hand, and we generated a set of variances. We then compiled two versions of Jikes RVM: one with and one without the *Hardware Performance Monitoring* (HPM) component. In our measurements, no HPM code was executed. In other words, we simply added a piece of non-executed code to Jikes RVM. The results are shown in the third column of Table 2. Note how the simple addition of some non-executed component to Jikes RVM can affect performance by up to 9.46%!

5.2 Benchmark specific factors

In this section, we extend our analysis to benchmark-specific factors which can also influence the performance. These properties include the relative number and distribution of reference fields, variation in GC

Benchmark	L.V.F.(%) of Code Shifting	L.V.F.(%) of Extra Component
compress	2.78	1.24
db	6.09	4.80
jack	2.04	5.19
javac	2.00	4.40
jess	2.69	6.39
mtrt	3.69	4.70
raytrace	3.21	6.42
antlr	0.89	5.75
bloat	1.49	9.46
fop	1.14	2.94
pmd	1.39	3.31
ps	1.89	1.62

Table 2: Impact of the code shifting in SableVM and adding an extra never executed component in Jikes RVM (L.V.F. for Largest Variation Found in execution time)

collection points and GC strategy, and relative cache sensitivity of the benchmarks.

In the experiments below we used an instrumented SableVM to gather heap-related data.

5.2.1 RS scanning gains

Section 3.1 presents the potential advantages of the RS strategy. By its nature, RS scanning will bring larger benefits when accessing long, contiguous reference sections. For objects with a single reference, the cost of RS scanning is greater than the cost of normal scanning. We thus measured the number of reference field in scanned objects in SPECjvm98 benchmarks.

We found that *db, mtrt* and *raytrace* have more than 40% objects with no reference at all. These objects are skipped by the RS strategy which leads to a significant improvement in GC speed over the original SableVM implementation. A relatively large number of single-reference objects are found in *jack* and especially *jess* (43.4%), for which our RS strategy brings less improvement. The behavior of *compress*, which has the lightest GC workload of all analyzed SPECjvm98 benchmarks, and of *javac*, which triggers four *forced* GCs, however, cannot be completely explained from the reference composition data alone.

5.2.2 GenMS behaviour analysis

Jikes RVM's garbage collector manages both application data and VM-specific data. Thus purely internal VM changes can be reflected in the workload experienced by applications. This can be seen as a major source of the anomalous behaviour of some of the DaCapo benchmarks under the GenMS GC strategy.

Our modifications to the Jikes RVM object model in the implementation of the RS strategy caused a slight change in GC workload. In particular, we noticed that the size of surviving objects, after collection, for these benchmarks was slightly different (by only a few Kilobytes) between the original and the RS implementations. Given the large heap size, we would not expect any significant impact from this when using a semi-space copying collector. But, in the case of a generational collector, where most of the work is done on a small nursery, a small size difference can have larger impacts. We found that *bloat* causes 27 GCs with the

RS version and 33 with the original. As a further complication, a lower number of GCs doesn't necessarily mean lower total GC time. In this case, the nursery-GC that follows a full-heap GC is much longer than other nursery GCs. The RS version is actually faster in this benchmark until near the end of the execution, when after a full-heap GC, one extra longer nursery-GC is triggered. This final GC eliminates all the prior gains. The aggregate GC time of *bloat* for both original and RS versions in two heap size settings(80M and 160M) are shown in Figure 9. In both heap settings, the RS version is faster than the original version at every step. In the 80M case, RS wins over original version consistently until the last step where an extra long nursery GC cycle is triggered. In the 160M case, RS reduces total garbage collection time by 4.4%.

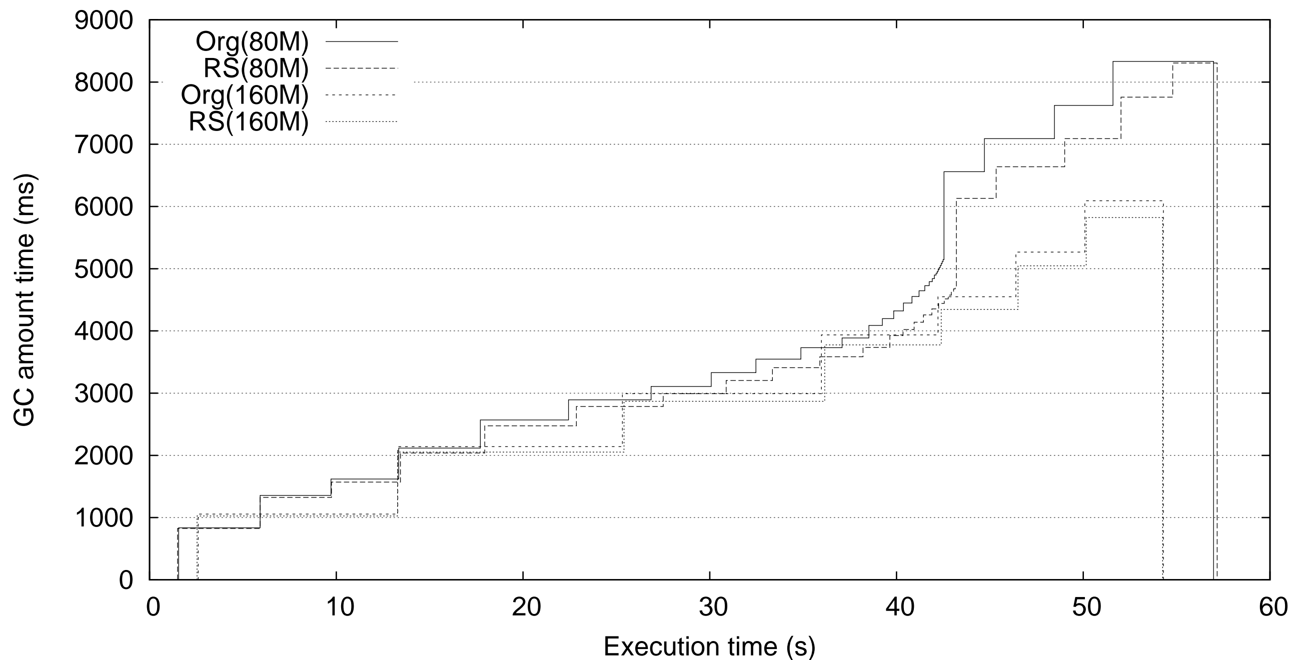

Figure 9: Bloat GenMS performance can be changed because of minor code differences

5.2.3 Generational or semi-space?

Specific benchmarks respond differently to different GC strategies under different input loads. This can also affect perception of performance.

Jikes RVM's GenMS GC, for instance, treats objects differently according to their potential lifetime, and in most cases provides better performance than SemiSpace GC. Many benchmarks, such as *jack, jess* and *ps*, are suitable for generational GC, where they can operate more than 10 times faster in ordinary heap size settings. On the other hand, the performance gains are sometimes lost when operating on large heaps in some benchmarks. In some cases, such as *db* and *pmd*, SemiSpace is actually nearly as fast or event faster than GenMS in normal heap settings. Since the performance of GC is a function of heap size, we examine how much the GenMS can win over SS in different heap sizes. We show the result for SPECjvm98 in Figure 10 and for DaCapo benchmarks in Figure 11. The y-axis expresses the GC time of SemiSpace normalized to that of GenMS. The heap size is shown as a multiple of the smallest heap size setting. Note that there is no SemiSpace GC on *compress* and *db* when the heap size is larger than 3.5X and 4X the minimum setting respectively.

Obviously the specific choice of GC strategy and heap size has a significant affect on performance. The impact on measuring optimizations is more subtle, and depends on the varying benchmark responses to these parameters. An optimization to a strategy being used in a sub-optimal situation may be more or less effective, affecting different benchmarks to different degrees.

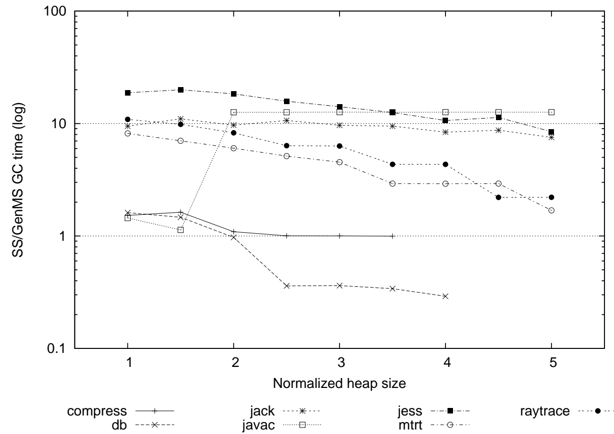

Figure 10: Performance comparison between SS and GenMS on SPECjvm98 benchmarks, the minimal heap size is 32M

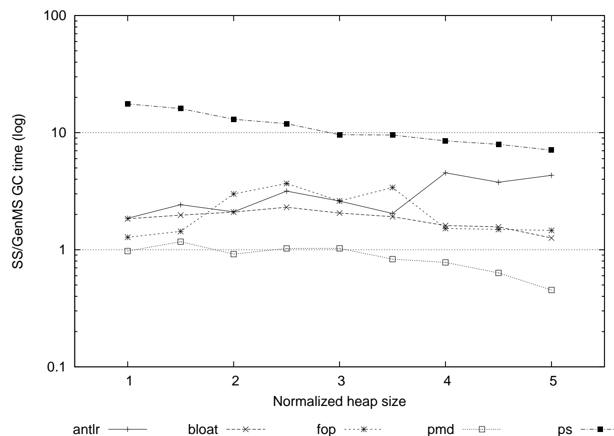

Figure 11: Performance comparison between SS and GenMS on DaCapo benchmarks, the minimal heap size is 40M

5.2.4 Hardware related benchmark characteristics

To further study the benchmark characteristics on hardware, we generated hardware event traces using Jikes RVM to show the hardware behaviour at each thread switching interval. These trace results show significant variations among benchmarks for different hardware events.

We will briefly discuss, here, the results for the following sample benchmarks: *compress, db* and *jack*. Their L1 instruction and data cache densities are respectively shown in Figures 12, 13 and 14.

Compress and *db* have more data than instruction cache misses, unlike *jack*. We note that all three benchmarks show some kind of recursive pattern in their instruction cache curve. Particularly, *db* exhibits many

high impulses in the instruction cache curve. In *jack*, the number of instruction cache misses is larger and the width of each recursive pattern is much wider. *compress* has lower instruction cache miss peaks, which is expected as it executes a comparatively smaller piece of code. These facts exhibit different optimization opportunities for these benchmarks. For example, data locality optimizations have higher potential benefits in *compress* and *db*. Moreover, the high impulses in *db* indicates the existence of small chunks of hot code in this benchmark. If the instruction cache behavior was somehow changed, the performance could potentially be affected. In other words, *db* is likely very sensitive to code motion.

Figure 15 shows the different cache performance bias of benchmarks. The center position is determined by the I-Cache (x-axis) and D-cache (y-axis) miss density. The rectangle area for each benchmark shows the standard cache density variation for the measurement intervals. Benchmark *compress* is an extremely data cache biased program, while a large number of benchmarks, such as *antlr*, are highly instruction cache biased programs. The performance of *db, mtrt* and *bloat* have similar dependency on these two caches. The two arrows for each benchmark represent the average of the top 10% largest cache miss variations between two continuous intervals of the benchmark. The length of the arrow is an indicator for the probability that a program phase transition points can be detected by monitoring the corresponding hardware event variation.

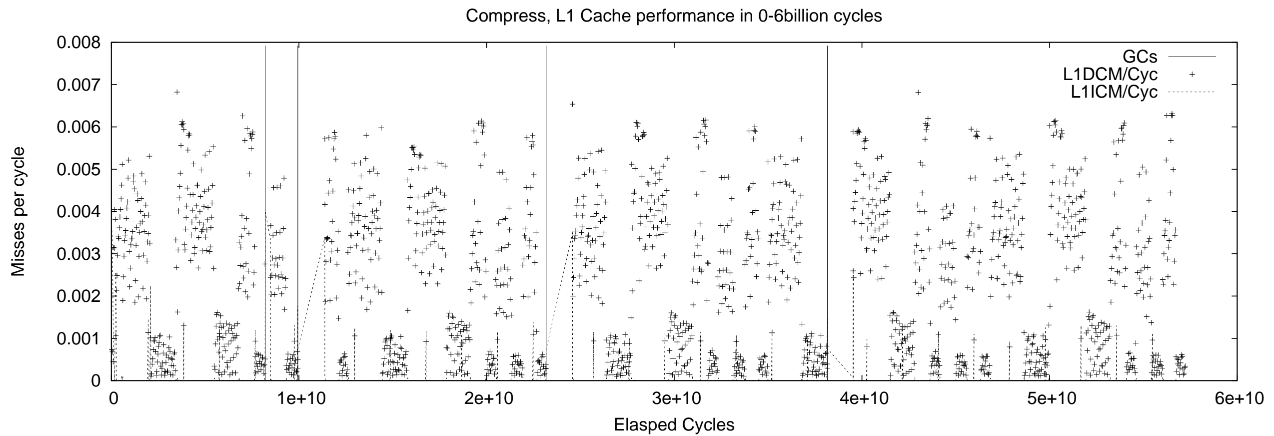

Figure 12: Compress hardware event trace

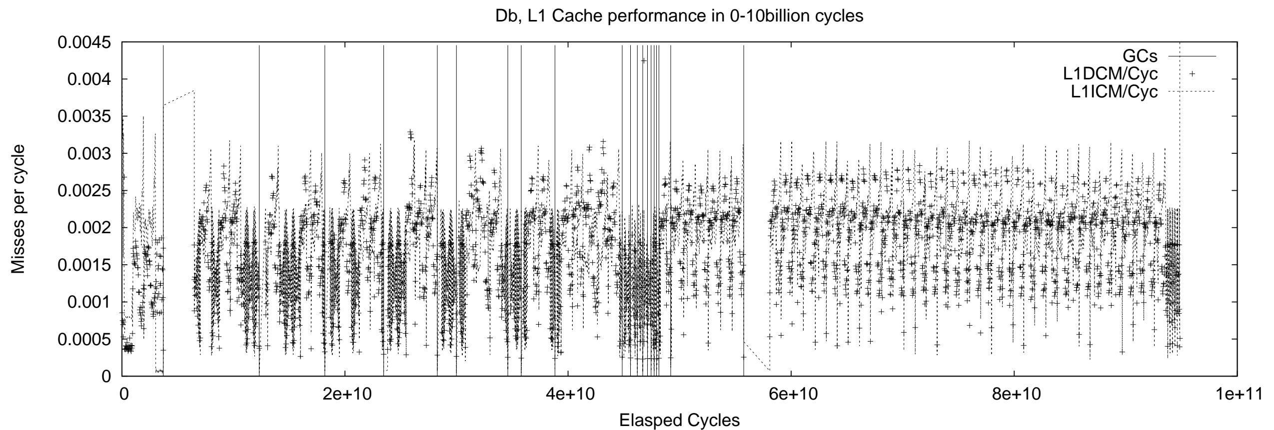

Figure 13: Db hardware event trace

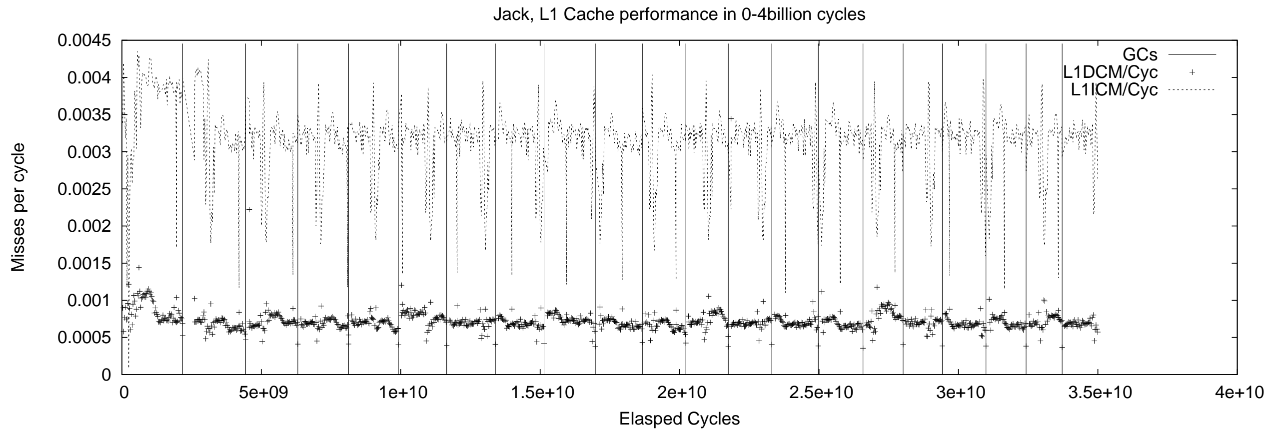

Figure 14: Jack hardware event trace

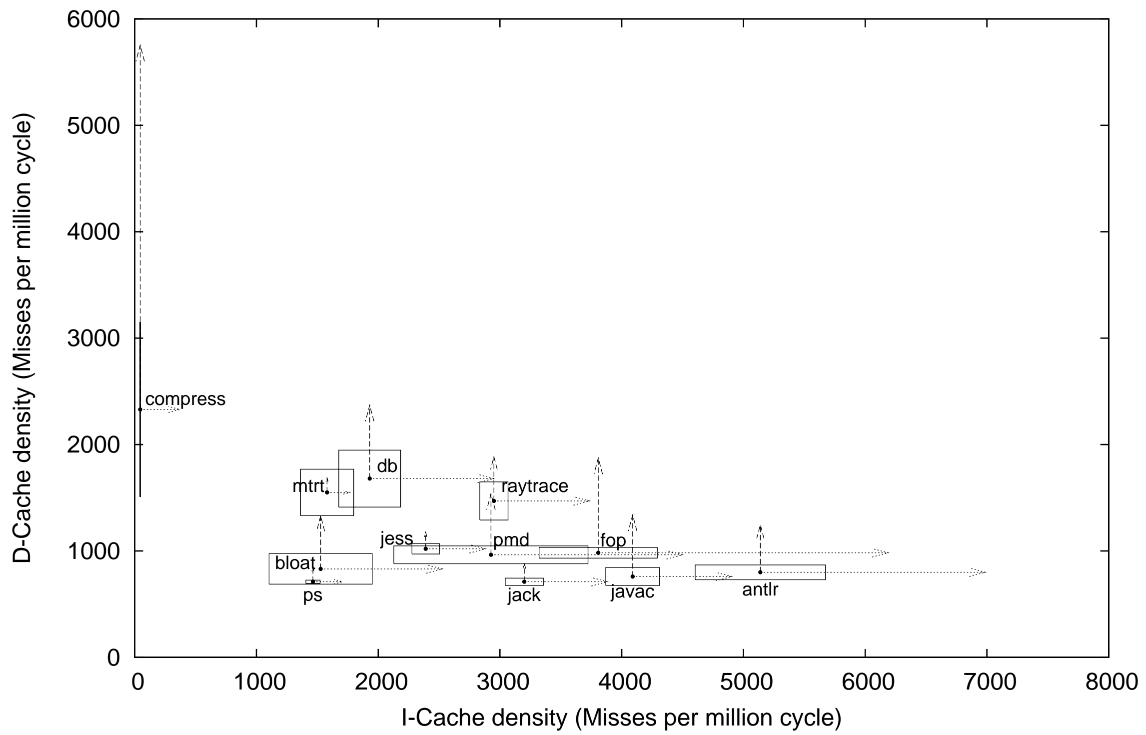

Figure 15: Benchmark cache bias

5.2.5 Summary

From the above data, we summarize that:

- The reference composition of the objects is an important factor to determine the suitability of applying the RS scanning strategy.
- The performance of Java virtual machines can be significantly affected by unintended code motion side-effects.
- Benchmarks show different sensitivity to code motion side-effects, as well as other hardware-level issues (data cache, etc.).
- Generational GC does not give a constant improvement over semi-space GC across all benchmarks. (This situation exhibits some potential for adaptively setting the nursery size to improve performance).
- The real performance of GC improvements is difficult to measure in hybrid systems like Jikes RVM, where internal VM-specific data is stored in the heap easily perturbing results.

6 Conclusions and future work

Optimizations in a modern virtual machine environment clearly have the potential for complex interactions with various systems aspects, high and low-level. Our GC optimization case study shows that these interactions are both subtle and significant. Cache effects dominate, and are a well known source of variance; their large impact and indirect causality is, however, surprisingly. Our experimental results also show how changes in GC timing, caused by code or data modifications, further contribute to performance variation. These basic concerns apply equally well to many other optimizations—certainly any that affect the placement of code or data, or which may alter the timing or parameters of GC. Unless these factors are controlled for, conservatively, real-time performance changes of less than 10% should be considered preliminary.

Of course a potential variance is also a potential source of optimization. At a fine grain the cache behaviour shows strong repetitive sequences, and at a coarse grain many benchmarks have a bias in their sensitivity toward instruction or data cache misses; future work on adaptive optimizations that branch on early detection of these qualities may be very applicable. Different, and more accurate (less-perturbing) measurements may also help decide on an optimization—we have shown that stable, average behaviour can be misleading, and development of appropriate measurement/evaluation strategies that (heuristically at least) give a good sense of potential variation would be quite useful. Currently we are focusing on techniques for more optimal code layout in order to better exploit the instruction cache.

References

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 314–324, Oct. 1999.

[2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, Nov. 1997.

[3] R. Berrendorf, H. Ziegler, and B. Mohr. Pcl-the performance counter library. `http://www.fz-juelich.de/zam/PCL/`.

[4] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 25–36, June 2004.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 137–146. IEEE Computer Society, May 2004.

[6] S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: getting around garbage collection gridlock. *SIGPLAN Not.*, 37(5):153–164, June 2002.

[7] H.-J. Boehm. Reducing garbage collector cache misses. In *ISMM '00: Proceedings of the 2nd international symposium on Memory management*, pages 59–64, Oct. 2000.

[8] S. Brown, J. Dongarra, N. Garner, K. London, and P. Mucci. PAPI. `http://icl.cs.utk.edu/papi`.

[9] C.-Y. Cher, A. L. Hosking, and T. N. Vijaykumar. Software prefetching for mark-sweep garbage collection: hardware analysis and software redesign. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 199–210, Oct. 2004.

[10] DaCapo Group. The DaCapo benchmark suite. `http://www-ali.cs.umass.edu/DaCapo`.

[11] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 220. IEEE Computer Society, Sep. 2003.

[12] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 149–168, Oct. 2003.

[13] L. Eeckhout, A. Georges, and K. De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 169–186, Oct. 2003.

[14] E. Gagnon. SableVM. http://www.sablevm.org/.

[15] E. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, McGill University, 2002.

[16] E. M. Gagnon and L. J. Hendren. SableVM:A Research Framework for the Efficient Execution of Java Bytecode. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, pages 27–40. USENIX Association, Apr. 2001.

[17] D. Gu, C. Verbrugge, and E. Gagnon. Code layout as a source of noise in JVM performance. *Studia Informatica Universalis*, 4(1):83–99, March 2005.

[18] S. Z. Guyer and K. S. McKinley. Finding your cronies: static analysis for dynamic object colocation. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 237–250, Oct. 2004.

[19] K. Hammond, G. L. Burn, and D. B. Howe. Spiking your caches. In J. T. O. Donnell and K. Hammond, editors, *GLA*, pages 58–68. Springer-Verlag, July 1993.

[20] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: understanding the behavior of object-priented applications. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 251–269, Oct. 2004.

[21] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 69–80, Oct. 2004.

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

[23] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *ISPASS '05: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, page 220. IEEE Computer Society, March 2005.

[24] F. Qian and L. Hendren. An adaptive, region-based allocator for java. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 127–138, June 2002.

[25] R. M. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W.-F. Wong. Compiler orchestrated prefetching via speculation and predication. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 189–198, Oct. 2004.

[26] S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 49–60, Oct 2004.

[27] Standard Performance Evaluation Corportion. SPECjvm98 benchmarks. `http://www.spec.org/osg/jvm98`.

[28] D. Stefanović, 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 *MSP '02: Proceedings of the 2002 workshop on Memory system performance*, pages 25–36, June 2002.

[29] D. Stefanović, K. S. McKinley, and J. E. B. Moss. Age-based garbage collection. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 370–381, Oct. 1999.

[30] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of Java applications. In *VM'04:Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, May 2004.

[31] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 157–167, Apr. 1984.

[32] X. Vera and J. Xue. Let's study whole program cache behavior analytical. In *International Symposium on High-Performance Computer Architecture (HPCA 8) (IEEE)*, pages 175–186, Feb. 2002.