# Relative Factors in Performance Analysis of Java Virtual Machines

Dayong Gu Clark Verbrugge
School of Computer Science
McGill University
Montréal, Québec, Canada
{dgu1, clump}@cs.mcgill.ca

Etienne M. Gagnon
Département d'informatique
Université du Québec à Montréal
Montréal, Québec, Canada
egagnon@sablevm.org

## 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 relative effects of 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, objective understanding of Java performance and optimization.

***Categories and Subject Descriptors*** C.4 [*Performance of Systems*]: Measurement Techniques, Design Studies

***General Terms*** Experimentation, Measurement, Performance

***Keywords*** performance analysis, Java, garbage collection, hardware counters, caches

## 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.

Many optimizations, however, result in small, single-digit performance changes, and in these cases uncontrolled factors arising from lower levels of execution may account for a significant amount of the performance difference, greatly distorting the perceived effect of an optimization. Unfortunately, not only is the breadth of potential influences not always obvious, but it is also not always clear which factors are most important with respect to perturbing performance measurements. This resulting lack of guidance often means that optimization researchers do not always give a sufficiently wide or deep consideration to potential confounding factors from the underlying systems.

In this paper we focus on two main concerns. First, an exposition of various external, and benchmark specific factors that can influence a VM or source-level optimization, and second a quantitative evaluation and comparison of the different influences. Using an in-depth case study of a simple garbage collection optimization we are able to show 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 [19], 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 give experimental results from both an interpreter and a JIT environment showing that side-effects of lower level execution can account for almost 10% of measured performance.
- We provide guidance on which factors are most critical by giving a quantitative and comparative analysis of a variety of potential confounding factors in assessing JVM performance.
- Using both VM and low-level hardware counter information, we provide an analysis and unique characterization of the SPECjvm98 [28] and the DaCapo [23] benchmarks with respect to their instruction and data cache sensitivity.
- Our optimization case study is also a new garbage collection optimization that can improve GC performance in both JIT (Jikes RVM) and interpreter (SableVM) environments.

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, we refine our investigation and give more detail on the



*VEE'06* June 14–16, 2006, Ottawa, Ontario, Canada.

impact of various performance influences. Section 6 provides directions for future work in this area and presents our conclusions.

## 2. Related work

Measuring the performance and understanding the behaviour of Java programs is challenging. Many factors, from program characteristics, VM techniques and implementations, and OS strategies, to hardware platform performance, can affect the final measured performance. B. Dufour *et al.* provides a set of concise and precisely defined dynamic metrics for Java programs [10]. At a lower level, L. Eeckhout *et al.* [11] 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* [19] 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.

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* [32] technique and more recent works on *Age-based* GC [30] *Older-first* GC [29] and *Beltway GC* [5], 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 [20, 17], or to reduce the memory access overhead of the collector [24]. Optimizations on data prefetching and lazy sweeping [8, 6] 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.

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 [18], 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 [22, 9], and recently for Jikes as well [26]. Our data is gathered using the *hardware performance counters* found in modern processors, and used in numerous low-level performance studies [2, 33, 25, 16, 31]. In our case we used the PCL [3] and PAPI [7] libraries for this low-level access.

## 3. Case study: GC optimization

In this section we briefly describe a simple GC optimization and its implementation in both SableVM and Jikes RVM. We will use this example optimization to show the number and subtlety of factors that need to be considered when examining performance results, as well as give concrete evidence of their relative impact.

Our optimization case study is based on a simple and general improvement to *tracing* garbage collectors. Tracing GCs are found in most Java virtual machines.

Starting from a set of root references (static variables, stack references), a tracing GC visits each *reachable* object seeking references to other reachable objects. Once the live set is determined the memory storage of non-reachable objects is reclaimed. Gagnon and Hendren proposed a *bi-directional* object layout [14] aiming to improve the performance of GC tracing, and here we present a *reference section* tracing strategy that attempts to validate and improve that work.

### 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*s. 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.

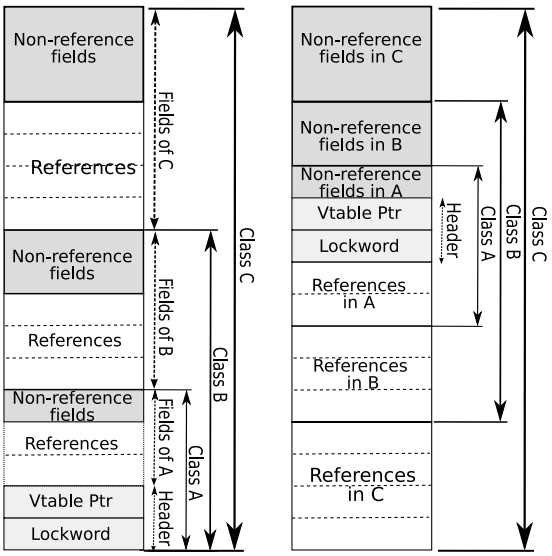


**Figure 1.** An instance of type **C** extending type **B** extending type **A** in both traditional and bi-directional object layouts

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 while GC is in progress.
- In copying collectors, using a work list allows for depth-first tracing instead of default breadth-first tracing. This usually leads to better cache locality [21].

### 3.2 Implementing RS scanning

We have implemented RS scanning in both SableVM[12] and Jikes RVM[1]. Here we give a brief overview; more implementation details can be found in [15].

**SableVM**

SableVM has a semi-space copying GC which uses a two-pointer scanning algorithm [21]. The *scan pointer* is used to trace references in copied objects, while the *free pointer* tracks the location of unallocated memory in the target semi-space.

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 blocks, and unused blocks are maintained in a free list, ready to be reused. Compared to the total size of the heap, the space required for this work list is very small. For SPECjvm98 [28] benchmarks, we needed at most five blocks (in *javac*), or 20K at the end of the *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 a significant GC performance improvement in SableVM.

**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 Memory Management Toolkit (MMTk). Modifications to the RVM are straightforward. 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 before the object header to the end of the object in order to avoid changing the location of references when the object is hashed.

Our GC work integrates with Jikes through the MMTk. 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 examine the effect of using the bi-directional layout and the RS scanning strategy, we collected performance data on the SPECjvm98 benchmarks [28] run with input size "s100" and on five benchmarks, *antlr, bloat, fop, pmd,* and *ps* of the DaCapo suite [23] using the default input size. We excluded *mpegaudio* from the SPEC benchmarks as it needs no garbage collection in SableVM's default heap settings. We excluded the *batik, chart, jython* and *xalan* DaCapo benchmarks as they either required unsupported graphical bindings or had reflection issues unsupported 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 under the debian Linux operating system on an Athlon 1.4G workstation with 1G memory, with some earlier results from a Pentium III 733MHz workstation with 512M memory. Both environments were isolated and minimized for testing, and each benchmark data points represent the average of the medium 3 values in 5 runs.

### 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 [13].

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*.

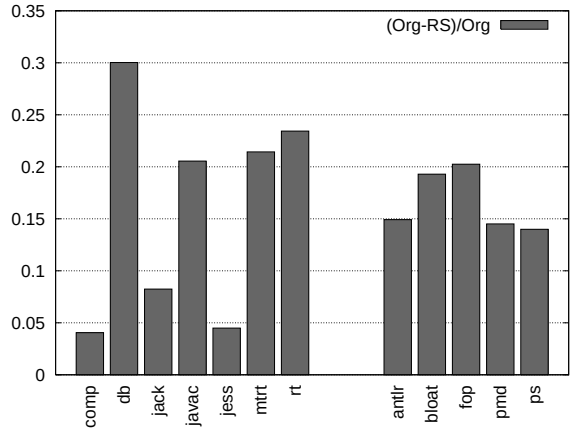


**Figure 2.** GC speedup in SableVM. Time is measured as cycles spent during GC execution. The vertical axis shows speedup, measured as: $\frac{(ExecutionTime_{Original} - ExecutionTime_{Optimized})}{ExecutionTime_{Original}}$

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 an anomalous performance decline in some benchmarks, most obviously *raytrace*. Equally surprising 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.

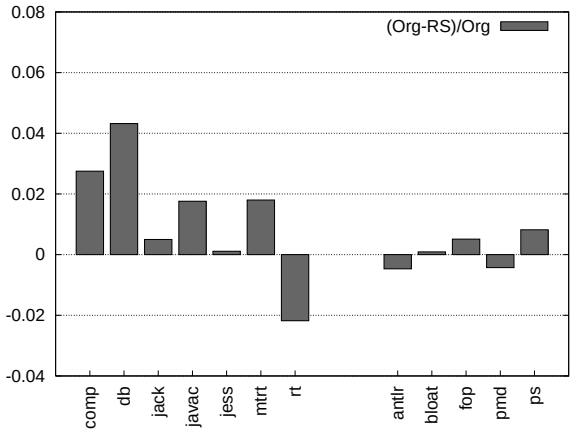


**Figure 3.** Whole program execution speedup in SableVM

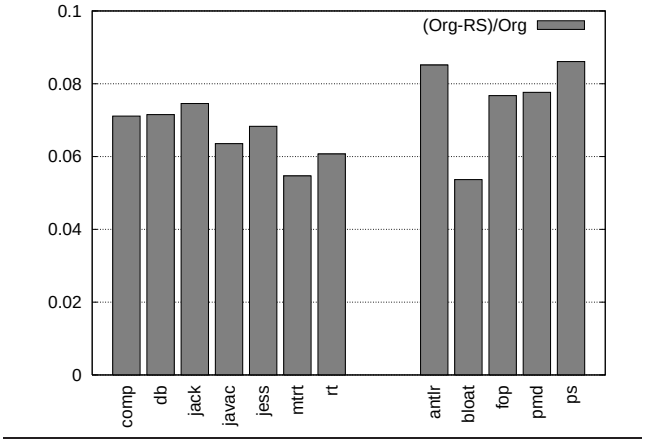


**Figure 4.** SS GC speedup in Jikes RVM

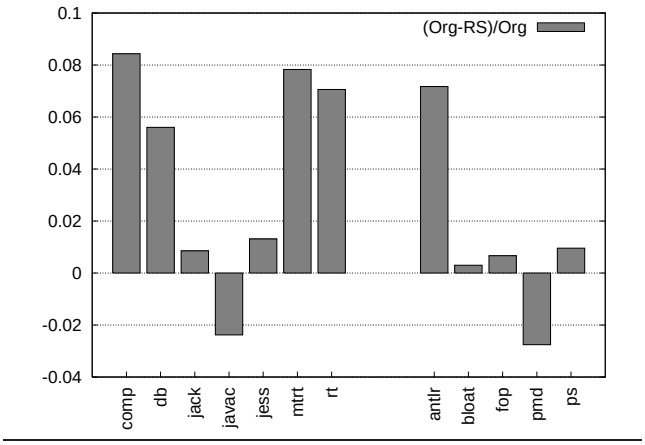


**Figure 5.** GenMS GC speedup in Jikes RVM

### 4.2 Jikes RVM results

We tested the RS scanning strategy in two types of GC in Jikes RVM: SemiSpace (SS) copying (basebaseSemispace configuration) and Generational-copying-Marksweep (GenMS) hybrid GC (basebaseGenMS configuration). We chose these two because they are representative GC configurations. The former is the classic tracing GC which can give better performance for some benchmarks when the heap size is large enough [27]. 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, the same as SableVM's default setting. For the DaCapo benchmarks, the heap size was set to 80MB, due to the larger data workloads than SPECjvm98. Note that our results here are not overly sensitive to heap size, and testing on other reasonable heap sizes produces similar results.

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%, using SS GC and the baseline compiler). Whole program execution performance is therefore highly dependent on the collector's performance.

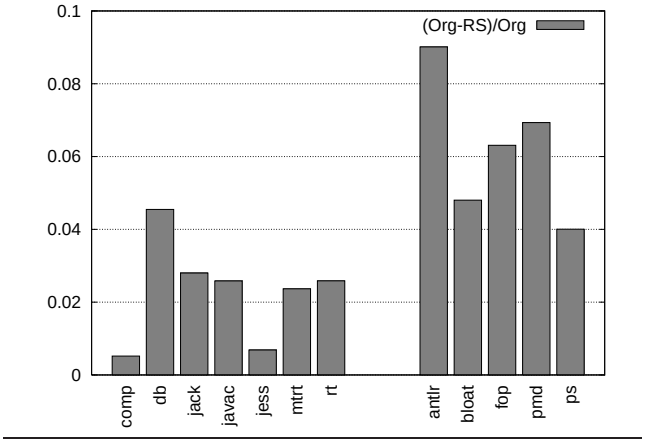


**Figure 6.** Whole program execution speedup when using SS GC in Jikes RVM

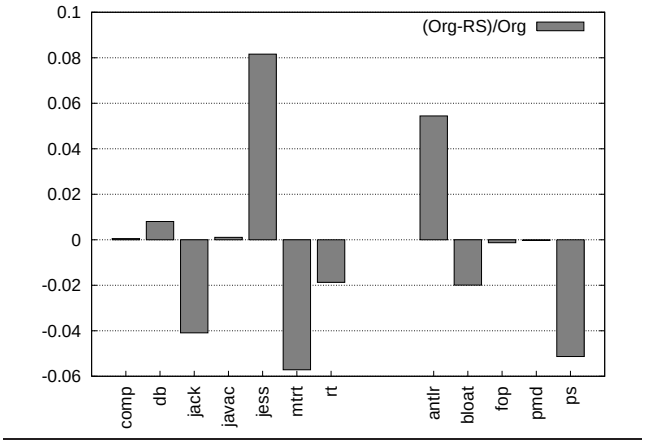


**Figure 7.** Whole program execution speedup when using GenMS GC in Jikes RVM

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 GC 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 to determine and compare the responsible influences is the subject of the next section.

## 5. Detailed performance analysis

To explain program performance on real platforms is tricky. Performance is affected by a variety of factors at different levels. Some factors have *system-wide* effects, such as available physical memory, fragmentation, number of simultaneously executing processes,

network bandwidth, disk usage, etc. While it is desirable, in theory, to study the effect of system factors on performance, it is also difficult to do so—such factors are very dependent on the specific system used and the workload at measurement time. In our experiments we perform our tests on a newly restarted, isolated system with a minimal workload, so as to minimize system-wide effects.

Other *general factors* affect the performance of Java programs. These factors can be divided into two sub-categories: *code related* such as overall instruction workload, hashcode location, and code positioning, and *data related* such as heap organization, data location, and scan order. In Section 5.1, we will study these general factors.

Finally, there are additional factors which can significantly affect the performance of particular Java benchmarks. We call these *benchmark-specific factors*. The selection of GC points, for instance, as well as the high level choice of GC algorithm can affect different kinds of benchmarks in different ways. Individual programs may also show a biased affinity or disaffinity for specific hardware or software designs or components. We study these factors in Section 5.2.

To measure low-level performance variations, we integrated the PCL [3] and PAPI [7] libraries into SableVM and Jikes RVM respectively.

### 5.1 General Factors: Code and Data Management

#### 5.1.1 Instruction workload

As the source code of a virtual machine 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 downstream 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 Hash code location

In support of the `java.lang.Object.hashCode()` method, many virtual machines derive object hash codes from heap addresses, and may also store calculated hashes in the object header. Use of hash code's thus can have an indirect effect on performance if the heap memory is laid out differently, and the position and value of an object's hash code is another implementation difference between our RS/bi-directional implementation and the original Jikes RVM implementation.

In practice, 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 to Jikes RVM), most objects are not hashed. Even in the *javac* benchmark, which exhibits the largest number of hashed objects, no more than 0.5% of copied objects are hashed. Measuring the precise effect of different hash values is of course quite difficult, but the limited use of hash codes in our benchmarks strongly suggests that any differences have a minimal impact.

#### 5.1.3 Code positioning

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. Our final code-related effect is thus a consideration of the effect of minor changes in code positioning on performance.

| **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: average number of cycles between cache misses in SableVM on a Pentium III workstation.

| **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)

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 these measurements no HPM code was executed; i.e., 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%! Performance changes due to changing code position clearly have potential to be quite large relative to other "noise" effects.

#### 5.1.4 Data location factors

Our case study optimization changes the position of fields in the object layout, and this has an obvious potential impact on the data cache. For the majority of objects with relatively few fields, however, proximity of data is maintained, and at least within the mutator these changes are expected to be both minor and amortized throughout execution.

A more subtle and important data cache effect can arise from the use of scanning GCs. In a tracing collection based system the order in which references are scanned has a direct impact on 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 the natural reference scan order in the bi-directional layout, where references of super classes are scanned last (after those of subclasses).

Figure 8 shows a 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 miss density in the mutator part, it is safe to assert that data locality changes due to scan order are not the key issue for the performance anomalies observed in Section 4.

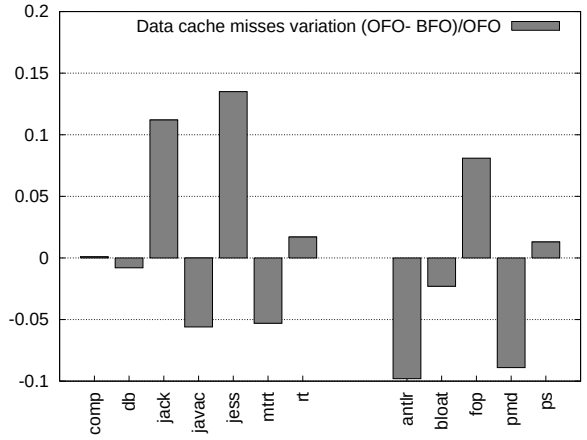


**Figure 8.** The effect of scan order on data cache performance in Jikes RVM

### 5.2 Benchmark specific factors

It is also the case that individual benchmarks may have properties that produce very different reactions to a given optimization. Below 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 (relatively unique to our particular optimization), and more generic effects such as variation in GC collection points and GC strategy, and relative cache sensitivity of the benchmarks.

#### 5.2.1 Reference field usage

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, however, the cost of RS scanning is greater than the cost of normal scanning. The number of reference fields typically found in objects can also thus influence performance, and so we measured the number of reference fields 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, leading 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 behaviour 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. For this we need to also consider more general properties of GC behaviour.

#### 5.2.2 GC frequency and workload

Our code and data modifications have strong potential to adjust the workload given to GC during program execution. This can have both obvious and subtle consequences. Jikes RVM's garbage collector, for instance, manages both application data and VM-specific data. Thus purely internal VM changes can be reflected in the workload experienced by applications; this may be a primary source for 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 also causes a slight change in GC workload. In particular, the size of surviving objects after a collection for these benchmarks is 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. However, in the case of a generational collector, where most of the work is done incrementally, a small size difference can have a much larger impact; e.g., the *bloat* benchmark has 27 GCs with the RS version and 33 with the original.

As a further complication, a lower number of GCs does not 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, eliminating 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 the 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%. It is well known of course that changing heap sizes can alter GC performance results; here we see that generational approaches can make GC performance very sensitive to relatively small variations in heap usage.

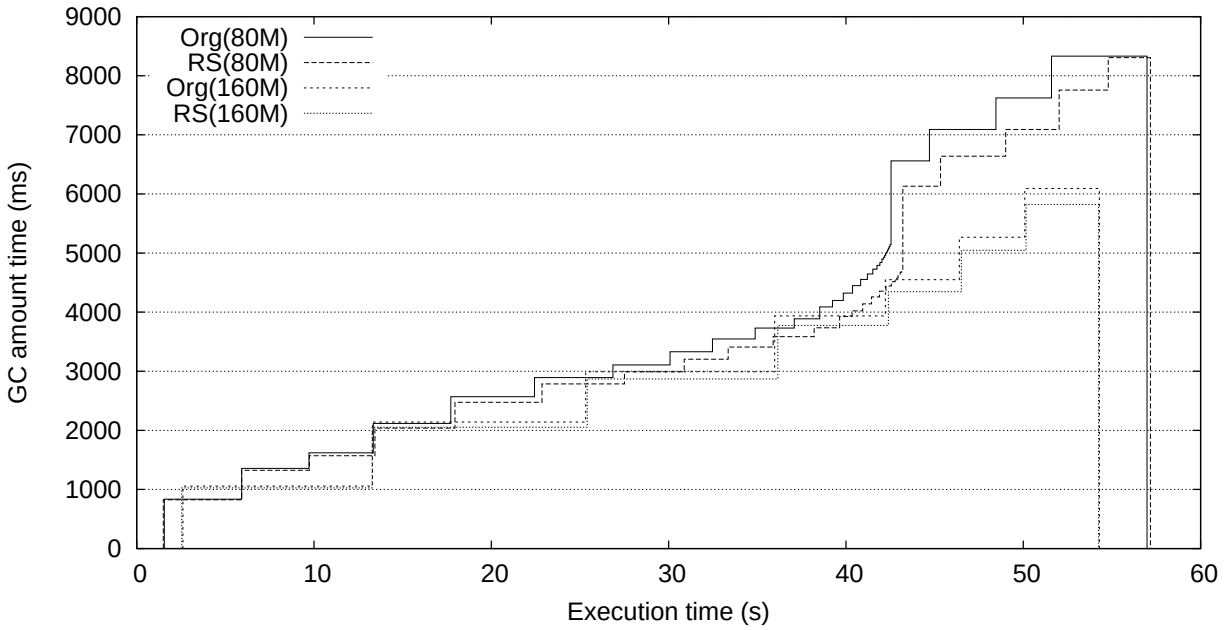


**Figure 9.** Bloat GenMS performance can be changed because of minor code and heap usage differences

#### 5.2.3 GC algorithm

A final consideration is with respect to the overall GC algorithm performed; specific benchmarks respond differently to different GC strategies. 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 cases, such as *db* and *pmd*, SemiSpace is actually nearly as fast or even faster than GenMS in normal heap settings. Since the performance of GC is strongly correlated with heap size, we examined how much GenMS can win over SS at different heap sizes. Figure 10 shows results for SPECjvm98 and Figure 11 shows the DaCapo benchmarks. The $y$-axis expresses the GC time of SemiSpace normalized to that of GenMS, and 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.

#### 5.2.4 Hardware related benchmark characteristics

Not all benchmark characteristics of interest are most easily seen as high level considerations, and so we also use an instrumented Jikes RVM to study benchmark behaviour through a variety of hardware events. Here we briefly discuss results on L1 instruction and data cache misses for some sample benchmarks, *compress, db* and *jack*. The corresponding cache miss data is shown in Figures 12, 13 and 14 respectively, and represent data gathered at each thread context switch.

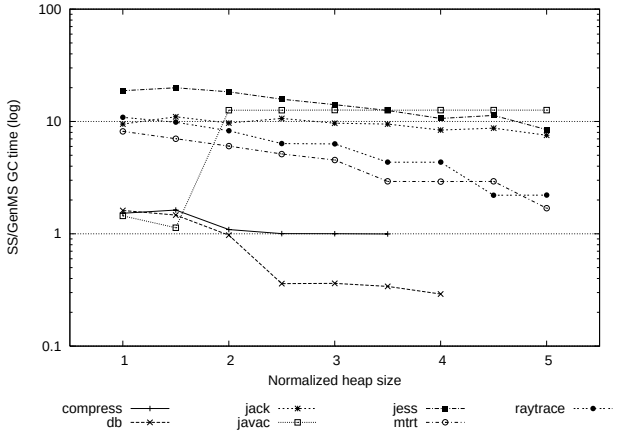


**Figure 10.** Performance comparison between SS and GenMS on SPECjvm98 benchmarks, the minimal heap size is 32M

All these benchmarks show recurrent patterns, particularly in the instruction cache miss rate. This corresponds to the various execution phases of these benchmarks. More interesting is the proportion of cache misses attributed to instruction or to data. In *compress* data cache misses dominate, whereas in *jack* instruction cache misses dominate; *db* lies between, with both kinds of misses equally important. Relative dominance of cache misses should correlate with the general sensitivity of benchmarks to instruction versus data cache effects; e.g., a benchmark with a dominant and tightly recurrent pattern of instruction cache misses likely contains a small but very "hot" section of code, and could be strongly affected by small changes in code positioning.

Figure 15 extends the idea of a cache sensitivity "bias" (I-Cache versus D-Cache) to all our benchmarks. In this graph a benchmark's position is determined by the I-Cache (x-axis) and D-Cache (y-axis) miss density. Benchmark *compress*, for instance, is quite biased toward the data cache, while a large number of

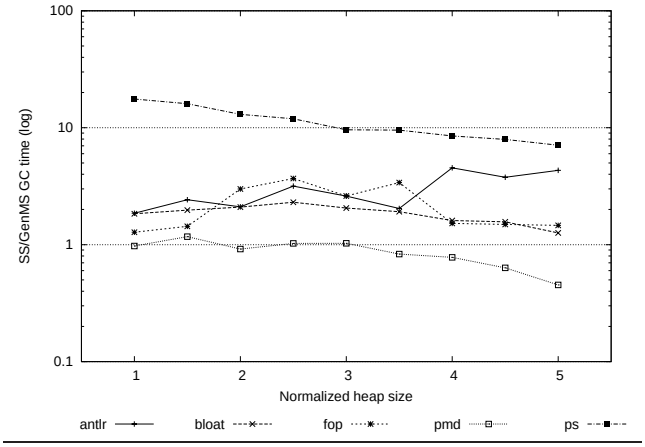


**Figure 11.** Performance comparison between SS and GenMS on DaCapo benchmarks, the minimal heap size is 40M

benchmarks, such as *antlr*, are highly biased toward the instruction cache. The performance of *db*, *mtrt* and *bloat* have similar relative dependencies on these two caches.

The rectangular area for each benchmark data point functions as error-bars, showing the size of one standard deviation in the variations between consecutive measurements. A box elongated in one direction represents a benchmark that has a larger variation in the corresponding hardware event, and thus a larger potential for variation due to optimizations; e.g., in *compress* data cache performance varies much more than I-Cache. The arrows associated with each point show the average of the top 10% largest cache miss variations between two consecutive sample points. A very long arrow thus means that the largest performance variation is very different from the more typical case, whereas a small arrow indicates a more uniform and stable result. The length of this arrow is thus a rough indicator of the validity of the measurement for detecting program phase transition points: a measurement that varies little will not be a good indicator of program behaviour changes.

The results in these figures are heuristic indicators only, but show that individual benchmarks may have very different properties with respect to how they respond to a particular optimization, even at a very low-level: these effects are not obviously and trivially amortized away by a long or complex execution. An optimization may be viewed quite differently given a set of benchmarks that are primarily I-Cache (resp. D-Cache) driven, and this can easily result in a spurious overall evaluation of the optimization effect.

#### 5.2.5 Summary

The above investigations and coarse taxonomy provides a number of insights into the sources of different influences on program and optimization performance. We have attempted to be exhaustive with respect to influences related to our specific optimization case study, while demonstrating both general principles and a typical, relative weighting of factors. From the analyses in this section we can summarize that:

- The performance of Java virtual machines can be significantly affected by unintended code motion side-effects. Instruction cache effects are not typically deeply considered in modern, high level optimization studies, but even in cases where an optimization does not intentionally alter I-Cache behaviour minor code position changes can induce a misleading understanding of the optimization effect.
- Our main approach has been to isolate and measure different influences. The real performance of GC improvements, however, is difficult to measure in hybrid systems like Jikes RVM, where internal VM-specific data is stored in the heap, easily perturbing results.
- The relation between kinds of benchmarks and design choices can be a complex source of variance, and cannot always be ignored as an amortized, unimportant cost. The reference composition of the objects, for instance, is an important factor in determining the suitability of our RS scanning strategy.
- Major VM components optimized for general cases do not give a consistent improvement across all benchmarks. Generational versus semi-space GC, for instance behaves differently depending on the specific benchmark and workload size. This situation exhibits some potential for adaptively setting the nursery size to improve performance.
- Benchmarks show a wide variation in sensitivity to code versus data cache effects. Which factor dominates for a given benchmark depends strongly on the benchmark itself. This highlights the impact of low level system design on high level performance, as well as the need to apply quantitative methods for ensuring benchmark representability.

## 6. Conclusions and future work

Optimizations in a modern virtual machine environment clearly have the potential for complex interactions with various system aspects, high and low level. Our GC optimization case study shows the relative impact of many confounding factors, and we have given a initial, high level taxonomy for the different sources of performance variation.

Within our categorization cache effects dominate, and are a well known source of variance. Their large impact and indirect causality is, however, much less known. Other effects have less overall influence, although changes in GC points and the sensitivity of particular benchmarks to specific system or optimization designs can also have a significant impact. Our study provides a guideline for the relative weight of these issues, and an example of the widely varying factors that must be considered. Different situations, benchmarks, systems, and optimizations will of course vary in the source and actual magnitude of performance change; nevertheless, the same basic concerns will apply to most other performance optimization analyses, certainly any that affect the placement of code or data, or alter the parameters or timing of GC. Unless these factors are appropriately investigated and controlled for, conservatively, real-time performance changes of up to 10% may be attributable to external factors.

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. Code layout optimizations that better exploit the instruction cache are also clearly necessary; the considerable performance variation we find just by moving code in a simplistic and arbitrary manner suggests significant performance improvements are still possible in the final stages of code generation and linking.

## Acknowledgments

This research has been supported by Le Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT) and the Natural Sciences and Engineering Research Council of Canada (NSERC). We also would like to thank the anonymous reviewers for their insightful comments.

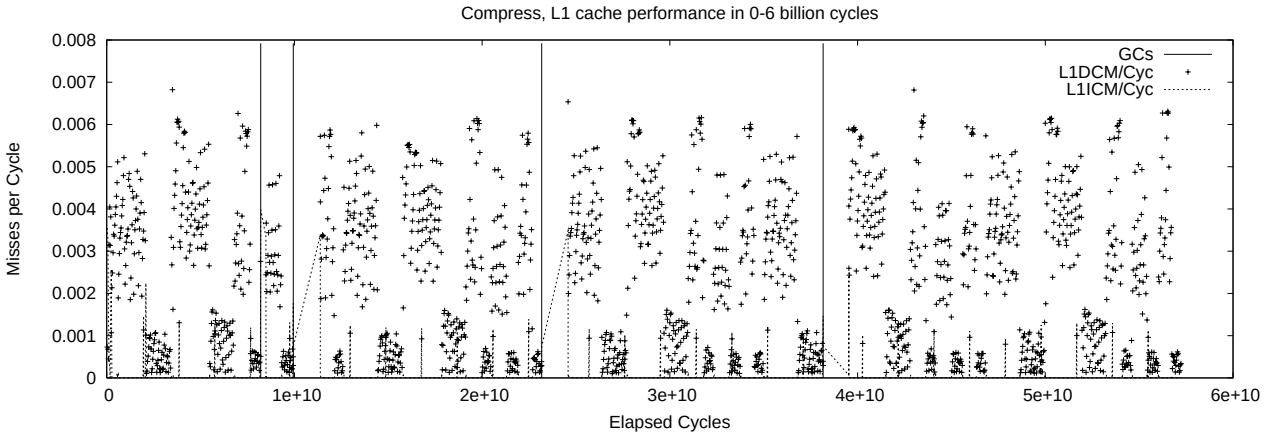


**Figure 12.** Compress hardware event trace

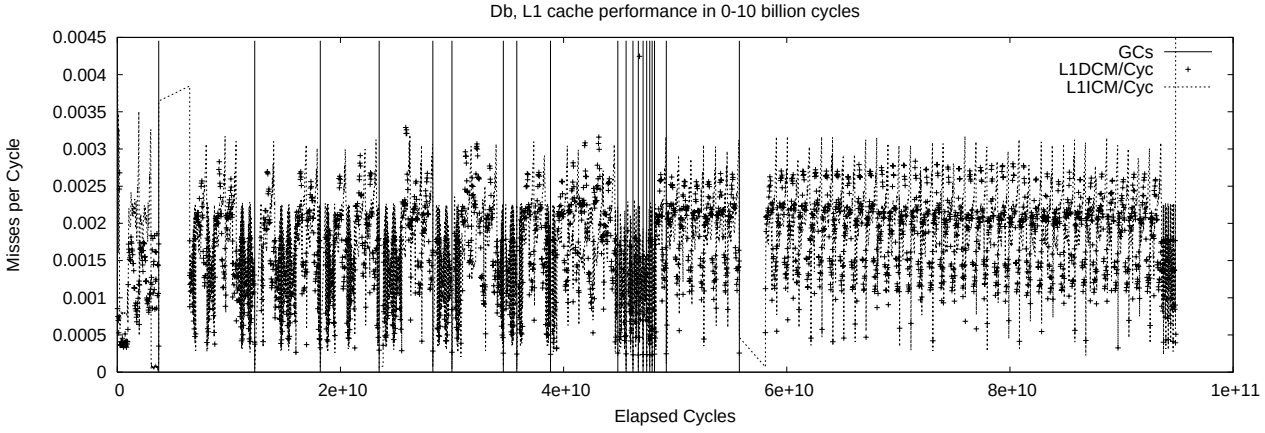


**Figure 13.** Db hardware event trace

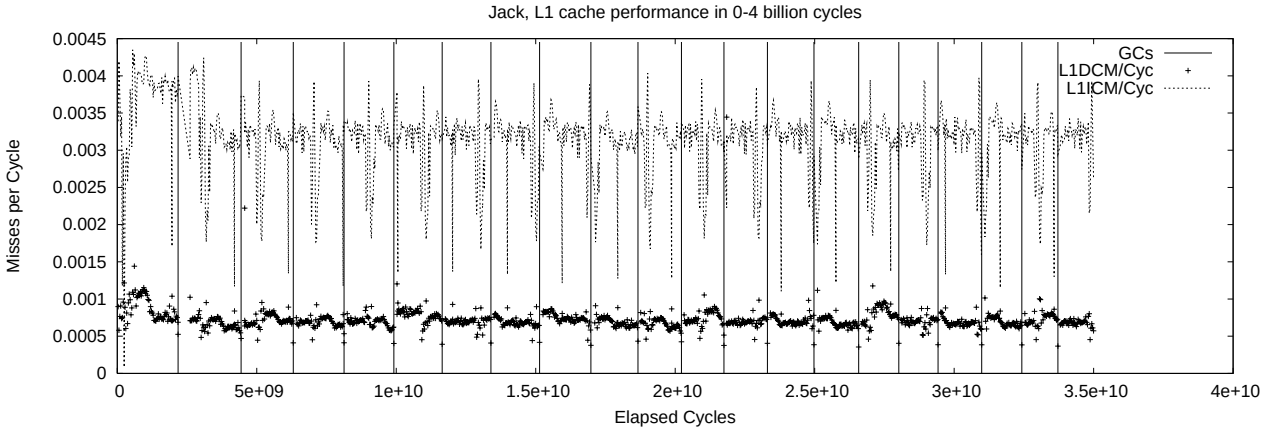


**Figure 14.** Jack hardware event trace

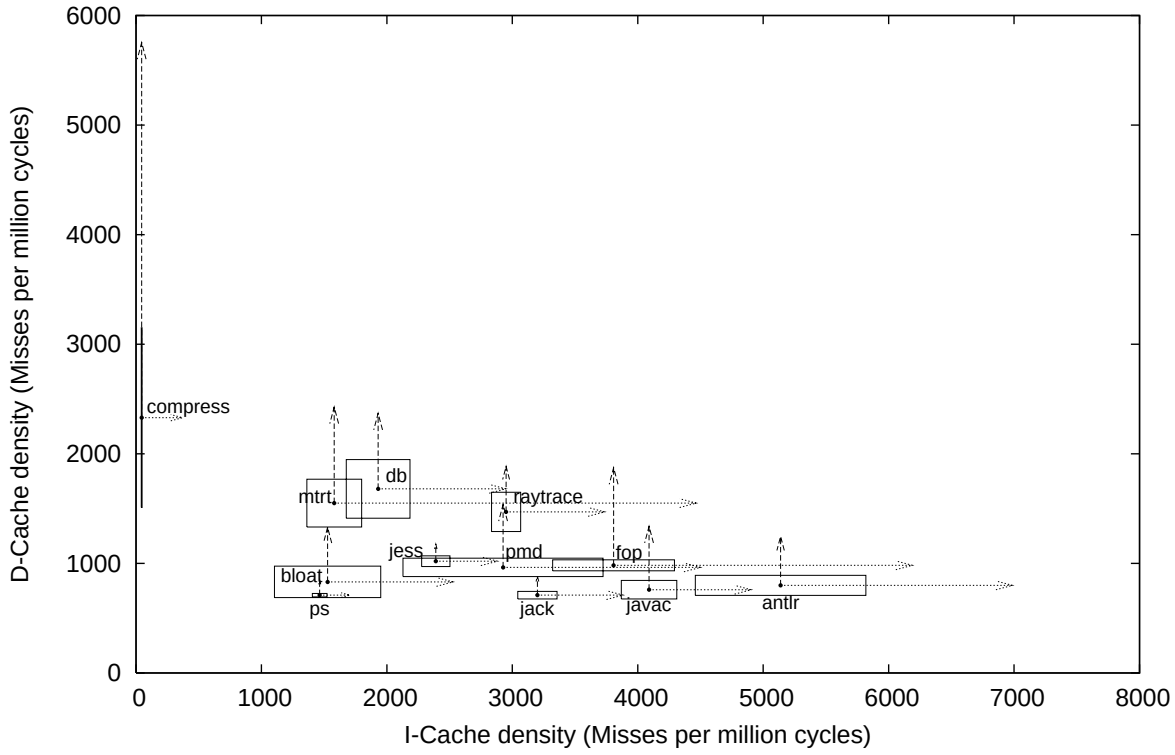


**Figure 15.** Benchmark cache bias

## 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. ISBN 1-58113-238-7.

[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. ISSN 0734-2071.

[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, R. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: getting around garbage collection gridlock. *SIGPLAN Not.*, 37(5):153–164, June 2002. ISSN 0362-1340.

[6] 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. ISBN 1-58113-263-8.

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

[8] 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. ISBN 1-58113-804-0.

[9] 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. ISBN 0-7695-2021-9.

[10] 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. ISBN 1-58113-712-5.

[11] 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. ISBN 1-58113-712-5.

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

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

[14] 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.

[15] D. Gu, C. Verbrugge, and E. Gagnon. Assessing the impact of optimization in Java virtual machines. Technical Report SABLE-TR-2005-4, Sable Research Group, McGill University, Oct. 2005.

[16] 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. ISBN 2-912590-31-0.

[17] 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. ISBN 1-58113-831-9.

[18] 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.

[19] 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. ISBN 1-58113-831-9.

[20] 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. ISBN 1-58113-831-9.

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

[22] 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.

[23] D. Project. The DaCapo benchmark suite (beta050224). `http://www-ali.cs.umass.edu/DaCapo/index.html`, 2003.

[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. ISBN 1-58113-539-4.

[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. ISBN 1-58113-804-0.

[26] F. Schneider and T. R. Gross. Using platform-specific performance counters for dynamic compilation. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC'05)*, oct 2005. to appear.

[27] 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. ISBN 1-58113-945-4.

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

[29] 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.

[30] 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. ISBN 1-58113-238-7.

[31] 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.

[32] 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. ISBN 0-89791-131-8.

[33] 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.