CODE LAYOUT AS A SOURCE OF NOISE IN JVM PERFORMANCE

DAYONG GU AND CLARK VERBRUGGE AND ETIENNE GAGNON

Abstract. We describe the effect of a particular form of "noise" in benchmarking. We investigate the source of anomalous measurement data in a series of optimization strategies that attempt to improve runtime performance in the garbage collector of a Java virtual machine. The results of our experiments can be explained in terms of the difference in code layout, and hence instruction and data cache behaviour. We show that unintended changes in code layout due to code modifications as trivial as symbol renaming can contribute up to 2.7% of measured machine cycle cost, 20% in data cache misses, and 37% in instruction cache misses.

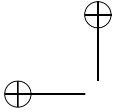
Keywords: java virtual machine, performance counter, benchmarking, middleware design, object layout, code layout, garbage collection

1. Introduction

Although large performance improvements in compiler and runtime optimization are highly desirable, many optimizations will produce only small effects for a large selection of benchmarks. When assessing optimizations in such a context the impact of external factors is thus important to minimize, if the optimization changes are to be properly identified through the "noise."

In this paper we describe the effect of a particular form of "noise" in benchmarking. We investigate the source of anomalous measurement data in a series of optimization strategies that attempt to improve runtime performance in the garbage collector of the SableVM Java virtual machine [Gag02a]. Through a number of validating experiments the counter-intuitive results of our experiments can be explained in terms of the difference in code layout, and hence instruction and data cache behaviour. We show that unintended changes in code layout due to code modifications as trivial as symbol or file renaming can contribute up to 2.7% of measured machine cycle cost, up to 20% in data cache misses, and up to 37% in instruction cache misses.

Date: 11/30/2004.



1.1. Contributions

Our main contribution is an experimental analysis and investigation of the impact of unexpected changes in code layout. We demonstrate this through analysis of several novel variations in object layouts and garbage collection algorithms. Our data is gathered in the context of an actual, realistic Java virtual machine, SableVM. Contributions thus include:

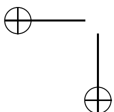
- We designed and analyzed a set of algorithmic changes and variations in object layouts for a Java virtual machine, and tested them in a non-trivial, realistic setting.
- We experimentally demonstrate that a significant source of noise in benchmark measurement is due to code layout.
- We have measured actual performance changes due to minor code layout differences, and thus provide guidance on the impact of this factor on benchmarks, and hence optimization assessment.
- Our work suggests that small, but measurable and significant improvements (or reductions) can be achieved by straightforward padding of symbols. A simple experimental validation can then indicate a locally optimal code shifting.

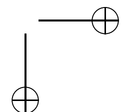
2. Related work

Our investigation into object layout changes was part of an effort to improve garbage collector performance in Java. Garbage collection techniques are well established, and an excellent reference on appropriate algorithms is Jones and Lin's well-known book [JL96]. Optimization approaches to GC are of course many; in [SHB$^+$02, SMM99, BJMM02], the researchers treat objects differently based on the *age* of the object, and thus reduce the overall execution time of GC. Other researchers try to decrease the *pause time* in the GC cycle [PD00]. Most of the latter is based on the basic idea of splitting the workload of GC into multiple concurrent processes, an idea which first appeared in Boehm's 1991 paper [BDS91]. Similar ideas are also used in implementing GC in multiprocessor environments [FDSZ01, OBYG$^+$02]. Our work includes a studying of the interaction between GC performance and the data cache behavior, as well as hardware in general, which is discussed firstly by Boehm [Boe00].

Hardware components such as data caches, instruction caches, branch prediction, etc, can have a significant effect on program performance, and are known optimization (and measurement) targets.

In [WL91] a large class of loop transformations is provided to improve data locality and reduce data cache misses. Unfortunately, it is difficult to get an



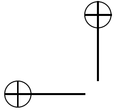


optimal arrangement of data. Another way to improve cache performance is by using *prefetching* techniques to eliminate the miss penalty. Lots of work has been done to improve prefetch effects and reduce the overhead in different situations [CSCT02, WH04]. That instruction location can cause a significant effect on hardware performance, especially instruction cache, has been noticed by researchers in various contexts. In [QHV02], Feng et al. describe one outlying experimental result where removal of unnecessary code slowed down execution, and attributed the anomaly to the instruction cache. Optimizations that focus on instruction cache behaviour are of course known; eg, in [PH90] it is suggested to align code of routines to the boundaries of cache blocks.

Hardware simulators or analysis frameworks, such as ATOM [SE94], SimPoint [SE02] and Simic [MCE+02], are popular tools for investigating performance at a low level. We make use of *performance counters* to monitor hardware performance. Most modern processors provide a *performance monitoring unit* (PMU), with which hardware related events can be measured. In general, the PMU is composed of several counter registers (eg, two for Pentium series, four for AMD Athlon), and a set of control registers to set the counted events, start/stop the counting, and collect the result in the counter registers. DCPI [ABD+03] is a software system that uses the hardware counters in the Alpha processors. IBM provides a library *pmapi* as an extension of the AIX kernel to access counters. Sweeney et al. use this library to develop a framework which can be used to explain the behavior of Java applications from the view of hardware events [SHC+03]. For Intel/AMD processors, PMC [Hel] and PCL [BZM] are libraries supporting hardware event counting. PCL also supports other platforms, including PowerPC, Alpha, R12000 and UltraSPARC I/II/III. Lieven et al [EGD03] measure the hardware events on Athlon Duron processors as part of an analysis of how Java programs can interact with Virtual Machines at the micro-architectural level. In Intel's 64 bit machine Itanium II [Int], the PMU is enhanced by adding many new features, such as an Event Address Register (EAR) and Branch Trace Buffer. Youngsoo Choi et al. [CKVW02] take advantages of these new features to make improvements in cache prefetching. We have used PCL for our experimentation, largely for its portability.

3. Object Layouts and Performance

The original motivation for this work was to investigate algorithms and object layouts that will improve performance of the copying garbage collector in SableVM [Gag02c]. Since a copying collector moves the entire live heap each collection cycle, data cache performance is obviously critical. Below we



describe the basic SableVM object layout, how the layout is used in garbage collection, and the potential optimizations we investigated.

3.1. Traditional SableVM Object Layout

The current, or "traditional" object layout is quite straightforward. Each object has a header composed of a *lockword,* which include information that can be used to obtain offsets of each object field, the size of the object, and a pointer to its virtual function table. After the header, we put the fields defined in each class, in order of inheritance from superclass to subclass. Within the fields defined in one class, we group the reference fields together and put them before the non-reference fields. Figure 1 shows the traditional layout in detail.

The GC algorithm for traditional layout is also simple. An initial *root set* of known live objects is copied from the existing heap into the new heap, and the reference fields of each live object are then scanned. The object header is used to find the offset of each reference field. Live references are copied to the end of the new heap, and thus themselves are eventually scanned. We refer the GC algorithm of Traditional layout as *Tr* from now on.

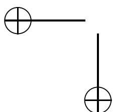
3.2. Alternative Object Layouts

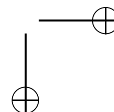
A number of variations are possible in the SableVM object layout. As potential improved designs for GC, we implemented two other layouts, a "bi-directional layout" and a hybrid "mixed layout".

Figure 2 shows the bi-directional layout. This approach groups all the references in a negative direction from the header and non-reference fields in the positive direction. Since all the references are grouped, during reference field scanning, we can just scan the references one by one copying targets until we meet the header of the object (identified by a special bit pattern). Once the header is reached, we can calculate the size of the object from the information saved in the header, and skip to the next one. This is the original algorithm for bi-directional layout, which we refer as *Bi*.

One problem with *Bi* is that we need to compare each value we load during the scan and verify whether it is a pointer (0 in the lowest bit) or a header (1 in the lowest bit). In order to eliminate these checks and save work further, we designed a new algorithm for the same layout, the Backward Pointer algorithm (*BP*).

The key idea of *BP* is to reuse the space in the copy of the current object in the old heap. Full details are beyond the scope of this paper, but this approach provides a number of improvements over *Bi*, at the cost of extra instructions.





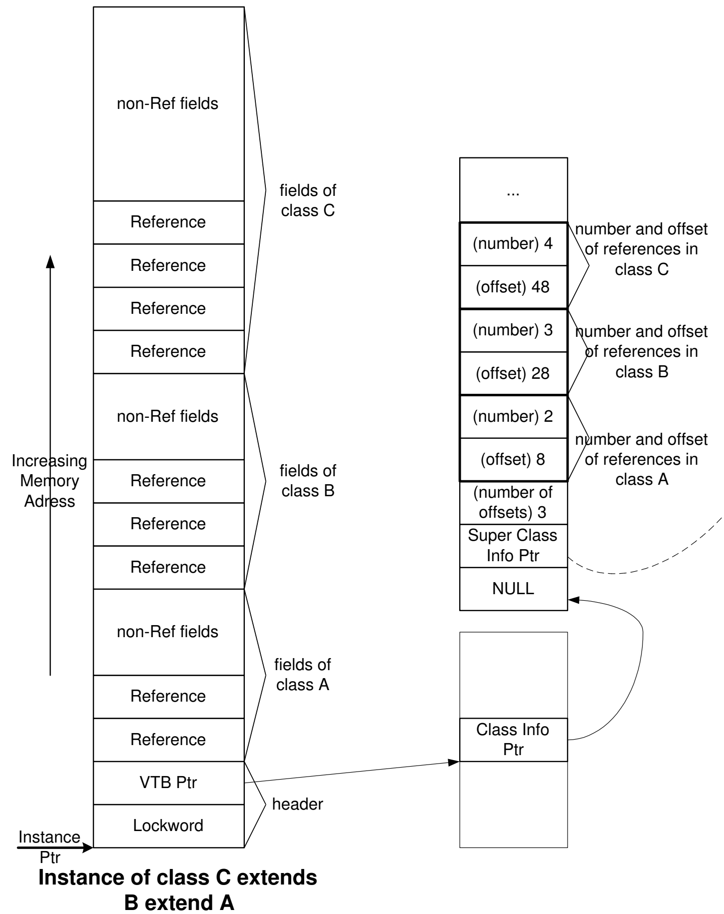


Figure 1: Traditional Object Layout

The third, "mixed" object layout, as its name suggests, is a hybrid of the first two layouts. Mixed layout is almost the same as bi-directional layout, except for reference arrays. For reference arrays mixed layout puts the header before the array items, as in traditional layout. The basic scan algorithm of mixed layout (*Mixed*) is thus the same as *Bi* except when scanning a reference array. For arrays this saves some pointer checks, and so should be an improvement over *Bi* .

We also developed a further variation on the mixed layout, *mixedRS* (or *RS*). This version improves on *Mixed* by further focusing on sections of grouped

references as the source of GC work rather than objects *per se*. Pointers to reference sections to be copied in the old heap are stored at the end of the new heap, and these sections are used as a work-list for the scanning procedure. *RS* has the advantage that it can skip scanning objects without references. It may, however, have different cache behaviour due to its use of the end of the new heap as a working area.

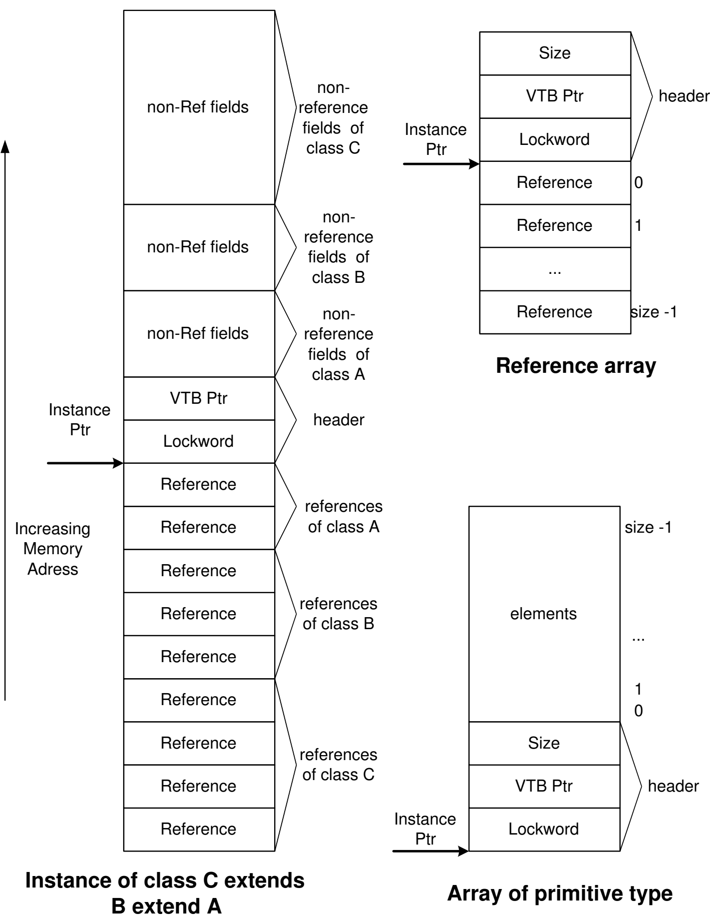


Figure 2: Bi-directional Object Layout

"main"
2004/12/1
page 7

3.3. Performance Analysis

We analyzed the relative performance of the proposed object layout and garbage collection schemes on the SpecJVM98 benchmark suite [SPE98], and one large internal benchmark, `sablecc` (a Java parser generator) [Gag02b]. Since GC accounts for a relatively minor portion of execution time in most of the benchmarks, we separated GC data from data collected from the rest of the program (the mutator) in order to more accurately see the effects. We examined the benchmark behavior under an instrumented SableVM using hardware counter data collected using the Performance Counter Library PCL [BZM]. This includes CPU cycle counts, instruction and data cache misses, instruction counts, and a number of other hardware events. Note that here we present only CPU cycle counts and cache miss data; other data events were examined, but were not more illustrative and so are not presented here for space reasons.

Figure 3 shows the relative change in CPU cycle counts for the GC component of the execution of the benchmarks as compared with the cycle counts of the original design, under our different layouts and strategies. Figure 4 shows the results for GC data cache misses. Data for the `mpegaudio` benchmark is not reported since it does not have a GC phase under our default configuration of heap size.

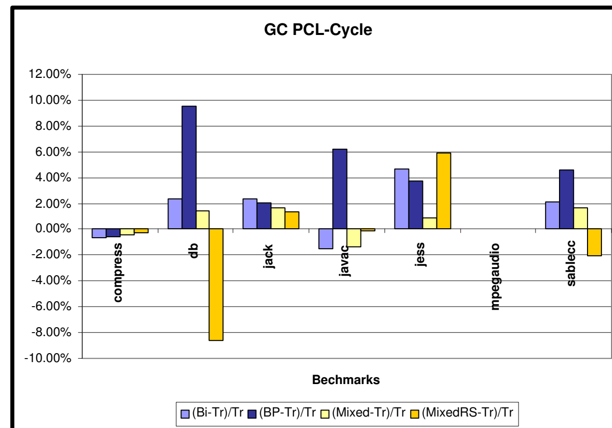


Figure 3: Cycle count changes in GC

Unfortunately, performance changes do not match expectations. In a general sense all the algorithms perform poorly; *RS* provides a marginal improvement on average, largely due to the relatively large improvement in `db`, but all others actually increase cycle counts.

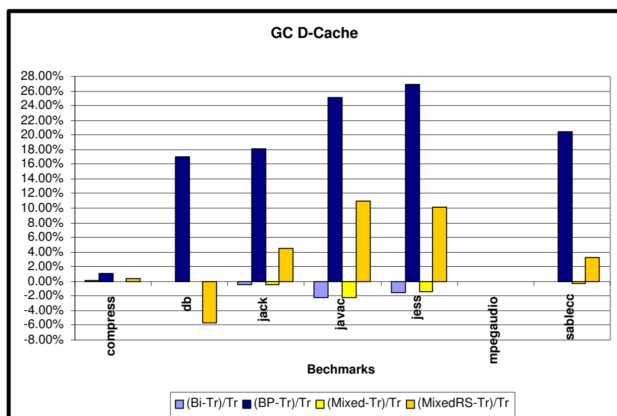


Figure 4: Data cache changes in GC

Because we use a copying GC, the number of data cache misses varies from 0.4% to 0.7% of the number of machine cycles. Considering the heavy cache miss penalty in advanced processors this is large enough to have an obvious impact on the performance. However, in these benchmarks, the data cache behaviour itself fails to fully explain the effects. In Figure 4, cache misses increase in general, but are not proportional, or even consistent with the relative changes in cycle counts for the same object layout strategy on the same benchmark. Other processor behaviour must necessarily be having a larger effect on overall performance.

Interestingly, as shown in Figure 5, the mutator's behaviour under all the layouts shows some small increases in cycles, but overall represents a reduction. Although changing object layouts necessarily perturbs runtime cache behaviour, the effect on the mutator should have been relatively neutral, and for several optimizations, object layout is at least initially identical. For `mpegaudio`, there is not even the effect of GC algorithms moving data to different places, but still measurable performance differences. This further suggests the effect of external influences.

Note that our measurements are both accurate and repeatable. In Figure 6 we show standard deviation divided by average for the measurements we use in this paper, collected using the original SableVM object and GC design (this accuracy is in fact consistent across all our implementations). Even the most varied measurement, cycles, differ by no more than 0.08%. This is an unrepresentable difference in most of our charts.

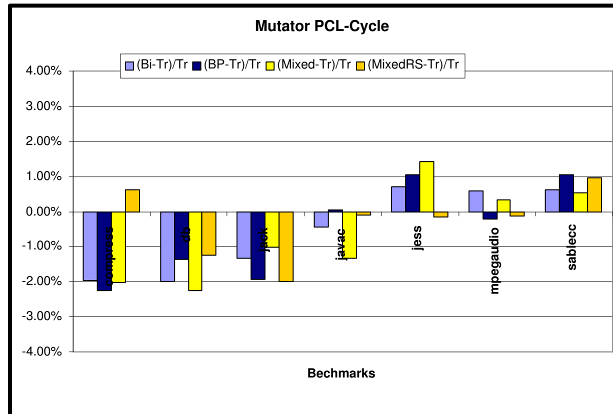


Figure 5: PCL-Cycle result of Mutator

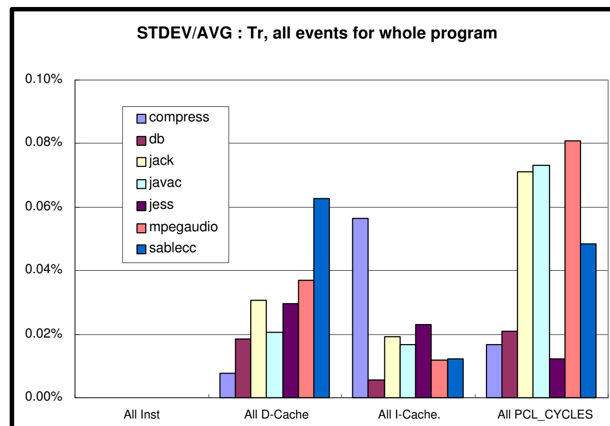


Figure 6: Standard deviation is very small

We thus designed a series of experiments to investigate the reason for the performance difference in the mutator, and to progressively eliminate sources of "noise." These are described in the next section.

4. Eliminating Noise

Since object layout changes have an obvious potential impact on heap usage and thus performance we first conducted a series of experiments to try and determine if the performance changes, particularly in the mutator, were related to the specific differences in heap usage. The main mechanism by which heap changes can affect the mutator is through changing the location of objects in the heap, or their associated field data. Differences in performance can thus arise from the following specific behaviours:

1. Hash code usage differences may exist. In SableVM, hash codes for objects are assigned lazily, upon first call to `Object.hashcode()`, and are built from the object's current location in the heap. Computed hash codes are then preserved during the copying phase of collection. Thus the hash code value may be different under our various strategies if the object header is not located in the same memory location when the hash code is first computed.
2. Cache effects due to field access patterns may change, since some of our optimization approaches place object fields in different relative locations to the object header.
3. The heap layout itself of course may have a large impact. After GC, the order of objects in the heap may be different due to the different scanning orders that correspond to how object fields are laid out. The traditional object layout and GC algorithm in SableVM corresponds to scanning object fields looking at (and hence copying) super class fields first, and subclass fields in hierarchical order. Other strategies we used such as bi-directional result in fields being examined in an inverse order to the inheritance hierarchy, inspecting fields and copying objects from subclasses up to superclasses.

Table 1 summarizes various permutations on our optimization design that we used to investigate the aforementioned problems. The second and third columns of the table give the object layout style and GC algorithm, as discussed in Section 3.2. The fourth column indicates how hash codes were computed, either from the location of the beginning of the object (which can vary according to how the object fields are organized), or from the location of the lockword, a uniform location in the object header that does not vary by layout. The last column indicates the scan order of fields, whether we look at reference fields from subclass fields to superclass fields or *vice versa*.

Hash codes turn out to be used quite rarely in most of our benchmarks. By tracing through virtual machine activity we found that most benchmarks only ever query the hash code of just one object. The `javac` benchmark is the only

"main"
2004/12/1
page 11

Name	Object Layout	Algorithm	Hash code	Heap Order
Tr	Traditional	Traditional(Tr)	beginning	super to sub
Tro	Traditional	Traditional(Tr)	beginning	sub to super
Bi	Bi-Directional	Bi-Directional(Bi)	lockword	sub to super
BiSHC	Bi-Directional	Bi-Directional(Bi)	beginning	sub to super
BP	Bi-Directional	Backward pointer(BP)	lockword	sub to super
BPR	Bi-Directional	Backward pointer(BP)	lockword	super to sub
Mixed	Mixed Layout	Hybird of Bi and Tr	lockword	sub to super
MixedSHC	Mixed Layout	Hybird of Bi and Tr	beginning	sub to super
RS	Mixed Layout	Reference Section(RS)	lockword	sub to super
RSR	Mixed Layout	Reference Section(RS)	lockword	super to sub

Table 1: Different optimization variations in SableVM

significant exception; though even for it, out of 4,786,236 runtime objects only 11039 had their hash code computed (under 0.3%). More importantly, using variations on our approaches that preserve the same hash code as traditional do not eliminate the apparent inconsistencies in data, nor does the extent of variation in hash code usage or calculation correspond in any obvious way with the measured performance.

Changing the layout of object fields is of course part of our optimization strategy, and differences should be expected based on that. There should though be some amount of correlation between approaches with identical object layouts. In Figure 7, however, it is apparent that even with identical layouts there is little connection between measured performance and strategies that employ the same layout. This is especially evident for `mpegaudio`, which does not have opportunity to produce different heap arrangements after garbage collection, yet still shows significant variation between strategies with the same layout.

The remaining factor is scan order. Once GC has occurred, objects that are scanned in a different order may be located at different locations in the new, copied heap. We show a comparison between our different algorithms when the scan order is reversed in Figure 8. Again, there is little correlation between this factor and performance; different algorithms do not consistently improve or degrade when scan order is switched.

Our measurements thus do not support that scan order, object layout, or hash codes have a direct or consistent effect on performance. The presence of external factors with greater, or at least complicating influence is therefore a strong possibility. A likely cause is suggested by the behaviour of `mpegaudio` under our different strategies. Since differences can be measured based on changes to garbage collection code despite the fact that it is not actually exercised at

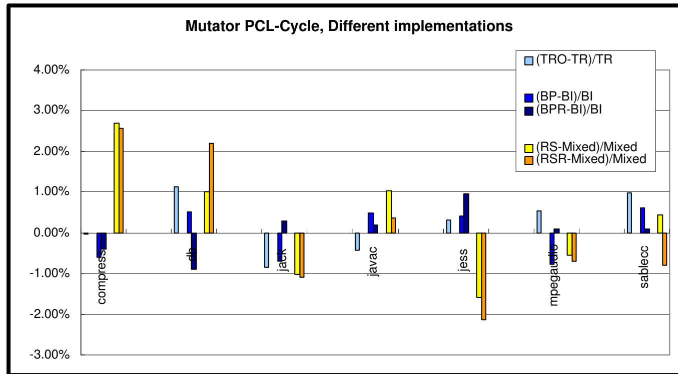


Figure 7: Performance differences among implementations for each object layout

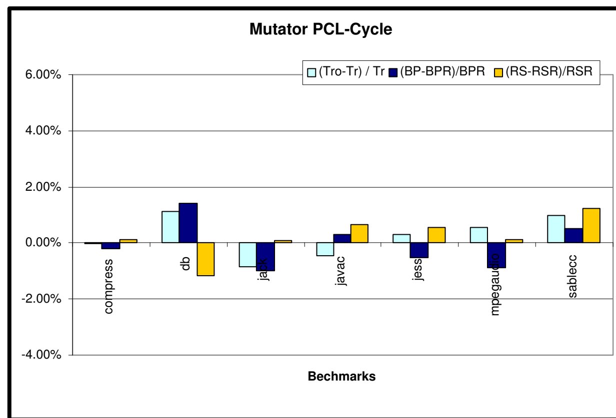


Figure 8: Comparison between heap layouts achieved by different scan orders

runtime, the mere existence of code changes must be the contributing factor. In the next section we explore the impact of code layout changes introduced by unused code changes on the instruction cache and performance.

5. Code Layout Effects

In order to determine the impact of (unused) code changes, we measured the instruction cache behaviour of our implementations. Figure 9 presents mutator

instruction cache misses in the same way as Figure 8 presents cycles. Unfortunately, again, the correlation with performance is weak. While instruction cache misses for db are dramatically higher for the *BP* strategy, and this corresponds nicely to db's relative performance under *BP*, other benchmarks and other strategies do not show good correlations in general.

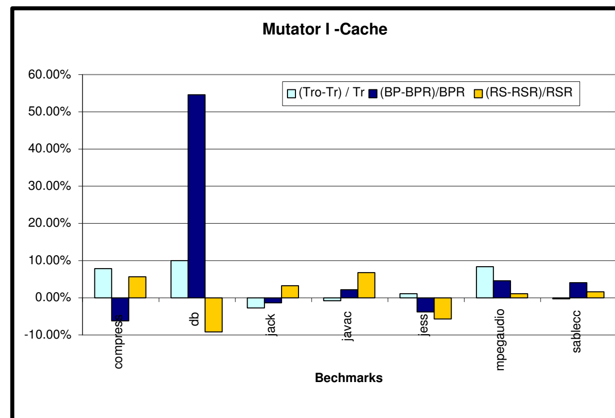


Figure 9: I-Cache performance of the mutator for different scan orders.

The reversed scanning order variations on our benchmarks are only slightly different from the original designs in terms of code. Using *RS* and *RSR* as a test case, we thus developed a series of implementations that progressively includes code making the transition from the *RS* to the *RSR* design. These new implementations are described in Table 2; note that although the code is incomplete in *RSR1* through *RSR4* all implementations are able to run since the code is never executed at runtime.

Version	Description
RSR0	the same code as RS
RSR1	switch two lines, perserve the code size
RSR2	change the code for scanning reference fields
RSR3	the same as RSR2, but add local variables
RSR4	the same as RSR3, initialize the newly added local variables
RSR5	the same code as RSR

Table 2: From RS to RSR, step by step

Figure 10 shows the corresponding experiment results in relation to the traditional performance. *RSR0* and *RSR1* are similar, as would be expected; neither code size nor executed code changed. The introduction of new code in *RSR2*, however, causes a significant decrease in instruction cache misses. Further code changes to add local variables and initialize them have no apparent effect. This is unsurprising; since this code is still incomplete, there are no uses for the introduced variables, and standard code optimization techniques eliminate unused variables during compilation. Once the complete implementation is included in *RSR5* performance changes again, as code size and placement is altered by the code difference.

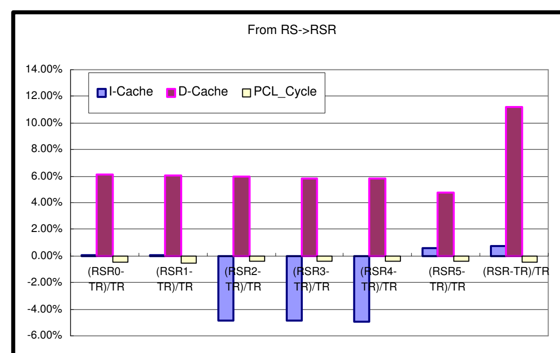


Figure 10: Performance of RS to RSR

The behaviour of the original *RSR* version is shown in the rightmost bars of the same graph. Interestingly, it is different from the data for *RSR5*, which contains exactly the same source code as *RSR* (executables were also stripped prior to testing to remove debug symbols). We therefore examined the differences in the binary program using the Linux `objectdump` utility. The results are presented in Table 3.

Code differences can show up in two main binaries constructed by the SableVM build; the executable and the sableVM library. We indicate whether code address changes were detected between the different versions in the two binaries in columns 2 and 3 of Table 3. The presence and degree of changes corresponds perfectly to the behaviour shown in Figure 10; even in the case of *RSR5* compared with *RSR* changes are detected. This change turns out to be caused by the path to the library being embedded in the executable, and hence shifting subsequent code.

The layout of methods is clearly an important reason for the performance differences. We provide further experiments that point to instruction cache

Version Pairs	SableVM executable	SableVM library
RSR0 Vs RSR1	the same	code shifts exist, but only for GC methods
RSR1 Vs RSR2	the same	code shifts exist for many methods, including non-GC methods
RSR2 Vs RSR3	the same	the same
RSR3 Vs RSR4	the same	the same
RSR4 Vs RSR5	the same	code shifts exist for many methods, including non-GC methods
RSR5 Vs RSR	code shifts	the same

Table 3: Method offset comparison between versions

behaviour, which is changed by the code layout, as a key factor in the next section.

5.1. Validating Icache Influence

In order to demonstrate that change of method layout is influencing performance through the instruction cache behaviour we examined the performance of a series of binaries. We measured 64 versions of SableVM built with progressively longer names, from 2 to 128 bytes (by 2s) longer than the original build name. This shifts the location of symbols in the executable by the same amount. The resulting effect on instruction cache, cycle count, and data cache are shown in Figures 12, 13, and 14 respectively. The graphs plot the differently named versions against absolute counts of hardware events.

The design of the Pentium III instruction cache is quite evident in these results. For instruction cache misses (Figure 12), and to a slightly lesser extent for CPU cycles (Figure 13), a cycle is apparent with a period of 64 in our naming scheme. Since each cache line is 32 bytes, and two cache lines are fetched at a time, a 64-byte cycle is quite consistent. The cycle is considerably less evident in the data cache results, but data cache misses in general are also less numerous and less expensive. A comparison is provided in Figure 11.

As previously seen in Figure 9, instruction cache misses do not correlate well with the performance of our different object layout strategies. They do, however, show a good correlation with code layout. These results are highly suggestive if not conclusive, and this confounding factor is a likely contributing cause to the original, counter-intuitive results of our experiments on object layouts.

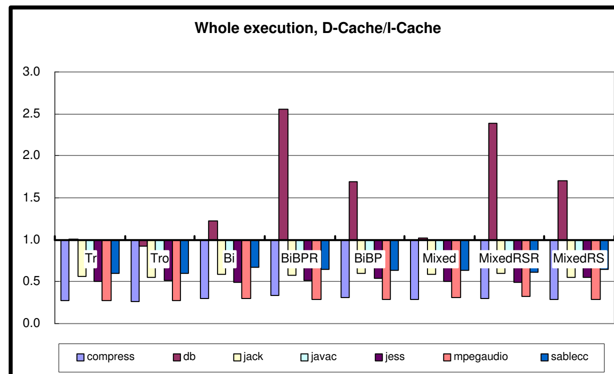


Figure 11: The number of instruction cache misses is larger in most cases

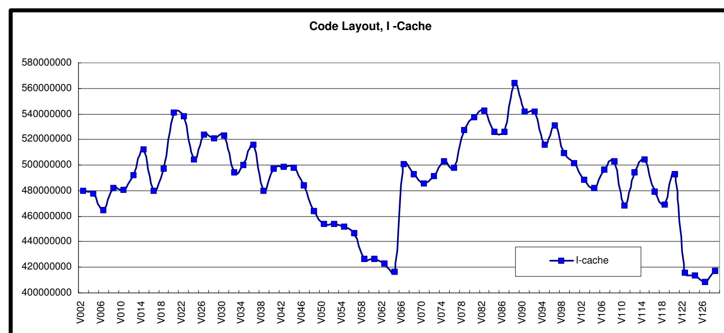


Figure 12: Effect of changing method layout on instruction cache misses

6. Conclusions and future work

Identifying external influences is critical to correctly assessing an optimization. We have described the observed impact of unintended code placement effects on industry standard benchmarks. These effects are non-trivial and can perturb intended measurements, and so are important to take into account in order to achieve accurate benchmarking.

Our examination is developed through a series of potential low-level optimizations, and their apparently anomalous results. We are currently exploring techniques to try and mitigate and/or exploit the impact of code layout on performance measurement. Careful naming strategies and coarse-grained

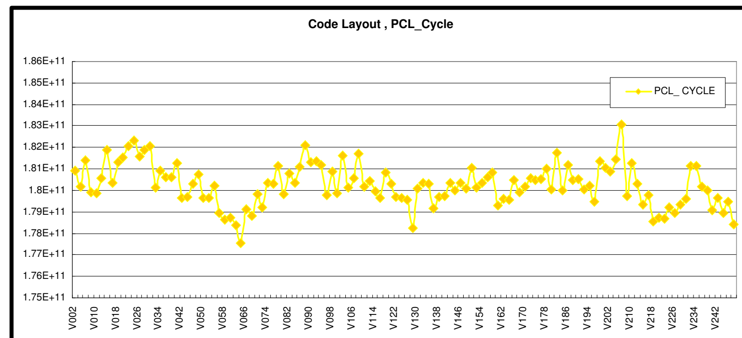


Figure 13: Effect of changing method layout on cycle counts

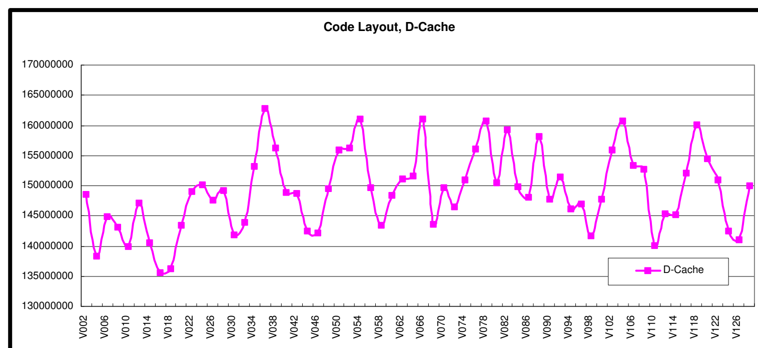


Figure 14: Effect of changing method layout on data cache misses

code alignment techniques may help provide consistent, comparable data that will generically apply to machines with different cache models. Compiler and link-time optimization techniques that exploit code alignment, such as the well known strategy of aligning methods and loops on cache boundaries may also help in achieving a more consistent result. We are actively developing a framework for investigating these issues.

7. Acknowledgments

This work has been supported by Le Fonds Québécois de la Recherche sur la Nature et les Technologies and the National Sciences and Engineering Research Council of Canada.

References

[ABD+03] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, R. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Cache memory behavior of advanced pde solvers. In *Processing of Parallel Computing 2003 (ParCo2003)*, Dresden, Germany, September 2003.

[BDS91] H-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, pages 157–164, 1991.

[BJMM02] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of PLDI'02 Programming Language Design and Implementation*. ACM Press, June 2002.

[Boe00] Hans-J. Boehm. Reducing garbage collector cache misses. In *ISMM*, pages 59–64, 2000.

[BZM] Rudolf Berrendorf, Heinz Ziegler, and Bernd Mohr. The performance counter library. http://www.fz-juelich.de/zam/PCL/.

[CKVW02] Youngsoo Choi, Allan Knies, Geetha Vedaraman, and Jeremiah Williamson. Design and experience: Using the Intel Itanium2 processor performance monitoring unit to implement feedback optimizations. In *EPIC 2*, November 2002.

[CSCT02] Jamison Collins, Suleyman Sair, Brad Calder, and Dean M. Tullsen. Pointer cache assisted prefetching. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 62–73. IEEE Computer Society Press, 2002.

[EGD03] Lieven Eeckhout, Andy Georges, and Koen 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. ACM Press, 2003.

[FDSZ01] Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, 2001.

[Gag02a] Etienne Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, McGill University, 2002.

[Gag02b] Etienne Gagnon. Sablecc. URL:http://www.sablecc.org/, 2002.

[Gag02c] Etienne Gagnon. Sablevm. URL:http://www.sablevm.org/, 2002.

[Hel] Don Heller. Performance monitoring counter. http://www.scl.ameslab.gov/Projects/Rabbit/.

[Int] Intel. *Intel Itanium2 Processor Reference Manual*. Intel Corp.

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

[MCE+02] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. In *IEEE Computer, 35(2)*, pages 50–58, February 2002.

[OBYG+02] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. In *Proceedings of the ACM SIGPLAN: Conference on Programming language design and implementation*, pages 129–140, 2002.

[PD00] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In *ISMM*, pages 143–154, 2000.

[PH90] K. Pettis and R. C. Hansen. Profile-guided code positioning. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[QHV02] Feng Qian, Laurie Hendren, and Clark Verbrugge. A comprehensive approach to array bounds check elimination for Java. In *International Conference on Compiler Construction*, April 2002.

[SE94] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[SE02] A. Srivastava and A. Eustace. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[SHB+02] Darko Stefanović, Matthew Hertz, Stephen M. Blackburn, Kathryn S. McKinley, and J. Eliot B. Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *Workshop on Memory System Performance (MSP)*, 2002.

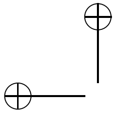
[SHC+03] Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove1, and Michael Hind. Using hardware performance monitors to understand the behavior of Java applications. In *VM'04*, 2003.

[SMM99] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In *Proceedings of the Conference on Object-Oriented Programing, systems, languages, and applications*, pages 370–381, 1999.

[SPE98] SPEC. SPEC JVM98 benchmarks. URL:http://www.spec.org/osg/jvm98, 1998.

[WH04] Dan Wallin and Erik Hagersten. Bundling: Reducing the overhead of multiprocessor prefetchers. URL:http://b6.hpsc.csiro.au/hpc-conferences/IPDPS2004/DATA/19_02_IPDPS.PDF, 2004.

[WL91] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, October 1991.



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

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