The Dissertation Committee for Xianglong Huang
certifies that this is the approved version of the following dissertation:

Improving Program Locality On-the-fly

Committee:

Kathryn S. McKinley, Supervisor

Stephen M Blackburn

Don Batory

Douglas C. Burger

Calvin Lin

Improving Program Locality On-the-fly

by

Xianglong Huang, B.S., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2006

Acknowledgments

I am fortunate to meet so many people who have been great help over the years of my graduate school. First I would like to thank my advisor, Kathryn McKinley. She has been an excellent advisor, mentor, guide and friend. I feel really lucky to be her student and to have the privilege of working with her for 6 years. I become a much more capable researcher under her guidance over these years.

I would like to thank my master's advisor, Steve Carr. He is such a wonderful person to work with. Especially during the time when I could not communicate fluently in English, he has been so patient and encouraging. He changed my view of research and because of him, I fell in love with research and decided to pursue my Ph.D. degree.

I can not overstate how important Steve Blackburn is to me on my research. In additional to providing high quality infrastructures and tools for my research, he also taught me a lot of the important principles and skills on programming. Without the help and knowledge I got from him, I will need another 6 years to finish all the work in my dissertation. He is my unofficial co-advisor.

My friends, especially my lunch bunch, Mike, Phoebe, Ben, Sam, Brendon, Emery, and Rich, they helped me over so many difficult times. Especially when I am down, having lunch is the highlight of my day and I enjoyed the

jokes, laughter, or just chatting. Sam, Brendon, and Emery also give me so many useful tips along the way of my Ph.D. study. I will miss my friends so much after I leave Austin.

My beautiful fiancee, Jingyuan (Ada), is my other strong support through the years. She has seen my darkest side, listened to most of my complaints when I feel stressed. She is always a good listener and supporter.

My parents and my sister are among the people who will be happy for me and they made me who I was before I came to the U.S. Without my family, I will never have the opportunity to come to the U.S. to study.

People in or outside my research group, Maria, Peter, Matt, Jenn, and others also give me so much help on my talks and on my written English. I really appreciate those help.

There are numerous people helped me and believed in me through the years. I can not enumerate them all, but I thank them and appreciate what I have got from them. I can not get here without you all helping me.

Improving Program Locality On-the-fly

Publication No. _____

Xianglong Huang, Ph.D.
The University of Texas at Austin, 2006

Supervisor: Kathryn S. McKinley

As increases in processor speed continue to outpace increases in cache and memory speed, programs are losing more performance to poor locality. Object-oriented languages exacerbate this problem by adopting new features such as just-in-time (JIT) compilation, dynamic class loading, and many small methods. However, they provide significant software engineering benefits and have become enormously popular. Solutions that bridge the memory gap will combine good performance with fast software development. We found although unique features of object-oriented languages, such as automatic memory management, dynamic compilation, and runtime monitoring systems, generate performance overhead, they also provide new opportunities for online optimizations which are not exploited by previous work. In this thesis, we take advantage of these opportunities with new approaches that improve data and instruction locality at runtime with low overhead.

To improve data locality, we first implement a new dynamic, low overhead, online class analysis to find data locality. Our algorithm detects hot

fields for hot objects and then reorders the objects according to their heat on-the-fly in a copying generational collector. The overall time variation between static orderings can be up to 25% and there is no consistent winner. In contrast, our new dynamic class reordering always matches or improves over the best static ordering since its history-based copying order tunes memory layout to program traversal.

To improve instruction locality, we develop two schemes for improving instruction locality in a Java Virtual Machine environment. We first describes a *partial* code reordering system, which reduces the program instruction working set and cache conflict misses with extremely low overhead. We then present a code management system that uses dynamic profiling to reorder *all* JIT-compiled code to improve instruction locality with novel efficient algorithms. Both systems show that the VM can dynamically improve instruction locality with little overhead.

These results indicate that the VM layer for modern languages are not just a cost-to-be-borne, but instead open up a new class of optimizations for monitoring and improving data and instruction locality, and code quality. Thus these results point to a future of programming languages that are robust, dependable, and high performance.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Due to wire scaling and clock rates, processors will soon only be able to access a fraction of the chip in one cycle. This trend will result in (1) partitioning of some components such as caches and register files, (2) longer latency to access the memory and more complex memory systems, and (3) multi-core architectures which will need to find and exploit more instruction level parallelism (ILP) for performance. The imbalance between memory and processor speeds is called the memory gap. The memory gap is already a serious performance bottleneck and could be getting worse because of the new hardware technologies. Software can help alleviate the memory gap by improving data and instruction locality and can consequently reduce long latency memory accesses. But the popular object-oriented programming languages, such as Java and C#, still lose significant performance due to poor locality. We measure the potential locality improvements for nine Java programs, and we find these programs spend on average 27% of their execution time waiting for cache misses from L1 and L2 in Figure 1.1 (see Section 4.3.2 for detailed analysis).

The goals of good software engineering and high performance are often at odds. Common wisdom holds that object-oriented languages offer software engineering benefits such as fewer errors and reduced development time, but

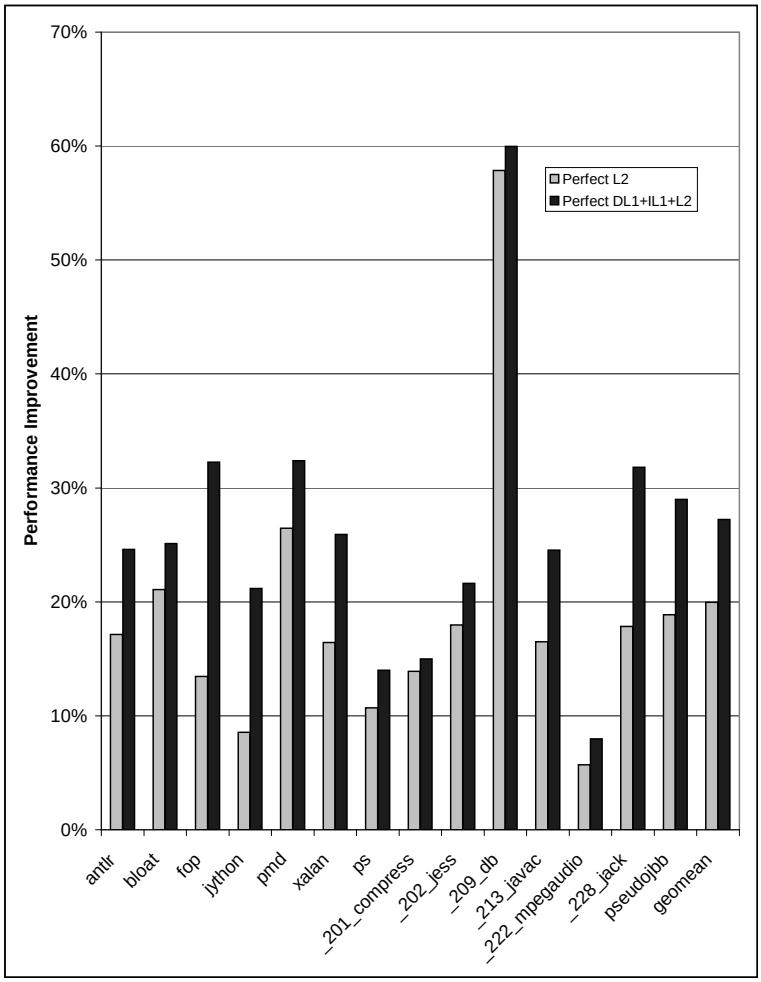


Figure 1.1: Java programs with Perfect Caches

these languages do so at a cost: New features such as garbage collection, JIT (just-in-time) compilation, and dynamic class binding add performance overhead to programs. Common wisdom also holds that traditional programming languages such as C offer performance benefits at a software engineering cost. With effort, programmers can free memory as soon as possible and use specialized allocators to gain memory efficiency and speed. For instruction accesses,

C has pre-compiled and pre-allocated instructions which eliminate the runtime overhead of dynamic code generation. However, C has a hidden performance cost. Because C cannot move objects and instructions without violating language semantics, it requires a non-moving allocator, such as one based on free lists. A free-list allocator places contemporaneously-allocated objects in whichever locations are free at that time, but these locations are not necessarily adjacent or even nearby in memory. Java can move objects and can thus use contiguous allocation to attain locality for contemporaneously-allocated objects and instructions, and Java can use copying collection to place objects and instructions with locality closer together, or even move objects and instructions. These dynamic optimization opportunities have not been exploited by previous research.

Our approach to improving program locality is to exploit the available opportunities in object-oriented program languages to generate powerful dynamic optimizations to improve program locality with low overhead. The programs thus can potentially achieve better data and instruction locality than programs written in traditional programming languages.

1.1 Improving Data Locality

There has been a lot of research on compile time optimizations for array accesses inside loops to improve data locality [17, 18, 33, 35, 43, 54]. Object-oriented languages usually use more pointer data structures than traditional programming languages and previous approaches only solve a small part of

the problems.

Copying generational collection is one of the best performing memory managers [11]. The allocator places contemporaneously allocated objects in contiguous memory. Generational copying collectors divide objects into newly allocated *nursery* objects and objects that have survived one or more collections [5, 51, 67]. Because most objects die young and the nursery is collected separately, generational copying collectors have better performance than non-generational collectors [5, 11]. Copying collectors promote reachable objects using transitive closure and place objects in breadth first [21] or depth first order [48, 69]. Prior research to improve data locality by changing copying orders has explored priori static orderings [48, 69], static class profiling [22, 61], and online object instance sampling [24]. Static orderings are problematic when program traversal patterns do not match the collector's single ordering. We show large differences ranging of up to 25% in total execution time due to the locality of static copy orders for some benchmarks. In a JIT (just-in-time) optimizing compiler for Java, dynamic class loading provides more non-determinism and therefore limits the generality of static profiling. Instance based reordering is potentially more powerful than the class based orderings we present, since objects with locality are not necessarily connected. However, the sampling space and time overheads for just the old objects are significant (6% in time for Cecil [24]) and miss the opportunity to improve locality when the collector promotes young (*nursery*) objects.

We introduce a novel low-cost dynamic class analysis that drives a *gen-*

erational [67] copying collector to use copy orders that match data access patterns and thus improve data locality. Our online class reordering analysis achieves its low cost (at most 2% of total time) by piggybacking on method sampling in an adaptive JIT compiler. The adaptive compiler in Jikes RVM uses timer-driven sampling to identify *hot* (frequently executed) methods, and they adaptive compiler recompiles them at progressively higher optimization levels. At compile time, online class analysis enumerates the field accesses in each method. During execution, when the adaptive compiler identifies a hot method (regardless of its optimization choice), adaptive class reordering analysis marks the fields the method accesses as hot. At garbage collection time, the collector preferentially copies referents of hot fields first, together with their parent. In this framework, we explore how quickly to decay heat to respond to phase changes, exploit Jikes RVM's static analysis to exclude cold basic blocks from the reordering analysis, and group objects of hot classes together in a separate copy space. More details about this research are presented in Chapter 5.

1.2 Improving Instruction Locality

Another locality problem is instruction locality. Previous research on code reordering usually uses static profiling to perform reordering either at compile time or at link time, which as we point out above is problematic for Java. Most studies try to reorder code on three different granularities, procedure reordering [36, 37, 58], basic block reordering [37, 58, 59], or proce-

dure splitting [58]. Their results show improvement in reducing instruction cache misses by up to 40% [37]. Because Java virtual machines have dynamic class loading and dynamic code generation, these static schemes are less effective. However, Java virtual machines also provide a new opportunity to apply similar optimizations during the execution of the program. We can pick the most frequently executed methods and avoid conflict misses on these methods. Most previous dynamic schemes include hardware components for detecting and removing conflict misses [10, 60]. Chen et al. [20] dynamically allocate procedure code at the time the procedure is invoked. But they only reorder the procedures inside libraries and they cannot move the instructions after they have been allocated. They show a comparable benefit to the profile-based algorithms.

We developed two online code reordering systems for improving instruction locality: one reorders all the methods in the heap and the other only does partial code reordering. An advantage of online schemes is that they can accommodate the dynamic properties of Java programs such as dynamic class loading, garbage collection, and dynamic code generation. However, the overhead at run time needs to be low to make our online algorithms beneficial.

We first describe the implementation of a *Whole Code Management* (WCM) system that is integrated into our managed runtime in Section 6.3. WCM uses dynamic profile information to reorganize all the compiled code at the granularity of a method. We show that our WCM can significantly improve performance. We also describe three new procedure layout algorithms that,

compared to previous approaches, reduce the cost of computing a new code placement. These algorithms specifically target ITLB misses, which typically have the greatest impact on performance because of their frequency and high cost. One of these algorithms, the Code Tiling is significantly faster both in worst case complexity and in practice than the best-known previous technique by Pettis and Hansen [58]. We demonstrate that Code Tiling generates code layouts that are better or comparable to those by the Pettis-Hansen algorithm.

Although our WCM algorithms have much lower overhead than the popular Pettis-Hansen algorithm, they are still too expensive for short running programs. We hence developed *Partial Code Reordering* (PCR) which performs partial online code reordering so that we achieve even lower low overhead than WCM. PCR improves instruction locality by attacking both capacity misses and conflict misses in the cache. PCR performs three optimizations using multiple code spaces: (1) interprocedural hot/cold method separation, (2) intraprocedural hot/cold code splitting, and (3) interprocedural hot code padding. To reduce capacity misses, PCR allocates hot and cold methods into separate spaces in the heap. PCR also performs code splitting of hot and cold basic blocks within the same method to further reduce the hot instruction working set size. PCR utilizes the adaptive sampling system in Jikes RVM to detect hot methods and collect the dynamic edge profile to determine hot basic blocks. To reduce conflict misses, PCR uses the dynamic call graph generated by dynamic stack profiling to find hot caller/callee pairs. If the hot caller and callee methods map to the same cache line in the cache,

they will have too many conflict misses. Therefore, PCR applies code padding on either caller or callee method (which ever it happens to be recompiling) to eliminate the potential conflict misses. Detailed algorithms of PCR are presented in Section 6.4.

1.3 Contributions

The main contributions of this thesis are:

1. Program characteristics study: Quantifying Java program memory behavior to find performance losses due to poor locality.

2. Improving data locality:

   (a) A novel, low-overhead, Online Object Reordering (OOR) system to improve data locality. We can improve over static copying orders by up to 25%, and the overhead of OOR is at most 2%.

   (b) A thorough evaluation on three architectures of the performance improvements of data reordering optimizations and the comparison of different static data copying schemes.

3. Improving instruction locality:

   (a) A Whole Code Management (WCM) system. This implementation of dynamic code reordering in a managed runtime is the first to our knowledge. Since it operates on-the-fly, it naturally copes with

dynamic features of languages like Java such as method recompilation and dynamic class loading. Results show, for example, that it reduces the execution time for a large benchmark by 6%.

   i. A new code placement algorithm called Code Tiling. This algorithm is fast enough to make dynamic code reorganization practical in a high-performance managed runtime. Placements computed by this algorithm perform as well and often better than those produced by the Pettis-Hansen procedure layout algorithm.

   ii. Detailed evaluation of the quality and overhead of dynamic call graphs which are generated using hardware performance counters on Intel Itanium architectures.

(b) An instruction locality optimization framework, Partial Code Reordering (PCR) system, which piggybacks on hotspot recompilation to achieve negligible overheads and reduced instruction cache footprints.

   i. The design of four code space optimizations: (1) one space with all code[1]; (2) two spaces: separate hot and cold methods; (3) three spaces: cold methods, hot blocks of hot methods, and cold blocks of hot methods; (4) three spaces with method padding for hot caller-callee pairs.

[1]This design is common in commercial VMs.

ii. A thorough evaluation on two architectures and in simulation of the potential and actual performance of code space optimizations. Simulation results show potential improvements are possible, but PCR has a negligible effect in practice because of the small instruction cache footprint of the benchmarks we tested.

This work is the first to exploit the available opportunities in object-oriented programming languages to improve program locality with low overhead dynamic optimizations. This dissertation not only proves the applicability of this approach, but also develops several novel dynamic optimizations which effectively improve data and instruction locality.

Chapter 2

Background

This chapter gives the background of the two dynamic optimization systems we use in this dissertation. However, our techniques are not specific to these systems and can be added to any systems with mechanisms to identify hot methods at runtime, dynamic code generation, and generational copying collector. All three features are common to current virtual machine systems for Java and C#. Our optimizations for improving program locality are taking advantage of opportunities that are not VM dependent.

We first describe how the adaptive sampling, compilation system, and code allocation in the IBM Jikes RVM [3, 4] works. Then we describe the generational copying collector from the Memory Management Toolkit (MMTk). Last, we describe the code generation in Intel Open Runtime Platform virtual machine (ORP) [25] because we implement two different code reordering systems in Jikes RVM and ORP respectively. This chapter is to set the stage to explain our online data/instructions reordering algorithms in the subsequent chapters.

2.1 Jikes RVM

Jikes RVM [4, 6, 7, 44] is an open source high-performance Java-in-Java virtual machine (VM) written almost entirely in a slightly extended Java. A few C code is needed for boot strapping and system calls. Our work of online object reordering and partial code reordering is in the context of Jikes RVM. We leverage the existing adaptive optimization system in Jikes RVM for both our data object reordering and code reordering system. In this section we briefly review the most relevant aspects of this adaptive optimization system.

Jikes RVM *does not have* a bytecode interpreter. Instead, a fast template-driven baseline compiler produces machine code when the VM first executes each Java method. Using timer-based profiling, the adaptive system periodically samples the currently executing code and records (1) the currently executing method and (2) the caller of the currently executing method. This profile data is fed into a cost-benefit model to identify methods that should be recompiled at a higher level of optimization. Methods selected by the system for recompilation are compiled asynchronously on a separate compilation thread by the Jikes RVM optimizing compiler. The profiled caller-callee relationships are used to maintain a weighted dynamic call graph that drives profile-directed inlining during optimizing compilation.

While generating code, the baseline compiler inserts instrumentation for every bytecode-level conditional branch to measure its execution frequency and its taken/not-taken distribution. The optimizing compiler uses this edge profile data to compute basic block frequencies and branch probabilities. A

number of optimizations exploit this information, and most relevant research to our work is basic block layout generation. At the lowest optimization level (O0), the compiler simply moves infrequently executed basic blocks to the bottom of the compiled method's code. At O1 and O2, it employs Pettis-Hansen's bottom-up positioning algorithm (Algo2) [58]. Note the Pettis-Hansen algorithm used here is for blocks reordering within a method, not the Pettis and Hansen algorithm which does inter-procedural method reordering and will be used for comparison in our instruction locality work.

2.2 Garbage Collection and MMTk

MMTk is an efficient, composable Java memory management toolkit used in Jikes RVM that implements a wide variety of high performance garbage collectors that reuse shared components [11, 12]. It is ideal for our experiments because we can easily apply different algorithms on the same data structure to systematically compare different policies.

We use the copying generational collection in MMTk, which is one of the best performing memory managers [11]. Copying generational collection divides the heap into two portions, a *nursery* containing newly allocated (i.e., *young*) objects, and a *mature space*, containing older objects [51, 67]. It further divides the mature space into two semispaces. It collects the nursery frequently (whenever the nursery fills up), by performing a transitive closure over the nursery objects, and copying them into one semispace of the mature space. When that semispace is full after a nursery collection, it copies reach-

able mature objects into the other semispace and flips the roles of the two semispaces.

Since the generational collector collects the nursery separately from the mature space, it requires the compiler to insert *write-barrier* code, which at run time records stores of pointers from mature to nursery objects in a *remembered set*. When the collector starts a nursery GC, the remembered set forms part of the set of *root pointers*, which also includes the stacks, registers, and static variables. It assumes all reachable objects are live. It copies any referents of the root pointers that lie in the nursery and iteratively enumerates the pointers in newly copied objects, copying their nursery referents, until it copies all reachable nursery objects. Mature space garbage collection proceeds similarly, except the remembered set is empty and the collector copies any reachable objects in both mature space and nursery. This scheme generalizes to multiple generations, but we use two.

We use the bounded generational copying collector (*GenCopy*) in our Online Object Reordering system for improving data locality. It follows Appel's flexible nursery [5], which shrinks the nursery as mature space occupancy grows, except that we never allow the nursery to exceed a fixed chosen bound (4MB) to reduce the average time to collect the nursery. When mature space occupancy approaches the maximum total heap size, bounded generational collector copying collector shrinks the nursery, until it reaches a lower bound (256KB) that triggers mature space collection. MMTk manages large objects (8KB or bigger) separately in a non-copying space and puts the compiler and a

few other system pieces into the boot image, an immortal space. See Blackburn et al. for additional implementation details [11, 12].

2.3 ORP: Intel Open Runtime Platform

Besides Jikes RVM, we also implemented our optimizations for improving instruction locality in Intel's Open Runtime Platform virtual machine (ORP). ORP is a managed runtime environment that supports Java and C# programs. ORP is more robust than Jikes RVM and ORP can run more benchmarks including large server benchmarks. Not like Jikes RVM, ORP is written in C++. Our core platform consists of the ORP and one or more JIT compilers. On IA-32, we use the optimizing O3 JIT [26] to compile JVM bytecodes. This JIT performs inlining, a number of global optimizations (e.g., copy propagation, dead code elimination, loop transformations, and constant folding), as well as CSE and array bounds check elimination.

In ORP, a JIT may emit compiled code for a method that consists of more than one separately-allocated *code block*, which is a set of basic blocks in one method. Because of this, code blocks are the units of code management in our algorithm, not methods, and WCM reorders code blocks. ORP also allocates code blocks in a region of memory that is separate from the garbage collected heap. It provides different subregions for code that is cold (infrequently executed) and warm (more often executed). ORP allocates code of equal "temperature" sequentially. ORP can collect dynamic profile information from either software instrumentation or from hardware Performance

Monitoring Unit (PMU) sampling. These dynamic profiles are used for optimizations such as basic block reordering within a method.

Chapter 3

Literature Survey

In this section, we discuss the most closely related work in the following areas: (1) memory performance studies for Java programs, (2) research on improving data locality for Java, and (3) research on code reordering to generate better instruction locality.

3.1 Memory Performance Studies

Several studies characterize the memory behavior and performance of Java programs via simulation [46, 50, 63]. Kim *et al.* [46] studied memory behavior by feeding Java memory access traces to cache simulators. The garbage collection algorithm they studied was mark-sweep GC. In our study, we will examine the behavior of Java programs in the context of bounded copying nursery generational garbage collectors, which have higher performance [11]. Their study concentrated on data locality but not instruction locality, while our work studies the effect of locality from both caches.

Li *et al.* [50] studied the performance characteristics of the SPECjvm98 Java programs. They used SimOS in their experiments. They did not differentiate the impact of mutator and GC, which, as we will show later, exhibit very different memory behaviors. SimOS does not have a cycle-level processor

model, affecting the accuracy of their results. Also, they did not have detailed study of instruction cache locality.

Shuf *et al.* [63] use a very similar methodology to Kim *et al.* They generated traces and simulated memory behavior by using the trace on a cache simulator on some of the benchmarks that we use. They adopted a very large heap size, ignoring the costs and benefits of GC. Also, because they use unusually large heaps, their results focus unnecessarily on TLB misses as a problem. In our study, we vary heap sizes and study the effects of GC and the interaction between mutator and GC. We find these applications very rarely stress the TLB since copying GC usually reduces the program's memory footprint.

3.2 Data Locality

The key investigations of our data locality work are (1) exploiting the object reordering that happens during copying generational garbage collection [51, 67], and (2) using online profiling to collect information for controlling the copying order. Much previous research in this area considers non-garbage collected languages (such as C) [22, 23, 69], or does not address the effects of copying collectors [47]. In other words, it neither considers nor exploits *moving* heap objects.

The related work most pertinent to ours falls into two categories: techniques that group objects to improve *inter*-object locality [22, 24, 48, 69], and those that reorder fields within an instance to improve *intra*-object local-

ity [22, 47]. This prior work relies on static analysis or offline profiling to drive object layout decisions and is class-oblivious for the most part, i.e., it treats all classes the same.

One can improve inter-object locality by clustering together objects whose accesses are highly correlated. The work in this area differs in how to define correlation and specific methods to cluster objects. Wilson et al. describe a hierarchical decomposition algorithm to group data structures of LISP programs using static-graph reorganization to improve locality of memory pages [69]. They found that using a two-level queue for the *Cheney scan* groups objects effectively. Lam et al. later conclude that hierarchical decomposition is not always effective [48]. They suggest that users supply object type information to group objects. We automatically and adaptively examine fine-grained field accesses to generate such type-based advice.

Chilimbi and Larus use a continuously running online profiling technique to track recently referenced objects and build a *temporal affinity graph* [24] based on the frequency of accesses to pairs of objects within a temporal interval. The object pair need not be connected by a pointer, but must lie in the same non-nursery generation to reduce overhead. Their dynamic instance-level profiling records in a buffer most pointers fetched from the heap. They report overheads of 6% for Cecil, a language which is not as well adopted as Java. Exploiting the timer-driven sampling, already in the adaptive optimization system of Jikes RVM, is much cheaper. We find copying cannot guarantee to improve every program by at least 6% so as to overcome instance

profiling costs by their approach. Their algorithm copies together objects with high affinity only during collection of the old generation whereas our system reorders objects during both nursery and old generation collections.

Chilimbi et al. split objects into hot and cold parts to group the hot parts together [22]. This technique is not fully automated and requires substantial programmer intervention. Chilimbi et al.'s clustering and coloring methods also rely on manual insertion of special allocation functions [23]. Our technique is automatic, but we do not support object splitting in our current system.

Intra-object locality can be improved by grouping hot fields together so that they will usually lie in the same cache line, and it is most useful for objects somewhat bigger than a cache line. The size of hot objects in Java benchmarks is close to and rarely exceeds 32 bytes [32], whereas typical L1 cache line sizes are 64 to 128 bytes and L2 line sizes are 64 to 256 bytes. Thus the performance improvement offered by field reordering alone is usually small. Kistler and Franz use an LRU stack to track the temporal affinity of object fields, and they partition and reorder fields based on their affinity graph [47]. They use a mark-sweep collector, where field reordering has no effect on the object order after collection. Chilimbi et al.'s field reordering depends on profiling to generate reordering advice [22]. The programmer then follows the advice to rewrite the code and reorder fields.

Rubin, Bodik, and Chilimbi developed a framework that attempts to pull together much prior work in this area [61]. Their approach involves the

following steps. (1) Produce an access trace with instance and field labels. (2) Compress the trace to fit in main memory and include only accesses relevant to a specific cache size and configuration. (3) Compute the objects with the most accesses and misses. (4) Use object properties (e.g., size, field access frequencies, field access correlations) to select optimizations. (5) Perform a hill-climbing search of possible field and object layout schemes, and model misses for each scheme on the compressed trace. Their framework would need significant changes to address moving collectors, and it is practical only as an offline tool. In contrast, we exploit the reordering of objects inherent in copying collection and our online analysis is inexpensive and robust to phase behavior.

3.3 Instruction Locality

Numerous researchers have studied the problem of restructuring programs to improve the memory performance of instruction accesses. Much of the early software-based work reduces virtual memory page faults. Some current work also tries to minimize these very expensive faults [70]. However, most recent work focuses on static and dynamic approaches for reordering code to reduce instruction cache and ITLB misses with offline and online profiling.

3.3.1 Static code placement

Researchers have explored code placement at compile or link-time at a number of different granularities: for example, at the granularity of basic blocks, groups of basic blocks, or entire procedures. A limitation of these

static layout approaches is that they produce a fixed static layout, which as we discussed in Section 1.2, is not suitable for a managed runtime. Furthermore, static schemes must assume that the profile data gathered on a training run is representative of all program executions and miss the opportunities available to a managed runtime of exploiting profile data from the program's current execution.

McFarling [53] uses profile data to lay out code to reduce misses in a direct-mapped instruction cache. His algorithm identifies those parts of a program that could overlap each other in the cache and those that should be placed in non-conflicting addresses.

Pettis and Hansen [58] perform profile-based code placement at all three granularities. 1) At the finest granularity, basic block positioning lays out basic blocks to straighten out common control paths and minimize control transfers. 2) Procedure splitting moves a procedure's never-executed basic blocks into a different allocation area from that of its other blocks. 3) At the coarsest granularity, a greedy algorithm starts with an undirected weighted call graph constructed from the profile data and progressively combines its nodes to place frequent caller-callee procedure pairs close together. Pettis and Hansen show that combining all three optimizations can improve performance up to 26% (average about 12%) with a 16 KB directly-mapped unified cache. However, the improvement they achieve is very sensitive to cache organization and their algorithms can not achieve similar improvement on current architectures with bigger cache size (see Section 6.3.4). Because it is both simple and effective,

their procedure ordering algorithm is generally considered the reference placement technique. It is the basis for several more recent algorithms. However, it has performance instability because small changes in the profile data often produce substantially different layouts as we will show in Section 6.3.

Cohn et al [29] describe the Spike post-link optimizer for Alpha/NT executables which includes the Pettis-Hansen procedure code layout algorithm. They report that, on a set of large benchmarks, Spike speeds up most by at least 5%, and often 10% or better. Ispike [52] is a post-link optimizer for the Itanium Process Family (IPF). It uses the IPF performance counters to collect low cost detailed profile information for instruction and data optimizations including inlining, branch forwarding, layout, and prefetching of both code and data. Their code layout optimization includes 1) basic-block chaining to lay out basic blocks in sequence if there is a frequently-executed control flow edge between them, 2) procedure splitting, and 3) procedure layout that keeps hot procedures close together. On a set of small benchmarks, they found that code layout by itself helps one-third of the benchmarks by over 4%.

Hashemi et al. [37] take the cache configuration into account to lay out procedures using cache line coloring. Their algorithm colors each cache line in the instruction cache and uses a greedy algorithm similar to Pettis and Hansen's to place procedures such that the most frequent caller-callee pairs will not occupy the same cache lines. In simulation, they achieve 17% lower instruction miss rate than Pettis and Hansen algorithm. Gloy and Smith [36] also compute procedure layouts that reflect the cache configuration. They

collect complete procedure interleaving information that in combination with the cache configuration and procedure sizes, they use to produce a layout that minimizes both cache conflicts and the instruction working set size. By making use of temporal locality information, their technique eliminates more cache conflict misses than Pettis and Hansen.

Ramirez et al. [59] developed a code reordering system, called the Software Trace Cache (STC), to improve the instruction cache hit rate and increase the processor's effective instruction fetch width. Using profile information, STC determines traces (hot basic block paths) then maps the resulting traces into memory locations that minimize cache conflicts. It also makes effective use of instruction cache lines while tending to keep sequentially-executed instructions in order. STC also reserves a region in the instruction cache for hot instructions to avoid conflict misses with cold instructions.

Since these static approaches generate code layouts ahead-of-time, they lose the flexibility of determining layouts using the actual information for a particular run of a program. They also cannot cope with different program phases. The time complexity of these algorithms is too high for a dynamic scheme. For example, Pettis and Hansen's algorithm has a time complexity of $O(n^3)$. If we use Pettis and Hansen's algorithm to These limitations make them less useful in the context of virtual machines.

3.3.2 Dynamic code placement

Dynamic schemes for improving instruction locality typically monitor system behavior and apply optimizations at runtime based on that behavior.

Chen and Leupen's just-in-time code layout technique places the procedures of Windows applications in the order of their invocation at runtime [20]. Their results show improvements similar to that of the Pettis and Hansen. It also substantially reduces the program's working set size, often by about 50%. Pettis-Hansen's procedure layout also reduces the working set, but because it is a static approach, it is less effective because the procedures executed do not exactly match those of the training run. Chen and Leupen's approach lays out procedures at allocation time, whereas our approach reorders hot procedures during recompilation or after the program's warm-up phase.

Scales' DPP (dynamic procedure placement) system uses runtime information to dynamically lay out procedure code [62]. DPP uses a loader component that is invoked on procedure calls. It copies the code of the called procedure to a new code region, where it will be close to the caller, then fixes up all references to the procedure to refer to the new copy. Because this system supports C and other languages that are not strongly typed, it deals with indirect calls by memory protecting the original code space, so that attempts to call a procedure at its original address result in a trap whose handler then invokes the new copy of that procedure. DPP's overhead is high because of the virtual memory protection traps and the many calls to the DPP loader. The DPP system can restart procedure placement to try to improve the layout, but each restart is expensive due to the overhead of the new loader calls. An extension of DPP supports runtime profiling: at each call to the loader, the call stack is recorded to build a profile of the calls. This information is used

later to improve the layout. However, this profiling is extremely expensive and slows down the program by a factor of ten or more.

Whaley [68] very briefly outlines a never implemented dynamic procedure code layout optimization for Jikes RVM. It also piggybacks on branch and call stack profiling, but suggests passing this information to the garbage collector as a hint to reorder code in the heap (see Figure 6.17(a)). In contrast, our PCR algorithm separates code from data objects in the heap which sometimes improves performance. Furthermore, PCR pads conflicting hot caller/callee pairs when the methods are recompiled, and it does not wait until garbage collection.

Recent research [14, 38, 39, 60] investigates code cache management for dynamic binary optimizing systems. This work focuses on frameworks for software managed code caches, creating basic block sequences (superblocks) for a trace cache, replacement policies for hardware instruction caches, and sharing between threads. Our work is complementary to theirs since we not only reduce the working set size by code splitting, and whole program reordering, but also reduce conflict misses by code padding.

Our system thus differs from the prior work in several key ways: it is not restricted to invocation order [20], nor does it rely on expensive page protection [62], nor does it require special hardware [39, 60], and it is implemented in a JVM [68].

Chapter 4

Methodology

We begin describing our experimental methodology, and relevant characteristics of the benchmarks we use. We also present different platforms used in our experiments in this section.

4.1 Jikes RVM

We use two methodologies for our experiments on Jikes RVM. (1) The *adaptive* methodology lets the adaptive compiler behave as intended and is non-deterministic. (2) The *compiler-replay* methodology is deterministic and eliminates memory allocation and mutator variations due to non-deterministic application of the adaptive compiler. We need this latter methodology because the non-determinism of the adaptive compilation system makes it a difficult platform for detailed performance studies. For example, we cannot determine if a variation is due to the system change being studied or just a different application of the adaptive compiler due to sampling variations.

In the adaptive methodology, the adaptive compiler uses non-deterministic sampling to detect hot methods and blocks, and then tailors optimizations for the hot blocks. Thus on different executions, it can optimize different methods and, for example, choose to inline different methods. Furthermore,

any variations in the underlying system induce variation in the adaptive compiler. We use this methodology for measuring the overhead of our system in Section 5.2.2 and for measuring programs running under multi-program environment in Section 6.4.2.5.

For all other experiments, we use a deterministic methodology that holds the allocation load and the optimized code constant. The compiler-replay methodology gives a mixture of optimized and un-optimized code that reflects what the adaptive compiler chooses, but is specified by an advice file from a previous run. We run each benchmark five times and profile the optimization plan of the adaptive compiler for each run. We pick the optimization plan of the run with best performance and store it in an advice file. For the performance measurement runs, we execute two iterations of each benchmark and report the second. We turn off the adaptive compiler, but not the adaptive sampling. In the first iteration, the compiler optimizes selected methods at the selected level of optimization according to the advice file. Before the second iteration, we perform a whole heap collection to flush the heap of compiler objects. We then measure the second iteration. Thus we have optimized code only for the hot methods (as determined in the advice file). This strategy minimizes variation due to the adaptive compiler since the workload is not exposed to varying amounts of allocation due to the adaptive compilation. We measure only the application behavior and exclude the compiler in this methodology.

We report the second iteration because Eeckhout et al. show that mea-

suring the first iteration, which *includes* the adaptive compiler, is dominated by the compiler rather than the benchmark behavior [34].

For each experiment we report, we measure the benchmark five times, interleaving the compared systems. We use the methodologies above, and take the fastest time. The variation between these measurements is low. We believe this number is relatively undisturbed by other system factors. When measuring the system overhead in the adaptive compiler, we believe the low variation from the fastest time reflects a stable application of the adaptive compiler.

4.2 Dynamic SimpleScalar

We conduct our experiments both on real machines whenever applicable and on our simulator for different cache configuration and some instrumented runs. The simulator we use is Dynamic SimpleScalar (DSS) [42], which is developed on the base of SimpleScalar [9, 15, 30].

Many current simulators do not have support for simulation of dynamic compilation, threads, or garbage collection–all of which Java Virtual Machines (JVMs) require. To experiment with effects of different cache configuration and to study detailed memory behavior of Java programs, we developed *Dynamic SimpleScalar* (DSS). DSS is a tool that simulates Java programs running on a JVM, using just-in-time compilation, executing on a simulated multi-way issue, out-of-order execution superscalar processor with a sophisticated memory system. Detailed implementation description and validation results are in our

technical report [42].

We use the same microprocessor configurations for our simulation experiments in this dissertation. The DSS configuration we use a processor model with five-stage pipeline. The details of this simulated microprocessor are as follows:

- Five-stage pipeline based on a 16 entry Register Update Unit (RUU), which combines the physical register file, reorder buffer, and issue window into a single data structure
- Out-of-order issue, including speculative execution
- Issue width, decode width, and commit width are 4
- 2-level branch predictor that uses its own 1 KB L1, 16 KB L2, and a 14 bit history register. The BTB is 2 way associative with 256 sets.
- An 8-entry load-store queue

4.3 DaCapo Benchmarks

DaCapo project is a multi-institution research project that aims to improve the performance of Java programs, with a particular focus on garbage collection and memory performance. As part of an ongoing effort with our collaborators in the DaCapo project [56], we collect several memory intensive Java programs for the DaCapo benchmark suite [13]. These benchmarks are intended to exercise garbage collection vigorously in order to reveal collector and platform induced differences.

1. **antlr**: parses one or more grammar files and generates a parser and lexical analyzer for each.
2. **bloat**: performs a number of optimizations and analysis on Java bytecode files
3. **fop**: takes an XSL-FO file, parses it and formats it, generating a PDF file.
4. **hsqldb**: executes a JDBC-like in-memory benchmark, executing a number of transactions against a model of a banking application
5. **jython**: interprets a series of Python programs
6. **pmd**: analyzes a set of Java classes for a range of source code problems
7. **ps**: reads and interprets a PostScript file
8. **xalan**: transforms XML documents into HTML
9. **ipsixql**: persistent XML database.
10. **postscript-fun**: a PostScript interpreter.

4.3.1 Benchmark Characteristics

Table 4.1 and Table 4.2 shows key characteristics of our benchmarks using the fixed workload and adaptive methodologies. We use the eight SPECjvm98 benchmarks, five DaCapo benchmarks, plus pseudojbb, a variant of SPECjbb2000 [64, 65] that executes a fixed number of transactions (70000), rather than running for a fixed time (for comparisons under a fixed work load). The *alloc* columns in Table 4.1 and Table 4.2 indicate the total number of megabytes allocated under adaptive and fixed work loads respectively. The *alloc:min* column lists

Benchmark	classes loaded	methods compiled	alloc (MB)	alloc: min
jess	155	507	403	25:1
jack	61	331	307	22:1
javac	160	821	593	23:1
raytrace	34	227	215	12:1
mtrt	35	225	224	11:1
compress	16	99	138	8:1
db	8	92	119	6:1
mpegaudio	59	270	51	4:1
ps-fun	347	522	8602	410:1
ipsixql	120	381	1777	105:1
hsqldb	90	432	6804	76:1
jython	175	1050	796	47:1
antlr	114	719	22	18:1
pseudojbb	13	92	339	7:1

Table 4.1: Benchmark Characteristics With Adaptive Run

the ratio of total allocation to the minimum heap size in which the program executes in MMTk. Including the adaptive compiler substantially increases allocation and collector load (compare *alloc* columns in Table 4.1 and Table 4.2, and *alloc:min* columns in Table 4.1 and Table 4.2). This behavior can obscure program behaviors and further confirms Eeckhout et al. [34]. Notice that mpegaudio allocates only 3MB, and with a 4MB heap is never collected; hence we exclude it from the remaining experiments. Also notice that the DaCapo benchmarks place substantially more load on the memory management system than the SPECjvm98 benchmarks. Therefore there are more opportunities for improving data locality in DaCapo benchmarks.

The *% nrs srv* column indicates the percent of allocation in the nursery that the collector copies (e.g., mpegaudio copies 0% means the survival rate of nursery object is less than 0.5% for mpegaudio). OOR can influence the subset of these objects with two or more non-null pointers. Notice that most programs follow the weak generational hypothesis, but that javac and ipsixql

Benchmark	alloc (MB)	alloc: min	% nrs srv	% wb take	alloc pointers			scan pointers			scan non-null pointers		
					0	1	many	0	1	many	0	1	many
jess	261	17:1	1	0.08	18%	40%	42%	1%	52%	47%	7%	49%	43%
jack	231	17:1	3	3.15	48%	31%	22%	21%	44%	35%	34%	53%	13%
javac	185	7:1	23	1.21	29%	34%	37%	5%	27%	68%	6%	34%	60%
raytrace	135	8:1	2	0.01	89%	1%	10%	55%	12%	33%	57%	14%	29%
mtrt	142	7:1	5	0.65	87%	2%	11%	55%	12%	33%	57%	14%	29%
compress	99	6:1	0	1.20	56%	34%	10%	41%	31%	29%	43%	37%	20%
db	82	4:1	9	1.21	4%	95%	1%	42%	53%	5%	42%	53%	5%
mpegaudio	3	1:1	0	0.00	76%	15%	10%	83%	5%	12%	83%	5%	12%
ps-fun	8589	409:1	0	0.00	95%	2%	3%	25%	30%	45%	25%	30%	44%
ipsixql	1739	102:1	31	1.17	40%	3%	56%	39%	2%	59%	39%	2%	59%
hsqldb	6720	75:1	4	0.01	44%	41%	16%	50%	0%	50%	50%	0%	50%
jython	722	42:1	1	0.103	0%	78%	22%	1%	62%	37%	2%	64%	34%
antlr	5	3:1	11	1.78	68%	23%	9%	25%	26%	48%	30%	41%	28%
pseudojbb	216	5:1	32	1.82	51%	26%	23%	36%	29%	35%	37%	29%	34%

Table 4.2: Benchmark Characteristics With Fixed Workload

are memory intensive while not being very generational. However, generational collectors still improve their performance [11].

The *% wb take* column shows the percent of all writes that the write barrier records in the remembered set. The remaining columns indicate the percentage of objects with 0, 1, or many pointer fields. The *alloc pointers* column indicates these proportions with respect to allocated objects. The *scan pointers* column indicates the proportions with respect to objects scanned at collection time, and *scan non-null pointers* indicates the proportions with respect to non-null pointers in objects scanned at collection time. Since OOR influences only objects with two or more non-null pointers, the final column in Table 4.2 indicates the proportion of scanned (copied) objects to which OOR can be applied.

We ran all the experiments we report here on all the benchmarks. For some benchmarks, performance variations due to different optimizations are small. For brevity and clarity, the results section focuses on programs that are

sensitive to our optimizations (positive or negative), and just summarizes the programs where our optimizations have little effect.

4.3.2 Program Locality Potential

This section we examine the locality characteristics of these benchmarks. We first use an unrealistic model of perfect cache as the caches used in Chapter 1 to measure the potential performance lost due to poor locality. In these perfect caches, there is never a miss, not even a compulsory miss.

Figure 4.1 shows DaCapo benchmarks and SPECjvm98 Benchmarks running under perfect caches. We use modest cache sizes for our experiments: both instruction L1 and data L1 caches are 32K 2-way set associative. 512K unified L2 cache. We run our benchmarks under five configurations: Base (regular caches); perfect L2 cache; perfect DL1 and L2 caches; perfect IL1 and L2 caches; and perfect DL1, IL1 and L2 caches. Also, to reduce the anomaly generated by mixing data and instruction together, we use separate spaces for data and instructions for this experiment. From Figure 4.1, we can see most of the performance is lost due to poor L2 cache locality (on average, 20% of the execution time is waiting for L2 cache misses). Because these benchmarks usually have small instruction footprints (as we will show later in Table 6.6), the L2 performance loss is mostly contributed by L2 data locality. As for L1 cache locality, instruction locality has more impact than data locality (5.7% vs. 1.7%). These results show the performance potential of improving L2 data locality and L1 instruction locality for these programs, as we will verify again in our results in the following chapters.

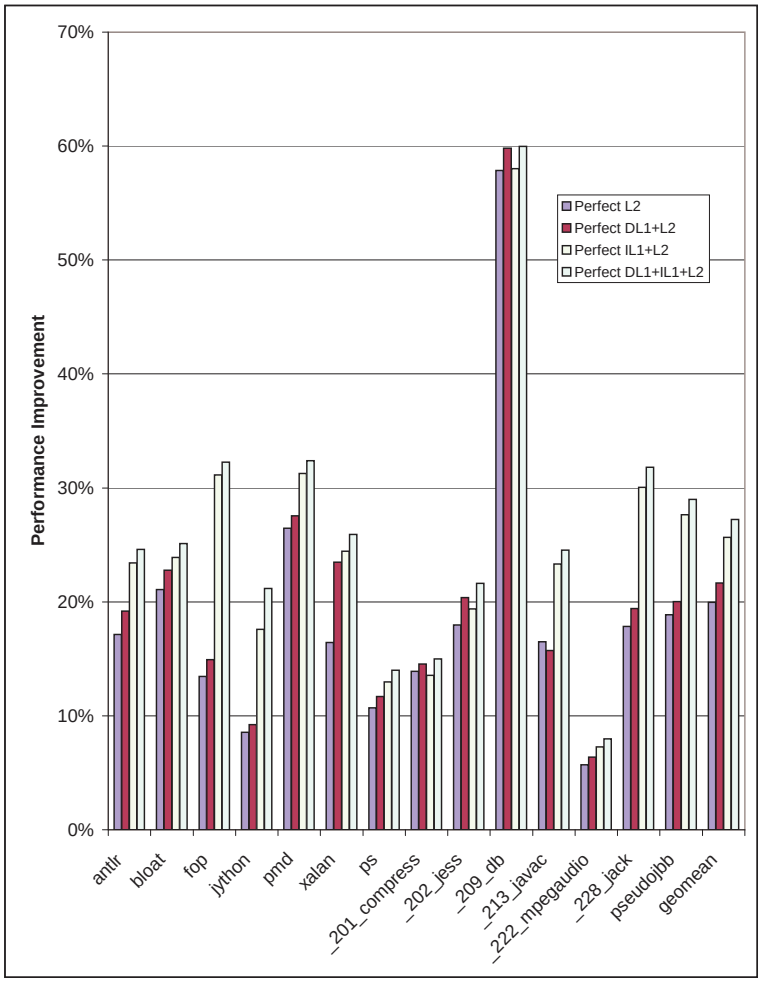


Figure 4.1: Java Benchmarks with Perfect Caches

4.4 Server Benchmarks

We use MiniBean and SPECjbb for our evaluation of full code layout algorithms to improve program locality. MiniBean is a large benchmark inspired by the SPECjAppServer2002 enterprise application server benchmark, but runs in a single process on a single machine. While MiniBean uses enterprise JavaBeans (EJB) functionality, it generates no network traffic itself.

Benchmark	IA32 Size	Methods	Calls/sec
SPECjbb	268K	758	2.04M
MiniBean	3.10M	15586	3.65M

Table 4.3: Benchmark characteristics

SPECjAppServer2002, on the other hand, must be run using multiple machines including a database server. When collecting our results, we run each benchmark five times and used the best time. Table 6.3 shows the code allocation characteristics of these benchmarks[1]. As for heap sizes, we use 512M for MiniBean, 256M for SPEC JBB2000. We reorganize code once at the end of application warm-up with MiniBean and SPEC JBB2000, and at the end of the first iteration of each program with SPEC JVM98.

4.5 Platforms

We report run-time results for our implementation on the following platforms:

3.2GHz P4 with hyper-threading enabled, a 64 byte L1 and L2 cache line size, an 8 KB 4-way set associative L1 data cache, a 12 Kμops L1 instruction trace cache, a 512 KB unified 8-way set associative L2 on-chip cache, and 1 GB main memory running Linux 2.6.0.

2.4GHz P4 The 2.4GHz Pentium 4 uses HyperThreading. It has a 64 byte L1 and L2 cache line size, an 8KB 4-way set associative L1 data cache,

[1]We measure the calls per second by running SPEC JBB2000 for 50 seconds and MiniBean for 200 seconds.

a 12Kμops L1 instruction trace cache, and a 512KB unified 8-way set associative L2 on-chip cache, 1 GB main memory, and runs Linux 2.6.0.

1.9GHz AMD Athlon XP 2600+ with a 64 byte L1 and L2 cache line size. The data and instruction L1 caches are 64KB 2-way set associative. It has a unified, *exclusive* 512KB 16-way set associative L2 cache. The L2 holds only replacement victims from the L1, and it does not contain copies of data cached in the L1. The Athlon has 1GB of main memory and runs Linux 2.6.0.

933MHz PPC The Apple G4 has a 933MHz PowerPC 7450 processor, separate 32KB on-chip L1 data and instruction caches, a 256KB unified L2 cache, 32 bytes L1 and L2 cache line size, 512MB of memory, and runs Linux 2.4.25.

1.6GHz PowerPC 970 with a 128 byte L1 and L2 cache line size, a 32 KB 2-way set associative L1 instruction and data (split) caches, a 512 KB unified 8-way set associative L2 on-chip cache, and 1 GB main memory running Linux 2.6.0.

1.5GHz Itanium 2 with 4×1.5GHz processors running Windows Server 2003. On each processor, the data and instruction L1 caches are both 16KB in size with 4-way set associativity and have a 64 byte line size. The 256KB unified L2 on-chip cache is 8-way set associative and has a 128-byte cache line. Also, the L3 cache is 9MB, has 128-byte cache lines and is 36-way associative. The ITLB on this machine is a two level TLB, where both

levels are fully-associative, The L1 ITLB has 32 entries while the L2 ITLB has 128 entries. Page size is 4KB.

2GHz Xeon with 4×2GHz Xeon processors. This machine runs Windows 2000 Advanced Server Edition and has a 400MHz system bus. Each processor has an 8K 4-way set associative L1 data cache, a 8-way 12Kμops L1 instruction trace cache, a 512K unified 8-way set associative L2 on-chip cache, and a 2MB 8-way L3 cache. The L1 and L2 cache line size is 64 bytes. This machine's ITLB has 128 entries and is 4-way set associative. We disable HyperThreading and use the default 4K page size.

Chapter 5

Online Object Reordering to Improve Data Locality

The generational copying collector can reorder the objects during garbage collection and from Section 4.3.2, we observe that there is a large performance improvement potential by improving data locality. We develop the *Online Object Reordering (OOR)* system to exploit this opportunity to improve program data locality at runtime. OOR analysis identifies the hot field accesses by piggyback to JIT compilation and use the hot field information to direct code copying policy during garbage collection. The OOR system thus generate data layout that matches the dynamic access pattern of the program. The OOR system is the first effective dynamic object reordering system for improving data locality with low overhead.

5.1 Online Object Reordering Algorithm

The Online Object Reordering (OOR) system is class-based, dynamic, and low-overhead. OOR consists of three components, each of which extends a subsystem of Jikes RVM: (1) static compiler analysis; (2) adaptive sampling for hot methods in the adaptive optimization subsystem; and (3) object traversal and reordering in garbage collection. Figure 5.1 depicts the structure and in-

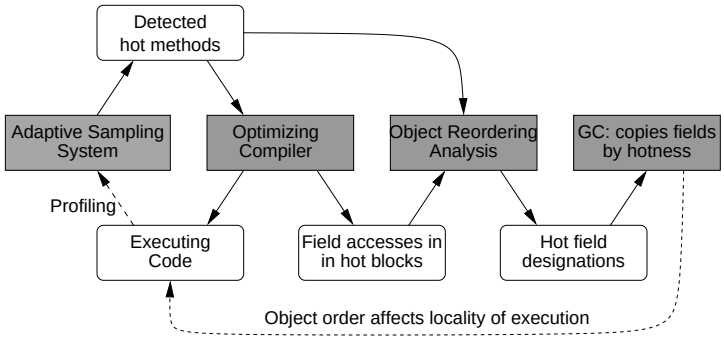


Figure 5.1: OOR System Architecture

teractions of the OOR system. When Jikes RVM initially compiles a method, we collect information about field accesses within that method. Later, the Jikes RVM adaptive compilation system identifies frequently executed (*hot*) methods and blocks using sampling (see Chapter 2). We piggyback on this mechanism to mark hot field accesses by combining the hot method information with the previously collected field accesses. We then use this information during garbage collection to traverse the hot fields first. The next three sections discuss each component in more detail.

5.1.1 Static Identification of Field Accesses

OOR analysis first identifies potentially hot fields by noting field accesses when first compiling each method. The Jikes RVM optimizing compiler uses a static analysis with a coldness threshold to mark cold basic blocks. OOR does not enumerate field accesses in cold blocks by using the compiler's default threshold (see Section 5.2.6). The compiler uses loop and branch prediction heuristics to estimate the execution frequency of basic blocks in a method. For

example, it marks exception handler basic blocks as cold, and basic blocks in loops as potentially hot. For each method, OOR analysis enumerates all the field accesses in potentially hot blocks, generating tuples of the form <`class`, `offset`>. The tuples identify the class and offset of any potentially hot field, and OOR associates each tuple with the compiled method. This analysis thus filters out field accesses the compiler statically determines are cold and associates a list of all non-cold field accesses with each compiled method. At present, we do not perform any field access analysis in the Jikes RVM baseline compiler. Since the Jikes RVM adaptive compilation framework recompiles hot methods with the optimizing compiler, we use it to apply our analysis selectively to hot methods. Jikes RVM also collects basic-block dynamic execution frequencies using a counter on every branch. We believe this information can improve the accuracy of OOR analysis, although we have not implemented this feature here.

5.1.2 Dynamically Identifying Hot Fields

The Jikes RVM adaptive sampling system detects hot methods by periodically sampling the currently executing method. When the number of samples for a method grows beyond a threshold the adaptive system invokes the optimizing compiler on it. OOR analysis piggybacks on this mechanism. The first time the system identifies a hot method, OOR changes all the potentially hot field accesses for the method to hot. Each time the sampling mechanism re-encounters a hot method (regardless of whether the adaptive system recompiles it), it updates the heat metric for the corresponding hot

```
DECAY-HEAT(method)
 1  for each fieldAccess in method do
 2    if POTENTIALLYHOT(fieldAccess) then
 3      hotField ← fieldAccess.field
 4      class ← hotField.instantiatingClass
 5      class.hasHotField ← true
 6      for each field in class do
 7        period ← NOW() − class.lastUpdate
 8        decay ← HI/(HI + period)
 9        field.heat ← field.heat ∗ decay
10        if field.heat < LO then
11          field.heat = 0
12      hotField.heat ← HI
13      class.lastUpdate ← NOW()
```

Figure 5.2: Pseudocode for Decaying Field Heat

fields.

Figure 5.2 shows OOR's decay mechanism for adapting to phase changes. Other policies are possible of course. The high and low heat thresholds, *HI* and *LO* (default values of 100 and 30 respectively) indicate the hottest field with heat *HI*[1] Any field cooler than *LO* is regarded as cold. Initially all fields are cold, with heat 0. When the timer goes off, the heuristic records the current sampling time, NOW(), and updates one or more heat fields in *class* for the method.

This heuristic decays heat for un-accessed fields based on the last time the analysis updated the instantiating class, *class.lastUpdate*. However, the heuristic does not decay field heat for all classes every sample period, since the cost would be prohibitive. Instead, it updates a class only when the adaptive compiler samples another method that uses a field instantiated by it. In the worst case of not strictly decaying field heat for all classes, the OOR collector will copy an old object using obsolete hot field information. Since none of

[1]The units for these thresholds are sample intervals, which are approximately 10ms: *HI* $\approx$ 1 sec, $LOW \approx 0.3$ sec.

the hot methods access this field, the order in which the collector copies these objects will simply be based on access orders further back in history and should not degrade performance because these fields are likely to be cold now. If these objects never become hot again, this mechanism does no harm. Otherwise, if their past accesses predict the future, program locality will benefit.

5.1.3 Reordering during Garbage Collection

The copying phase of the collector applies OOR advice. For each instance of a class, the collector traverses the hot fields (if any) first. At class load time, the OOR system constructs an array for each class, with one integer representing the heat of each field in the class. Initially all fields have a heat of zero. OOR analysis uses the algorithm in Figure 5.2 to set the heat value for each field and thus identify hot fields to the collector. The OOR collector then copies and enqueues the hot fields first. Figure 5.3 shows how the collector copies data. For a nursery collection, it begins by processing the remembered sets (these are empty in a full heap collection), and then processes the roots. ADVICE-PROCESS() places all uncopied objects (line 2) in the copy buffer, and updates the pointer for already copied objects. ADVICE-SCAN()) then copies all the hot fields first (line 3), and enqueues the remaining fields to process later. Without advice, all fields are cold.

We also experiment with using a *hot space* that segregates hot objects from the others to increase their spatial locality, which should improve cache line utilization, reduce bus traffic, and reduce paging. We refine hot objects to *hot referents*—instances referred to by hot fields, and *hot parents*—instances

of classes that instantiate hot fields. When copying an object, it is identified as a hot parent if the *hasHotField* value of the object's class is true. Hot referents are discovered when traversing hot fields. The *hot space* contains all the hot objects and is part of the older space; during nursery garbage collection, the collector copies into the hot space all objects that contain hot fields and the objects to which the hot fields point. During older generation collection, the collector copies objects in the hot space to a new hot space. It always copies all other objects into a separate space in the older generation. We do not need to change the write barrier to add a hot space since we always collect it at the same time as other objects in the older generation. Therefore, this change does not influence write barrier overhead in the mutator.

An advantage of advice-directed traversal is that it is not exclusive. For those objects without advice, we can use the best static traversal order available to combine the benefit of both methods. In our current implementation, the default copy order is pure depth first for cold objects, last child first, because this static order generally generates good performance, as we will show in the following section.

5.2 Experimental Results

We now present evaluation of our online object reordering system. We begin with results that show that the overhead for the reordering analysis, including its use by the collector, adds at most 1 to 2% to total time. We then show some programs are sensitive to copying order. Comparisons with

```
ADVICE-BASED-COPYING()
 1  Objects ← EMPTYQUEUE()
 2  Cold ← EMPTYQUEUE()
 3
 4  for each location in Remsets do
 5    ADVICE-PROCESS(location)
 6  for each location in Roots do
 7    ADVICE-PROCESS(location)
 8  repeat
 9    while Objects.NOTEMPTY() do
10      ADVICE-SCAN(Objects.DEQUEUE())
11    while Cold.NOTEMPTY() do
12      ADVICE-PROCESS(Cold.DEQUEUE())
13  until Objects.ISEMPTY()

ADVICE-PROCESS(location)
1  obj ← *location
2  if NEEDSCOPYING(obj) then
3    Objects.ENQUEUE(COPY(obj))
4  if FORWARDED(obj) then
5    *location ← NEWADDRESS(obj)

ADVICE-SCAN(obj)
1  for each field in obj.fields() do
2    if field.isHot(location)   // advice
3      then ADVICE-PROCESS(obj.field)
4      else Cold.ENQUEUE(obj.field)
```

Figure 5.3: Pseudocode for Advice Based Copying

OOR show that it essentially matches or improves over the oblivious orders. A series of experiments demonstrates the sensitivity of OOR to the decay of field heat to respond to phase changes, the use of a hot space, cold block analysis, and hot method analysis. We also compare OOR with class-oblivious copying on three additional architectures. Static ordering performance is not always consistent across architectures. However, OOR consistently attains essentially the same performance as the best static order across these platforms.

5.2.1 Experimental Platform

We perform our experiments on four platforms and find similarities across these. Section 5.3 reports on cross architecture results. For brevity

and unless otherwise noted, we report experiments on a 3.2GHz Pentium 4 machine described in Section 4.5

We instrument MMTk and Jikes RVM to use the CPU's performance counters to measure cycles, retired instructions, L1 and L2 cache misses, and TLB (translation look-aside buffer) misses of both the mutator and collector, as we vary the collector algorithm, heap size, and other features. Because of hardware limitations, each performance counter requires a separate execution. We use version 2.6.5 of the *perfctr* Intel/x86 hardware performance counters for Linux with the associated kernel patch and libraries [57].

5.2.2 Overhead of Reordering Analysis

To explore the overhead of the analysis, we measure the first iteration of the benchmark (where the compiler is active) with the adaptive compiler on a moderate heap size (1.8 $\times$ maximum live) and pick the fastest of 5 runs. We consider the fastest run to be the one which has the least disturbance from other factors in the system, therefore we can measure only the impact of our changes. This experiment performs the additional run-time work to record hot class fields and examines the results at collection time, but it never acts on those results. Therefore, the system does all the work of class reordering, but obtains no benefit from it. Table 5.1 compares the original adaptive system with the augmented system. The table shows some improvements as well as degradations. At worst, OOR adds a 2% overhead, but this overhead is obscured by larger variations due to the timer-based sampling. For the exact same program, VM, and heap size, the timer-based sampling can cause

Benchmark	Default	OOR	Overhead
jess	4.39	4.43	0.84%
jack	5.79	5.82	0.57%
raytrace	4.63	4.61	-0.59%
mtrt	4.95	4.99	0.7%
javac	12.83	12.70	-1.05%
compress	8.56	8.54	-0.2%
pseudojbb	13.39	13.43	0.36%
db	18.88	18.88	-0.03%
antlr	0.94	0.91	-2.9%
gcold	1.21	1.23	1.49%
hsqldb	160.56	158.46	-1.3%
ipsixql	41.62	42.43	1.93%
jython	37.71	37.16	-1.44%
ps-fun	129.24	128.04	-1.03%
mean			**-0.19%**

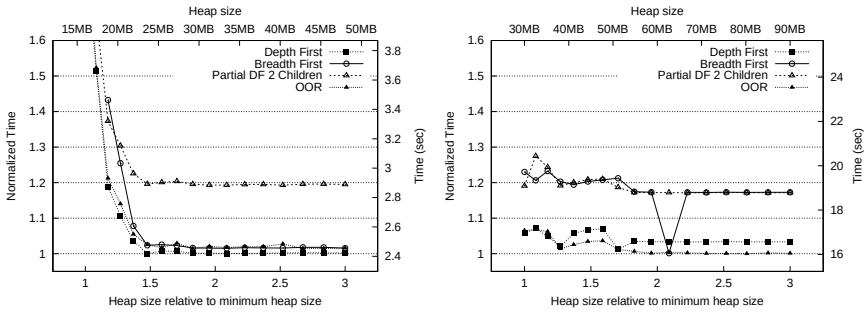
Table 5.1: OOR System Overhead

variations up to 5% because of the non-determinism, and this variation is the dominant factor, not the OOR analysis.

5.2.3 Class Sensitive vs. Class Oblivious

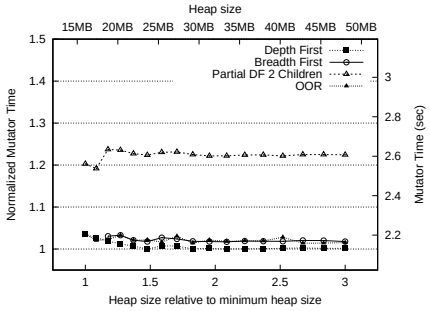
This and all remaining sections apply the compiler-replay methodology, reporting only application behavior. This section compares static and OOR copying orders. OOR uses a hot space (Sections 5.1.3 and 5.2.5), the decay function described in Section 5.1.2, and excludes field accesses from cold blocks (Sections 5.1.1 and 5.2.6). This configuration produces the best results across all architectures.

Most of the benchmark programs vary due to copy order by less than 4%. However, four programs (jython, db, jess, and javac) show variations of up to 25% due to copying order, so we focus on them. Figure 5.4 (jess) and Figure 5.5 (jython, db, javac) compare OOR with three static, class-oblivious

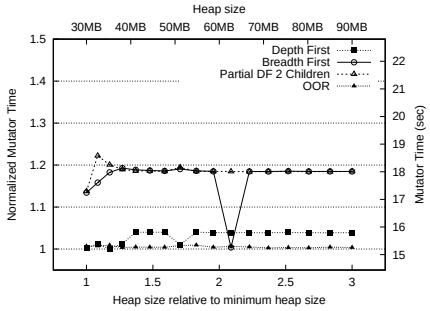


(a) jess Total Time

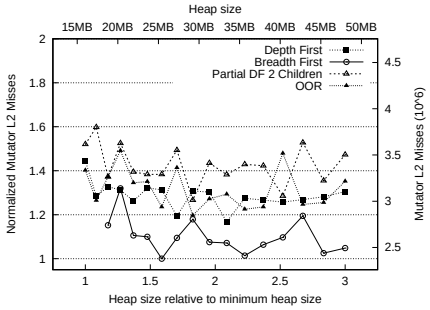
(b) jython Total Time



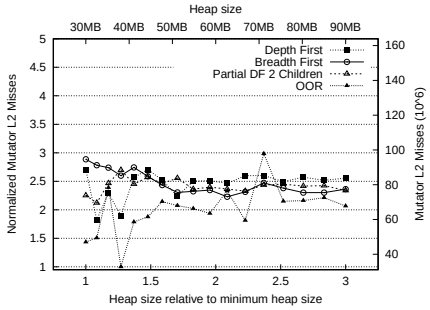
(c) jess Mutator Time



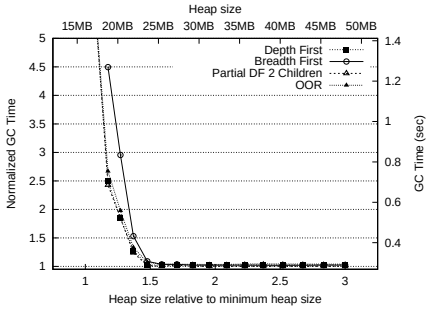
(d) jython Mutator Time



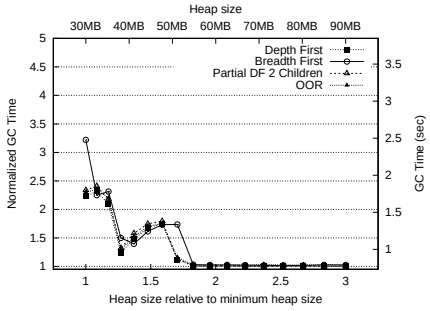
(e) jess L2 Mutator Misses



(f) jython L2 Mutator Misses



(g) jess GC Time



(h) jython GC Time

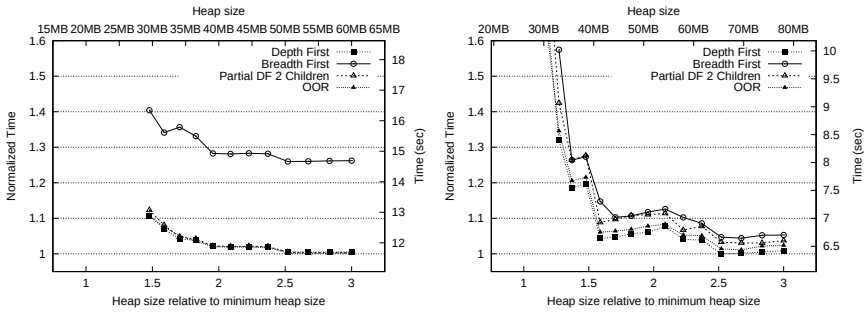
Figure 5.4: OOR vs. Class-Oblivious Traversals [jess & jython]

orders: breadth first, depth first, and partial depth first using the first two children (a hierarchical order). The figures present total time, mutator time, mutator L2 misses (from performance counters), and garbage collection time. Notice that the total time of jess and javac and the mutator L2 misses of jython use scales different from the other benchmarks in the figures.

First consider variations due to a priori breadth or depth first on db and jython (Figure 5.5). In db, class-oblivious depth first and partial depth first using the first two children perform over 25% better in total time than breadth first copying order. For jess (Figure 5.4), partial depth first is more than 20% worse than breath first. For jython, depth first performs about 18% better than breadth first and partial depth first. Locality explains these differences as shown in the mutator time and L2 miss graphs. For a few other programs, partial depth first offers a minor improvement (1 to 4%) over the best of breadth or depth first. The wide variation in performance is a pathology of static copying orders and is of course undesirable.

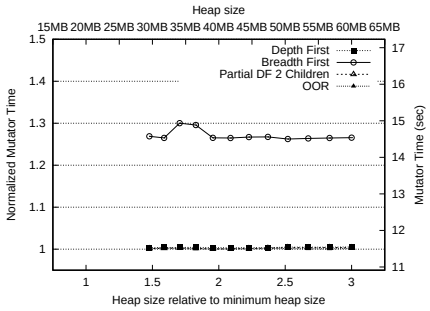
Figures 5.4 and 5.5 show that OOR is not subject to this variation and matches or improves over the best static orders. In javac and jess, OOR sometimes degrades mutator time by 2 to 3% which degrades the total performance by 2%. The worst case for OOR on all benchmarks and platforms is 4% degradation for ipsixql on the 3.2 GHz P4. For all other benchmarks on these architectures, OOR matches or improves over the best mutator locality and total performance.

These results are consistent with cache and page replacement algo-

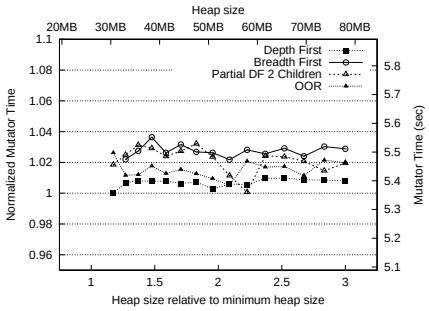


(a) db Total Time

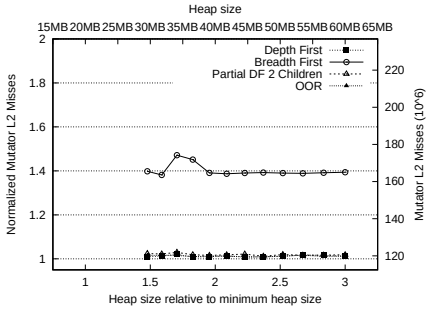
(b) javac Total Time



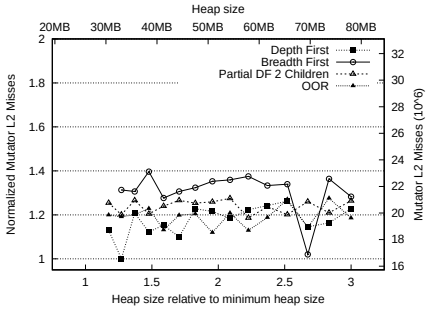
(c) db Mutator Time



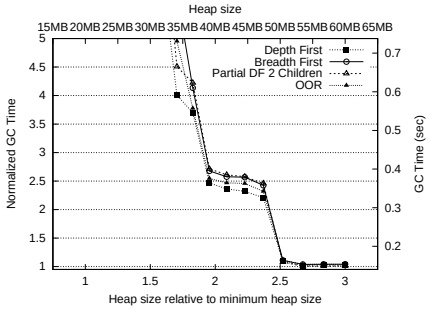
(d) javac Mutator Time



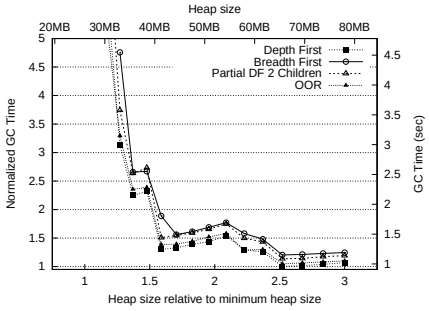
(e) db L2 Mutator Misses



(f) javac L2 Mutator Misses



(g) db GC Time



(h) javac GC Time

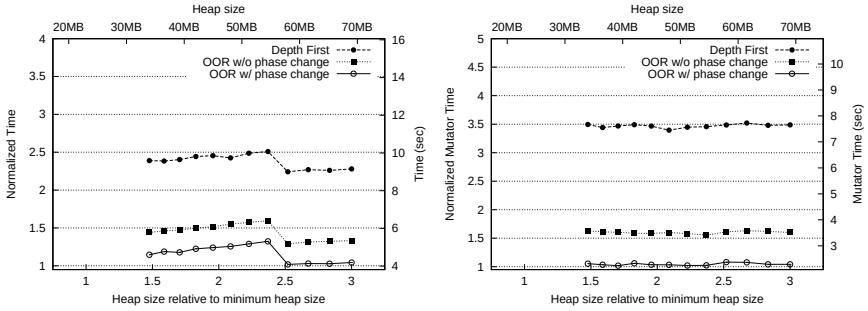
Figure 5.5: OOR vs. Class-Oblivious Traversals [db & javac]

rithms, among others, that use past access patterns to predict the future. OOR dynamically tunes itself to program behavior and thus protects copying garbage collection from the high variations that come from using a single static copying order that may or may not match program traversal orders.

5.2.4 Capturing Phase Changes

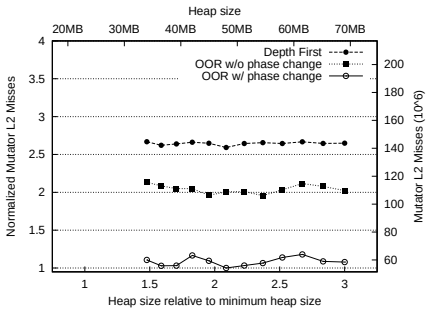
OOR can adapt to changes *within* the execution of a given application. Section 5.1.2 describes how the decay model ensures that field heat metrics adapt to changes in application behavior. We now examine the sensitivity of this approach. We use a synthetic benchmark, phase, which exhibits two distinct phases. The phase benchmark repeatedly constructs and traverses large trees of arity 11. The traversals favor a particular child. Each phase creates and destroys many trees and performs a large number of traversals. The first phase traverses only the 4th child, and the second phase traverses the 7th child.

Figure 5.6 compares the default depth first traversal in Jikes RVM against OOR and OOR without phase change detection on the phase benchmark. Phase change detection improves OOR total time by 25% and improves over the default depth first traversal by 55%. Mutator performance is improved by 37% and 70% respectively (Figure 5.6(b)). Much of this difference is explained by reductions in L2 misses of 50% and 61% (Figure 5.6(c)). Figure 5.7 compares OOR with and without phase change detection on jess, jython, javac, and db. These and the other benchmarks are insensitive to OOR's phase change adaptivity, which indicates that they have few, if any, traversal order



(a) phase Total Time

(b) phase Mutator Time



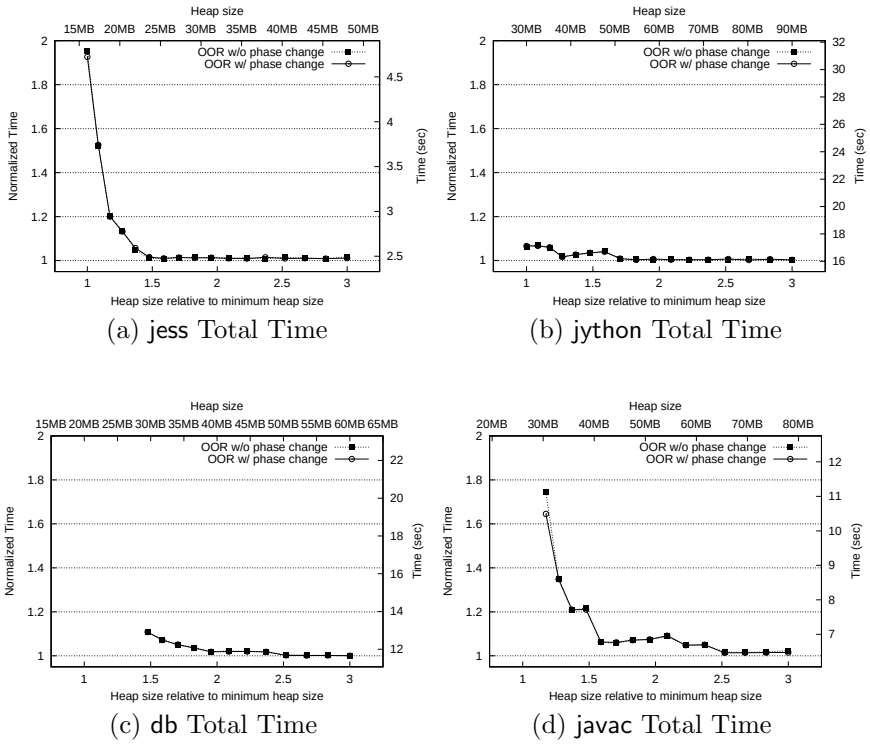
(c) phase Mutator L2 Misses

Figure 5.6: Performance Impact of Phase Changes Using a Synthetic Benchmark

phases.

5.2.5 Hot Space

In order to improve locality further, OOR groups objects with hot fields together in a separate copy space within the mature space, as described in Section 5.1.3. Figure 5.8 shows results from four representative benchmarks for OOR with and without a hot space. On average, these configurations perform similarly. However, in our experiments for other platforms, we found OOR with the hot space usually has slightly better results (see Figure 5.11(b) in Section 5.3). The hot space generally reduces the footprint of the hot objects



(a) jess Total Time (b) jython Total Time

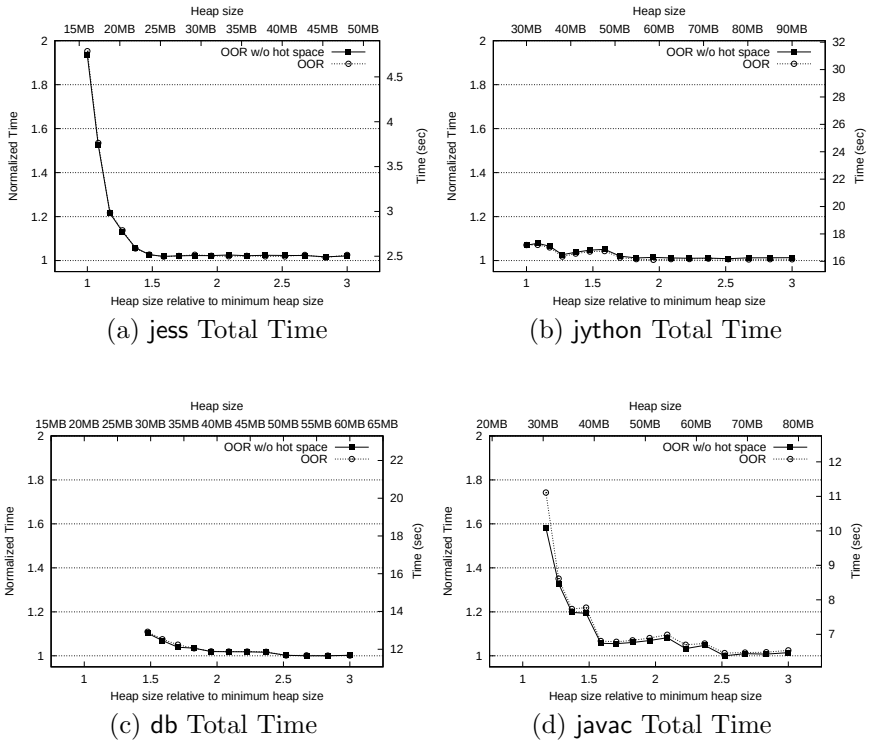
(c) db Total Time (d) javac Total Time

Figure 5.7: Absence of Phasic Behavior in Standard Benchmarks

but this benefit is not as significant as copying order.

5.2.6 Hot Field Analysis

We now explore the impact of the Jikes RVM static analysis thresholds for basic block heat on OOR (see Section 5.1.1). The Jikes RVM optimizing compiler assigns a heat value to basic blocks based on static loop iteration estimates (or counts if available) and branches. It then classifies them as hot or cold based on a run-time configuration threshold. OOR directly uses this classification to enumerate field accesses in hot basic blocks. The default configuration marks the fewest blocks cold (BB1 in Figure 5.9). BB20 through



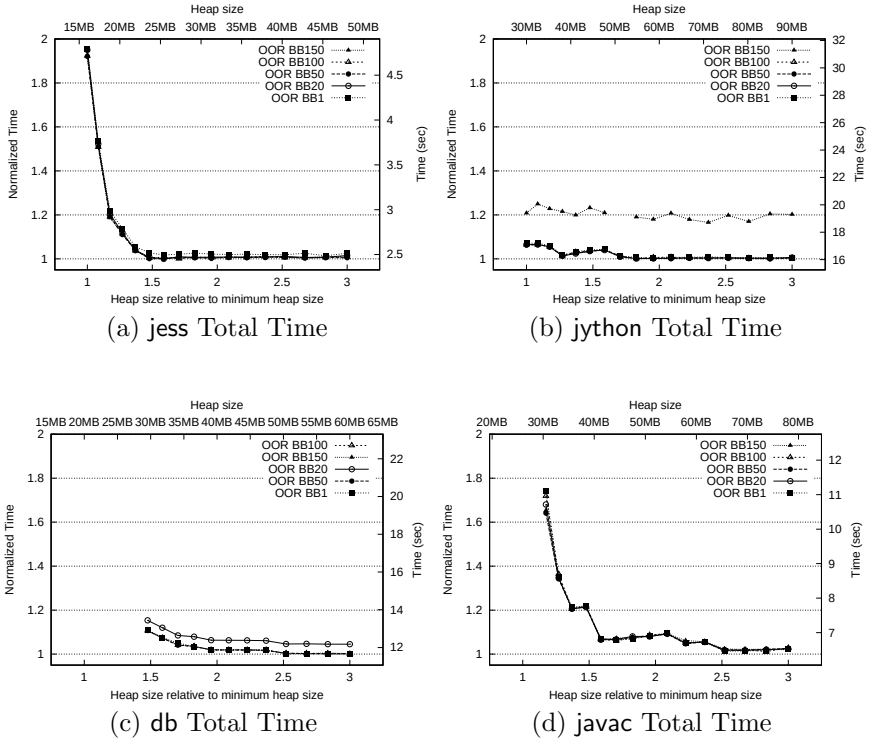
(a) jess Total Time (b) jython Total Time

(c) db Total Time (d) javac Total Time

Figure 5.8: OOR without Hot Space

BB150 mark increasingly more basic blocks cold. Figure 5.9 presents the sensitivity of OOR to this threshold. Most of the benchmarks, including jess and javac, are fairly insensitive to it, but jython is particularly sensitive, with a worst case degradation of 20%. For db, when OOR marks only basic blocks with heat greater than 20 as hot, the program has the worst performance. One possible explanation is that this threshold causes OOR to distribute an important data structure between the hot and cold spaces. With thresholds higher and lower than 20, OOR probably tends to put the whole data structure in one space or the other. Based on these results, we use the Jikes RVM default

and mark as hot any basic block with a heat greater than or equal to one.



(a) jess Total Time (b) jython Total Time

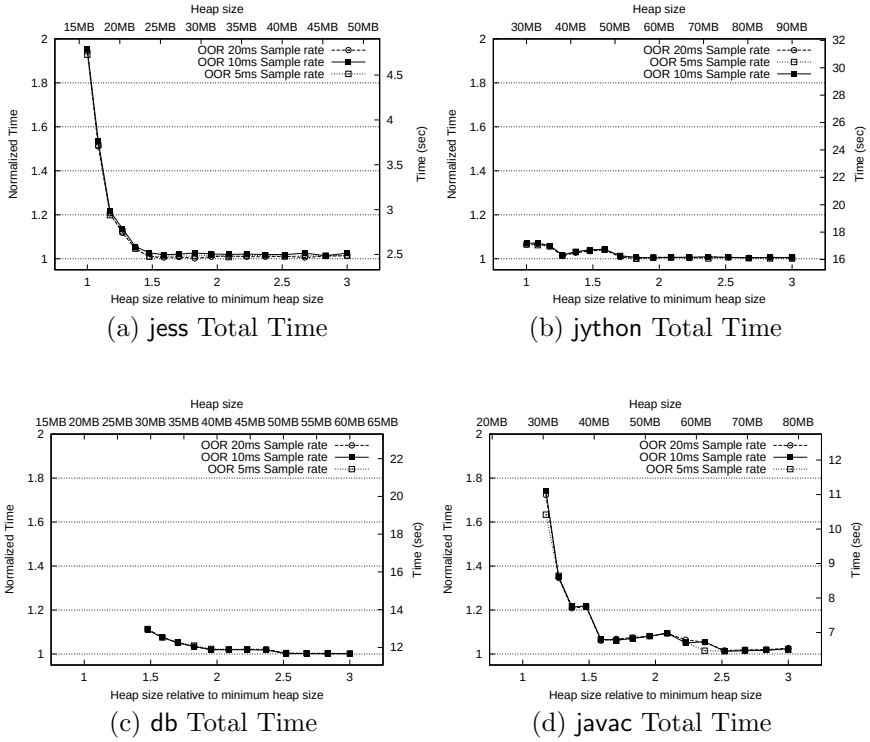
(c) db Total Time (d) javac Total Time

Figure 5.9: Using Different Policies to Determine Cold Fields

5.2.7 Hot Method Analysis

Finally, Figure 5.10 examines the sensitivity of the sampling frequency for selecting hot methods. Hot methods are identified according to the number of times the adaptive optimization infrastructure samples them. Figure 5.10 shows OOR with sampling rates of 20ms, 10ms, and 5ms. More frequent sampling marks more methods as hot. OOR is quite robust with respect to this threshold. One possible explanation for this insensitivity is that method

heat tends to be bimodal—methods are either cold or very hot. Another explanation is that warm methods (those neither hot nor cold) tend not to impact locality through field traversal orders.



(a) jess Total Time (b) jython Total Time

(c) db Total Time (d) javac Total Time

Figure 5.10: Using Different Policies to Determine Hot Methods

5.3 Different Platforms

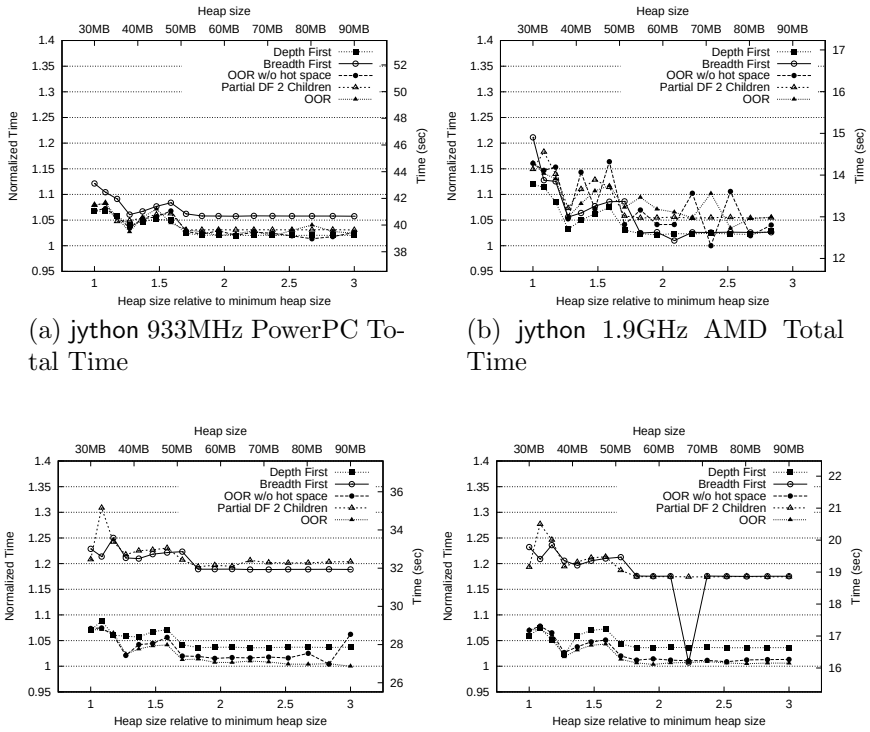
This section examines the sensitivity of OOR to architecture variations, including processor speed and memory system. We run the same experiments as before on an three additional architectures (933MHz PPC, 1.9GHz AMD, and 2.4GHz P4 as described in Section 4.5.

We present a representative benchmark with variations due to locality. Figure 5.11 shows jython on all four architectures. Not surprisingly, the two Intel Pentium 4 architecture graphs have very similar shapes and the 3.2GHz P4 is faster. Comparing between architectures shows that the memory architecture mainly dictates differences among traversal orders. The 1.9GHz AMD and 933MHz PPC are less sensitive to locality because they have larger and relatively faster caches compared to the P4s which have higher clock speeds. Interestingly, the slower AMD processor achieves the best, performance, possibly due to its large non-inclusive caches. However, on all four architectures, OOR consistently provides the best performance, across all benchmarks and architectures.

5.4 The Copying Advantage

We now present evidence confirming the locality advantages of copying. We first examine mutator locality by comparing a standard copying collector with a non-copying mark-sweep collector. We then compare the mutator time of a non-copying mark-sweep collector with the total time of the copying collector to see whether the benefits of copying can ever outweigh the cost of garbage collection.

Figure 5.12(a) compares just the mutator performance of the bounded (4MB) nursery generational copying collector using OOR to a whole heap mark-sweep collector [12], labeled OOR and Mark-Sweep respectively. The figure shows mutator time as a function of heap size for javac, a represen-



(a) jython 933MHz PowerPC Total Time

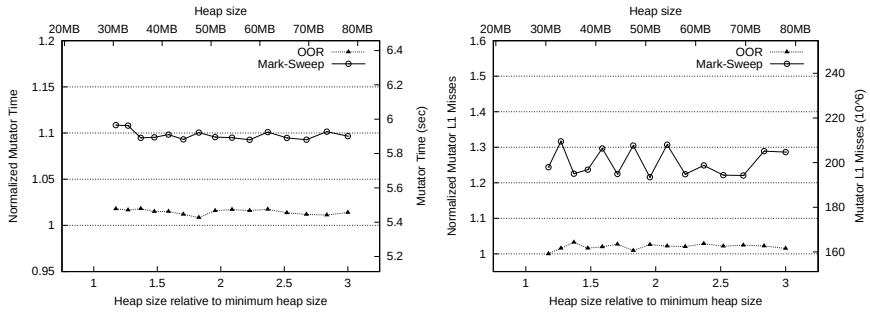
(b) jython 1.9GHz AMD Total Time

(c) jython 2.4GHz P4 Total Time

(d) jython 3.2GHz P4 Total Time

Figure 5.11: Performance on Different Architectures

tative program. OOR has a mutator-time advantage of around 8-10% over Mark-Sweep due to fewer L1 misses on javac (Figure 5.12(b)). The L2 and TLB misses follow the same trend, and this advantage holds across all of our benchmarks, ranging from a few percent on jython and compress, to 15% on pseudojbb and 45% on ps-fun. Our analysis confirms a prior result [11]: it is *locality* rather than the cost of the free-list *mechanism* that accounts for the performance gap. Note that this result is contrary to the oft-heard claim that non-moving collectors 'disturb the cache less' than do copying collectors.



(a) Mutator Time (b) Mutator L1 Cache Misses

Figure 5.12: Mutator Performance for Copying and Mark-Sweep Collectors on javac

We now examine the overall impact of garbage collection when the locality advantage and collection overhead are combined. Figure 5.13 compares the *total* execution time of the copying collector (labeled OOR) with the *mutator* time of mark-sweep (Mark-Sweep), which we regard as an approximation to the performance of explicit memory management. We use a standard free-list allocator [11, 49] and subtract the cost of garbage collection. The approximation is imperfect. On one hand, the application does not pay the cost of `free()`. On the other hand, it does not reclaim memory as promptly as explicit memory management does. In both graphs, the performance of the copying collector is normalized against the mutator time for Mark-Sweep. A result less than 1 indicates that the total time for the copying collector is less than the Mark-Sweep mutator time.

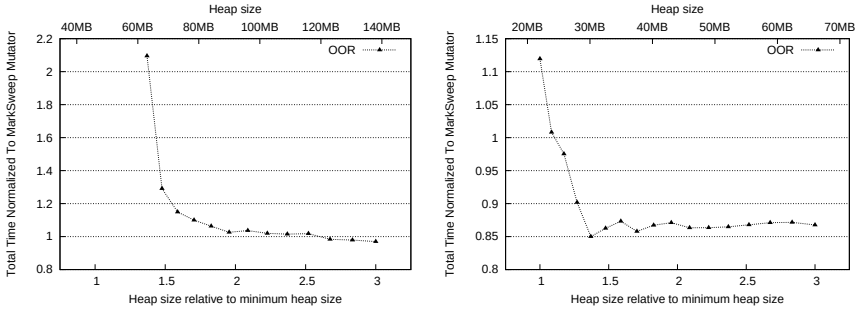
Three patterns emerge in our results. Figure 5.13(a) shows three representative benchmarks: pseudojbb, ps-fun, and ipsixql. ipsixql is the only outlier where the Mark-Sweep mutator actually has consistently better performance

than the copying collector. Seven benchmarks are like pseudojbb. In modest to large heaps, the locality advantage of copying garbage collection compensates for its collection costs, to the point where the total time of OOR is the same as the mutator time of Mark-Sweep. ps-fun represents five benchmarks, where the locality advantage is so significant that OOR improves over the mutator time of Mark-Sweep, even in small heap sizes. Figure 5.13(b) is remarkable because it shows that for one of the largest and most realistic benchmarks in our suite, garbage collection produces a net performance *win*. These results stand against the conventional wisdom that garbage collection always comes at a performance price.

5.5 Summary

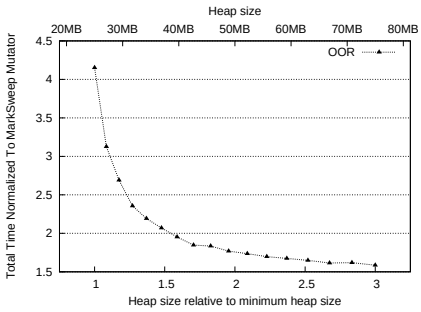
We show that the performance of class-oblivious traversal orders can be unpredictable and expose programmers to variations outside of their control. We show that our online object reordering system eliminates copying order *gambling*. It has a negligible overhead, is amenable to the virtual machine context, and adaptively matches or improves over the best static, class-oblivious order for a given program. Our results show that most of the performance benefit of applying OOR comes from the reduction of L2 cache misses.

Common wisdom holds that the software engineering benefits of garbage collection come with a performance penalty. We show that copying collectors have a locality advantage over the free-list organizations of explicitly managed memory. Copying collectors achieve good locality by placing contemporane-



(a) Total Time, pseudojbb

(b) Total Time, ps-fun



(c) Total Time, ipsixql

Figure 5.13: Garbage Collection vs. Idealized Mark-Sweep

ously allocated objects together in memory and copying connected objects together in the mature space. OOR further adapts copying order to program access patterns. Since future processors will demand locality to achieve high performance, we can look forward to a future where garbage collection combines software engineering and performance benefits.

Chapter 6

Dynamic Systems for Improving Instruction Locality

Now we change our focus to improving instruction cache locality. The importance of the instruction cache to Object-oriented programming languages is not clear from previous work, therefore we conduct a series of studies to understand the potential of performance improvement by improving instruction cache and the different types of instruction cache misses for Java programs. After that, we describe two approaches that we implemented for improving instruction locality. (1) One approach is to reorder all the code in the heap using a greedy algorithm after the warm-up phase of the program run. This approach is suitable for long running applications (like server application). We developed several new algorithms for full code layout algorithm in Section 6.3 (2) An incremental approach calculates the position of a method at allocation time using runtime information. This approach incrementally changes the code layout to improve code footprint and to avoid conflict misses in direct-mapped instruction caches. This second approach generates negligible overhead so it is suitable for even short-running applications. To our best knowledge, this work is the first to perform dynamic code reordering for JIT compilation.

6.1 Different Types of Instruction Cache Misses

Figure 6.1 reports data for instruction cache accesses. From the graph, we can see that for javac, there are more cache misses in the instruction L1 cache than in the data L1 cache. This result holds for all the SPECjvm98 benchmarks except db. Although all the programs we measured have better instruction locality (lower cache miss rate) than data locality, the absolute number of instruction cache misses is sometimes higher than data cache misses because there are usually 2 to 3 times more instruction accesses than data accesses. Therefore, there is a significant performance loss due to poor instruction locality as well as data locality.

To find out what kind of optimizations for improving instruction locality are effective for Java programs, we try to separate the instruction cache misses into conflict misses and capacity misses. We measure the impact of conflict misses in the instruction cache by comparing the results of a direct-mapped cache with a fully associative cache. Also, to find out how hard it is to remove the conflict misses, we use a two-way set associative cache for comparison. Comparing a direct-mapped cache to a fully associative cache, most benchmarks show a large reduction in misses. For example, we are able to reduce 83% of the instruction misses for jess. The detailed numbers are in Table 6.1.

The results show that, except for jess, using two-way set associative cache can reduce most of the conflict misses and approaches the full-associative cache. Although the improvement on reducing cache misses is significant,

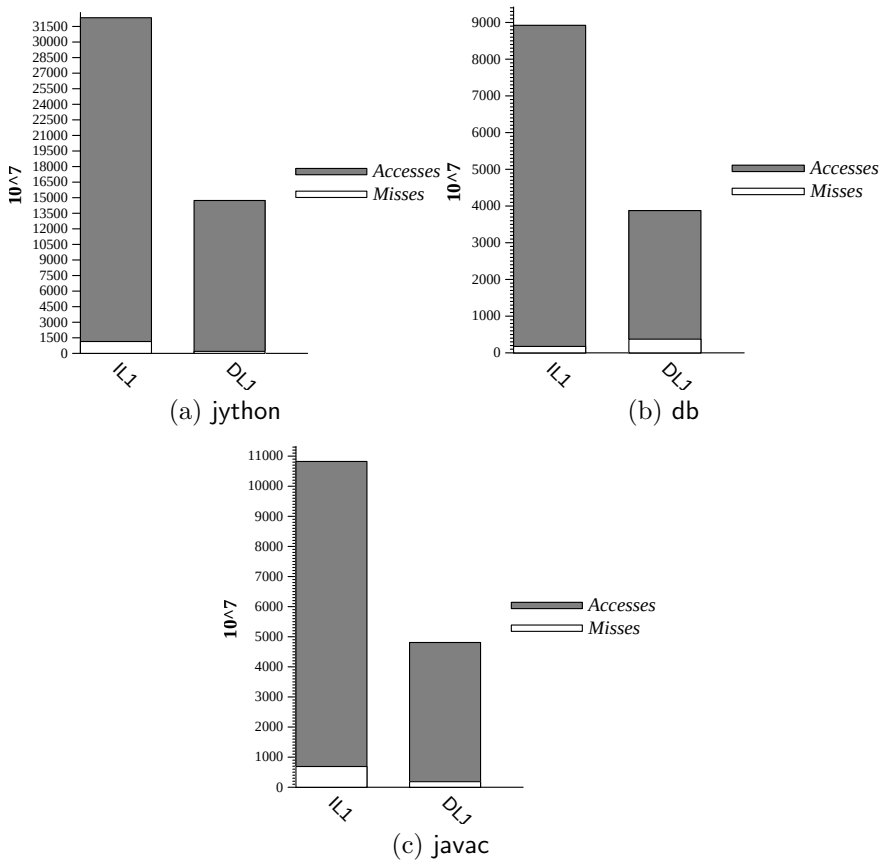
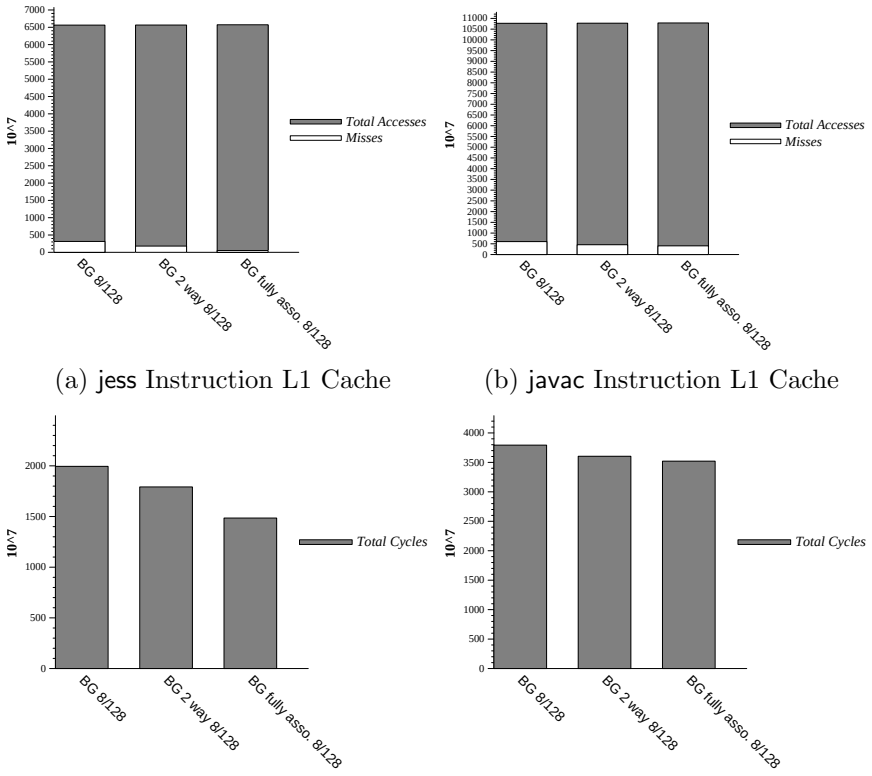


Figure 6.1: Instruction L1 vs. Data L1

the performance of the program does not always improve as much as the improvement in cache misses because half of the programs do not have a lot of misses even using a direct-mapped cache. If we use the same latency for all three cache configurations, jess has the most improvement of 25% and javac has 8% in total cycles . All other benchmarks have little improvement (less than 5%). The results of instruction cache accesses and total cycles for javac and jess is shown in Figure 6.2.

Benchmark	Direct-mapped	Two-way	Fully-asso.
jess	314	180	52
javac	603	457	405
jack	277	243	212
mtrt	137	117	89
compress	14	-	4
db	45	12	5

Table 6.1: Number of Conflict Misses (10^7)



(a) jess Instruction L1 Cache (b) javac Instruction L1 Cache

(c) jess Total Cycles (d) javac Total Cycles

Figure 6.2: Direct-mapped vs. Fully Associative

This experiment shows that the potential performance loss from poor instruction cache locality is significant and the locality can be improved relatively easily (two-way set associativity can remove more than half of the loss, but it will not solve the entire problem).

6.2 Whole Program and Incremental Code Layout

In the previous section, we find that there are usually more L1 instruction cache misses than L1 data cache misses, partly because the instruction cache is direct-mapped. Also, the JVM treats code objects just like any other objects. Unlike C and other languages, Java code can be allocated anywhere in the heap and mixed with data. Therefore there is a larger chance of having conflict misses among Java code if code allocation is not carefully implemented. In addition to reordering code during allocation, we can also reorder code after it has been allocated at runtime to improve locality just like the data objects.

In Section 3, we show that previous work mostly improves full code layout. But the algorithms are too expensive to apply dynamically, even for long running applications. We developed new algorithms which are orders of magnitude faster and produce better layouts, which make them more suitable for long running programs. These algorithms are presented in Section 6.3

Because full code layout is expensive for modest to short-running benchmarks, we developed an online Partial Code Reordering (PCR) system to perform a procedure-level, low overhead, and dynamic code allocation and reordering. PCR has two major components: avoiding code conflicts during code allocation and code reordering to remove code conflicts at runtime. The first component to reduce instruction working set size during code allocation by utilizing method execution frequency information at runtime. The second component, reordering code at runtime, includes three steps: (1) adaptive sampling to generate dynamic call graphs with weighted edges to represent the

heat, (2) runtime analysis for detecting conflicts among procedures in the call graph, and (3) code reordering to remove conflicts among procedures, during recompilation of hot methods.

In the rest of this section, we will describe full code layout generation system in Section 6.3 and the low overhead systems for short-running programs in Section 6.4 in greater details.

6.3 Whole Program Dynamic Code Management System

This section describes the implementation of a Whole Code Management system (WCM) that is integrated into a managed runtime. WCM uses dynamic profile information to reorganize the compiled code at the granularity of a method. We show that WCM can significantly improve performance. We also describe three new procedure layout algorithms that, compared to previous approaches, reduce the cost of computing a new code placement. These algorithms specifically target ITLB misses, which typically have the greatest impact on performance because of their frequency and high cost. One of these algorithms, Code Tiling, is significantly faster both in worst case complexity and in practice than the best-known previous technique by Pettis and Hansen [58]. We demonstrate that Code Tiling generates code layouts that are better or comparable to those by the Pettis-Hansen algorithm.

6.3.1 Whole Code Management overview

This section overviews the Whole Code Management system (WCM), and the following section describes its implementation and design choices.

As the application executes and its behavior changes, WCM reorganizes compiled code as necessary. When miss rates for the ITLB or instruction cache become too high, it calculates a new layout, moves method code, and updates code pointers and offsets in methods, thread stacks, registers, and data structures of the managed runtime to reflect the new locations. The miss rates for the ITLB or instruction cache can be detected by hardware performance counter or software estimation. WCM currently uses user-defined execution point for code reorganization. Different schemes for detecting program instruction locality behavior are beyond the scope of this work.

WCM gathers profile information on callers and callees and uses it to build a dynamic call graph. The dynamic call graph (DCG) is an undirected graph with a node for each method, and edges from a method to any methods it invokes, weighted by their dynamic frequency. When the system triggers a reorganization, WCM computes a new placement for each method's code based on the DCG. WCM then moves the code and does any required code updates. Figure 6.3 depicts WCM's components and their interactions.

Software instrumentation or hardware performance monitoring unit (PMU) can provide the dynamic call graph profiles. WCM can use one of a number of different code layout algorithms. The Pettis-Hansen procedure layout algorithm is one, and we describe three others in Section 6.3.3. These

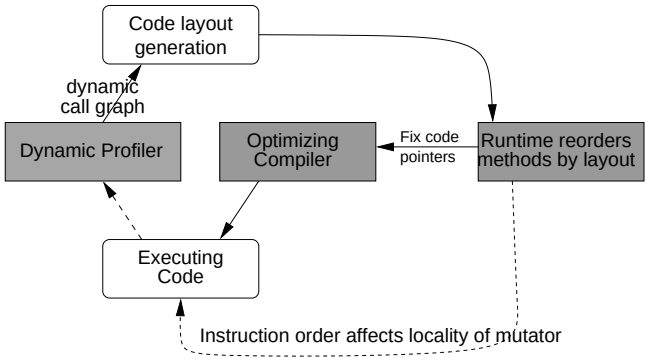


Figure 6.3: Whole Code Management

algorithms attempt to improve performance by, for example, reducing the number of frequently executed code pages to minimize ITLB misses.

To minimize the number of managed runtime components that need to understand JIT-compiled code, WCM relies on the JIT to make any necessary adjustments to relocated code. This division gives JITs more freedom in how they emit code, and the code management system and the rest of the managed runtime do not need to understand the representation of JIT-compiled code.

6.3.2 WCM Implementation on IA-32

We implemented WCM in a managed runtime environment that supports Java and C# programs. Our core platform consists of the Open Runtime Platform virtual machine (ORP) [25] and one or more JIT compilers. On IA-32, we use the optimizing O3 JIT [26] to compile JVM bytecodes. This JIT performs inlining, a number of global optimizations (e.g., copy propagation, dead code elimination, loop transformations, and constant folding), as well as common subexpression elimination and array bounds check elimination.

ORP allocates compiled code in a region of memory that is separate from the garbage collected heap. It provides different subregions for code that is cold (infrequently executed) and hot (more often executed). ORP allocates code of equal "temperature" sequentially. Our IA-32 O3 JIT emits a single block of code for each compiled method. As a result, the granularity of reorganization for the IA32 is methods; for the IPF, it is a *code block* which typically divides a method into two parts: one for its hot basic blocks and another for its cold ones. We return to the IPF implementation in Section 6.3.5, but the remainder of this section and the next two discuss our IA-32 implementation.

To support Whole Code Management, we modified O3 to emit relocatable code. This simplifies moving code during code reorganization, but usually requires updating pointers to compiled methods used in that code, including references into code of other methods as well as into the same method. WCM calls the JIT (here O3) to update code after it has moved it.

ORP can collect dynamic profile information from either software instrumentation or from hardware Performance Monitor Units (PMU) sampling. However, on the IA-32, we found that using the PMU is too expensive. In particular, capturing the LBR (last branch record) requires hundreds of cycles since it requires flushing the pipeline and performing memory fences. As a result, ORP uses software instrumentation on IA-32. We could have modified our JITs to do the instrumentation, but we chose a simpler approach. ORP interposes on method calls to record the caller and the callee. It does this by generating a small machine code stub for each compiled method that is exe-

cuted first whenever a call is made to the associated method. When entered, this stub records the caller/callee information and then transfers control to the start of the intended callee. This stub approach handles indirect calls and hot-cold method splitting. Our implementation of software instrumentation has the feature that it can be turned on or off, and when turned off has no impact at all on the application's execution.

To reorganize code, WCM performs the following steps:

1. Stop all managed threads.

2. Compute a new layout.

3. Allocate new code storage. It would be possible to reorganize code in place, but moving to a newly-allocated code region is simpler. It also simplifies debugging our WCM implementation, since it is easy to recognize a reference to an old code location.

4. For each method, move the code and call the JIT to fix it up. Also fix up the method's meta data recorded by the managed runtime.

5. Update the call stack of each thread. In particular, fix up any code addresses such as return addresses currently on stacks. Also, update any registers containing code addresses.

6. Restart the managed threads.

To compute the new layout, WCM uses one of several different code layout algorithms. Each of these operates on the dynamic call graph (DCG)

WCM Step	Cache-Aware Pettis-Hansen	Code Tiling
Compute new layout	28,459	417
Allocate new code space	5	5
Move and update code	90	87
Update thread stacks	1	1

Table 6.2: Breakdown of time to reorganize MiniBean's code (ms)

produced during profiling and creates a code layout. This layout identifies sequences of code that should be placed together in memory in that order. So far, we have implemented four code layout algorithms.

Our experience is that most of the time needed to reorganize code is due to the new layout calculation; WCM finishes the remaining steps quickly. To illustrate this, for MiniBean, the Code Tiling layout algorithm requires 417ms of the 510ms total reorganization time, while Cache-Aware Pettis-Hansen requires 28,459ms of the total 28,555ms. The other steps require about 100ms. These times are shown in Table 6.2.

6.3.2.1 Current Status

Our IA-32 WCM implementation currently reorganizes code at GC time for simplicity. Since our garbage collector stops all threads during a GC, we reorganize code then. Despite this implementation, WCM itself is completely independent of GC and could reorder code at any time.

We do not currently support a mechanism to automatically trigger code reorganization since we have not yet developed a technique to determine when it would be be productive to do so. In the future, we plan to enhance WCM's use of PMU information to monitor ITLB and other instruction-related misses

in order to determine when reorganizations are needed. Currently, the user specifies on the ORP command line the GC at which to reorganize code, and ORP invokes WCM at the end of that GC. Although this interim solution allows only a single reorganization, WCM itself is capable of reorganizing code multiple times. Eventually, we will use WCM to reorder code whenever necessary.

Our WCM implementation also stops application threads while it does all reorganization work. However, much of WCM's work—in particular, calculating the new code layout—could be done concurrently with application threads to minimize pause time. Those threads need to be stopped only during the update of the metadata and thread stacks.

6.3.2.2 Discussion

In many ways, WCM resembles a copying garbage collector. It moves objects (method code) and updates any pointers to those objects. It is intended to improve program locality, but that is also a partial goal of many garbage collectors including ones that compact the heap or place objects to improve their locality [41, 45]. WCM also supports pinning of objects that would be too hard or too expensive to relocate. For example, we pin methods containing Java `JSR` bytecodes since these bytecodes are relatively rare and the resulting code is complex. Like many garbage collectors, WCM could also do much of its work in parallel with application threads, even though it does not currently do this. New code layouts could be computed in parallel, for example. WCM could also reclaim no-longer-needed code: code that is currently not referenced

by any thread and not likely to be needed again.

6.3.2.3 Alternatives to WCM

One alternative to WCM is to use large pages for code which will reduce the number of ITLB entries and misses. Unfortunately, not all operating systems support large pages. For example, IA-32 versions of Windows do not support them. However, large pages will consume a larger portion of the virtual address space and may suffer higher fragmentation, which may be a problem if the virtual address space is relatively small. In addition, using large pages does not address instruction cache misses.

Another alternative to WCM is method recompilation. Many JITs support profile-based recompilation of frequently executed (hot) methods, or methods in which a significant amount of execution time is spent. If the managed runtime or JIT allocates code sequentially, these recompiled hot methods will tend to be located close together, which is likely to improve code locality. However, for very large applications with large instruction foot prints and many hot methods, the natural benefits of JIT compilation are unlikely to consistently provide good code locality.

6.3.3 The Code Layout Algorithms

This section describes the Pettis-Hansen and our new code layout algorithms. All the algorithms use the same underlying data structure, the dynamic call graph (DCG), and produce a new code layout. We first implemented Pettis-Hansen algorithm, and find it usually improved performance

of large applications. However, it is too expensive. Pettis-Hansen creates a new layout for SPEC JBB2000 (758 methods) in less than 150ms, but requires minutes for MiniBean (15,586 methods). This overhead leads us to develop three new, faster algorithms.

6.3.3.1 Pettis-Hansen Algorithm

Pettis-Hansen places methods using a greedy "closest is best" strategy from the original call graph. Each step combines two nodes in the DCG and specifies their code layout. Each of the call graph's nodes initially contains a single method. The algorithm repeatedly chooses an edge $A \rightarrow B$ of highest weight in the entire graph (i.e., greatest calling frequency), then merges the nodes and outgoing edges of A and B. It lays out the code in the new node using the heuristic described by Gloy and Smith [36] (line 7 of procedure MergeNodes in Figure 6.4). Pettis-Hansen finds the hottest call edge between two methods from node A and B in the original call graph, and then orders the merged methods to minimize the distance between these two methods. Here the distance is measured in bytes (details are in Gloy and Smith [36]'s paper). It leaves the ordering within A and B the same. When it merges outgoing edges, if two point to the same node C, it merges them and weights the edge with the sum of the original weights. The algorithm terminates when no edges remain. Figure 6.4 shows the pseudo-code for the Pettis-Hansen algorithm.

Time Complexity The complexity of finding the maximum edge is $O(n^2)$ since in the worst case, there are n edges connected to each node. The max-

```
PettisHansen(Graph)
1  while (edge ← HeaviestEdge(Graph))! = NULL
2  do (nodeA, nodeB) ← edge.getNodes()
3      MergeNodes(nodeA, nodeB);

HeaviestEdge(Graph)
1  maxEdge ← NULL
2  for each node in Graph do
3    for each edge in node.edgeList do
4      if (maxEdge = NULL)||(edge.heat > maxEdge.weight) then
5        maxEdge ← edge
6  return maxEdge

MergeNodes(nodeA, nodeB)
1  for each edgeB in nodeB.edgeList do
2    for each edgeA in nodeA.edgeList do
3      if edgeB.equals(edgeA)
4        then edgeA.weight ← edgeB.weight + edgeA.weight
5             RemoveEdge(edgeB)
6  nodeA.edgeList.attach(nodeB.edgeList)
7  nodeA.blockList.GSattach(nodeB.methodList)
```

Figure 6.4: Pettis-Hansen procedure layout algorithm

imum number of edge merges for each node merge is also $O(n^2)$. So the asymptotic time complexity of the algorithm is $O(n*(n^2+n^2)) = O(n^3)$. This complexity explains the dramatic increase in time required to place the code for large applications such as MiniBean.

There are data structures such as priority queue and Fibonacci heap that can speed up searching for the maximum edge and inserting the new edges generated by merging. However, these will not help Pettis-Hansen much since the most expensive part of that algorithm is the edge merge.

6.3.3.2 Cache-Aware Pettis-Hansen Algorithm

In our search for a faster placement algorithm, we realize that there is little locality benefit in putting two methods on different pages, even if the two methods are on consecutive pages. We modify Pettis-Hansen to stop merging

```
CacheAwarePettisHansen(Graph)
1  while (edge ← HeaviestEdge(Graph))! = NULL
2  do (nodeA, nodeB) ← edge.getNodes()
3     if (nodeA.Size() > PAGE_SIZE)||(nodeB.Size() > PAGE_SIZE)
4        then RemoveEdge(edge)
5        else MergeNodes(edge, nodeA, nodeB)
```

Figure 6.5: Cache-Aware Pettis-Hansen algorithm

methods into the current code after enough methods have been merged to fill a page. We find that, with this optimization, the total time to calculate a layout is reduced by a factor of 10. This new *Cache-Aware Pettis-Hansen algorithm* is shown in Figure 6.5.

Cache-Aware Pettis-Hansen may generate a different layout from Pettis-Hansen. For example, Pettis-Hansen generates a layout of $DABC$ for the DCG in Figure 6.6 (merging order: AB, DAB, $DABC$). But if node A and node B are both larger than the page size, the new algorithm generates layout $ABCD$. Note here that C and D are adjacent, but are not with the original Pettis-Hansen layout. The different layout Cache-Aware Pettis-Hansen produces may or may not improve application performance. For example, assume that A and B are both two pages in size, and C and D are half a page. If every invocation of another method on a different page triggers a page fault, our layout generates fewer page faults than Pettis and Hansen because the method A and B would be on 4 pages instead of 5 pages in Pettis and Hansen layout. But with other node sizes, the result could be different. None of the four algorithms is guaranteed to produce the best layout. However, our main concern is the layout generation time, and we show in the next section that Cache-Aware Pettis-Hansen runs much faster.

A further refinement for Cache-Aware Pettis-Hansen is the following.

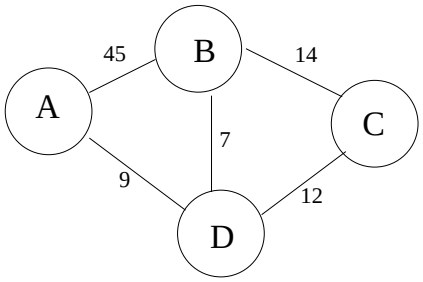


Figure 6.6: An example dynamic call graph

In a direct-mapped cache, we do not want two methods mapped on the same cache set if they frequently call each other. To avoid causing cache interference in direct-mapped instruction caches, the algorithm should use the cache size instead of the page size.

Time Complexity Complexity of the Cache-Aware Pettis-Hansen algorithm is the same as the Pettis-Hansen algorithm. However, this algorithm removes edges from the graph as it operates. As a result, in practice, finding the heaviest edge and merging the edges of two nodes are both less expensive than in the Pettis-Hansen algorithm.

6.3.3.3 Code Tiling Algorithm

A major cost in Cache-Aware Pettis-Hansen is finding the heaviest edge in the entire graph every time. To reduce this cost, we developed the *Code Tiling* algorithm that uses a simpler approximation. This algorithm traverses the nodes of the DCG one at a time. Assume the current node is A. As long as A's code occupies less than a page, it selects the heaviest edge $A \rightarrow B$ and merges A with the node B. If this algorithm merges any nodes, it produces a

```
CodeTiling(Graph)
 1  for  each node in Graph do
 2     currentNodeSize ← 0
 3     node.isVisited ← true
 4     STAY :
 5     maxEdge ← NULL
 6     for  each edge in node.edgeList do
 7        if isVisited(edge)
 8           then RemoveEdge(edge)
 9           else  if (maxEdge = NULL)||(edge.heat > maxEdge.heat)
10                    then maxEdge ← edge
11     if (currentNodeSize > PAGE_SIZE)||(maxEdge = NULL)
12        then RemoveEdge(maxEdge)
13        else  nodeB ← maxEdge.otherNode(node)
14              currentNodeSize ← currentNodeSize + nodeB.size()
15              MergeNodes(node, nodeB);
16               goto STAY

isVisited(edge)
 1  (nodeA, nodeB) ← edge.getNodes()
 2  return (nodeA.isVisited)&&(nodeB.isVisited);
```

Figure 6.7: Code Tiling algorithm

different layout than either Pettis-Hansen or Cache-Aware Pettis-Hansen since it only considers that part of the graph immediately connected to the current node. However, this layout may occasionally be better than Pettis-Hansen since it can give the best possible locality to the single hottest path, if one exists. Performance results in the next section show this algorithm computes good layouts and computes them faster.

Time Complexity If we have n nodes in the graph, we must scan n nodes. Since in the worst case, there are n out edges for each node, the time to find the heaviest edge for one node is still $O(n)$. However, in practice this case does not occur. The maximum number of edge merges requires $O(n^2)$. As a result, the worst case asymptotic complexity of the Code Tiling algorithm is $O(n * (n + n^2)) = O(n^3)$. It is the same as the Cache-Aware Pettis-Hansen algorithm, but since we avoid the work of repeatedly searching for the heaviest

edge in the whole graph, we expect layout generation to be faster than with the Cache-Aware Pettis-Hansen algorithm.

6.3.3.4 Linear Scan Algorithm

To further reduce the cost of generating a code layout from the dynamic call graph, we also tried a straightforward algorithm that has linear time complexity, the *Linear Scan algorithm*. In this algorithm, we scan each node in the graph in breadth-first traversal order, but we ignore cold edges (ones with weight less than some threshold). We show this algorithm in Figure 6.8. Notice that when AttachNodes merges two nodes, it does not merge their out edges: even if two edges connect to the same node, they are not merged. We do eliminate this step because merging edges is especially expensive. Notice also, that when merging one node B into another node A, AttachNodes simply attaches B's edge list to A without updating any of the edges in B's edge list. As a result, the edge data structure for B's edge list will still record B instead of the correct A. Not updating the data structure does not cause a problem because we never attach an already-visited node like B again. By reducing the work done during node merges, Linear Scan scans every edge exactly once and achieves linear time to the number of edges.

Time Complexity If there are n^2 edges (worst case for n nodes) in the graph, the Linear Scan algorithm scans n^2 edges. If an edge E has both ends visited before, edge E is removed. Otherwise, both nodes connected by edge E are merged. When Linear Scan merges nodes, it simply attaches one edge

```
LinearScan(Graph)
1  for each node in Graph do
2     node.isVisited ← true
3     for each edge in node.edgeList do
4        if (edge.heat > Threshold)&&(!isVisited(edge))
5           then AttachNodes(edge, node)
6           else RemoveEdge(edge)

AttachNodes(edge, node)
1  (nodeA, nodeB) ← edge.getNodes()
2  if (!nodeA.isVisited)
3     then nodeB ← nodeA
4  nodeB.isVisited ← true
5  node.edgeList.attach(nodeB.edgeList)
6  node.blockList.attach(nodeB.methodList)
```

Figure 6.8: Linear Scan algorithm

list to the other node and takes a constant time. This means the Linear Scan algorithm's asymptotic complexity is O(n^2).

6.3.4 WCM Results

We evaluate WCM by measuring its benefit and runtime overhead for various benchmarks using each of the four code layout algorithms.

6.3.4.1 Experimental framework

To do our experiments, we use a 4-way Intel Xeon server with 2GHz Xeon processors (details are in Section 4.5. Our experiments use the SPEC JVM98, SPEC JBB2000, and MiniBean benchmarks. We execute each application stand-alone. For heap sizes, we use 512M for MiniBean, 256M for SPEC JBB2000, and 50M for SPEC JVM98 to accommodate their different working set (data and code) sizes. Because we focus our efforts on server applications and are not changing the garbage collector, we do not consider the trade offs of different heap sizes.

Benchmark	IA32 Size	Methods	Calls/sec
SPECjbb	268K	758	2.04M
MiniBean	3.10M	15586	3.65M

Table 6.3: Benchmark characteristics

To measure performance, we divide application execution into three components: (1) profiling (warm-up), (2) code reorganization, and (3) steady-state execution. This methodology is widely used for reporting long running server programs in the literature and industry [65], and where WCM should be most effective. We measured these three components separately. We re-organized code once at the end of application warm-up with MiniBean and SPEC JBB2000, and at the end of the first iteration of each program with SPEC JVM98.

6.3.4.2 WCM Overhead

The overhead of WCM has two main components: the time to generate a dynamic call graph, and the time to generate a new code layout. We evaluate both overhead components separately.

Dynamic Call Graph Generation Overhead We use software instrumentation on IA-32. To determine the overhead of software instrumentation for DCG creation, we run MiniBean two times: once with no instrumentation and a second time with our software-based DCG generation. Each time, we run MiniBean up to the same point in its execution, when it completed its warm-up phase. We found that MiniBean required 54 seconds with no in-

strumentation, but 66 seconds when we used software instrumentation. This overhead is high.

Our software instrumentation overhead is high because it uses an untuned, unspecialized call-counting stub requiring additional procedure calls, memory allocation, and locking. We do not tune this stub because it is only used during warmup. A production WCM implementation would use optimized and inlined JIT-compiled code. However, although this instrumentation overhead is high today, it only exists while the DCG is being generated. After WCM reorganizes code, it turns off software instrumentation and removes the instrumentation stubs. As a result, there is no overhead after code reorganization. For long running server benchmarks like MiniBean, the time during which WCM uses software instrumentation is relatively short and so the overall impact on the benchmark is low.

We also expect and do observe high overheads due to software instrumentation implementation on the SPEC JVM98 benchmarks. The overheads are shown in Figure 6.9. The bars for _227_mtrt are cut off since they are about 3000%. The geometric mean of the overheads is about a factor of 3. This result indicates that if software instrumentation is used, a faster implementation is needed, especially for smaller applications. Jikes RVM use adaptive sampling to collect dynamic call graph which has much lower overhead and therefore suitable for small application.s

Code Layout Generation Overhead Another overhead of WCM is the time required by the code layout algorithms to generate a code layout. We

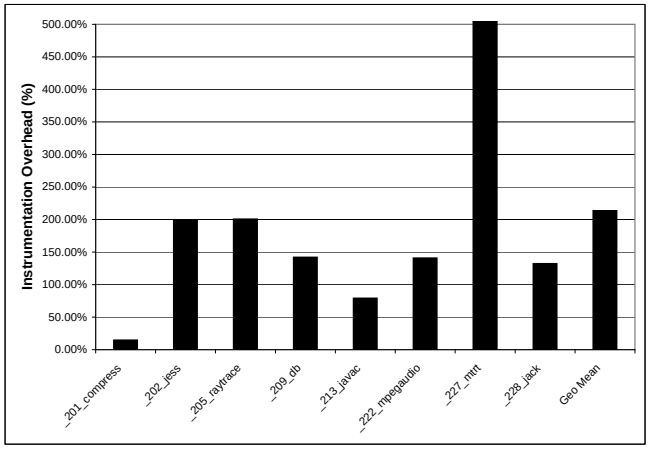


Figure 6.9: SPEC JVM98 software instrumentation overheads

Algorithm	**MiniBean**	**SPECjbb**
Pettis-Hansen	2215127	503
Cache-aware Pettis-Hansen (16K)	26012	197
Cache-aware Pettis-Hansen (4K)	28840	186
Code Tiling (16K)	508	17
Code Tiling (4K)	352	15
Linear Scan (1)	3611	29
Linear Scan (10)	820	23
Linear Scan (100)	295	21
Linear Scan (1000)	254	19
Linear Scan (10000)	253	21

Table 6.4: MiniBean and SPEC JBB2000 layout creation times (ms)

show the times needed for MiniBean and SPEC JBB2000 in Table 6.4.

The Pettis-Hansen procedure layout algorithm requires 37 minutes to reorder MiniBean's code which is much too long to be practical in a dynamic code reordering system. Our new algorithms are much faster, especially Code Tiling which takes just 0.35 seconds for MiniBean when using a 4KB page as the cut off threshold. This time is less than most of MiniBean's garbage

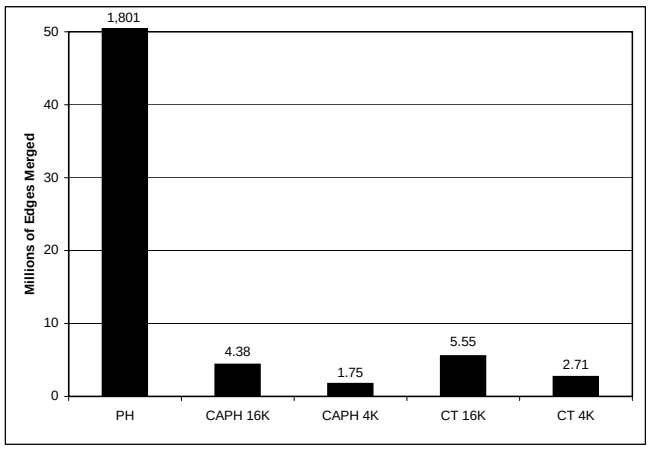


Figure 6.10: Number of edge merges for MiniBean

collection times.

Pettis-Hansen merges many more edges than Code Tiling or our other layout algorithms as illustrated in Figure 6.10. In this figure and the following ones, "PH" stands for the Pettis-Hansen algorithm, "CAPH" for Cache-Aware Pettis-Hansen, "CT" for Code Tiling, and "LS" for Linear Scan. The figure shows the number of edge merges (in millions) needed for MiniBean with Pettis-Hansen and the other algorithms. Note that the bar for Pettis-Hansen is cut off since it merges about 1.8 billion edges. When we explore where Pettis-Hansen spent its time, we find that merging edges required most of the time for SPEC JBB2000 and MiniBean.

Generating the new code layouts for the SPEC JVM98 benchmarks is generally much faster than for SPEC JBB2000. Our new algorithms are up to 6.33 times faster (for Code Tiling with a 4KB threshold). The results are shown in Figure 6.11.

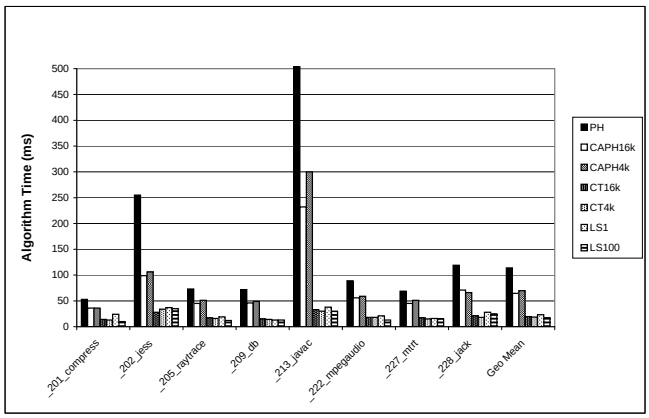


Figure 6.11: SPEC JVM98 layout generation times (ms)

6.3.4.3 WCM Performance Results

Figure 6.12 shows the performance benefit of dynamic code reorganization with both the MiniBean and SPEC JBB2000 benchmarks using the four different code layout algorithms. The base code layout is the default one used by our managed runtime, which is based on invocation order and already provides some locality benefit, as we noted in Section 1.2. The MiniBean rate reported in the second column is the harmonic mean of the four throughput rates it reports. For SPEC JBB2000, we report the 8-warehouse score.

These results show that WCM with Code Tiling can significantly improve MiniBean's performance. However, it has essentially no impact on SPEC JBB2000. One reason for this difference is the size of the two benchmarks. The IA-32's 128-entry ITLB can map 512K of simultaneous code space with the default 4K pages which is much smaller than MiniBean's 3.1MB of JIT-compiled code. Optimizations that improve MiniBean's code locality should

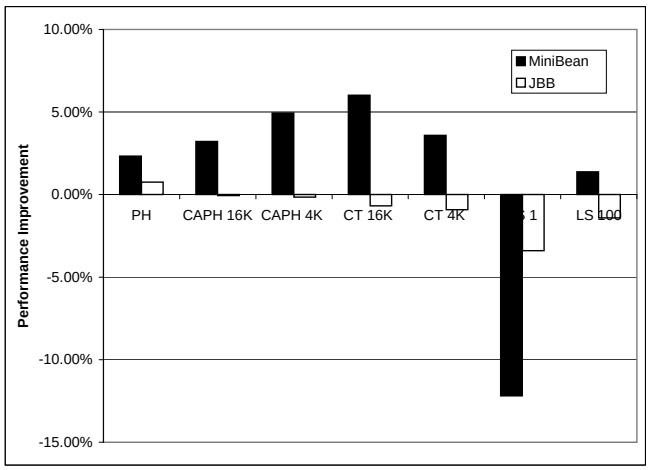


Figure 6.12: MiniBean and SPEC JBB2000 performance

thus improve its performance. SPEC JBB2000, on the other hand, only has 268K of code, so it fits within the ITLB span. Reorganizing this code to improve locality has little benefit, at least as long as SPEC JBB2000 is the only application running on the machine. If multiple programs are running, there may be some benefit since improving a program's code locality will reduce its working set, which allows more applications to run simultaneously without ITLB misses.

Figure 6.13 shows the run times for the SPEC JVM98 benchmarks with different code layout algorithms. We measure the times for the second iteration of the benchmark runs, so these times do not include any instrumentation or code reorganization overhead. There is no clear performance benefit from using any layout algorithm. Since these benchmarks are so small, with instruction working sets often less than 32K (see Section 6.4.2.2, this result is

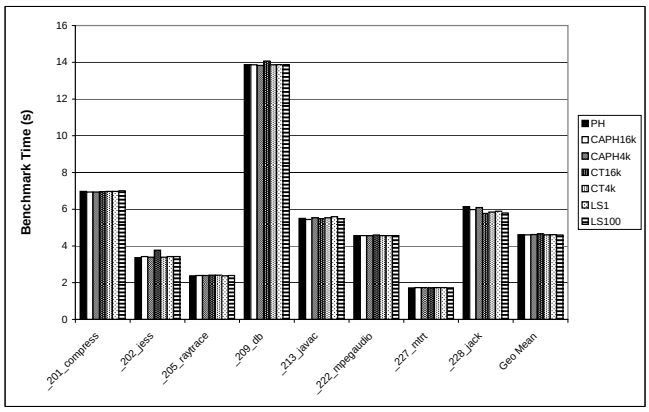


Figure 6.13: SPEC JVM98 performance

not surprising.

6.3.4.4 Discussion

This section's results demonstrate that Whole Code Management can significantly improve the performance of the large MiniBean benchmark. In previous work, we found that Pettis-Hansen improves the performance of the even larger SPECjAppServer benchmark by 4.2%. SPECjAppServer has approximately 19,000 compiled methods compared to MiniBean's 15,586. However, we have not measured the benefit of WCM using Code Tiling for SPECjAppServer.

The benefit of WCM depends on application size. This section shows that code reorganization helps large applications more than small ones. These results also demonstrate that Code Tiling is much more suitable for online code reorganization than the classic Pettis-Hansen algorithm. It executes much faster and can produce better performance.

6.3.5 PMU-based Code Reorganization

This section describes our Itanium Processor Family (IPF) WCM implementation and presents our experience with using hardware Performance Monitoring Unit (PMU) sampling on IPF to reorder compiled code. After describing our implementation, we discuss its overhead. We also present performance results using PMU-generated DCGs.

On IPF, we prefer PMU sampling since it is less expensive than software instrumentation[1]. Our PMU sampling implementation periodically examines the processor's Branch Trace Buffer to find the recent taken branches. By filtering the branches to extract those with source and target addresses in different methods, WCM discovers information about recent method calls. This information identifies both the caller and the callee methods, or more precisely, their code blocks. We separate call instructions from return instructions by checking if the target address is at the beginning of a code block.

6.3.5.1 Experimental Framework

For our IPF results, we use the 1.5GHz Itanium 2 described in Section 4.5. We use StarJIT [1] which has a high-performance dynamic compiler that uses a single SSA-based intermediate representation and global optimization framework to compile JVM bytecodes. StarJIT typically emits two code blocks for each method. These blocks separate the method's hot and cold

[1]PMU monitoring can be adjusted dynamically to keep its overhead to 1% or so. If 1% overhead is still too great, PMU monitoring can be done periodically and disabled between monitoring.

code; the cold code includes exception handlers, for example. The granularity of code reorganization on IPF, then, is a code block instead of a method.

Our IPF WCM implementation is not complete. We have not completed the StarJIT changes needed for it to update compiled code during a code reorganization. However, we are able to use *static code layout* to get an approximation of what WCM might provide. This approximation uses a separate profiling run to build the DCG, runs the code layout algorithm, and writes the resulting code layout to a file. Then subsequent runs use this layout file to place their compiled code. When static code layout is used, VM first reads the layout, then uses its placement information when allocating code for JITs.

6.3.5.2 PMU-based DCG Generation Overhead

On IPF, we first study the overhead for dynamic call graph generation with PMU sampling. We vary the sampling interval from 10 (1 sample every 10 branches) to 100,000 (1 sample every 100,000 branches). The times required to generate the dynamic call graph for MiniBean at different sampling intervals are shown in Figure 6.14. These times are from program start until WCM generates and applies the new code layout and thus do not reflect any benefit from using the new code layout. As we increase the sampling interval to 10,000 or higher, the overhead drops to less than 1%. In addition, our PMU driver has the ability to change the hardware sampling interval at runtime. We could sample more frequently for a short period of time and then revert back to longer-interval sampling as necessary.

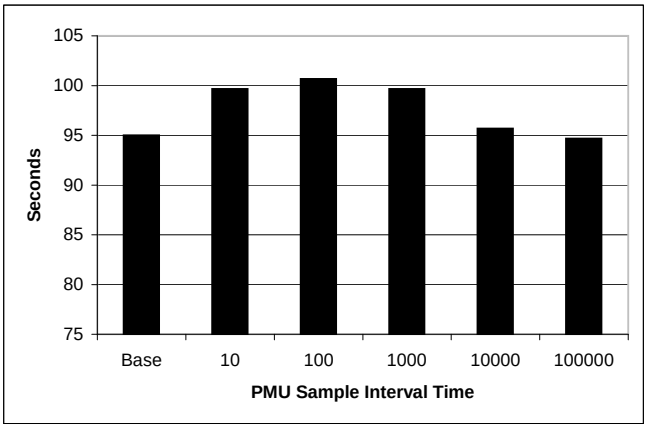


Figure 6.14: MiniBean DCG creation times with PMU sampling

We also measure the overhead of using the PMU to generate the dynamic call graph for the SPEC JVM98 benchmarks. Figure 6.15 shows the results similar to those for MiniBean. As the sampling interval increases to 10,000 or more, the overhead drops to less than 2%. This result shows that using hardware sampling is a plausible method to gather dynamic calling information even for small applications. We measure the overhead by comparing one run using PMU code layout with a second run for the base case (no PMU sampling, no code reordering). The first run does all the work of generating a new code layout but did not actually apply it.

6.3.5.3 Code reorganization results

Since our WCM system is not fully implemented on IPF, we could not collect performance results there for fully dynamic code reorganization. One drawback of using static code layout instead is that it can't cope with methods that were never compiled in the profiling run. Different methods can be

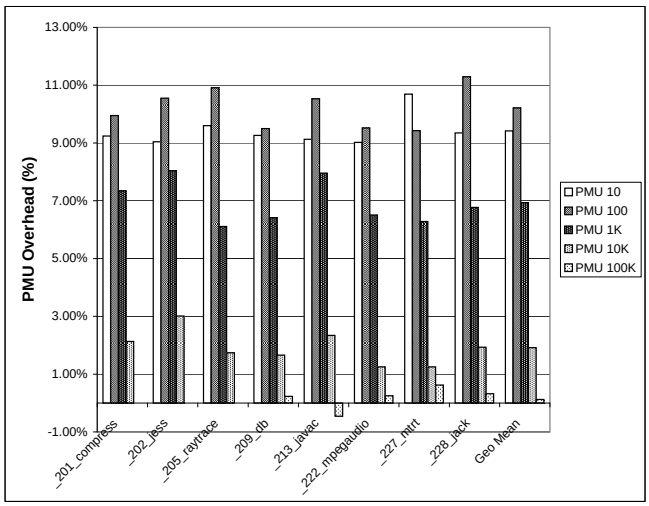


Figure 6.15: PMU overhead for the SPEC JVM98 benchmarks

compiled in different runs of the same program with same input because of dynamic class creation and loading (which is done by MiniBean and SPECjAppServer2002). As a result, the benefit of static code layout may be less than dynamic layout would achieve.

We use static code layout with the Pettis-Hansen layout algorithm to determine its performance impact for the MiniBean benchmark. We find the performance improvement was slightly negative, -1%, which probably indicates only that for this benchmark on IPF, code reorganization has little impact. While it is possible that WCM would provide better performance, its benefit is still likely to be less than it was on IA-32. One reason for this poor result is the large L3 cache (9MB) on our IPF machine, which holds nearly all of MiniBean's code (11MB on IPF). Another reason is the short memory stall time on the Itanium 2 processor: the latency is approximately 6 cycles for

	Base	Pettis-Hansen
FE Flush	5655	5570
TLB Stall	18904	13876
Instruction Cache Miss Stall	45484	46583
Any of 4 Branch Recirculates	6780	7155
Recirculate for Fill Operation	935	975
Branch Bubble Stall	66228	68481
Instruction Buffer Full Stall	115852	117431
Sum	259838	260070

Table 6.5: IPF front-end stalls using static code layout

the L2 cache and 12 cycles for the L3 cache. These mean that reordering method code will have little impact on IPF programs unless those programs have working sets much larger than the L3 cache size.

6.3.5.4 Effectiveness of PMU sampling

The Itanium 2 processor's PMU support makes it possible to get detailed performance information at low cost and with low impact on the running program. Even though we cannot collect performance results there for dynamic code reorganization, we can still use its hardware performance counters to study the impact of static reorganization.

Table 6.5 shows the number of IPF front-end stall cycles when running MiniBean. It compares the number of stalls for the default code layout as well as one using the Pettis-Hansen layout algorithm. For the latter, we use a static code layout using a DCG based on PMU sampling. Among the front-end stalls we measure are ITLB miss and instruction cache miss stalls.

There is a 26.60% improvement in ITLB miss stalls with the static code layout. However, there is no noticeable improvement in either the total front-

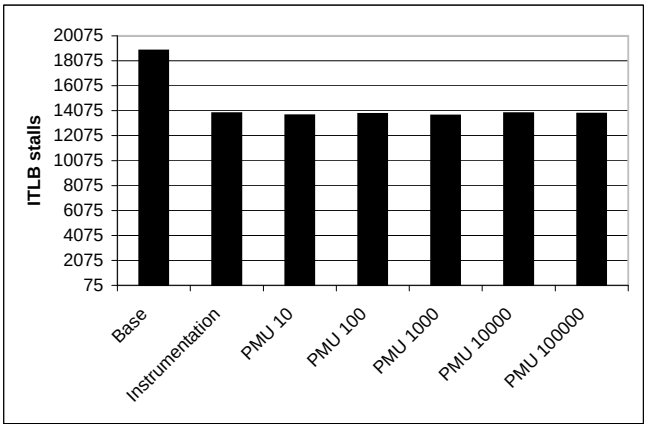


Figure 6.16: Effect of PMU-based call graphs on ITLB misses

end stalls or MiniBean's overall performance since the ITLB stalls are only a small percentage of the overall stalls.

We also measure the effectiveness of PMU-generated dynamic call graphs. Using PMU sampling at different branch intervals, we compute static code layouts based on the resulting DCGs. We then collect the total ITLB misses for MiniBean using the IPF performance counters. The results are shown in Figure 6.16 and indicate that, for MiniBean, PMU-generated dynamic call graphs are as effective as the precise software instrumentation graphs for DCG. While there is a trade off between PMU sampling frequency and accuracy of the resulting DCGs, Figure 6.16 illustrates that the lower-precision DCGs do not have a significant impact on the effectiveness of the generated code layouts. These results are consistent with our previous experience that PMU instrumentation has the same benefit as software instrumentation, but at lower cost.

6.3.5.5 Discussion

While Section 6.3.4 shows that Whole Code Management can significantly improve the performance of larger applications such as MiniBean, this section's results show that the benefit depends on the processor's microarchitecture and cache hierarchy. Code reorganization has more impact on IA-32 than IPF. Our IPF machine's 9MB L3 cache and short memory stall times means that applications must have substantially more code than MiniBean before they will benefit from code reorganization.

This section's results also demonstrate that PMU sampling for collecting dynamic call graph has low overhead. The results also show that, for short-running programs, PMU sampling can have lower impact on the programs than software-based instrumentation.

6.4 Fast and Efficient Partial Online Code Reordering

This section introduces Partial Code Reordering (PCR) which performs online code reordering with low overhead by piggybacking on just-in-time (JIT) recompilation. PCR seeks to improve instruction locality by attacking capacity and conflict cache misses. It uses dynamic call graph and basic block profiles. It performs three optimizations using multiple code spaces: (1) interprocedural hot/cold method separation, (2) intraprocedural hot/cold code splitting, and (3) interprocedural hot code padding. To reduce capacity misses, PCR allocates hot and cold methods into separate spaces in the heap. PCR also performs code splitting of hot and cold basic blocks within the same method to

further reduce the hot instruction working set size. To reduce conflict misses for the current method, PCR examines the dynamic call graph and finds hot caller/callee pairs. If they map to the same lines in the cache, they will have too many conflict misses. Therefore, PCR applies code padding on either caller or callee method (whichever it happens to be recompiling) to eliminate the potential conflict misses.

PCR piggybacks on adaptive *hotspot* compilation. PCR performs its code layout optimizations when the dynamic recompilation system has already selected a method to recompile at a higher level, and thus must generate and allocate space for the code anyway. PCR uses the dynamic call graph and edge profile for the current method, and never examines the entire graph nor re-allocates the code, as does the prior work [58] and the WCM approach in the previous section. This design reduces the overhead of PCR to a negligible level.

We run our experiments on two architectures, an Intel Pentium 4 and an IBM PowerPC 970, using SPEC [64, 65] and DaCapo [13] Java benchmarks. Because the instruction working set sizes for these benchmarks are modest compared to the available instruction cache (or trace cache), PCR does not improve most of these programs. However, a few programs are sensitive to instruction code layout: compared to the Jikes RVM default configuration which mixes code and data in the heap, a simple instruction code space improves total performance on average by around 6% and on one benchmark by 30%. The PCR optimizations improve one benchmark by 5%, but sometimes

degrade performance and on average have a negligible impact.

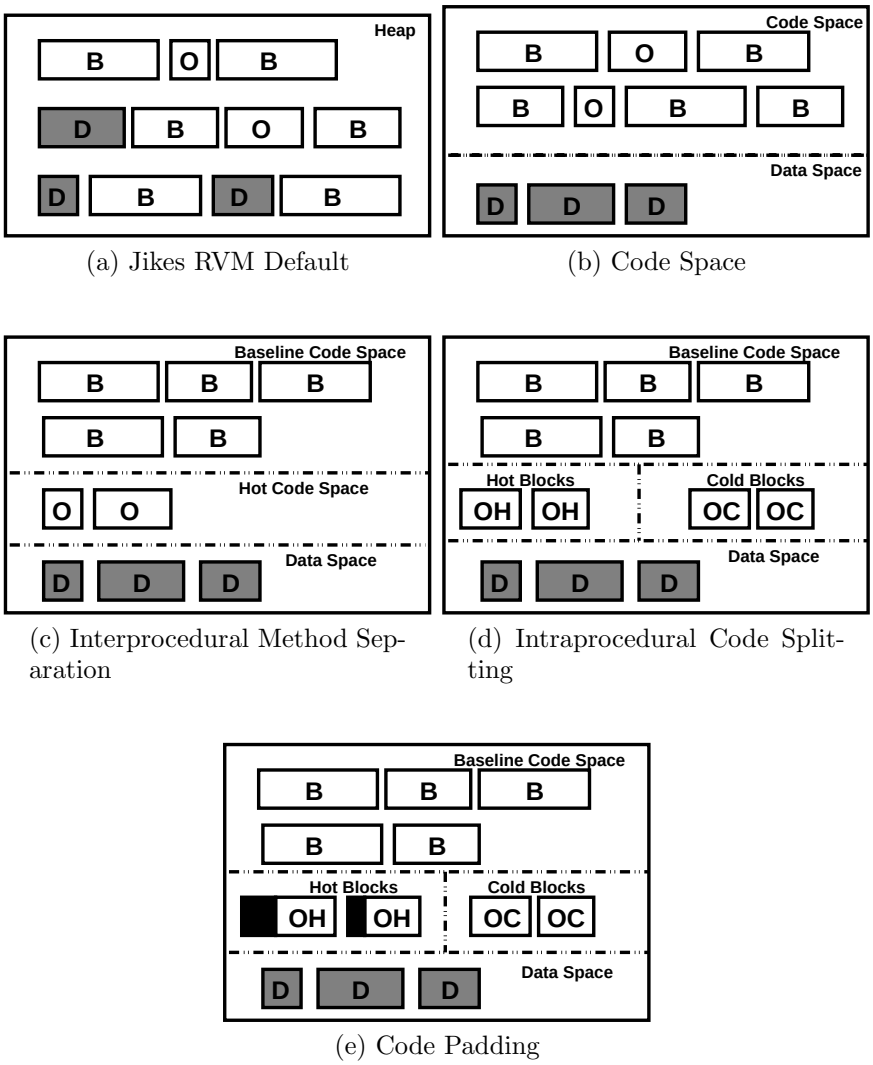
PCR demonstrates that a dynamic optimization system can reduced instruction cache footprints with negligible overheads by exploit the opportunities in JIT compilation.

6.4.1 Partial Code Reordering

The Partial Code Reordering (PCR) system is designed to be extremely low overhead and to exploit dynamic program behavior. By default, Jikes RVM allocates code in the heap with all the other VM and application objects. PCR first adds a separate space for all code. (This design is prevalent in commercial JVMs for code locality and ease of implementation for JVMs written in C.)

PCR performs two types of optimizations: *interprocedural* and *intraprocedural* code reordering. When Jikes RVM initially compiles a hot method with its optimizing compiler, PCR allocates the hot method in a separate space from baseline compiled code. PCR also splits the hot method into hot and cold basic blocks based on their execution frequencies and allocates them into different spaces. PCR determines whether a basic block is hot or cold by computing its relative execution frequency from online edge profile information and then applying a simple threshold. It also identifies frequent caller/callee pairs by applying a threshold to the dynamic call graph edges. PCR calculates and inserts padding in front of the hot portion of each optimized method to minimize the likelihood of conflicts with its callers and callees in the instruction cache.

Figure 6.17(a) shows code and data layout for the default configuration of Jikes RVM. In the figure, 'B' denotes baseline compiled code; 'O' denotes optimized compiled code; and 'D' denotes data objects. Figure 6.17(b) shows



(a) Jikes RVM Default

(b) Code Space

(c) Interprocedural Method Separation

(d) Intraprocedural Code Splitting

(e) Code Padding

Figure 6.17: Code Reordering Heap Layouts: *B: baseline code; O: optimized code; D: application objects; OH: hot basic blocks; CC: cold basic blocks*

the separation of code and data into separate spaces; this design is typical of most current JVMs. When a method is compiled by either the baseline or optimizing compiler, PCR allocates the code into the single shared code space.

6.4.1.1 Interprocedural Method Separation

As described in Section 2.1, hot methods are detected by software sampling in Jikes RVM and recompiled by optimizing compiler, and cold methods are compiled by baseline compiler. Because of lazy compilation and dynamic class loading, baseline compiled code (code methods) and optimized code (hot methods) will mix in a single code space. The first PCR optimization, *interprocedural method separation*, simply separates hot and cold methods by separating optimized code and baseline compiled code. When the optimizing compiler recompiles a method, PCR allocates these hot methods into a separate hot code space, as shown in Figure 6.17(c). This optimization reduces the code footprint of the hot methods and consequently may reduce L2 cache residency, L2 cache misses, and paging. We manage the optimized compiled code spaces as a contiguously allocated (*bump-pointer copy*) space in MMTk [12].

6.4.1.2 Intraprocedural Code Splitting

The existing optimizing compiler uses the edge profiling instrumentation from the baseline compiled code to push hot basic blocks to the beginning of the generated code and cold ones to the end. PCR further separates the hot and cold basic blocks by allocating them into different regions of the optimized code space. The generated layout is shown in Figure 6.17(d), where

'OH' denotes hot basic blocks of a method and 'OC' denotes cold blocks of a method.

PCR splits code during code generation. We implement PCR system on x86 and PowerPC architectures, which both have short pc-relative branch instructions for a short jump. We conservatively use long branches if a branch is crossing the two partitions of the same method. This conservative choice increases the code size if the branch was a short branch before code splitting. PCR allocates 16 KB size chunks for hot and cold block allocation to avoid having a branch distance larger than the upper bound of a conditional branch. Therefore the hot and cold blocks are approximately interleaved within the heap in 16 KB chunks.

6.4.1.3 Code Padding

PCR uses the dynamic call graph to find frequent caller/callee pairs, based on the threshold used to identify recompilation candidates. The frequent caller/callee pairs may generate conflict misses if they are mapped into the same line in the instruction cache. After PCR splits a method A into hot/cold blocks, it checks all of the frequent callers and callees of method A to see if their mappings in the cache overlap method A's mapping. If PCR detects overlaps, it employs a simple and fast algorithm to calculate a padding size that avoids conflicts. PCR does not attempt to find an optimal padding size that minimizes the expected number of conflict misses and wasted code space. However, our experience is that the number of potentially conflicting methods for a method is often *one* and therefore this simple and efficient algorithm is

```
CODE-PADDING(methodA, DCG)
1  currentPadding ← 0
2  repeat
3          for each methodB in GET-ADJACENT-NODES(methodA, DCG) do
4            if CHECK-CONFLICTS(methodA, methodB) then
5               padding ← CALCULATE-PADDING(methodA, methodB)
6               currentPadding ← currentPadding + padding
7               if currentPadding < methodA.size then
8                  methodA.address ← methodA.address + padding
9    until (padding == 0)||(currentPadding >= methodA.size)

CHECK-CONFLICTS(methodA, methodB)
1  offsetA ← methodA.address&(CACHE_SIZE − 1)
2  offsetB ← methodB.address&(CACHE_SIZE − 1)
3  if offsetA < offsetB
4     then return (offsetA + methodA.size > offsetB)
5     else return (offsetB + methodB.size > offsetA)

CALCULATE-PADDING(methodA, methodB)
1  offsetA ← methodA.address&(CACHE_SIZE − 1)
2  offsetB ← methodB.address&(CACHE_SIZE − 1)
3  padding ← offsetB + methodB.size − offsetA
4  if (offsetB > (offsetA + methodA.size))
5     then return 0
6     else return padding
```

Figure 6.18: Pseudocode for Code Padding

usually sufficient. To avoid wasting space, we use the method size as an upper bound on the amount of padding we insert.

The detailed algorithm is in Figure 6.18. For each hot caller-callee pair, CHECK-CONFLICTS computes where in the given cache size they map and their overlap. If they overlap, it computes a padding, accumulating any non-zero padding unless the padding size exceeds the method size. Because PCR contiguously allocates with a bump pointer in the code space, PCR applies the padding by simply adding the padding size to the bump pointer before allocating the hot blocks of method A. Figure 6.17(e) depicts this code layout.

	Baseline	O0	O0H	O1	O1H	O2	O2H
DaCapo Benchmarks							
antlr	1,385,368	109,232	48,892	118,252	57,928	17,656	10,052
bloat	1,178,616	193,716	103,104	307,020	123,484	140,836	39,968
fop	1,841,528	37,868	17,352	41,396	15,872	4,068	2,484
hsqldb	516,800	15,628	6,024	284,332	74,328	104,956	33,324
jython	1,217,868	13,916	9,184	8,992	2,384	43,824	11,432
pmd	1,166,364	59,932	31,132	48,708	20,996	51,892	25,144
xalan	1,397,848	20,356	10,232	97,388	32,004	4,528	1,016
ps	205,472	16,212	9,068	17,648	10,004	5,264	3,432
SPEC Java Benchmarks							
compress	173,432	2,208	1,392	180	112	4,248	2,108
jess	355,296	8,400	4,012	29,724	9,820	6,104	3,628
raytrace	220,508	13,560	7,232	15,808	10,736	1,228	960
db	175,640	2,476	1,156	0	0	5,804	3,412
javac	612,128	93,032	42,900	53,720	27,784	2,168	836
mpegaudio	546,512	21,968	8,320	22,104	8,280	6,464	4,116
mtrt	221,032	14,124	7,500	14,700	9,988	1,336	960
jack	465,028	9,964	4,440	36,756	21,008	4,352	2,604
pseudojbb	404,512	85,456	43,368	24,916	15,240	2,588	2,028
Arithmetic mean	**710,820**	**42,238**	**20,900**	**65,979**	**25,880**	**23,960**	**8,677**

Table 6.6: Benchmark Code Size Characteristics in Bytes with Replay Compilation

6.4.2 Experimental Results

This section evaluates PCR and compares it to Jikes RVM with and without a separate code space. For our evaluation, we first perform simulations to expose the magnitude of the performance loss due to instruction cache conflicts of our Java applications, and the benefits of padding in a controlled setting. We find that for a direct mapped cache, programs lose around 6% on average and up to 17% of their performance to instruction cache conflict misses. We further

explore the performance impact of PCR using two architectures: Pentium 4 and PowerPC; and two Jikes RVM configurations: one that excludes most compilation and thus consists mostly application time, and one that mixes the adaptive compilation and the application. The latter experiment more accurately reflects a multiprogrammed workload and is when PCR is most effective. A simple code space improves the default Jikes RVM configuration by about 6%. Because the code footprint of our benchmarks is small, additional PCR optimizations have little impact, occasionally improving them and occasionally slowing them down.

6.4.2.1 Application and Compiler Mix

We use two Jikes RVM compiler and application mixes for our experiments, which we call *second run* and *adaptive*.

(1) The *second run* methodology uses profiling of the adaptive compiler from previous runs (compiler replay [11, 41]) to *deterministically* optimize methods to their highest level when the method first executes. We then perform a whole heap collection to flush the heap of compiler objects and execute the benchmark again. Some additional, but minimal recompilation may take place during the second run of the benchmark. We report measurements of this second run because it consists almost entirely of application execution and it is easier to understand and measure [13]. Eeckhout et al. show that measuring the first iteration on SPECjvm98, which *includes* the adaptive compiler, is dominated by the compiler rather than the benchmark behavior [34]. This methodology gives a simple code space an advantage because more compila-

tion takes place early and together (as we show below). This methodology would also mimic the Arnold et al. system that combines offline and online profiles to drive compilation [8].

(2) The *adaptive* methodology lets the optimizing compiler behave as intended, is non-deterministic, and measures compilation and application time. Section 6.4.2.4 reports these results, which because the compiler *competes* with the application, are most indicative of a multiprogrammed workload, and may be more indicative of results on programs with larger icache footprints than our benchmarks. For example, SPECjAppServer loses significant performance to poor instruction cache behavior [28].

6.4.2.2 Benchmarks and Instruction Code Sizes

We use the SPECjvm98 [64], SPECjbb [65], and DaCapo benchmarks [13]. Other work [13, 32, 41] characterizes the memory behavior and memory system performance of the *data* for these benchmarks. Table 6.6 shows instead the code size characteristics of these benchmarks in bytes. We use the *replay compilation* methodology to measure the size of generated code at each optimization stage. Therefore the numbers here only include the final optimized code for every method, since replay specifies exactly at which level to compile each method. An adaptive compilation would instead compile a very hot method M at multiple levels, e.g., baseline compiled first, and then optimizing compiled at level O0, level O1, and level O2. Replay compilation only optimizes method M at level O2. Table 6.6 thus shows the amount of compilation at each level, and each method is compiled once at one level (although inlining

produces copies of some code). Column one lists the benchmarks. The *baseline* column shows the total amount of baseline compiled code in bytes, which ranges from 173 KB up 1841 KB. These volumes clearly exceed the capacity of typical 8 to 32 KB instruction caches and demonstrate that for the most part, the DaCapo benchmarks have larger code footprints than SPEC. For each of the three levels of optimization (O0, O1, O2), the next six columns divide the methods into hot (indicated with a suffix 'H') and cold code. We use the edge profile to determine the hot basic blocks. The SPEC Java benchmarks always produce less than 8 KB of hot code in the O2H space, and the total size of the hot methods at O1 and O2 is always less than 32 KB. For the DaCapo benchmarks at O2, there are two programs with a hot code size of greater than 32 KB, and at O1 plus O2, there are five of eight. The table thus indicates that the working set of code (i.e., the hot code) in these programs is not putting very much pressure on the instruction cache.

6.4.2.3 Simulation Results

Because on a real architecture, we have limited information available and less control with different hardware configurations (for example, cache associativity), we use simulation to understand our PCR optimizations better. We use Dynamic SimpleScalar (DSS) [42], a variant of widely used SimpleScalar simulator [15] that is extended to run Java programs. We simulate a fully associative instruction cache and compare with a direct-mapped cache with the same access time to show how much performance is lost to instruction cache conflict misses.

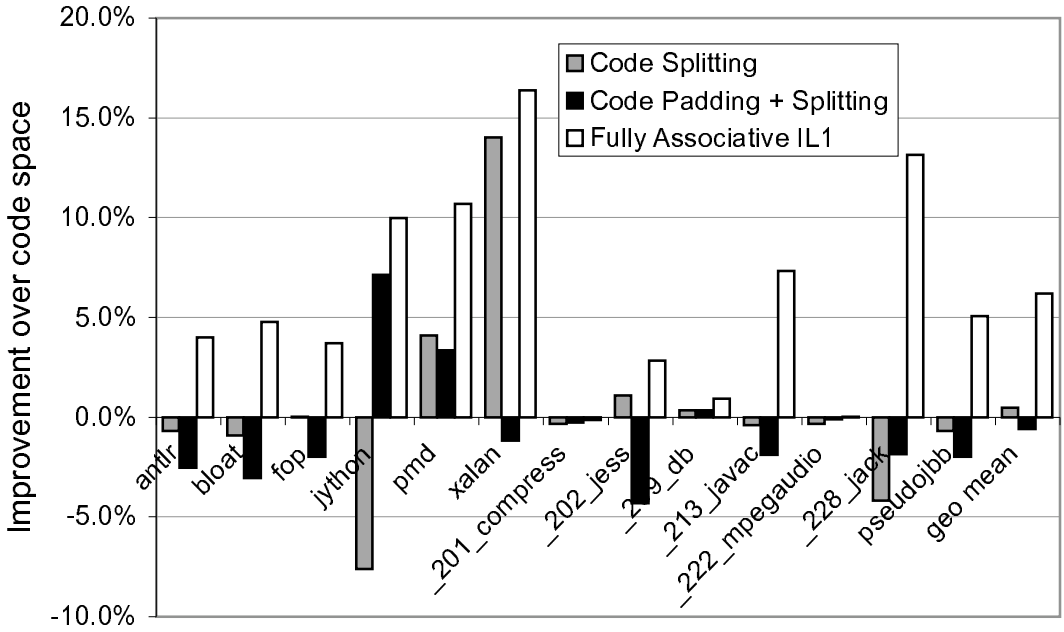
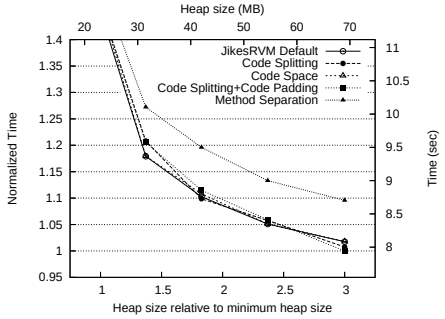


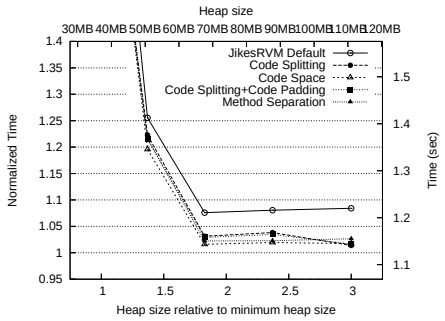
Figure 6.19: Simulation Results for Directed-Mapped and Fully Associative 32 KB IL1, 512 KB L2

We use two instruction cache configurations for these simulations. (1) A 32 KB *direct-mapped* instruction cache and 512 KB unified L2 cache; L1 access latency is 2 cycles and L2 access latency is 5. (2) A 32 KB *fully-associative* instruction cache and 512 KB unified L2 cache with the same latencies as configuration (1). We make the hit latency of (1) and (2) the same to examine the potential performance improvement if we have no conflict misses on a direct-mapped instruction cache. We use the *second run* methodology described above. We perform functional simulation for the first iteration, turn off the adaptive compiler, and then switch to cycle level simulation right before the second iteration, and then simulate 2 billion instructions.

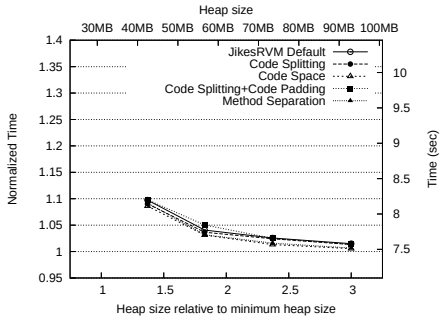
Figure 6.19 compares the relative performance of PCR using as its baseline hardware instruction cache configuration (1) with a simple code space. It shows the benefits of PCR code splitting, code splitting, padding, and a fully associative instruction cache (hardware configuration (2)). PCR code splitting and padding performs 7.1% better than a simple code space on jython although just PCR code splitting itself degrades the performance 7.6%. Because jython has 11K hot blocks after code splitting, which can easily fit in the instruction cache, we think the benefit we have from use code padding is from avoid conflicts between jython's application code and the Jikes RVM code. Jikes RVM code is allocated ahead of time in a separate region from the application. PCR has the opposite trend on xalan where code splitting improves performance by 14.0% but combined with code padding the performance degrades by 1.2%. This is likely to be incurred by the space overhead of code padding. Although the geometric mean of PCR performance over all benchmarks is about the same (0.5% better or 0.6% worse) as a simple code space on these benchmarks with modestly sized instruction footprints, we believe that by carefully choosing the PCR optimizations for each individual program, we may be able to achieve better average results. The performance of the fully associative cache shows that even these relatively small applications lose on average around 6% to instruction cache conflicts, but that PCR is not consistently able to achieve that potential.



(a) antlr



(b) fop



(c) Geometric Mean

Figure 6.20: Code Optimizations on Pentium 4

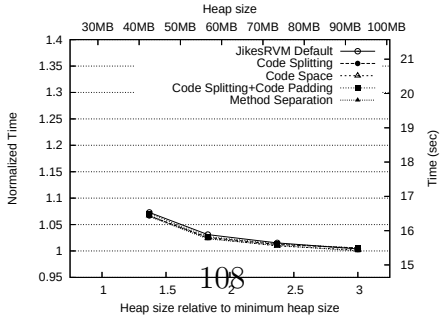


Figure 6.21: Code Optimizations on PowerPC 970: Geometric Mean

6.4.2.4 Application Performance

We report run-time results for our implementation on the 3.2GHz Pentium 4 and 1.6GHz PowerPC described in Section 4.5. We use the *second run* methodology in experiments we report in this section, and thus measure only the application behavior. We first compare the performance of PCR with the method separation and the default configuration in Jikes RVM (code is mixed in with data in the heap) on the Pentium 4. Most of the benchmarks are not sensitive to the code layout, but we found that a few benchmarks have some sensitivity. Figure 6.20 shows two of these programs (antlr and fop), and the geometric mean of all programs. All the performance numbers report relative heap size (bottom), actual heap size in KB (top), the normalized times on the left legend, and seconds on the right legend. We normalize the time to the best time on each figure, so it is easy to see the relative performance difference between the configurations. Although most systems use separated spaces for code and data, method separation, which further separates optimized compiled code from baseline compiled code, is the worst performing configuration for antlr. This could be caused by certain property of trace cache or the different sized branch instructions generated by different code layout. For fop, mixing code and data in the heap degrades its performance and PCR optimizations perform worse than just having a simple code space. PCR optimizations perform about the same as Jikes RVM default configuration for the geometric mean over all benchmarks.

We also perform the same experiments on the PowerPC which has

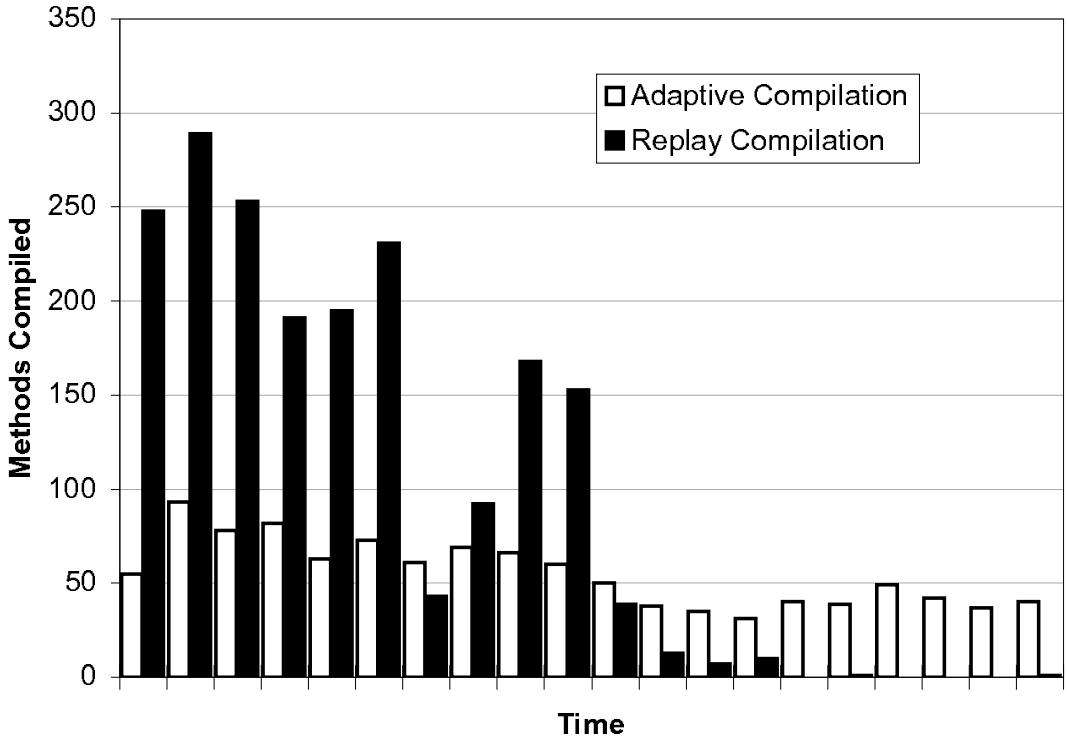


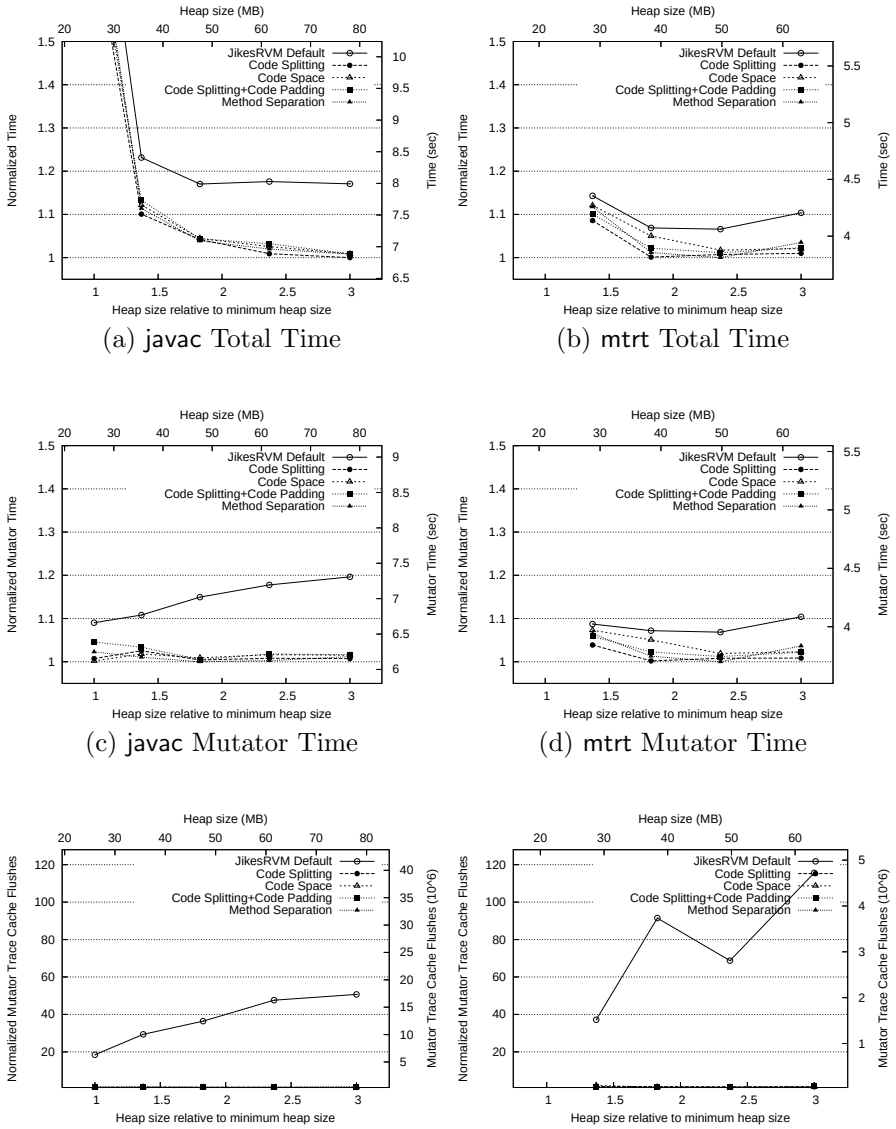
Figure 6.22: Compiler Activity Histogram on First Iteration

a traditional instruction cache instead of the instruction trace cache on the Pentium 4. Figure 6.21 shows these results. PCR has even less impact on performance on the PowerPC than on the Pentium 4 because the PowerPC has a larger instruction cache (32 KB) and 2-way set associativity. It is thus large enough to contain the working set of our benchmarks and its associativity reduces the conflict misses. As the previous section showed in simulation, a large capacity (32 KB) direct-mapped cache does however lose performance to instruction cache misses (Figure 6.19).

6.4.2.5 Mix of Compiler and Application

This section reports on experiments using the *adaptive* methodology which includes a mix of the application and the compiler as it finds hot methods and compiles them at progressively higher levels. The compiler histograms in Figure 6.22 show the differences between when the recompilation takes place in the first run with adaptive compiler versus the first run using replay compilation. We divide each of the two executions of the program into twenty buckets and then record the number of methods compiled at level O0 or higher, and sum over all programs. When we use compiler replay (to eliminate non-determinism from adaptive recompilation), compilation happens earlier in the program. We see this behavior because replay compilation compiles to the highest level of optimization in the profile on the first execution of the method, instead of recompiling at multiple levels. The adaptive methodology is thus running the compiler throughout the execution of the program. The periodic execution of the compiler displaces application code from the instruction cache and thus we believe the adaptive methodology is a suitable environment in which to study instruction cache performance programs because the competition for the cache mimics a multiprogramming environment. In this case, the application and JIT compiler interfere with each other.

Figure 6.24 shows the performance of various configurations of PCR when using the *adaptive* methodology. The figures show the total time, mutator time (program only without garbage collection), and the trace cache flushes. We report trace cache flushes using the Pentium 4 performance coun-

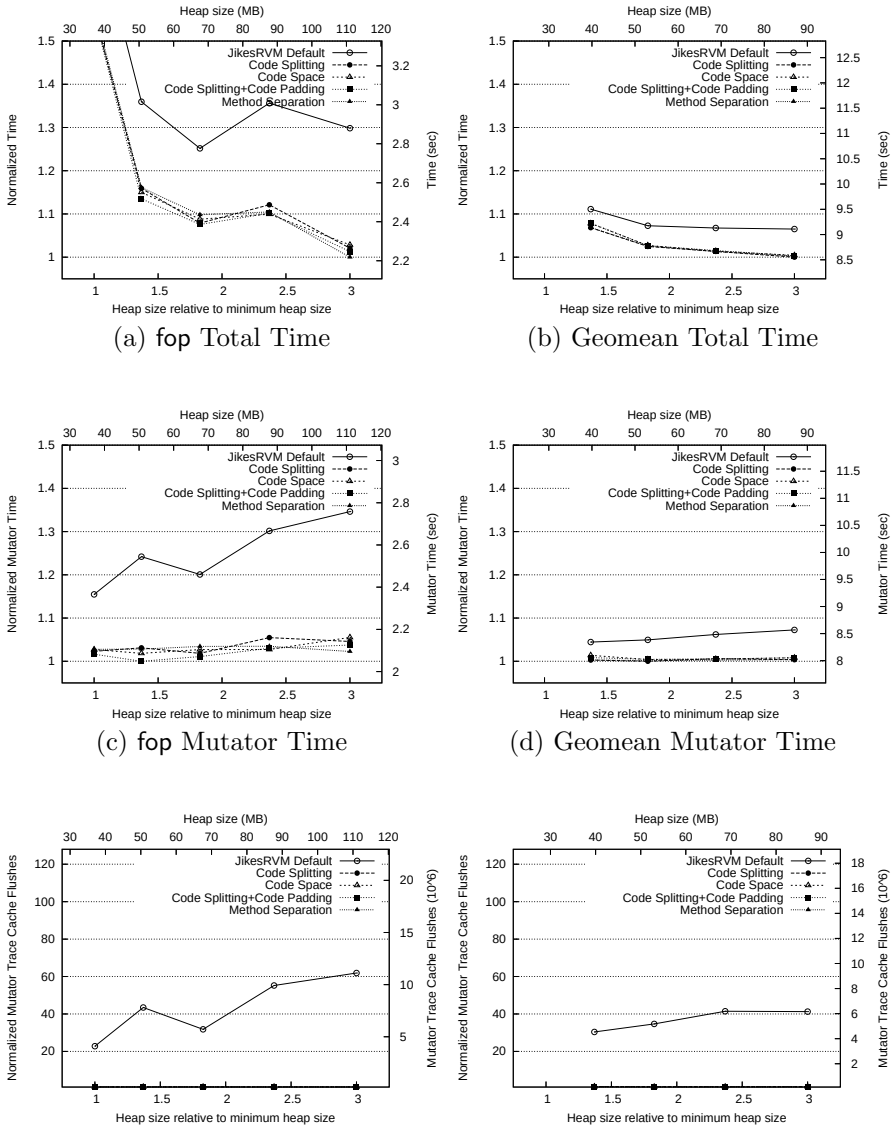


(a) javac Total Time (b) mtrt Total Time

(c) javac Mutator Time (d) mtrt Mutator Time

(e) javac Trace Cache Flushes (f) mtrt Trace Cache Flushes

Figure 6.23: Total time, mutator time, and trace cache flushes for a simple code cache, Jikes RVM default and various PCR configurations.



(a) fop Total Time (b) Geomean Total Time

(c) fop Mutator Time (d) Geomean Mutator Time

(e) fop Trace Cache Flushes (f) Geomean Trace Cache Flushes

Figure 6.24: Total time, mutator time, and trace cache flushes for a simple code cache, Jikes RVM default and various PCR configurations.

ters configured for this measurement. We measure all the programs and present the geometric mean and three programs (javac, mtrt, fop) across four heap sizes (the bottom axis reports relative to the smallest and the top axis reports in MB). Again, we normalize the total and mutator time figures to the best point to show relative differences.

The results show that a simple code space improves over the Jikes RVM default configuration by 6% on average and by 30% on fop. There are two reasons for this large difference in performance.

1. The Pentium 4's trace cache is flushed whenever the program writes to a page in the trace cache (e.g., when the VM writes new code on to a page or writes data on the same page as code). By mixing data and code together in the heap, Jikes RVM greatly increases the possibility of flushing the trace cache because writing to the data space happens more often than writing to the instruction space. This effect is very clear in Figures 6.24(c),(f),(i) and (l) and is the reason that the corresponding mutator time increases as the heap size grows for each of the benchmarks and the geometric mean.

2. By scattering instructions into the heap, Jikes RVM destroys the instruction spatial locality between methods. This effect is especially crucial for architectures with hardware prefetch for instructions.

PCR optimizations offer some additional, but modest improvements on a few programs. For example, on mtrt, PCR code splitting is most effective and

improves over a code space by up to 5%. These results demonstrate that PCR has no appreciable overheads and has the potential to improve performance over a basic code space in a multiprogrammed environment. Programs with larger instruction cache working sets may benefit, but our programs do not exercise this space.

Chapter 7

Applicability of Results to Other VMs

Our whole code layout reordering system, WCM, is implemented on Open Source Platform (ORP) which is written in C++. We do not discuss applying WCM to other virtual machines because most other VMs are written in C/C++ and it is more straightforward to port the WCM implementation to other VMs. Jikes RVM is implemented in Java which is different from most other virtual machines. By implementing Jikes RVM in Java, it reduces the cross-boundary overhead between Java/non-Java code. There are also new optimization opportunities. For example, inlining of the library methods is easier when both the caller and callee are written in Java. Although we implement our Online Object Reordering (OOR) and Partial Code Reordering (PCR) systems in Jikes RVM, our designs are not VM dependent. Most current Virtual Machines compile every method with a set of quick optimizations at the first invocation of a method [2, 16, 19, 31]. Sophisticated system [27, 40, 55, 66] can identify hot methods and apply more optimizations to these hot methods. A significant portion of our system can be implemented as an extension of these two features. In this section, we describe how to apply our systems on other virtual machines.

7.1 Adding OOR to Other VMs

The OOR system consists of three components, static compilation analysis, dynamic sampling for hot methods and copying garbage collection for object reordering. We can apply the same static analysis during compilation in other Virtual Machines. The runtime sampling technique for detecting hot methods is built on top of dynamic feedback system in Jikes RVM. However, the information we need does not require a general mechanism for online adaptive feedback. Most current VMs are able to dynamically select a subset of all methods as hot methods. Which is enough for our system to find hot field accesses. The generational copying collector is a popular choice in current VMs. Virtual Machines with a copying garbage collector implementation can easily be extended to support object reordering by hot fields.

7.2 Applying PCR on Other VMs

For the Partial Code Reordering system, we need more sophisticated support from the virtual machine. Our implementation includes two major code reorderings. First, we allocate the methods in the order they were invoked. During the process of allocation, we also separate methods by the execution frequency and split a method so that the hot blocks and cold blocks are in different spaces. We also check whether there is a conflict between the current method and its caller. Because most VMs compile each method with a fixed set of optimizations the first time it is invoked, the same optimization can be applied during the code generation just as in Jikes RVM. The com-

plication is that some VMs use a free-list allocator, in which it is harder to control the method address. By padding and requesting a larger block of the same size for every conflicting methods, the VM can overcome this limitation.

For the second phase of code reordering, PCR uses runtime feedback to find hot calling sequences and check for potential conflicts in the call graph. This component of Jikes RVM is not language dependent; therefore the same adaptive sampling system can be implemented in other language inside other virtual machines as well, but of course requires certain knowledge of the hardware (like the cache size and associativity for code padding).

Copying the generated code to a new memory region is very common in VMs because they dynamically generate code for methods and then select a subset of methods to be recompiled and move the new code into a new region. The method is either not invoked at the time it is moved, which does not require any other action. If the method is on the stack we can either give up or use on-stack replacement. Jikes RVM and Sun Hotspot [55] compilers both provide on-stack replacement.

Chapter 8

Conclusions and Future Work

Research on improving memory performance has been active for decades. Previous research has been concentrated on improving hardware or statically optimizing software using a compiler or by hand. Java programs run on top of a Java Virtual Machines, which have total control of the data object and instruction layout. This flexibility provides new opportunities to improve program locality at run time. In this dissertation, we show that dynamically optimized data and instruction layouts for Java program can consistently improve program performance and that there is room for further improvement. The key investigations of this thesis are:

1. A study of Java programs to explore the potential performance loss due to poor locality.

2. A novel low-overhead, online data object reordering system that consistently matches or improves the best static object reordering.

3. A Whole Code Management system which is the first implementation of dynamic code reordering in a managed runtime.

4. We describe a new placement algorithm, Code-Tiling, that specifically addresses expensive ITLB misses. It is much faster than the widely-used

Pettis-Hansen procedure layout algorithm, and the layouts generated by Code-Tiling algorithm often perform better.

5. We developed a new scheme for dynamic code allocation that reduces instruction working set size at runtime.

6. A new low-overhead, dynamic code padding optimizations which reduces instruction cache conflict misses.

Despite enormous effort, performance is still an issue for object-oriented programming languages compared to traditional languages like C. Prior to this work, there were no dynamic optimizations that can dynamically adjust the data and instruction layout to match program behavior in order to achieve better locality with low overhead. We discovered and innovatively exploited this available opportunities for efficiently monitoring program data and instruction locality in object-oriented programming languages. Our results demonstrate that extremely low overhead (1% or less) monitoring of data and instruction locality yields sufficient precision to drive powerful, effective data and instruction optimizations.

8.1 Future Work

Besides showing how effective our optimizations are at improving program locality, our potential results (Figure 4.1) also show that there is still room for further improvement. We believe we can develop even more effective optimizations by improve our understanding of virtual machine environments.

There are three major problems for us to address regarding design and implementation of dynamic optimizations:

- *What kind of information to collect at runtime?* This question includes what potential information we can collect at runtime, and how important is certain information is to our optimizations.

- *How to get the information with the lowest overhead?* Some information is valuable but too expensive to collect using traditional methods. Instead, we could find a low overhead estimation of the expensive information.

- *How to apply the optimization with little overhead or low enough to show the benefit?* We should leverage the available sub-systems, such as the memory manager, compiler, or runtime, to apply our optimization with low overhead.

There are many innovative answers to the above questions. The following are some of the applications.

8.1.1 Performance Counters

Most modern processors, such as Intel x86, AMD x86, IBM PowerPC, support performance counters and the trend is toward richer, more complicated support. Hardware performance counters are perfect candidates for collecting dynamic profiling information for virtual machines because they are fast and low-overhead. We use the Performance Monitor Unit (PMU) on Intel Itanium

processors to collect dynamic call graphs for Java programs. We find the quality of the sampling information is as good as the information we collect using software instrumentation but with a much lower overhead. We plan to further explore the hardware performance counters for collecting different types of information to help dynamic optimizations.

8.1.2 Potential Applications of Dynamic Monitoring

Virtual machines allow us to monitor the execution of the program and collect information from different components of the system. Besides dynamic optimizations, we can check for errors in the programs by using this information. For example, we can detect memory leaks (objects that are no longer used by the program but are still pointed by some pointer), security vulnerabilities, model violations, binary rewriters, etc. The advance of these approaches will make object-oriented programming languages safer, more robust, and provide more stability.

Optimizations in virtual machines also need to adjust to new hardware trends. Recent trends in processors, such as multi-core, require changes in the designs and optimizations of virtual machines to adapt. New dynamic optimizations such as localizing cache accesses, reducing unnecessary synchronization, will become more important than before.

Instead of considering the VM layer as a cost to be borne, we have opened up a new class of dynamic optimizations for monitoring and improving program locality, and code quality. With the advance of these dynamic optimizations, we will create programming languages that are both robust,

reliable, and high performance.

Bibliography

[1] A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. S. Menon, B. R. Murphy, M. Serrano, and T. Shpeisman. The StarJIT compiler: a Dynamic Compiler for Managed Runtime Environments. *Intel Technology Journal*, 7(1), February 2003.

[2] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. In *ACM Conference on Programming Languages Design and Implementation*, pages 280–290. ACM Press, 1998.

[3] Bowen Alpern et al. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 314–324, Denver, CO, November 1999.

[4] Bowen Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[5] A. W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[6] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented*

Programming Systems, Languages, and Applications, pages 47–65, Minneapolis, MN, October 2000.

[7] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Architecture and policy for adaptive optimization in virtual machines. Technical Report 23429, IBM Research, November 2004.

[8] Matthew Arnold, Adam Welc, and V. T. Rajan. Improving virtual machine performance using a cross-run profile repository. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 297–311, 2005.

[9] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. 35(2):59–67, February 2002.

[10] B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, San Jose, CA, USA, 5–7 October 1994.

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

[12] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with JMTk. In *Proceedings of*

the International Conference on Software Engineering, pages 137–146, Scotland, UK, May 2004.

[13] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, S. Z. Guyer, A. Hosking, M. Jump, J. E. B. Moss, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java benchmarking development and analysis. Technical Report TR-CS-06-01, Dept. of Computer Science, Austrailian National University, March 2006. http://ali-www.-cs.umass.edu/DaCapo/Benchmarks.

[14] D. Bruening, V. Kiriansky, T. Garnett, and S. Banerji. Thread-shared software code caches. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 28–38, NY, NY, March 2006.

[15] D. Burger and T. M. Austin. The SimpleScalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[16] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141. ACM Press, 1999.

[17] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for

subscripted variables. In *ACM Conference on Programming Languages Design and Implementation*, pages 53–65, June 1990.

[18] J.B. Carter, W.C. Hsieh, L.B. Stoller, M.R. Swanson, L. Zhang, and S.A. McKee. Impulse: Memory system support for scientific applications. *The Journal of Scientific Programming*, 7(3-4):195–209, 1999.

[19] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 1–15. ACM Press, 1991.

[20] J. Bradley Chen and B. D. D. Leupen. Improving instruction locality with just-in-time code layout. In *Proceedings of the USENIX Windows NT Workshop*, pages 25–32, 1997.

[21] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[22] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *ACM Conference on Programming Languages Design and Implementation*, pages 13–24, Atlanta, GA, May 1999.

[23] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *ACM Conference on Programming Languages Design and Implementation*, pages 1–12, Atlanta, GA, May 1999.

[24] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *ACM International Sym-*

posium on Memory Management, pages 37–48, Vancouver, BC, October 1998.

[25] Michał Cierniak, Marsha Eng, Neal Glew, Brian Lewis, and James Stichnoth. Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. *Intel Technology Journal*, 7(1), February 2003. Available at `http://intel.com/\linebreak[0]technology/\linebreak[0]itj/\linebreak[0]2003/\linebreak[0]volume07issue01/\linebreak[0]art01\_orp/\linebreak[0]p01\_abstract.htm`.

[26] Michał Cierniak, Guei-Yuan Lueh, and James Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000.

[27] Michal Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing judo: Java under dynamic optimizations. In *ACM Conference on Programming Languages Design and Implementation*, pages 13–26. ACM Press, 2000.

[28] C. Click. Personal communication, Jan 2006.

[29] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin. Spike: An Optimizer for Alpha/NT Executables. In *USENIX Windows NT Workshop*, pages 17–24, 1997.

[30] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 2001 symposium on Software reusability*, pages 266–277. ACM Press, 2001.

[31] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *ACM Conference on Programming Languages Design and Implementation*, pages 297–302. ACM Press, 1984.

[32] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 92–115, June 1999.

[33] Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *ACM Conference on Programming Languages Design and Implementation*, pages 229–241. ACM Press, 1999.

[34] L. Eeckhout, A. Georges, and K. De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 244–358, Anaheim, CA, October 2003.

[35] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. 5(5):587–616, October 1988.

[36] N. Gloy and M. D. Smith. Procedure Placement Using Temporal-Ordering Information. *ACM Transactions on Programming Languages and Systems*, 21(5):977–1027, 1999.

[37] A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient Procedure Mapping Using Cache Line Coloring. In *ACM Conference on Programming Languages Design and Implementation*, pages 171–182, 1997.

[38] K. Hazelwood and R. Cohn. A cross-architectural interface for code cache manipulation. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 17–27, NY, NY, March 2006.

[39] K. Hazelwood and J. E. Smith. Exploring code cache eviction granularities in dynamic optimization systems. In *International Symposium on Code Generation and Optimization*, pages 89–99, Palo Alto, CA, March 2004.

[40] Urs Holzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, 1996.

[41] 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 *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 69–80, 2004.

[42] X. Huang, J. E. B.Moss, K. S. McKinley, S. Blackburn, and D. Burger. Dynamic SimpleScalar: Simulating Java virtual machines. Technical Report TR-03-03, University of Texas at Austin, Department of Computer Sciences, February 2003.

[43] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. pages 179–188, July 1995.

[44] Jikes Research Virtual Machine (RVM). http://jikesrvm.sourceforge.net.

[45] R. E. Jones and Rafael D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996.

[46] J. Kim and Y. Hsu. Memory system behavior of Java programs: Methodology and analysis. In *ACM Conference on Measurement & Modeling Computer Systems*, pages 264–274, Santa Clara, CA, June 2000.

[47] T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems*, 22(3):490–505, May 2000.

[48] M. S. Lam, P. R. Wilson, and T. G. Moher. Object type directed garbage collection to improve locality. In Y. Bekkers and J. Cohen, editors, *ACM International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 404–425, St. Malo, France, September 1992. Springer-Verlag.

[49] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html, 1997.

[50] Tao Li, Lizy Kurian John, Vijaykrishnan Narayanan, Anand Sivasubramaniam, Jyotsna Sabarinathan, and Anupama Murthy. Using complete system simulation to characterize SPECjvm98 benchmarks. Santa Fe, New Mexico, May 2000.

[51] H. Lieberman and C. E. Hewitt. A real time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[52] C.-K. Luk, R. Muth, H. Patil, R. S. Cohn, and P. G. Lowney. Ispike: A Post-link Optimizer for the Intel Itanium Architecture. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 15–26, 2004.

[53] S. McFarling. Program Optimization for Instruction Caches. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, 1989.

[54] K. S. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[55] Sun Microsystem. The java hotspot performance engine architecture, 1999. http://java.sun.com/products/hotspot/whitepaper.html.

[56] J. E. B. Moss, K. S. McKinley, S. M. Blackburn, E. D. Berger, A. Diwan, A. Hosking, D. Stefanovic, and C. Weems. The DaCapo project. Technical report, 2004. http://ali-www.cs.umass.edu/DaCapo/.

[57] M. Pettersson. Linux Intel/x86 performance counters, 2003. http://user.it.uu.se/ mikpe/

[58] K. Pettis and R. C. Hansen. Profile-guided code positioning. In *ACM Conference on Programming Languages Design and Implementation*, pages 16–27, 1990.

[59] A. Ramirez, J.-L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero. Software Trace Cache. In *International Conference on Supercomputing*, pages 119–126, 1999.

[60] E. Rotenberg, S. Bennett, and J. E. Smith. A Trace Cache Microarchitecture and Evaluation. *IEEE Transactions on Computers*, 48(2):111–120, 1999.

[61] S. Rubin, R. Bodik, and T. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *ACM Symposium on the Principles of Programming Languages*, pages 140–153, Portland, OR, 2002.

[62] D. Scales. Efficient Dynamic Procedure Placement. Technical Report WRL-98/5, Compaq WRL Research Lab, May 1998.

[63] Y. Shuf, M. J. Serran, M. Gupta, and J. P. Singh. Characterizing the memory behavior of Java workloads: A structured view and opportunities

for optimizations. In *ACM Conference on Measurement & Modeling Computer Systems*, pages 194–205, Cambridge, MA, June 2001.

[64] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[65] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001.

[66] T. Suganama, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM Systems Journal*, 39(1), 2000.

[67] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, April 1984.

[68] J. Whaley. Dynamic Optimization Through the Use of Automatic Runtime Specialization. Master's thesis, Massachusetts Institute of Technology, May 1999.

[69] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *ACM Conference on Programming Languages Design and Implementation*, pages 177–191, Toronto, Canada, June 1991.

[70] B. Zorn. Performance in the Age of Trustworthy Computing, January 2004. Presentation at the DaCapo winter meeting. The University of Colorado, Boulder, CO.

Index

Vita

Xianglong Huang was born in Tianjin, China in Jan. 1975, the son of Jianan Huang and Yuying Li. He received a Bachelor of Science degree in Computer Science from the University of Science and Technology of China, Hefei, China in May 1998. He entered the graduate program in Computer Science at the Michigan Technological University in Fall 1998 and got a Master of Science degree in Computer Science in Summer of 2000. He entered the Ph.D. program in Computer Sciences at the University of Massachusetts at Amherst in Fall 2000. In Fall 2001, he joined the PhD. program in Computer Science at the University of Texas at Austin.

Permanent address: 2314 Wickersham Ln
Apt. 1005
Austin, Texas 78741

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.