Fast and Efficient Partial Code Reordering: Taking Advantage of Dynamic Recompilation

Xianglong Huang
The University of Texas at Austin
xlhuang@cs.utexas.edu

Stephen M Blackburn
Intel
Steve.Blackburn@intel.com

David Grove
IBM Research
groved@us.ibm.com

Kathryn S McKinley *
The University of Texas at Austin
mckinley@cs.utexas.edu

Abstract

Poor instruction cache locality can degrade performance on modern architectures. For example, our simulation results show that eliminating all instruction cache misses improves performance by as much as 16% for a modestly sized instruction cache. In this paper, we show how to take advantage of dynamic code generation in a Java Virtual Machine (VM) to improve instruction locality at run-time. We develop a dynamic code reordering (DCR) system; a low overhead, online approach for improving instruction locality. DCR has three optimizations: (1) Interprocedural method separation; (2) Intraprocedural code splitting; and (3) Code padding. DCR uses the dynamic call graph and an edge profile that most VMs already collect to separate hot/cold methods and hot/cold code within a method. It also puts padding between methods to minimize conflict misses between frequent caller/callee pairs. It incrementally performs these optimizations only when the VM is optimizing a method at a higher level. We implement DCR in Jikes RVM and show its overhead is negligible. Extensive simulation and run-time experiments show that a simple code space improves average performance on a Pentium 4 by around 6% on SPEC and DaCapo Java benchmarks. These programs however have very small instruction cache footprints that limit opportunities for DCR to improve performance. Consequently, DCR optimizations on average show little effect, sometimes degrading performance and occasionally improving performance by up to 5%. Our work shows that the VM has the potential to dynamically improve instruction locality incrementally by simply piggybacking on *hotspot* recompilation.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Incremental Compilers, Compilers, Optimization

General Terms Languages, Performance, Experimentation, Algorithms

Keywords dynamic, locality, instruction, JIT compilation

* This work is supported by NSF ITR CCR-0085792, NSF CCR-0311829, NSF EIA-0303609, DARPA F33615-03-C-4106, ARC DP0452011, Intel, and IBM. Any opinions, findings and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISMM'06 June 10–11, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-221-6/06/0006...$5.00.

1. Introduction

The imbalance in memory and processor speeds creates a memory gap that software can help alleviate by improving data and instruction locality, and consequently reduce long latency memory accesses. This paper focuses on improving instruction locality in managed run-times to keep the processor pipeline fed and prevent pipeline stalls. Programming languages with managed runtimes, such as Java and C# are gaining enormous popularity and features such as JIT compilation provide opportunities *at run-time* to reorder instruction, improving their locality.

Most previous work [14, 15, 23, 24, 25] uses static schemes to improve instruction locality. They first profile dynamic call graphs and basic block execution frequencies. Then they apply an expensive algorithm on the profile to generate a code layout which colocates frequent caller/callee pairs together to create good spatial locality and to avoid conflict misses. They apply the pre-computed code layouts at compile or link time. Because these algorithms are expensive (e.g., Pettis/Hansen is $O(n^3)$, where n is the number of methods), they are not suitable for dynamic systems. Dynamic schemes for improving instruction locality include Chen et al. [9]. This approach allocates methods in invocation order and achieves performance improvements. Java Virtual Machines implicitly use this order when they perform lazy compilation. Huang et al. [19] develop several new dynamic code management algorithms based on a complete call graph. Their algorithms regenerate all the code in the program at once, causing a large pause, and are therefore only suitable for long running programs.

This paper introduces dynamic code reordering (DCR) which performs online code reordering with low overhead by piggybacking on just-in-time (JIT) recompilation. DCR 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, DCR allocates hot and cold methods into separate spaces in the heap. DCR 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, DCR 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, DCR applies code padding on either caller or callee method (whichever it happens to be recompiling) to eliminate the potential conflict misses.

DCR piggybacks on adaptive *hotspot* compilation. DCR 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. DCR uses the dynamic call graph and edge profile for the cur-

rent method, and never examines the entire graph nor re-allocates the code, as does the prior work [19, 24]. This design reduces the overhead of DCR to a negligible level.

We run our experiments on two architectures, an Intel Pentium 4 and an IBM PowerPC 970, using SPEC [28, 29] and DaCapo [6] Java benchmarks. Because the instruction working set sizes for these benchmarks are modest compared to the available instruction cache (or trace cache), DCR 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 DCR optimizations improve one benchmark by 5%, but sometimes degrade performance and on average have a negligible impact.

This paper makes the following contributions:

- An instruction locality optimization framework (DCR) that piggybacks on hotspot recompilation to achieve negligible overheads and reduced instruction cache footprints.
- The design of four code space optimizations: (1) one space with all code*; (2) two spaces: separate hot/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.
- 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 DCR has a negligible effect in practice because of the small instruction cache footprint of these benchmarks.

2. Background

We build DCR on the existing adaptive optimization system in Jikes RVM [1, 2, 21], an open source Java virtual machine written almost entirely in a slightly extended Java. We now briefly review relevant aspects of this system. The features we use, dynamic profiling and hotspot recompilation, are typical of many VMs. Section 3 explains DCR and how we integrate it into this framework.

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, and then a separate *optimizing* compiler recompiles frequently executed methods at progressively higher levels of optimization. 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. Jikes RVM feeds this profile data into a cost-benefit model to identify methods to optimize further. Jikes RVM recompiles and optimizes these methods asynchronously on a separate compilation thread. The system uses the profiled caller-callee relationships to build a weighted dynamic call graph that drives profile-directed inlining during optimizing compilation.

When 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 in the optimizing compiler exploit this information. Most relevant to our work is basic block layout. 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 and Hansen's bottom-up positioning algorithm (Algo2) [24].

3. Dynamic Code Reordering

The Dynamic Code Reordering (DCR) system is designed to be 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. DCR 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.)

DCR performs two types of optimizations: *interprocedural* and *intraprocedural* code reordering. When Jikes RVM initially compiles a hot method with its optimizing compiler, DCR allocates the hot method in a separate space from baseline compiled code. DCR also splits the hot method into hot and cold basic blocks based on their execution frequencies and allocates them into different spaces. DCR 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. DCR 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 1(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 1(b) shows 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, DCR allocates the code into the single shared code space.

3.1 Interprocedural Method Separation

Because of lazy compilation and dynamic class loading, baseline compiled code and optimized code will mix in a single code space. The first DCR optimization, *interprocedural method separation*, simply separates hot and cold methods. When the optimizing compiler recompiles a method, DCR allocates these hot methods into a separate hot code space, as shown in Figure 1(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 [5].

3.2 Intraprocedural Code Splitting

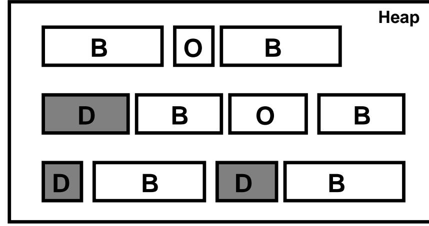
The existing optimizing compiler uses the edge profiling instrumentation from the baseline compiled code to push hot basic blocks to the beginning of the generated code and cold ones to the end. DCR further separates the hot and cold basic blocks by allocating them into different regions of the optimized code space. The generated layout is shown in Figure 1(d), where 'OH' denotes hot basic blocks of a method and 'OC' denotes cold blocks of a method.

DCR splits code during code generation. We implement DCR 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. DCR 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.

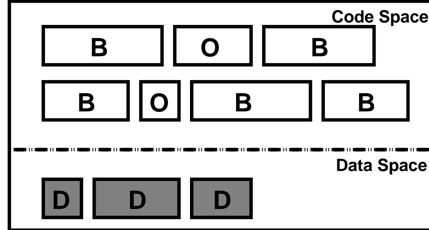
3.3 Code Padding

DCR 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 map into the same line in the instruction cache. After DCR splits a method A into hot/cold blocks, it checks all of the frequent

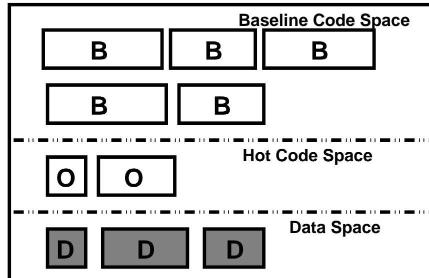
* This design is common in commercial VMs



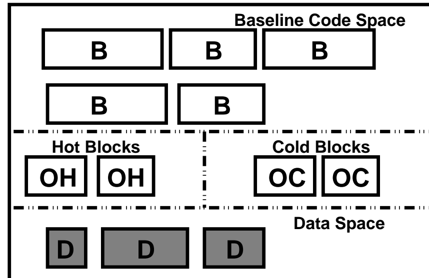
(a) Jikes RVM Default



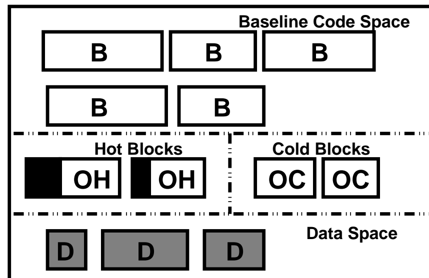
(b) Code Space



(c) Interprocedural Method Separation



(d) Intraprocedural Code Splitting



(e) Code Padding

Figure 1. Code Reordering Heap Layouts: *B: baseline code; O: optimized code; D: application objects; OH: hot basic blocks; CC: cold basic blocks*

```
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 2. Pseudocode for Code Padding

callers and callees of method A to see if their mappings in the cache overlap method A's mapping. If DCR detects overlaps, it employs a simple and fast algorithm to calculate a padding size that avoids conflicts. DCR 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 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 2. For each conflicting method, 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 DCR contiguously allocates with a bump pointer in the code space, DCR applies the padding by simply adding the padding size to the bump pointer before allocating the hot blocks of method A. Figure 1(e) depicts this code layout.

4. Experimental Results

This section evaluates DCR 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 DCR 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 DCR 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 DCR optimizations have little impact, occasionally improving them and occasionally slowing them down.

	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							
_201_compress	173,432	2,208	1,392	180	112	4,248	2,108
_202_jess	355,296	8,400	4,012	29,724	9,820	6,104	3,628
_205_raytrace	220,508	13,560	7,232	15,808	10,736	1,228	960
_209_db	175,640	2,476	1,156	0	0	5,804	3,412
_213_javac	612,128	93,032	42,900	53,720	27,784	2,168	836
_222_mpegaudio	546,512	21,968	8,320	22,104	8,280	6,464	4,116
_227_mtrt	221,032	14,124	7,500	14,700	9,988	1,336	960
_228_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 1. Benchmark Code Size Characteristics in Bytes with Replay Compilation

4.1 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 [4, 20]) 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 [6]. 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 [13]. This methodology gives a simple code space an advantage because more compilation 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 [3].

(2) The *adaptive* methodology lets the optimizing compiler behave as intended, is non-deterministic, and measures compilation and application time. Section 4.5 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 [10].

4.2 Benchmarks and Instruction Code Sizes

We use the SPECjvm98 [28], SPECjbb [29], and DaCapo benchmarks [6]. Other work [6, 12, 20] characterizes the memory behavior and memory system performance of the data for these benchmarks. Table 1 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 1 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.

4.3 Simulation Results

To examine the potential of DCR's capability to remove conflict misses, we use Dynamic SimpleScalar (DSS) [18], a variant of widely used SimpleScalar simulator [8] 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. The base DSS configuration uses an aggressive processor model with five-stage pipeline. The details of this standard 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

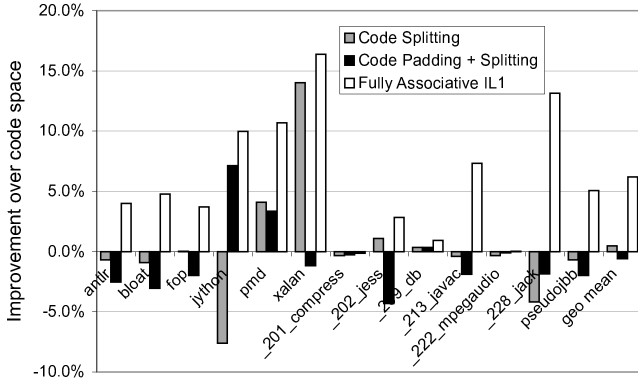


Figure 3. Simulation Results for Directed-Mapped and Fully Associative 32 KB IL1, 512 KB L2

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 3 compares the relative performance of DCR using as its baseline hardware instruction cache configuration (1) with a simple code space. It shows the benefits of DCR code splitting, code splitting, padding, and a fully associative instruction cache (hardware configuration (2)). DCR code splitting and padding performs 7.1% better than a simple code space on jython although just DCR code splitting itself degrades the performance 7.6%. DCR 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%. Although the geometric mean of DCR 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 DCR 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 DCR is not consistently able to achieve that potential.

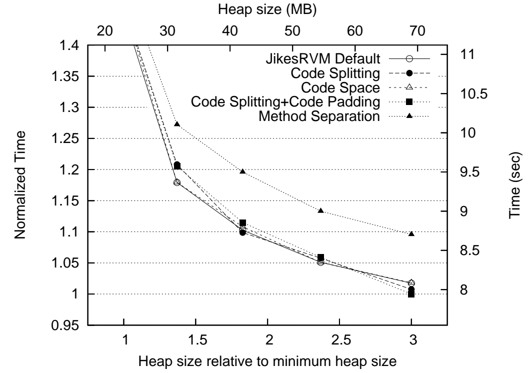
4.4 IA32 and PowerPC Performance Results

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

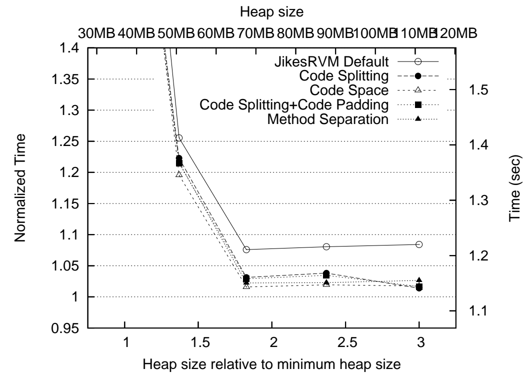
3.2GHz Pentium 4 with hyper-threading enabled, a 64 byte L1 and L2 cache line size, an 8 KB 4-way set associative L1 data cache, a 12 Kμops L1 instruction trace cache, a 512 KB unified 8-way set associative L2 on-chip cache, and 1 GB main memory running Linux 2.6.0.

1.6 GHz 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.

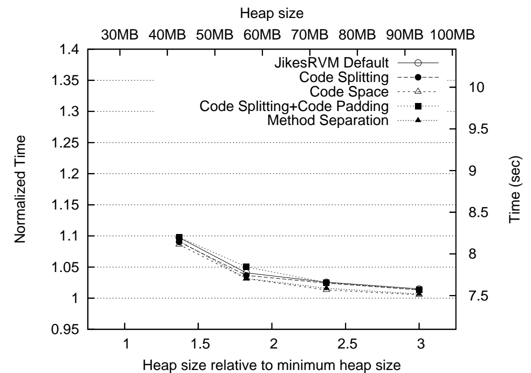
For each experiment we report, we run the experiment five times, interleaving the compared systems. We use the methodologies



(a) antlr



(b) fop



(c) Geometric Mean

Figure 4. Code Optimizations on Pentium 4

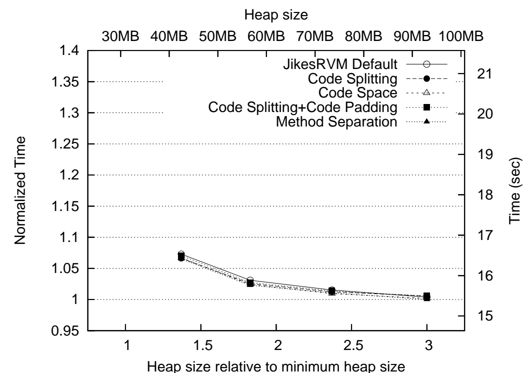


Figure 5. Code Optimizations on PowerPC 970: Geometric Mean

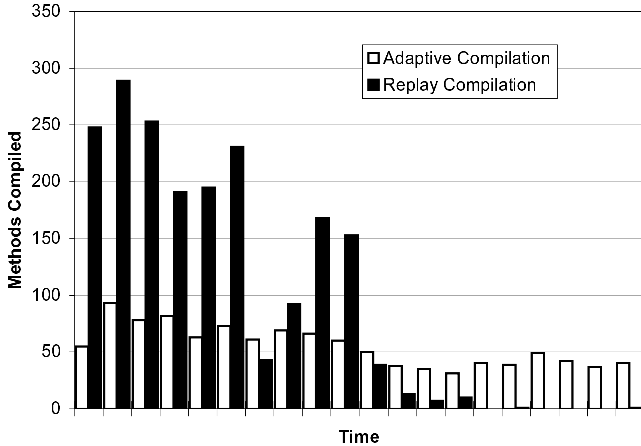


Figure 6. Compiler Activity Histogram on First Iteration

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 good application of the adaptive compiler. We use a generational copying collector with a 4MB bounded nursery with five heap sizes ranging from the smallest in which the application runs up to three times larger [4, 6].

4.5 Application Performance

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 DCR 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 4 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. For fop, mixing code and data in the heap degrades its performance and DCR optimizations perform worse than just having a simple code space. DCR optimizations perform about the same as Jikes RVM default configuration for the geometric mean over all benchmarks.

We also performed the same experiments on the PowerPC which has a traditional instruction cache instead of the instruction trace cache on the Pentium 4. Figure 5 shows these results. DCR 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 3).

4.6 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 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 7 shows the performance of various configurations of DCR 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 counters configured for this measurement. We measure all the programs and present the geometric mean and three programs (_213_javac, _227_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.

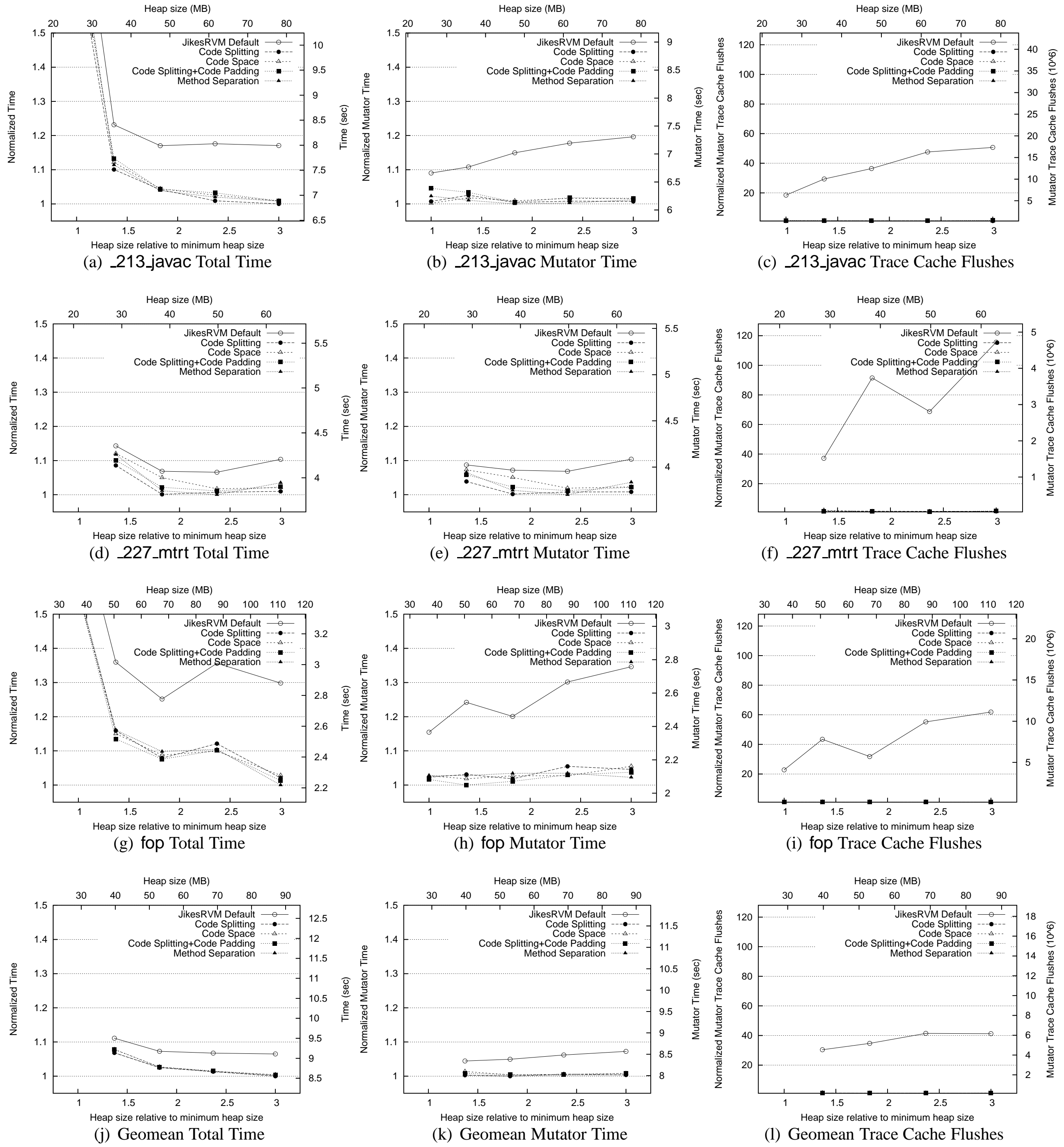
The results show that a simple code space improves over the Jikes RVM default configuration by 6% on average and by 30% on fop. There are two reasons for this large difference in performance.

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 7(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.

DCR optimizations offer some additional, but modest improvements on a few programs. For example, on _227_mtrt, DCR code splitting is most effective, and improves over a code space by up to 5%. These results demonstrate that DCR 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.

5. Related work

Numerous researchers have studied the problem of restructuring programs to improve memory performance. Much of the early software-based work was aimed at reducing virtual memory page faults. Some current work also tries to minimize these very expensive faults [31]. However, most recent work focuses on static and dynamic approaches for reordering code to reduce instruction and ITLB misses with offline and online profiling.



(a) _213_javac Total Time

(b) _213_javac Mutator Time

(c) _213_javac Trace Cache Flushes

(d) _227_mtrt Total Time

(e) _227_mtrt Mutator Time

(f) _227_mtrt Trace Cache Flushes

(g) fop Total Time

(h) fop Mutator Time

(i) fop Trace Cache Flushes

(j) Geomean Total Time

(k) Geomean Mutator Time

(l) Geomean Trace Cache Flushes

Figure 7. Total time, mutator time, and trace cache flushes for a simple code cache, Jikes RVM default and various DCR configurations.

5.1 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, 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 [23] 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 [24] 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. 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 [19].

Cohn et al [11] 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 [22] 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. [15] 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 better performance than Pettis and Hansen. Gloy and Smith [14] 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. [25] developed a code reordering system, called the Software Trace Cache (STC), that not only tries to improve the instruction cache hit rate, but also 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 $O(n^3)$. These limitations make them less useful in the context of virtual machines.

5.2 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 [9] . 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 being a static approach, it is less effective because the procedures executed typically 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.

Scales' DPP (dynamic procedure placement) system uses runtime information to dynamically lay out procedure code [27]. DPP uses a loader component that is invoked on procedure calls. This 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 [30] 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 1(a)). In contrast, DCR separates code from data objects in the heap which sometimes improves performance. Furthermore, DCR pads conflicting hot caller/callee pairs when the methods are recompiled, and does not wait until garbage collection.

Huang et al. [19] developed several more efficient algorithms for generating a code layout. Their algorithms are up to 6000 times faster than the popular Pettis-Hansen algorithm. However, they use fairly expensive instrumentation to gather their profile data and perform a complete reorganization of all the compiled code. Both of these factors result in large overheads for short running applications. Our techniques try to allocate code in the right place when it is generated and piggyback on the natural actions of the adaptive recompilation system. This organization achieves significantly lower overheads than previous approaches, and may have the potential to obtain speedups on programs with modest running times.

Recent research [7, 16, 17, 26] 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, 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 [9], nor rely on expensive page protection [27], nor does it require special hardware [26, 17], and it is implemented in a JVM [30].

6. Conclusions

This work develops and throughly evaluates a dynamic code reordering system specifically tailored for use in a virtual machine. By exploiting existing online profiling mechanisms and piggybacking on the activities of the adaptive recompilation system, DCR seeks to improve instruction locality in a completely online fashion with negligible overhead. DCR employs three optimizations (1) interprocedural hot/cold method separation, (2) intraprocedural hot/cold code splitting, and (3) interprocedural hot code padding that together improve instruction locality by reducing both capacity and conflict misses.

We have completely implemented DCR in Jikes RVM for both the IA32 and PowerPC architectures and present experimental results using the SPEC Java and DaCapo benchmark suites on Intel Pentium 4 and PowerPC 970 machines. We also present simulation results that demonstrate the effectiveness of DCR at eliminating conflict misses. Overall, the results show that instruction locality can have an important impact on overall performance of some Java applications. Although on average the improvement is minor, DCR occasionally improves performance by improving instruction locality and thus merits further investigation on larger benchmarks.

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Architecture and policy for adaptive optimization in virtual machines. Technical Report 23429, IBM Research, Nov. 2004.

[2] 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.

[3] M. Arnold, A. 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.

[4] 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.

[5] 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.

[6] 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, Mar. 2006. http://ali-www.cs.umass.edu/DaCapo/-Benchmarks.

[7] 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, Mar. 2006.

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

[9] J. B. 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.

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

[11] 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.

[12] 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.

[13] L. Eeckhout, A. Georges, and K. D. 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, Oct. 2003.

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

[15] 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.

[16] 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, Mar. 2006.

[17] 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.

[18] 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, Feb. 2003.

[19] X. Huang, B. T. Lewis, and K. S. McKinley. Dynamic code management: Improving whole program code locality in managed runtimes. In *International Conference on Virtual Execution Environments*, Ottawa, Canada, June 2006.

[20] X. Huang, Z. Wang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, and P. Cheng. The garbage collection advantage: Improving program locality. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, BC, 2004.

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

[22] 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.

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

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

[25] 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.

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

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

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

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

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

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