Dynamic Code Management

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

Brian T Lewis
Intel Corperation
brian.t.lewis@intel.com

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

ABSTRACT

Poor code locality degrades application performance, especially for large programs such as server applications, by increasing the memory stalls caused by instruction TLB and instruction cache misses. This locality problem is a particular issue for languages such as Java and C# that provide just-in-time (JIT) compilation, dynamic class loading, and dynamic recompilation. Managed runtimes for these languages offer the opportunity to dynamically profile an application's execution and adapt that execution to improve performance. This paper describes the dynamic code management system (DCM) of our managed runtime that uses dynamic profile information to reorder JIT-compiled code when necessary to improve locality.

To place method code, our DCM system supports the widely-used Pettis-Hansen procedure placement algorithm. In addition to this algorithm, we also developed and implemented three new placement algorithms. Our new algorithms can run up to 6000 times faster than the Pettis-Henson algorithm and still achieve the same or better performance improvement, which makes them more suitable for use in high-performance managed runtimes. The new algorithms specifically target ITLB misses, which typically have the greatest impact on performance.

Using the DCM, we demonstrate a 6% performance improvement with our new Graph-Walking layout algorithm for the large MiniBean benchmark, a single-process version of the SPECjAppServer2002[1] enterprise application server benchmark. This performance is 4% better than that provided by Pettis-Hansen placement. Furthermore, instead of the prohibitively expensive 35 minutes Pettis-Hansen layout requires, Graph-Walking layout needs just 0.35 second (6000 times faster).

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

[1]We use the SPEC benchmarks only as benchmarks to compare the performance of the various techniques within our own VM. We are not using them to compare our VM to any other VM and are not publishing SPEC metrics of any kind.

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.
VEE'06, June 14-16, 2006, Ottawa, Canada.
Copyright 2006 ACM ...$5.00.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Memory management (garbage collection)*

General Terms

Languages, Performance, Experimentation, Algorithms

Keywords

dynamic, locality, instruction, compiler, JIT

1. Introduction

Because of the widening gap between processor and memory speed, memory latency has become a major performance bottleneck. Applications spend much of their time waiting because of cache and TLB misses. On the Intel® Itanium® 2 processor, for example, the SPEC JBB2000 server benchmark spends 54% of its cycles waiting for memory. While the majority of stalls are from data access, instruction-related misses are also important. The stalls caused by instruction cache misses, instruction TLB (ITLB) misses, and branch mispredictions can be a major performance bottneck for large programs such as server applications.

There are a number of commonly-used techniques used by compilers to improve code layout. For example, compilers often arrange basic blocks within a procedure to decrease branch mispredictions. They may also use procedure splitting [11]: by allocating frequently-called basic blocks separately from infrequently-called ones, they reduce the number of active code pages and so the number of ITLB misses.

A number of after-compilation tools have also been developed to optimize code placement. For example, Microsoft uses a profile-based tool [15] to reorder the code of their applications for better performance. Similarly, the Spike post-link optimizer [5] uses a number of techniques to improve code locality, and has been used to improve the performance of several large programs including the Oracle 11i application server and the TPC-C benchmark by as much as 30%.

In this paper we are concerned with code layout in Java and C# managed runtimes. These runtimes employ JIT (just-in-time) compilers, dynamic code generation, and often reoptimization. They typically allocate compiled code sequentially. This tends to keep callers close to their callees, which Chen and Leupen [2] showed can improve performance. However, even if the original code layout performs well, methods may be recompiled. Furthermore, new classes may be loaded at runtime, often varying based on application data. The result is that when one method calls another, the caller's code may be some distance away from that of the callee. In general, the code for applications, especially large applications, can occupy many virtual memory pages. If control jumps around the code region frequently, there will be numerous costly ITLB misses as well as many instruction cache misses.

Because of the above, we believe managed runtimes should include *code management* in addition to JIT compilation, garbage collection, exception handling, and the other services they provide to executing code. Code management actively manages JIT-allocated code in order to improve its locality and so improve application performance. Managed runtimes should use profile information from hardware- or software-based monitoring to reorder method code when necessary to maintain good instruction locality.

This paper describes the implementation of a dynamic code management system (DCM) that is integrated into our managed runtime. The DCM uses profile information to reorganize the compiled code of methods. We show that our DCM 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 (e.g., on IPF, they require processing by the operating system). One of these algorithms, the Graph-Walking procedure layout algorithm, is significantly faster both in complexity and in practice than the best-known previous technique, Pettis-Hansen procedure layout [11]. We demonstrate that Graph-Walking generates code layouts that are comparable to those generated by Pettis-Hansen layout.

This paper makes the following novel contributions:

- A DCM system. The first implementation of dynamic code reordering in a managed runtime. 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% on a 4-processor Intel® Xeon® processor.
- A new code placement algorithm called Graph-Walking layout. 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.

The rest of this paper is organized as follows. Section 2 presents an overview of our DCM system while Section 3 describes our current implementation. Next, Section 4 describes the Pettis-Hansen procedure layout algorithm and presents our new layout algorithms. We present our experimental results in Section 5. Related work is discussed in Section 6.

2. Dynamic code management overview

This section gives an overview of our dynamic code management system. We describe its implementation in more detail next, in Section 3.

As the application executes and its behavior changes, the DCM reorganizes compiled code as necessary when miss rates for the ITLB or instruction cache become too high. A new layout is calculated, methods are moved, and code pointers in methods, thread stacks, and data structures of the managed runtime are updated to reflect the new locations.

Figure 1 depicts the the dynamic code management system's components and their interactions.

Briefly, code reorganization works as follows: Profile information is gathered during application execution and used to build a dynamic call graph. The dynamic call graph (DCG) is an undirected graph with a node for each method and, when one method calls another, an edge between their nodes weighted by the number of calls. When a reorganization is triggered, the DCM uses a code layout algorithm that

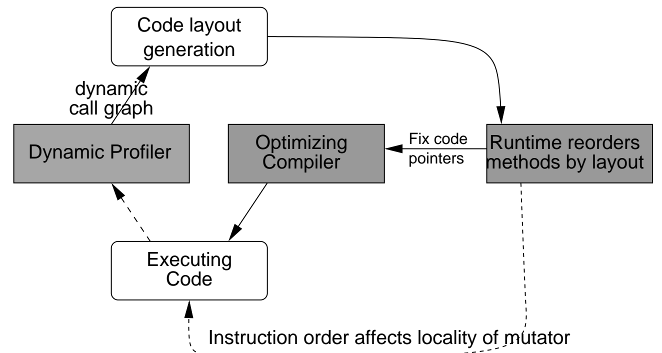

Figure 1: Dynamic Code Management

computes a new placement for each method's code based on the DCG. The DCM then moves the code and does any required code updates.

Profile information can be gathered using software instrumentation or PMU (hardware performance monitoring unit) sampling. PMU sampling is often less expensive than software instrumentation, as it is on the Intel® Itanium® Processor Family (IPF) processors[2]

The DCM 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 4. These 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, code can be updated by having the JIT fix up any code that it compiled. This gives JITs more freedom in how they emit code. It also means the code management system and the rest of the managed runtime do not need to understand the representation of JIT-compiled code.

In addition to dynamic code layout, our managed runtime also supports *static code layout*. This resembles the DCM, but uses a separate profiling run to build the DCG, run the code layout algorithm, and write the resulting code layout to a file. Then subsequent runs can use this layout file to place their compiled code. When a static code layout is used, our managed runtime first reads the layout, then uses its placement information when later allocating code for JITs.

Our DCM implementation is incomplete on IPF. As a result, we use static code layout there to explore the use of PMU-based profiling and to estimate the performance benefit that DCM can provide.

3. DCM Implementation

We implemented our DCM system in a managed runtime environment that supports Java and C# programs. Our core platform consists of the Open Runtime Platform virtual machine (ORP) [3] and one or more JIT compilers. On IPF, we use StarJIT[1], a high-performance dynamic compiler that uses a single SSA-based intermediate representation and global optimization framework to compile Java (i.e., JVM bytecodes). On IA-32, we use the optimizing O3 Java JIT that 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.

StarJIT does hot-cold method splitting, so a method's compiled code may consist of more than one separately-allocated *code block*. Because of this, code blocks are the units of code management in our DCM system, not methods, and the DCM reorders code blocks.

ORP allocates code in a region of memory that is separate from the

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

garbage collected heap. Different subregions are provided for cold (infrequently executed) code and warm (more often executed) code. Code of equal "warmth" is allocated sequentially.

To support dynamic code management, we modified StarJIT and O3 to support our interface for updating code after it has been moved. The DCM calls the JIT to fix up each relocated code block. The JIT corrects any pointers to compiled code used in the code block, including references into other code blocks as well as into the same code block. Since StarJIT may hold code addresses in processor registers, it must also update any registers holding code addresses.

We collect dynamic profile information from either software instrumentation or from a hardware performance monitoring unit (PMU). On IPF, we prefer PMU sampling since this is less expensive than software instrumentation[3]. 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, the DCM discovers information about recent method calls. This information allows us to identify both the caller and the callee methods, or more precisely, which code blocks have been called along with their calling code blocks. We separate calls instructions from *return* instructions by checking if the target address is at the beginning of a code block.

On the Intel Pentium 4 processor, we found that using the PMU is too expensive. In particular, capturing the LBR (last branch record) on this processor requires hundreds of cycles to access since it needs to flush the pipeline and do a memory fence. As a result, on IA-32 we use software instrumentation for profiling. We could have modified our JITs to do the instrumentation, but we decided on a simpler approach: we interpose on method calls to record the caller and the callee. This is done by generating a small machine code stub for each compiled method that is actually entered 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. However, it is relatively expensive. Software instrumentation can also be used on IPF machines.

To reorganize code, the DCM 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 we found it easier to move all code to a newly-allocated code region. This also simplifies debugging our reorganization implementation, since this makes it easy to recognize a reference to an old code location.
4. Fix up the metadata for each method.
5. Call the JIT to fix up each code block.
6. Update the call stack of each thread. In particular, fix up any code addresses such as return addresses on stacks.
7. Restart the managed threads.

To compute the new layout, the DCM uses one of several different code layout algorithms. Each of these operates on the DCG produced during profiling and creates a code layout. This layout identifies sequences of code blocks that should be placed together in memory in that order. So far, four code layout algorithms have been implemented.

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

The DCM currently stops application threads while it does all reorganization work. Much of the DCM's work—in particular, calculating the new code layout—could be done concurrently with application threads to minimize the pause time. Those threads need to be stopped only during the update of the metadata and thread stacks. Our experience is that most of the time needed to reorganize code is due to the new layout calculation; the remaining steps are done quickly.

Our DCM 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, the DCM is completely independent of GC and can be done 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 the DCM's use of PMU information to monitor ITLB and other instruction-related misses in order to determine when reorganizations are needed. We currently allow the user to specify when to do a reorganization, and we typically reorganize code once at the end of application warm-up. Despite this, our DCM is capable of reorganizing code multiple times and whenever necessary.

In many ways, our DCM system resembles a copying garbage collector. It moves objects (methods) 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 [8]. It also supports pinning of objects that would be too hard or too expensive to relocate. In addition, although our DCM does not currently do this, it would be straightforward to modify it to do much of its work (e.g., new layout calculation) in parallel with application threads. In the future, we might also investigate having the DCM reclaim no-longer-needed code: code that is currently not referenced by any thread and not likely to be needed again soon.

4. The code layout algorithms

This section describes the code layout algorithms we evaluated in our work. These all use the same underlying data structure, the dynamic call graph (DCG), and produce a new code layout. The first of these algorithms is the Pettis-Hansen algorithm. While the Pettis-Hansen algorithm usually improved performance of large applications, it was expensive to run. Pettis-Hansen could create a new layout for SPEC JBB2000 (758 methods) in less than 150ms, but it required minutes for MiniBean (15,586 methods). This led us to develop three new, faster algorithms. The remainder of this section discusses each algorithm, while Section 5 presents results for their run times and the resulting performance improvements.

4.1 Pettis-Hansen algorithm

Pettis-Hansen places methods using a greedy "closest is best" strategy. During the algorithm's operation, each node contains one or more code blocks that have already been placed in some order, and this sequence is preserved in the final code layout. Each of the call graph's nodes initially contain a single code block. The algorithm repeatedly chooses the edge $A \rightarrow B$ of highest weight (i.e., greatest calling frequency), then combines the nodes A and B. This consists of placing A and B's code next to each other then merging their outgoing edges. If outgoing edges point to some node C, then the merged edge is given a weight equal to the sum of the original weights.

When nodes A and B are merged, their code is placed using the heuristic described by Gloy and Smith [6] (line 7 of Figure **??**). We find the hottest call edge between one node of A and one of B in the original call graph, and then order the merged code blocks to minimize that edge's call distance in bytes. The algorithm terminates when no

```
PETTISHANSEN(Graph)
1  while (edge ← HEAVIESTEDGE(GRAPH))! = NULL
2  do (nodeA, nodeN) ← 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 2: Pettis-Hansen procedure layout algorithm

```
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 3: Cache-Aware Pettis-Hansen algorithm

edges remain. The pseudo-code for the Pettis-Hansen algorithm is shown in Figure 2.

4.1.1 Time complexity

The Pettis-Hansen algorithm's asymptotic complexity is $O(ne)$, where n is the number of nodes in the graph and e the number of edges. Pettis-Hansen merges at most n nodes. For each node merge, the edge with maximum weight must be found, the code blocks of the connected nodes combined, and their outgoing edges merged. Finding the maximum edge is $O(e)$. While the number of edges can be as high as n^2, in practice the Pettis-Hansen complexity is likely to be $O(n^2)$. This complexity explains the dramatic increase in time required to place the code for large applications such as MiniBean.

4.2 Cache-Aware Pettis-Hansen algorithm

In our search for a faster placement algorithm, we realized that there is little locality benefit in putting two methods on different pages, even if the two methods are on consecutive pages. We modified Pettis-Hansen to stop merging methods into the current code after enough methods have been merged to fill a page. We found 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 3

Cache-Aware Pettis-Hansen is not guaranteed to generate the same layout as Pettis-Hansen. For example, Pettis-Hansen generates a layout of *DABC* for the DCG in Figure 4. But if node *A* and node *B* are both larger than the page size, the new algorithm generates layout *CDAB*. Note here that *C* and *D* are adjacent, but are not with 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. But with other node sizes, the result could be very different. None of the four algorithms is guaranteed to produce the best improvement. However, our main concern is the layout generation time, and we show in the next section that Cache-Aware Pettis-Hansen runs much faster.

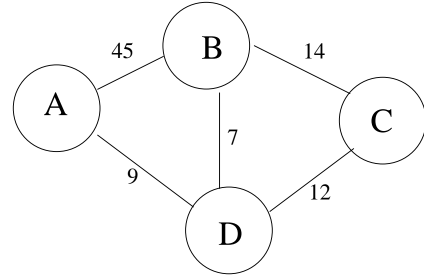

Figure 4: An example dynamic call graph

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

Figure 5: Graph Walking algorithm

A further refinement for Cache-Aware Pettis-Hansen is the following. 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 this, if the architecture has a direct-mapped instruction cache, the algorithm should use the cache size instead of the page size.

4.2.1 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.

4.3 Graph Walking 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 *Graph Walking 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 connected to *A* and merges *A* with the node *B* at the other end of that edge. If this algorithm merges any nodes, it produces a 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. Performance results in the next section show this algorithm performs well both in run time and in the locality benefit it produces.

4.3.1 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 $O(n)$. As a result, the asymptotic complexity of the Graph Walking algorithm is $O(n^2)$, which is less than the asymptotic complexity of Pettis-Hansen.

4.4 Linear Scan algorithm

```
LINEARSCAN(Graph)
1  for each node in Graph do
2    for each edge in node.edgeList do
3      if edge.heat > Threshold
4        then ATTACHNODES(edge, node)
5        else REMOVEEDGE(edge)

ATTACHNODES(edge, nodeA)
1  nodeB ← edge.OTHERNODE(nodeA)
2  nodeA.edgeList.ATTACH(nodeB.edgeList)
3  nodeA.blockList.ATTACH(nodeB.methodList)
```

Figure 6: 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). This algorithm is shown in Figure 6. Notice here that when merging two nodes, we do not merge their out edges: even if two edges connect to the same node, they are not merged. We do this because merging edges is especially expensive.

4.4.1 Time complexity

Assume there are n nodes in the graph. The algorithm scans n nodes. For each node, it examines each out edge once, and in the worst case there are n out edges. As a result, the asymptotic complexity of Linear Scan is $O(n^2)$ but in practice is much smaller.

5. Results

We evaluated our DCM code management system by measuring its benefit and runtime overhead for the SPEC JVM98, SPEC JBB2000, and MiniBean benchmarks. This was done using each of the four code layout algorithms. We also studied the benefit and cost of using PMU information on the IPF (Itanium® processor family) machines to generate profile information.

5.1 Experimental framework

We performed our experiments using two 4-processor machines with different architectures. The default 4KB page size was used on both systems.

1. **Xeon® System** The first machine we used is a 4-way Intel® Xeon® server with 2GHz Xeon processors running Windows 2000 Advanced Server Edition. This machine has a 400MHz system bus. Each processor has an 8KB 4-way set associative L1 data cache, a 8-way 12Kμops L1 instruction trace cache, a 512KB unified 8-way set associative L2 on-chip cache, and a 2MB 8-way L3 cache. In addition, the L1 and L2 cache line size was 64 bytes. This machine's ITLB has 128 entries and is 4-way set associative. HyperThreading was disabled.

2. **Itanium 2® System** The second machine we used is a 4-way Intel® Itanium® 2 system with 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.

For our experiments, we used the SPEC JVM98, SPEC JBB2000, and MiniBean benchmarks. MiniBean is a single machine, single process version of SPECjAppServer2002. When collecting our results, we ran the benchmarks five times to collect each result, then chose the best time. Table 1 shows the characteristics of these benchmarks. We ran MiniBean using a 512M heap, SPECjbb with a 256M heap, and the SPECjvm98 benchmarks with a 50M heap.

Benchmark	IA32 Size	IA64 Size	Methods	Invocations
SPECjbb	268K	468K	758	N/A
MiniBean	3.10M	10.97M	15586	729.46M[4]

Table 1: Benchmark characteristics

Interval	Time (s)
10	99.67
100	100.67
1000	99.67
10000	95.67
100000	94.67
Base	95.00

Table 2: MiniBean DCG creation times with PMU sampling

5.2 DCM overhead

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

5.2.1 Dynamic call graph generation overhead

As we described before, DCM generates the dynamic call graph using information from either software instrumentation or performance counters.

We studied the overhead for dynamic call graph generation with PMU sampling on the IPF machine. The sampling interval was varied 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 Table 2. The times shown here are from program start until the DCM generates and applies the new code layout. This means that the times do not reflect any benefit from using the new code layout. As we increased the sampling interval to 10,000 or higher, the overhead dropped to less than 1%. In addition, our PMU driver has the ability to change the hardware sampling interval at runtime. As a result, if it proved necessary, we could sample more frequently for a short period of time, then revert back to longer-interval sampling.

We also measured the overhead of using the PMU to generate the dynamic call graph for the SPEC JVM98 benchmarks. The results are shown in Figure 7. The results are 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 measured the overhead by comparing one run using PMU-based code layout with a second run for the base case (no PMU sampling, no code reordering). The first run did all the work of generating a new code layout but did not actually apply it.

On the IA-32 system, we used software instrumentation because it is expensive to use the Intel Pentium 4 PMU to capture the LBR (last branch record). The runtime overhead for our software instrumentation is shown in Table 3. Again, the time shown here is the time from program start until the DCM generates and applies the new code layout. As expected, this software instrumentation is expensive: 21.57% for MiniBean. It executes a small number of data structure operations and an extra branch on every method call. This overhead could be decreased substantially by inlining it into JIT-compiled code and by using a different data structure.

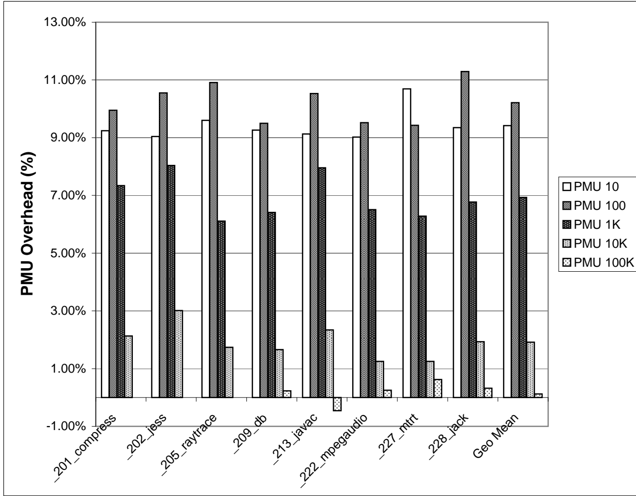

Figure 7: PMU Overhead for the SPEC JVM98 Benchmarks

Frequency	Time (s)
Software Instrumentation	66.167
Base	54.429

Table 3: MiniBean DCG creation times with software instrumentation

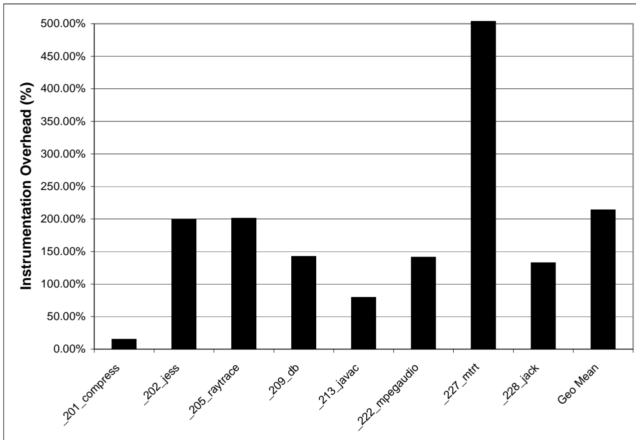

Figure 8: SPEC JVM98 software instrumentation overheads

We expected the overhead of our software instrumentation implementation would be high for the SPEC JVM98 benchmarks, and this was confirmed by our measurements. The overheads are shown in Figure 8. The geometric mean of the overheads is about a factor of 3. This indicates that if software instrumentation is used, a faster implementation is needed, especially for smaller applications. As we discussed earlier, it would be much better to have the JIT compiler insert code to do this, and to use a more efficient data structures to collect the call information.

5.2.2 *Code layout generation overhead*

Another overhead of our DCG system is the time needed for the code layout algorithms to generate a code layout. We show the times required for MiniBean and SPEC JBB2000 on the IA-32 machine in Table 4.

The Pettis-Hansen procedure layout algorithm requires 37 minutes

Algorithm	MiniBean	SPECjbb
Pettis-Hansen	2215127	503
Cache-aware Pettis-Hansen (16K)	26012	197
Cache-aware Pettis-Hansen (4K)	28840	186
Graph Walking (16K)	508	17
Graph Walking (4K)	352	15
Linear Scan (1)	203631	164
Linear Scan (10)	138014	80
Linear Scan (100)	69855	42
Linear Scan (1000)	29952	39
Linear Scan (100000)	257	29

Table 4: MiniBean and SPEC JBB2000 layout times (s)

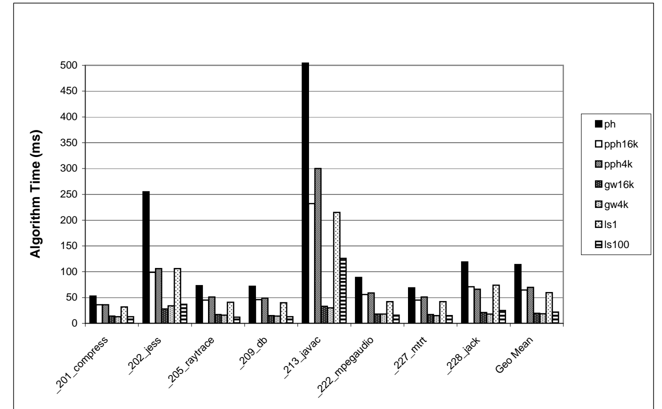

Figure 9: SPEC JVM98 layout generation times (ms)

to reorder MiniBean's code. This is much too long to be practical in a dynamic code reordering system. Our new algorithms are much faster. This is especially true for the Graph Walking algorithm, which takes just 0.3 seconds for MiniBean when using a 4KB page as the cut off threshold. This is less than most of MiniBean's garbage collection times.

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 Graph Walking with a 4KB threshold). The results are shown in Figure 9. In this figure and the following ones, "ph" stands for the Pettis-Hansen algorithm, "pph" for Cache-Aware Pettis-Hansen, "gw" for Graph-Walking, and "ls" for the Linear Scan algorithm.

5.3 DCM performance results

Table 5 shows the performance benefit of dynamic code reorganization on our IA-32 machine with both the MiniBean and SPEC JBB-2000 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 the invocation order. This already provides some locality benefit, as we noted in Section 1. No other applications were run at the same time. The MiniBean rate reported in the second column is the harmonic mean of the four throughput rates it reports. For SPEC JBB2000, we used the 8-warehouse score.

These results show that our DCM system with the Graph-Walking algorithm can significantly improve MiniBean's performance. However, it has essentially no impact on SPEC JBB2000. The reason for these difference is the size of the two benchmarks. The IA-32's 128-entry ITLB can map only 512K of simultaneous code space with the default 4K pages. This is much less than MiniBean's 3.1MB of JIT-compiled code, so optimizations that improve its code locality can help performance. SPEC JBB2000, on the other hand, only has 268K

Algorithm	MnBean	Improv	JBB	Improv
Base	7.263	——	42737	——
Pettis-Hansen	7.432	2.33%	43058	0.75%
Cache-Aware PH (16K)	7.497	3.22%	42710	-0.06%
Cache-Aware PH (4K)	7.622	4.94%	42671	-0.15%
Graph Walking (16K)	7.700	6.02%	42447	-0.68%
Graph Walking (4K)	7.524	3.59%	42346	-0.91%
Linear Scan (1)	6.378	-12.19%	41282	-3.40%
Linear Scan (100)	7.363	1.38%	42136	-1.41%

Table 5: MiniBean and SPEC JBB2000 performance on IA-32

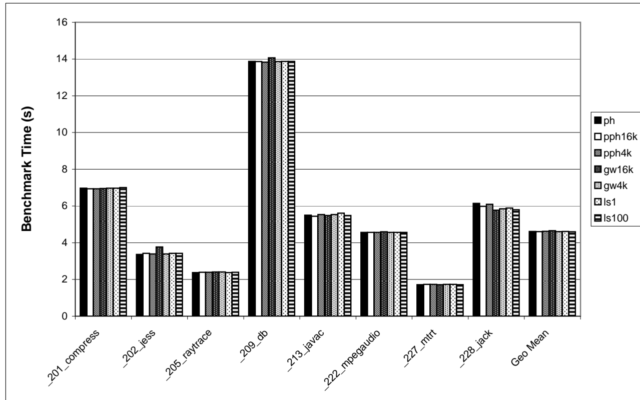

Figure 10: SPEC JVM98 performance on IA-32

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 applications are running, there may be some benefit since improving a program's code locality will reduce its working set, allowing more applications to run simultaneously without ITLB misses.

Figure 10 shows the run times for the SPEC JVM98 benchmarks with different code layout algorithms on the IA-32 machine. We measure the times for the second iteration of the benchmark runs, so these times do not include any overhead from instrumentation or code reorganization. There is no clear performance benefit from using any layout algorithms. Since the benchmarks are all smaller than SPEC JBB2000, this is not surprising.

Since our DCG system is not fully implemented on IPF, we could not collect performance results there for dynamic code reorganization. However, we were able to use static code layout to get an approximation of what DCG might provide. As we mentioned above, static code layout uses a separate profiling run to build the dynamic call graph, run the code layout algorithm, and write the new layout to a file. This layout file is used on subsequent executions to place JIT-compiled code. One drawback of static code layout is that it can't cope with methods that were never compiled in the profiling run. This can happen because of dynamic class creation and loading (which is done by MiniBean and SPECjAppServer2002) or if the application chooses to call methods that were never called in the profiling run. As a result, the benefit of static code layout can be less than dynamic layout.

On our IPF system, we used static code layout with the Pettis-Hansen layout algorithm to determine its performance impact for the MiniBean benchmark. As shown in the Table 6, the performance improvement is actually negative. While it is possible that DCG would provide better performance, its benefit is still likely to be less than it was on the IA-32 machine. One reason for this poor result is the large L3 cache (9M) on our IPF machine, which allows it to hold

Algorithm	Score	Improvement
Base	17.96	
Pettis-Hansen	17.78	-1.00%

Table 6: Pettis-Hansen performance estimate on IPF

	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 7: Front-end stalls using static code layout on IPF

Base	Soft	PMU10	PMU100	PMU1000	PMU10000
18904	13876	13706	13825	13681	13882

Table 8: Effect of PMU-based call graphs on ITLB misses

nearly all the code (11MB) of MiniBean. Another reason is the short memory stall time on the Itanium® 2 processor: the latency is approximately 6 cycles for 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.

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 7 shows the number of 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 used a static code layout based on a DCG based on software instrumentation. Among the stalls we measured are ITLB misses and instruction cache misses. We also collected the front end stalls, which are the number of cycles the back end of the Itanium® 2's execution pipeline spent waiting for instructions from the pipeline's front end.

There is a 26.60% improvement in ITLB miss stalls with the static code layout even though there is no noticeable improvement in either the total front-end stalls or MiniBean's overall performance.

We also measured the effectiveness of PMU-generated dynamic call graphs. Using PMU sampling at different branch intervals, we computed static code layouts based on the resulting DCGs. We then collected the total ITLB misses for MiniBean using the IPF performance counters. The results are shown in Table 8. These results indicate that, for MiniBean, PMU-generated dynamic call graphs are as effective as software instrumentation-based graphs for DCG. These results are consistent with our other experience that PMU-based instrumentation has the same benefit as software-based instrumentation, but at lower-cost.

5.5 Discussion

These results demonstrate that dynamic code management can significantly improve the performance of larger applications such as MiniBean. However, this benefit depends on both the application size and the processor's microarchitecture and cache hierarchy. Code reorganization helps large applications more than small ones. It also 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 can benefit from code reorganization.

This section's results also demonstrate that our Graph-Walking layout algorithm is more suitable for code reorganization while an application is running than the classic Pettis-Hansen algorithm. It runs much faster and can produce better performance.

Finally, these results show that PMU sampling is low-overhead and can be more effective as software-based instrumentation.

6. 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; see, for example [15]. However, most recent work has focused on reducing instruction and ITLB misses. These efforts can be organized into static and dynamic approaches. The static techniques are used at compile- or link-time to reorder code. Dynamic techniques are done at runtime while the program executes. Profile information is often used by both kinds of approaches. We discuss static approaches first.

6.1 Static code placement

Code layout at compile-time or link-time has been an active research area. Researchers have explored code placement at a number of different granularities: for example, at the granularity of basic blocks, groups of basic blocks, or entire procedures. A common 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. Another drawback is that the layouts they generate reflects the profile data used, so that data must be representative of other program executions.

McFarling [10] uses profile data to lay out a program's code to reduce misses in a direct-mapped instruction cache. His algorithm identifies those parts of a program that should overlap each other in the cache and those that should be placed in non-conflicting addresses. However, it is not clear how to apply his techniques to multi-level caches.

Pettis and Hansen [11] present techniques to do 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 16K directly-mapped unified cache. However, the improvement they achieve is very sensitive to cache organization: for example, it drops to 5% with a 2-way set-associative cache. Also, modern cache hierarchies usually have more than one level of cache. Because it is both simple and effective, their procedure ordering algorithm is generally considered the reference placement technique. It has also been used as the basis for several newer algorithms. Despite this, their algorithm has the drawback that small changes in the profile data often produce substantially different layouts.

Hashemi et al [7] 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 that of Pettis and Hansen's to place procedures such that the most frequent caller-callee pairs will not occupy the same cache lines. By a simulation estimation, they achieve better performance than Pettis and Hansen's procedure ordering.

Gloy and Smith [6] also compute procedure layouts that reflect the cache configuration. They collect complete procedure interleaving information which, in combination with information about the cache configuration and procedure sizes, allows them 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's algorithm.

Ramirez et al. [12] 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 effect 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 these from having conflict misses with cold instructions.

Cohn et al [4] describe the Spike post-link optimizer for Alpha/NT executables. Among its optimizations is code layout, which uses the Pettis-Hansen procedure 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 [9] is a post-link optimizer for IPF processors. It uses the IPF performance counters to collect at low cost (and low program impact) detailed profile information that Ispike uses for several instruction- and data-related optimizations including inlining, branch forwarding, and 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%.

Since they generate code layouts ahead-of-time, these static approaches lose the flexibility of determining layouts using the actual information for each run of a program. They also cannot cope with different program phases. These limitations make them less useful for dynamic languages like Java.

6.2 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 [2] developed a just-in-time code layout technique that places the procedures of Windows applications in the order of their invocation at runtime. Their results show that the resulting code layout achieves improvement similar to that of the Pettis and Hansen approach. 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 won't typically match those of the training run. Chen and Leupen's approach lays out procedures at allocation time while we can reorder all compiled code after it has been allocated and as often as necessary.

Scales [13] DPP (dynamic procedure placement) system uses runtime information to dynamically lay out procedure code. 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 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. This profiling is extremely expensive, however, and slows down the program by a factor of ten or more.

Whaley [14] very briefly outlines a never implemented dynamic procedure code layout optimization for the Jikes RVM (Research Virtual Machine). Using a combination of instrumented code and timer-based call stack profiling, it collects method call frequencies and calling relationships to determine which compiled methods should be located near each other. This location information is then passed to the garbage collector as a hint to reorder code (Jikes allocates compiled code in the garbage-collected heap). While Whaley's optimization would dynamically reorder compiled code in a managed runtime to improve its locality, it was never implemented and differs from our DCM system in the following ways: 1) For the Jikes garbage collector to relocate compiled code, it must understand that code and how to modify it. This means either additional complexity for a Jikes garbage collector since it must understand the representation of the code emitted by each Jikes JIT, or it means there are restrictions on the code JITs can emit. Our scheme has the JIT explicitly cooperate in moving code, so it can emit aggressively optimized code. 2) Whaley's system also differs from ours by only reordering code as part of a garbage collection. However, many programs would benefit from having their code reordered at times other than a garbage collection.

7. Conclusions

Managed runtimes for languages like Java provide the opportunity to dynamically monitor program execution and make adaptations to improve performance. We take advantage of this capability to reorganize JIT-compiled code at runtime to improve its locality and reduce instruction-related misses. This is especially important for large programs like enterprise applications. Our DCM system is the first implementation of dynamic code reordering in a managed runtime. We also describe a new placement algorithm, Graph-Walking layout, that specifically addresses expensive ITLB misses. It is much faster than the widely-used Pettis-Hansen procedure layout algorithm, and its layouts often perform better.

The results demonstrate that DCM with the Graph-Walking algorithm is practical and effective as an optimization in highperformance managed runtimes. DCM improved performance of the large MiniBean benchmark by 6% on a 4-way Intel® Xeon® SMP server machine. On MiniBean, Pettis-Hansen was only able to achieve a 2.3% improvement. In addition, the Graph-Walking algorithm can generate a new layout for the more than 15,586 compiled methods of MiniBean in only 0.35 seconds, less than the time required for one of its typical garbage collections, and much less than the 35 minutes needed by Pettis-Hansen.

8. REFERENCES

[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] 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.

[3] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. *Intel Technology Journal*, 7(1), February 2003. Available at `http://intel.com/technology/itj/2003/volume07issue01/art01_orp/p01_abstract.htm`.

[4] 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.

[5] R. Flower, C.-K. Luk, R. Muth, H. Patil, J. Shakshober, R. Cohn, and P. G. Lowney. Kernel Optimizations and Prefetch with the Spike Executable Optimizer. In *Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

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

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

[8] 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.

[9] 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.

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

[11] K. Pettis and R. C. Hansen. Profile-guided code positioning. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 16–27, New York, NY, USA, 1990. ACM Press.

[12] 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.

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

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

[15] B. Zorn. Performance in the Age of Trustworthy Computing, January 2004. Slides for a presentation at the University of Colorado and other universities.