Dynamic Code Management: Improving Whole Program Code Locality in Managed Runtimes

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

Brian T Lewis
Intel Corporation
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 by increasing memory stalls due to instruction cache and TLB misses. This problem is particularly an issue for large server applications written in languages such as Java and C# that provide just-in-time (JIT) compilation, dynamic class loading, and dynamic recompilation. However, managed runtimes also offer an opportunity to dynamically profile applications and adapt them to improve their performance. This paper describes a Dynamic Code Management system (DCM) in a managed runtime that performs whole program code layout optimizations to improve instruction locality.

We begin by implementing the widely used Pettis-Hansen algorithm for method layout to improve code locality. Unfortunately, this algorithm is too costly for a dynamic optimization system, $O(n^3)$ in time in the call graph. For example, Pettis-Hansen requires a prohibitively expensive 35 minutes to lay out MiniBean which has 15,586 methods. We propose three new code placement algorithms that target ITLB misses, which typically have the greatest impact on performance. The best of these algorithms, *Code Tiling*, groups methods into page sized *tiles* by performing a depth-first traversal of the call graph based on call frequency. Excluding overhead, experimental results show that DCM with Code Tiling improves performance by 6% on the large MiniBean benchmark over a baseline that orders methods based on invocation order, whereas Pettis-Hansen placement offers less improvement, 2%, over the same base. Furthermore, Code Tiling lays out MiniBean in just 0.35 seconds for 15,586 methods (6000 times faster than Pettis-Hansen) which makes it suitable for high-performance managed runtimes.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization, Run-time environments, Memory management (garbage collection)

General Terms Algorithms, Measurement, Performance, Design, Experimentation

Keywords Locality, Code Layout, Code Generation, Performance Monitoring, Dynamic Optimization, Virtual Machines

[*] 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. Introduction

The gap between processor and memory speed continues to present a major performance bottleneck. Applications spend much of their time waiting because of cache and TLB misses. For example, on the Intel® Itanium® 2 processor, the SPEC JBB2000[1] server benchmark spends 54% of its cycles waiting for memory. While the majority of stalls are from data access, instruction related misses such as instruction cache misses, ITLB misses, and branch mispredictions also contribute. Zorn [18] notes that Word XP, Excel Xp, and Internet Explorer have large code working sets, and that much of the latency users perceive is caused by page faults fetching code.

There are a number of common techniques compilers use to improve code layout. For example, compilers often arrange basic blocks within a procedure to decrease branch mispredictions. They may also use procedure splitting [13]: by allocating frequently executed basic blocks separately from infrequently executed ones, they reduce the number of active code pages and so the number of ITLB misses.

Researchers have developed a number of offline tools to optimize code placement. For example, Microsoft uses a profile-based tool [18] to reorder the code of their applications for better performance. Similarly, the Spike post-link optimizer [6] 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 profile-based aggressive dynamic optimizations. 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 these issues, we believe managed runtimes should include *code management* in addition to JIT compilation, garbage

[1] We use the SPEC 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.

collection, exception handling, and other services. Code management actively manages JIT-allocated code in order to improve its locality and consequently application performance. Managed runtimes should use profile information from hardware or software monitoring to reorder method code when necessary to maintain good instruction locality.

This paper describes the implementation of a dynamic code management (DCM) system that is integrated into our managed runtime. DCM uses dynamic profile information to reorganize JIT-compiled code at the granularity 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. One of these algorithms, Code Tiling is significantly faster both in worst case complexity and in practice than the best-known previous technique by Pettis and Hansen [13]. We demonstrate that Code Tiling generates code layouts that are better or comparable to those by the Pettis-Hansen algorithm.

This paper makes the following novel contributions:

- We present the first implementation of a dynamic code reordering system in a managed runtime (to our knowledge). Since DCM operates on-the-fly, it naturally copes with dynamic features of languages like Java such as method recompilation and dynamic class loading.
- A new code placement algorithm called Code Tiling. This algorithm is fast enough to make dynamic code reorganization practical in a high-performance managed runtime. Our results show that that DCM with Code Tiling reduces the execution time for MiniBean by 6% on a 4-processor Intel® Xeon® processor, which is better than with Pettis-Hansen.

The rest of this paper is organized as follows. Section 2 presents an overview of our DCM system. We have implemented DCM for IA-32 and the Itanium Processor Family (IPF), but our IA-32 implementation is more complete. We begin by describing the IA-32 implementation in Section 3. Section 4 then describes the Pettis-Hansen procedure layout algorithm and our new layout algorithms. Section 5 presents our experimental results on code reorganization and application times. Next, we describe the IPF implementation in Section 6, and discuss our experience using performance monitoring unit (PMU) sampling on the IPF to collect profile information for code reorganization. We then discuss related work in Section 7. Section 8 concludes and lists future work.

2. Dynamic code management overview

This section overviews the dynamic code management system, and the following section describes its implementation and design choices.

As the application executes and its behavior changes, DCM reorganizes compiled code as necessary. When miss rates for the ITLB or instruction cache become too high, it calculates a new layout, moves method code, and updates code pointers and offsets in methods, thread stacks, registers, and data structures of the managed runtime to reflect the new locations.

DCM gathers profile information on callers and callees and uses it to build a dynamic call graph. The dynamic call graph (DCG) is an undirected graph with a node for each method, and edges from a method to any methods it invokes, weighted by their dynamic frequency. When the system triggers a reorganization, DCM computes a new placement for each method's code based on the DCG. DCM then moves the code and does any required code updates. Figure 1 depicts DCM's components and their interactions.

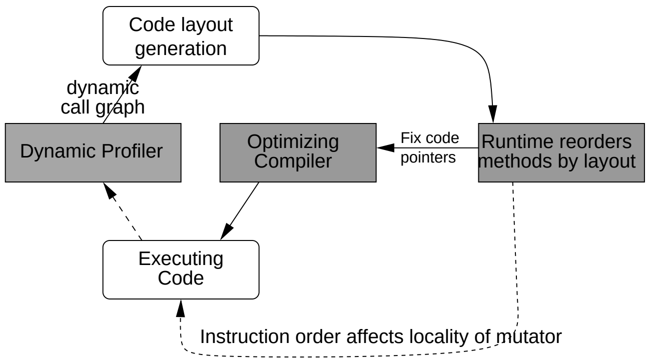

Figure 1. Dynamic Code Management

Software instrumentation or hardware performance monitoring can provide the dynamic call graph profiles. 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, DCM relies on the JIT to make any necessary adjustments to relocated code. This division gives JITs more freedom in how they emit code, and the code management system and the rest of the managed runtime do not need to understand the representation of JIT-compiled code.

3. DCM Implementation on IA-32

We implemented DCM 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 IA-32, we use the optimizing O3 JIT [4] to compile JVM bytecodes. This JIT performs inlining, a number of global optimizations (e.g., copy propagation, dead code elimination, loop transformations, and constant folding), as well as common subexpression elimination and array bounds check elimination.

ORP allocates compiled code in a region of memory that is separate from the garbage collected heap. It provides different subregions for code that is cold (infrequently executed) and hot (more often executed). ORP allocates code of equal "temperature" sequentially. Our IA-32 O3 JIT emits a single block of code for each compiled method. As a result, the granularity of reorganization for the IA32 is methods; for the IPF, it is a *code block* which typically divides a method into two parts: one for its hot basic blocks and another for its cold ones. We return to the IPF implementation in Section 6, but the remainder of this section and the next two discuss our IA-32 implementation.

To support dynamic code management, we modified O3 to emit relocatable code. This simplifies moving code on a code reorganization, but usually requires that the code be fixed up to update pointers to compiled methods used in that code, including references into code of other methods as well as into the same method. DCM calls the JIT (here O3) to update code after it has moved it.

ORP can collect dynamic profile information from either software instrumentation or from PMU sampling. However, on the IA-32, we found that using the PMU is too expensive. In particular, capturing the LBR (last branch record) requires hundreds of cycles since the pipeline must be flushed and a memory fence performed. As a result, ORP uses software instrumentation on IA-32. We could have modified our JITs to do the instrumentation, but we chose a simpler approach. ORP interposes on method calls to record the

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

Table 1. Breakdown of time to reorganize MiniBean's code (ms)

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

To reorganize code, 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 moving it to a newly-allocated code region is simpler. It also simplifies debugging our DCM implementation, since this makes it easy to recognize a stale reference to an old code location.
4. For each method, move the code and call the JIT to fix it up. Also fix up the method's metadata recorded by the managed runtime.
5. Update the call stack of each thread. In particular, correct any code addresses such as return addresses currently on stacks. Also, update any registers containing code addresses.
6. Restart the managed threads.

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

Our experience is that most of the time needed to reorganize code is due to the new layout calculation; DCM finishes the remaining steps quickly. To illustrate this, for MiniBean, the Code Tiling layout algorithm requires 417ms of the 510ms total reorganization time, while Cache-Aware Pettis-Hansen requires 28,459ms of the total 28,555ms. The other steps require about 100ms. These times are shown in Table 1.

3.1 Current status

Our IA-32 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 implementation, DCM itself is completely independent of GC and could reorder code at any time.

We do not currently support a mechanism to automatically trigger code reorganization since we have not yet developed a technique to determine when it would be be productive to do so. In the future, we plan to enhance DCM's use of PMU information to monitor ITLB and other instruction-related misses in order to determine when reorganizations are needed. Currently, the user specifies on the ORP command line the GC at which to reorganize code, and ORP invokes DCM at the end of that GC. Although this interim solution allows only a single reorganization, DCM itself is capable of reorganizing code multiple times. We will eventually use DCM to reorder code whenever necessary.

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

3.2 Discussion

In many ways, DCM resembles a copying garbage collector. It moves objects (method code) and updates any pointers to those objects. It is intended to improve program locality, but that is also a partial goal of many garbage collectors including ones that compact the heap or place objects to improve their locality [10]. DCM also supports pinning of objects that would be too hard or too expensive to relocate. For example, we pin methods containing Java `JSR` bytecodes since these bytecodes are relatively rare and the resulting code is complex. Like many garbage collectors, DCM could also do much of its work in parallel with application threads, even though it does not currently do this. New code layouts could be computed in parallel, for example. In the future, we might also investigate having DCM reclaim no-longer-needed code: code that is currently not referenced by any thread and not likely to be needed again soon.

3.3 Alternatives to DCM

One alternative to DCM is to use large pages for code which will reduce the number of ITLB entries and misses. Unfortunately, large pages are not supported by a number of operating systems, including IA-32 versions of Windows. Large pages also consume a larger portion of the virtual address space and may suffer higher fragmentation, which may be a problem if the virtual address space is relatively small. In addition, using large pages does not address instruction cache misses.

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

4. The code layout algorithms

This section describes the Pettis-Hansen and our new code layout algorithms. All the algorithms use the same underlying data structure, the dynamic call graph (DCG), and produce a new code layout. We first implemented the Pettis-Hansen algorithm, and found it usually improved performance of large applications. However, it was too expensive: Pettis-Hansen creates a new layout for SPEC JBB2000 (758 methods) in less than 150ms, but requires minutes for MiniBean (15,586 methods). This expensive overhead led us to develop three new, faster algorithms.

4.1 Pettis-Hansen algorithm

Pettis-Hansen places methods using a greedy "closest is best" strategy from the original call graph. Each step combines two nodes in the DCG and specifies their code layout. Each of the call graph's nodes initially contains a single method. The algorithm repeatedly chooses an edge $A \rightarrow B$ of heaviest weight in the entire graph (i.e., greatest calling frequency), then merges the nodes and outgoing edges of A and B. It lays out the code in the new node using the heuristic described by Gloy and Smith [7] (line 7 of procedure

```
PettisHansen (Graph)
while ($edge \leftarrow$ HeaviestEdge(Graph))! = NULL
do $(nodeA, nodeB) \leftarrow edge$.getNodes()
    MergeNodes(nodeA, nodeB);

HeaviestEdge (Graph)
$maxEdge \leftarrow NULL$
for each node in Graph do
  for each edge in node.edgeList do
    if $(maxEdge = NULL) || (edge.heat > maxEdge.weight)$ then
      $maxEdge \leftarrow edge$
return maxEdge

MergeNodes (nodeA, nodeB)
for each edgeB in nodeB.edgeList do
  for each edgeA in nodeA.edgeList do
    if edgeB.equals(edgeA)
      then $edgeA.weight \leftarrow edgeB.weight + edgeA.weight$
           RemoveEdge(edgeB)
nodeA.edgeList.attach(nodeB.edgeList)
nodeA.blockList.GSattach(nodeB.methodList)
```

Figure 2. Pettis-Hansen procedure layout algorithm

```
CacheAwarePettisHansen (Graph)
while ($edge \leftarrow$ HeaviestEdge(Graph))! = NULL
do $(nodeA, nodeB) \leftarrow edge$.getNodes()
    if $(nodeA.\text{Size}() > PAGE\_SIZE) || (nodeB.\text{Size}() > PAGE\_SIZE)$
      then RemoveEdge(edge)
      else MergeNodes(edge, nodeA, nodeB)
```

Figure 3. Cache-Aware Pettis-Hansen algorithm

MergeNodes in Figure 2). Pettis-Hansen finds the hottest call edge between a method in node *A* and one of *B* in the original call graph, and then orders the merged methods to minimize that edge's call distance in bytes. It leaves the sequence of methods within *A* and *B* the same. When it merges outgoing edges, if two point to the same node *C*, it merges them and weights the edge with the sum of the original weights. The algorithm terminates when no edges remain. Figure 2 shows the pseudo-code for the Pettis-Hansen algorithm.

4.1.1 Time complexity

The complexity of finding the maximum edge is $O(n^2)$ since in the worst case, there are *n* edges connected to each node. The maximum number of edge merges for each node merge is also $O(n^2)$. So the asymptotic time complexity of the algorithm is $O(n*(n^2+n^2)) = O(n^3)$. This complexity explains the dramatic increase in time required to place the code for large applications such as MiniBean.

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

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 node 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 may generate a different layout from 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 *ABCD*. Note here that *C* and *D* are adjacent, but are not with the original Pettis-Hansen layout. The different layout Cache-

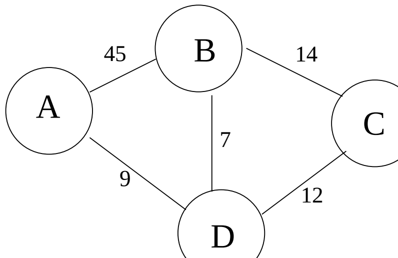

Figure 4. An example dynamic call graph

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 different. None of the four algorithms is guaranteed to produce the best layout. However, our main concern is the layout generation time, and we show in the next section that Cache-Aware Pettis-Hansen runs much faster.

A further refinement for Cache-Aware Pettis-Hansen is the following. In a direct-mapped cache, we do not want two methods mapped on the same cache set if they frequently call each other. To avoid causing cache interference in direct-mapped instruction caches, the algorithm should use the cache size instead of the page size.

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 Code Tiling algorithm

A major cost in Cache-Aware Pettis-Hansen is finding the heaviest edge in the entire graph every time. To reduce this cost, we developed the *Code Tiling* algorithm that uses a simpler approximation. This algorithm traverses the nodes of the DCG one at a time. Assume the current node is *A*. As long as *A*'s code occupies less than a page, it selects the heaviest edge $A \rightarrow B$ and merges *A* with the node *B*. If this algorithm merges any nodes, it produces a different layout than either Pettis-Hansen or Cache-Aware Pettis-Hansen since it only considers that part of the graph immediately connected to the current node. However, this layout may occasionally be better compared with Pettis-Hansen since it can give the best possible locality to the single hottest path, if one exists. Performance results in the next section show this algorithm computes good layouts and computes them faster.

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 still $O(n)$. However, in practice this case does not occur. The maximum number of edge merges requires $O(n^2)$. As a result, the worst case asymptotic complexity of the Code Tiling algorithm is $O(n*(n+n^2)) = O(n^3)$. It is the same as the Cache-Aware Pettis-Hansen algorithm, but since we avoid the work of repeatedly searching for the heaviest edge in the whole graph, we expect layout generation to be faster than with the Cache-Aware Pettis-Hansen algorithm.

```
CodeTiling(Graph)
for each node in Graph do
  currentNodeSize ← 0
  node.isVisited ← true
  STAY :
  maxEdge ← NULL
  for each edge in node.edgeList do
    if isVisited(edge)
      then RemoveEdge(edge)
      else if (maxEdge = NULL)||(edge.heat > maxEdge.heat)
              then maxEdge ← edge
  if (currentNodeSize > PAGE_SIZE)||(maxEdge = NULL)
    then RemoveEdge(maxEdge)
    else nodeB ← maxEdge.otherNode(node)
         currentNodeSize ← currentNodeSize + nodeB.size()
         MergeNodes(node, nodeB);
         goto STAY

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

Figure 5. Code Tiling algorithm

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

AttachNodes(edge, node)
(nodeA, nodeB) ← edge.getNodes()
if (!nodeA.isVisited)
  then nodeB ← nodeA
nodeB.isVisited ← true
node.edgeList.attach(nodeB.edgeList)
node.blockList.attach(nodeB.methodList)
```

Figure 6. Linear Scan algorithm

4.4 Linear Scan algorithm

To further reduce the cost of generating a code layout from the dynamic call graph, we also tried a straightforward algorithm that has linear time complexity, the *Linear Scan algorithm*. In this algorithm, we scan each node in the graph in breadth-first traversal order, but we ignore cold edges (ones with weight less than some threshold). We show this algorithm in Figure 6. Notice that when AttachNodes merges two nodes, it does not merge their out edges: even if two edges connect to the same node, they are not merged. We eliminate this step because merging edges is especially expensive. Notice also, that when merging one node *B* into another node *A*, AttachNodes simply attaches *B*'s edge list to *A* without updating any of the edges in *B*'s edge list. As a result, the edge data structure for *B*'s edge list will still record *B* instead of the correct *A*. Not updating the data structure does not cause a problem because we never attach an already-visited node like *B* again. By reducing the work done during node merges, Linear Scan scans every edge exactly once and achieves linear time in the number of edges.

4.4.1 Time complexity

If there are n^2 edges (worst case for n nodes) in the graph, the Linear Scan algorithm scans exact n^2 edges. If an edge E has both ends visited before, edge E is removed. Otherwise, both nodes connected by edge E are merged. When Linear Scan merges nodes, it simply attaches one edge list to the other node and takes a constant time. This means the Linear Scan algorithm's asymptotic complexity is just O(n^2).

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

Table 2. Benchmark characteristics

5. DCM Results

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

5.1 Experimental framework

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

Our experiments used the SPEC JVM98, SPEC JBB2000, and MiniBean benchmarks. MiniBean is a large benchmark inspired by the SPECjAppServer2002 enterprise application server benchmark, but runs in a single process on a single machine. While MiniBean uses enterprise JavaBeans (EJB) functionality, it generates no network traffic itself. SPECjAppServer2002, on the other hand, must be run using multiple machines including a database server. When collecting our results for each benchmark, we used the same input for each of its runs. We also ran each benchmark five times and used the best time. Table 2 shows the characteristics of these benchmarks[2]. We executed each application stand-alone. For heap sizes, we used 512M for MiniBean, 256M for SPEC JBB2000, and 50M for SPEC JVM98. Because we focus our efforts on server applications and are not changing the garbage collector, we do not consider the trade offs of different heap sizes.

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

5.2 DCM overhead

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

5.2.1 Dynamic call graph generation overhead

We use software instrumentation on IA-32 because it is expensive to use the Intel Pentium® 4 PMU to capture the LBR (last branch record) to get call information. To determine the overhead of software instrumentation for DCG creation, we ran MiniBean two times: once with no instrumentation and a second time with our software-based DCG generation. Each time, we ran MiniBean up to the same point in its execution, when it completed its warm-up phase. We found that MiniBean required 54 seconds with no instrumentation, but 66 seconds when we used software instrumentation. This overhead is high.

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

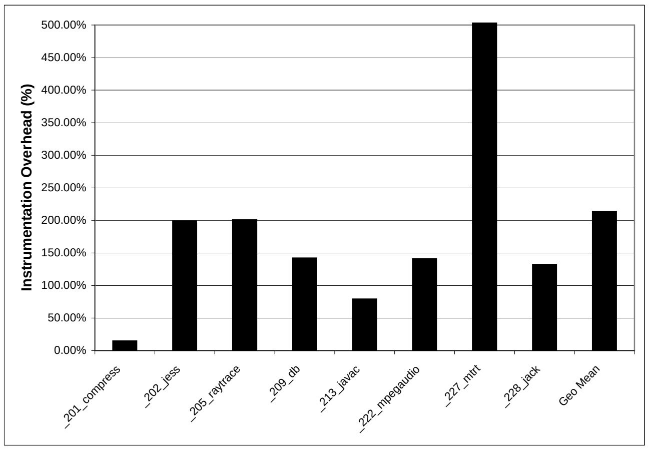

Figure 7. SPEC JVM98 software instrumentation overheads

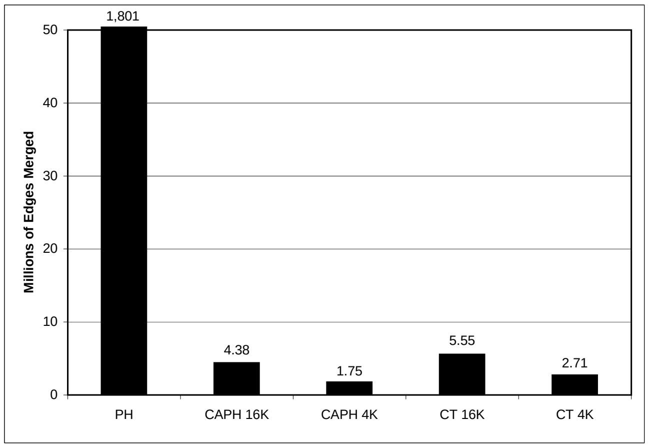

Figure 8. Number of edges checked for merging for MiniBean

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

Table 3. MiniBean and SPEC JBB2000 layout creation times (ms)

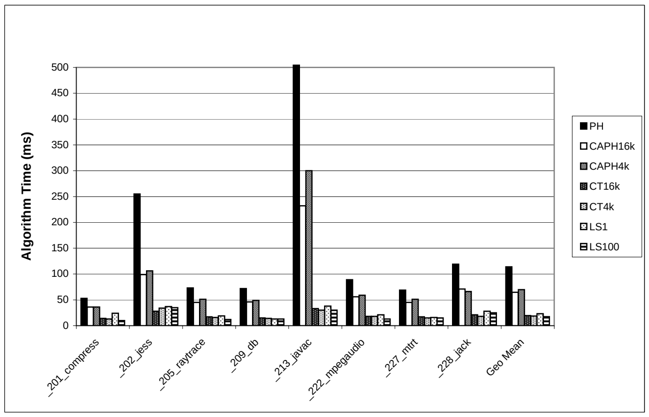

Figure 9. SPEC JVM98 layout generation times (ms)

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

We also expected and did observe high overheads due to the software instrumentation implementation on the SPEC JVM98 benchmarks. The overheads are shown in Figure 7. The bars for _227_mtrt are cut off since they are about 3000%. The geometric mean of the overheads is about a factor of 3. This result indicates that if software instrumentation is used, a faster implementation is necessary, especially for smaller applications.

5.2.2 Code layout generation overhead

Another overhead of DCM is the time required by the code layout algorithms to generate a code layout. We show the times needed for MiniBean and SPEC JBB2000 in Table 3.

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

When we explored where Pettis-Hansen spent its time, we found that most of its time for SPEC JBB2000 and MiniBean was spent merging edges, or more precisely, checking for edges to be merged. Pettis-Hansen checks many more edges for merging than Code Tiling or our other layout algorithms as illustrated in Figure 8, which shows the number of edge checks (in millions) needed for MiniBean with Pettis-Hansen and the other algorithms. In this figure and the following ones, "PH" stands for the Pettis-Hansen algorithm, "CAPH" for Cache-Aware Pettis-Hansen, "CT" for Code Tiling, and "LS" for Linear Scan. Note that the bar for Pettis-Hansen is cut off since it checks about 1.8 billion edges. Although Cache-Aware Pettis-Hansen checks about the same number of edges as Code Tiling, Code Tiling is faster because its working set is smaller at each step, allowing it to have better memory locality when it traverses the node and edge data structure.

Generating the new code layouts for the SPEC JVM98 benchmarks is generally much faster than for SPEC JBB2000. Our new algorithms are up to 6.33 times faster (for Code Tiling with a 4KB threshold). The results are shown in Figure 9.

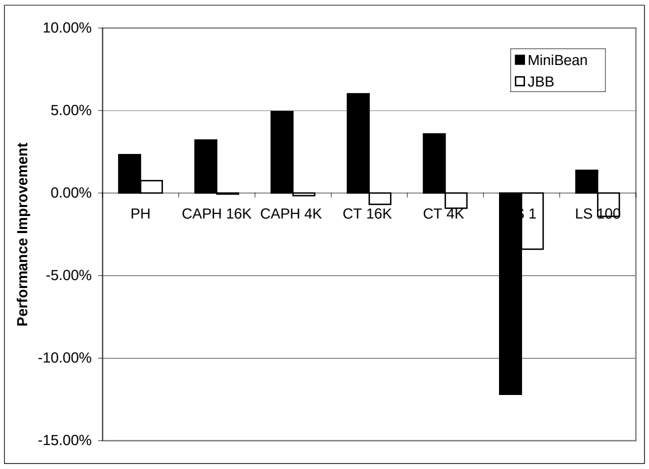

Figure 10. MiniBean and SPEC JBB2000 performance

5.3 DCM performance results

Figure 10 shows the performance benefit of dynamic code reorganization with both the MiniBean and SPEC JBB2000 benchmarks using the four different code layout algorithms. The base code layout is the default one used by our managed runtime, which is based on invocation order and already provides some locality benefit, as we noted in Section 1. 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 DCM with Code Tiling can significantly improve MiniBean's performance. However, it has essentially no impact on SPEC JBB2000. One reason for this difference is the size of the two benchmarks. With the default 4K pages, the IA-32's 128-entry ITLB can map 512K of simultaneous code space, which is much smaller than MiniBean's 3.1MB of JIT-compiled code. Optimizations that improve MiniBean's code locality should thus improve its performance. SPEC JBB2000, on the other hand, only has 268K of code, so it fits within the ITLB span. Reorganizing this code to improve locality has little benefit, at least as long as SPEC JBB2000 is the only application running on the machine. If multiple programs are running, there may be some benefit since improving a program's code locality will reduce its working set, which allows more applications to run simultaneously without ITLB misses.

Figure 11 shows the run times for the SPEC JVM98 benchmarks with different code layout algorithms. We measure the times for the second iteration of the benchmark runs, so these times do not include any instrumentation or code reorganization overhead. There is no clear performance benefit from using any layout algorithm. Since these benchmarks are so small, with instruction working sets often less than 32K, this result is not surprising.

5.4 Discussion

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

The benefit of DCM depends on application size. This section shows that code reorganization helps large applications more than

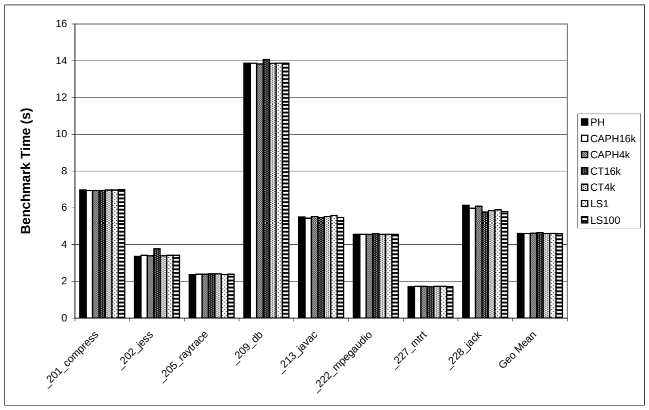

Figure 11. SPEC JVM98 performance

small ones. These results also demonstrate that Code Tiling is much more suitable for online code reorganization than the classic Pettis-Hansen algorithm. It executes much faster and can produce better performance.

6. IPF implementation and PMU-based DCM

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

On IPF, we prefer PMU sampling since it is less expensive than software instrumentation[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, DCM discovers information about recent method calls. This information allows us to identify both the caller and the callee methods. We separate calls instructions from return instructions by checking if the target address is at the beginning of a code block.

6.1 Experimental framework

For our IPF results, we used 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. We used the IPF's fundamental 4KB page size.

We also used StarJIT [1] which is a high-performance dynamic compiler that uses a single SSA-based intermediate representation and global optimization framework to compile JVM bytecodes. StarJIT typically emits two code blocks for each method. These contain the method's hot and cold code; the cold code includes exception handlers, for example. The granularity of code reorganization on IPF, then, is a code block instead of a method.

Our IPF DCM implementation is not fully complete. We have not completed the StarJIT changes needed for it to update compiled

[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.

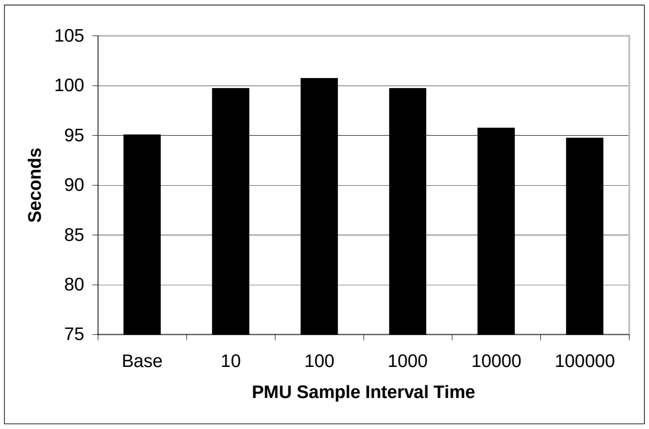

Figure 12. MiniBean DCG creation times with PMU sampling

code during a code reorganization. However, we are able to use *static code layout* to get an approximation of what DCM might provide. This approximation 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 use this layout file to place their compiled code. When static code layout is used, ORP first reads the layout, then uses its placement information when allocating code for JITs.

6.2 PMU-based DCG generation overhead

On IPF, we studied the overhead for dynamic call graph generation with PMU sampling. We varied the sampling interval from 10 (1 sample every 10 branches) to 100,000 (1 sample every 100,000 branches). The times required to generate the dynamic call graph for MiniBean at different sampling intervals are shown in Figure 12. These times are from program start until 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 13. 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 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.

6.3 Code reorganization results

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

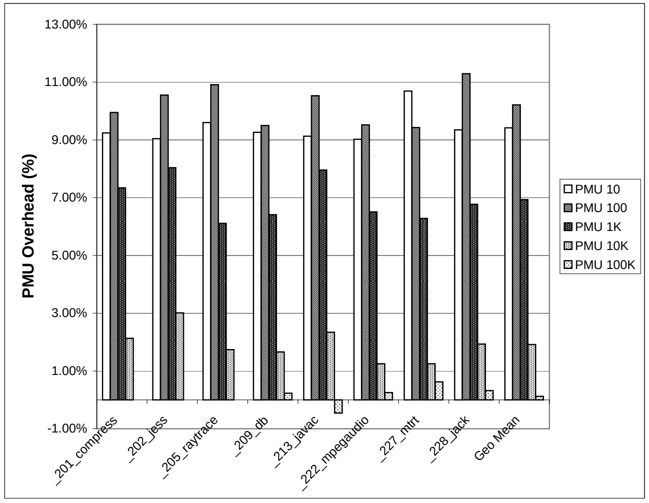

Figure 13. PMU overhead for the SPEC JVM98 benchmarks

	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 4. IPF front-end stalls using static code layout

We used static code layout with the Pettis-Hansen layout algorithm to determine its performance impact for the MiniBean benchmark. We found the performance improvement was slightly negative, -1%, which probably indicates only that for this benchmark on IPF, code reorganization has little impact. While it is possible that DCM would provide better performance, its benefit is still likely to be less than it was on IA-32. One reason for this poor result is the large L3 cache (9MB) on our IPF machine, which allows it to hold nearly all of MiniBean's code (11MB on IPF). Another reason is the short memory stall time on the Itanium 2 processor: the latency is approximately 6 cycles for the L2 cache and 12 cycles for the L3 cache. These mean that reordering method code will have little impact on IPF programs unless those programs have working sets much larger than the L3 cache size.

6.4 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 4 shows the number of IPF front-end stall cycles when running MiniBean. It compares the number of stalls for the default code layout as well as one using the Pettis-Hansen layout algorithm. For the latter, we used a static code layout using a DCG based on PMU sampling. Among the front-end stalls we measured are ITLB miss and instruction cache miss stalls.

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

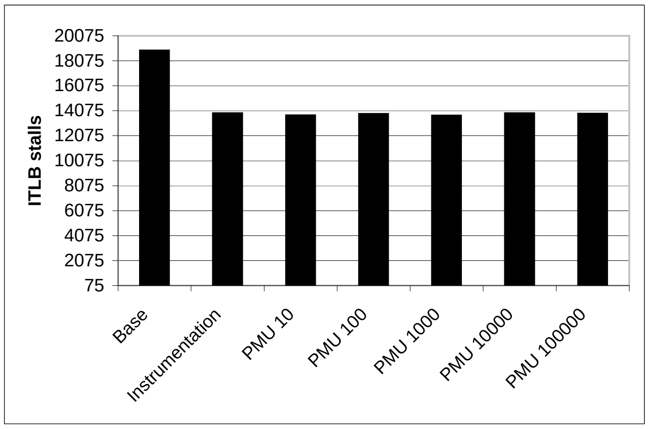

Figure 14. Effect of PMU-based call graphs on ITLB misses

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

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 Figure 14. These results indicate that, for MiniBean, PMU-generated dynamic call graphs are as effective as the precise software instrumentation graphs for DCM. While there is a trade off between PMU sampling frequency and accuracy of the resulting DCGs, Figure 14 illustrates that the lower-precision DCGs do not have a significant impact on the effectiveness of the generated code layouts. These results are consistent with our previous experience that PMU instrumentation has the same benefit as software instrumentation, but at lower cost.

6.5 Discussion

While Section 5 showed that dynamic code management can significantly improve the performance of larger applications such as MiniBean, this section's results show that the benefit depends on the processor's microarchitecture and cache hierarchy. Code reorganization has more impact on IA-32 than IPF. Our IPF machine's 9MB L3 cache and short memory stall times means that applications must have substantially more code than MiniBean before they can benefit from code reorganization.

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

7. Related work

Numerous researchers have studied the problem of restructuring programs to improve memory performance. Much of the early software work was aimed at reducing virtual memory page faults. Some current work also tries to minimize these very expensive faults; see, for example [18]. 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.

7.1 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 reflect the profile data used, so that data must be representative of other program executions.

McFarling [12] 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 [13] 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. They 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 [8] 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-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-Hansen's procedure ordering.

Gloy and Smith [7] 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-Hansen's algorithm.

Ramirez et al. [14] 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 [5] 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 [11] 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.

7.2 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-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 [16] 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 and 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 [17] 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 feature 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 proposed 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.

A few researchers [15, 9] investigate code cache management policies for dynamic optimizing systems. Their work focuses on creating basic block sequences (superblocks) for a trace cache and replacement policies for hardware instruction caches. By moving hot caller/callee pairs closer together, DCM helps the trace cache find more hot traces since the trace cache is limited by address range.

8. Conclusions and future work

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 which 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, Code-Tiling, 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 Code Tiling is practical and effective as an optimization in high performance 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, Code Tiling 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.

Our plans for the future include enhancing DCM's use of PMU information to monitor ITLB and other instruction-related misses in order to automatically determine when to trigger code reorganizations. Furthermore, we want to implement a suggestion by one of the anonymous reviewers to relocate code only if the total code grows to a point where this is likely to pay off. We also intend to explore techniques to automatically determine the best page size cut-off threshold for the Code Tiling algorithm.

Acknowledgments

We want to thank Neal Glew and the anonymous reviewers for their feedback on this paper. We would especially like to thank James Stichnoth for his ideas and continuous support of the code management work. We are grateful for Brian Murphy's help with the StarJIT compiler. In addition, we also thank Todd Anderson for the insightful discussions about algorithms.

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] M. Cierniak, G.-Y. Lueh, and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

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

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

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

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