Identifying the Sources of Cache Misses in Java Programs Without Relying on Hardware Counters

Hiroshi Inoue and Toshio Nakatani

IBM Research - Tokyo
NBF Canal Front Building, 5-6-52, Toyosu, Tokyo, 135-8511, Japan
{inouehrs, nakatani}@jp.ibm.com

Abstract

Cache miss stalls are one of the major sources of performance bottlenecks for multicore processors. A Hardware Performance Monitor (HPM) in the processor is useful for locating the cache misses, but is rarely used in the real world for various reasons. It would be better to find a simple approach to locate the sources of cache misses and apply runtime optimizations without relying on an HPM. This paper shows that pointer dereferencing in hot loops is a major source of cache misses in Java programs. Based on this observation, we devised a new approach to identify the instructions and objects that cause frequent cache misses. Our heuristic technique effectively identifies the majority of the cache misses in typical Java programs by matching the hot loops to simple idiomatic code patterns. On average, our technique selected only 2.8% of the load and store instructions generated by the JIT compiler and these instructions accounted for 47% of the L1D cache misses and 49% of the L2 cache misses caused by the JIT-compiled code. To prove the effectiveness of our technique in compiler optimizations, we prototyped object placement optimizations, which align objects in cache lines or collocate paired objects in the same cache line to reduce cache misses. For comparison, we also implemented the same optimizations based on the accurate information obtained from the HPM. Our results showed that our heuristic approach was as effective as the HPM-based approach and achieved comparable performance improvements in the SPECjbb2005 and SPECpower_ssj2008 benchmark programs.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: [Programming Languages]: Processors – Compilers, Optimization, Memory management.

General Terms Measurement, Performance, Experimentation.

Keywords Hardware performance monitor, Object placement optimization

1. Introduction

Cache miss stalls are one of the major sources of performance bottlenecks in high performance processors. Hence, it is important for compilers and language runtime systems to use the processor cache efficiently, especially on multicore processors, which have limited memory bandwidth compared to the huge computation resources. Previous techniques [1-4] showed that cache miss profiles were useful for runtime systems in reducing cache misses and improve the performance of cache-miss-intensive programs. These previous techniques used an HPM (Hardware Performance Monitor) in the processor to obtain cache miss profiles. However, for a compiler to use the HPM in is difficult because the HPM functions are often specific to the processor, the HPM may require a special device driver and super-user privilege, and only one process can use the HPM at a time.

In this paper, we identify the source of cache misses without relying on hardware support. We used an HPM for a thorough study of various Java programs and identified the hot loops that cause frequent cache misses. We found that many of them can be classified into a small set of patterns that can be heuristically detected as simple idioms without relying on an HPM. In general, the idioms correspond to repeated indirect loads from the Java heap in hot loops. Typical object-oriented programs heavily use complicated data structures, such as hashmaps and linked lists, and many cache misses come from accesses to such data structures. Our basic idioms work well for many Java programs because they can effectively capture such accesses. We experimentally showed that our heuristic approach effectively identified a large part of the L1 and L2 cache misses in many Java programs, including SPECjbb2005, SPECpower_ssj2008, SPECjvm2008, and the DaCapo benchmark suite. On average, our technique selected only 2.8% of the load and store instructions generated by the JIT compiler and these instructions accounted for 47.3% of the L1D cache misses and 48.9% of the L2 data cache misses caused by the JIT-compiled code. Compared to the total number of load and store instructions and cache misses caused by hot methods that we apply our analysis, our technique achieved about 63.6% and 69.2% coverage for the L1 and L2 cache misses by selecting 14.2% of the load and store instructions in the hot methods.

We demonstrate the effectiveness of our technique for compiler optimizations. We prototyped two types of object placement optimizations based on our heuristic approach in a Java VM with a JIT compiler. We compared the performance improvements from the optimizations based on our heuristic approach against the similar optimizations based on accurate cache miss statistics obtained from the HPM. Our optimizations showed performance improvements in two benchmarks with

http://doi.acm.org/10.1145/2258996.2259014

many cache misses, SPECjbb2005 and SPECpower_ssj2008. These performance improvements were close to the gains based on the accurate cache miss statistics from the HPM.

The main contributions of this paper are two-fold. (1) We present a technique to identify the instructions and objects that frequently cause cache misses in Java programs without relying on an HPM. (2) We prototyped the online optimizations in a Java JIT compiler using our heuristic approach and compared to the HPM-based approach. Our results showed that our technique is effective in implementing optimizations in dynamic compilers.

The rest of the paper is organized as follows. Section 2 discusses related techniques. Section 3 presents our no-HPM technique to identify the instructions that cause frequent cache misses. Section 4 describes the experimental environment and our results. Section 5 explains how we use the pattern-matching-based heuristic approach in compiler optimizations. We also show the performance gains from our optimizations and compare them to the HPM-based approach. Section 6 summarizes our work.

2. Related Work

In this paper, we identify the instructions and objects that tend to cause many cache misses. Burtscher *et al.* [5] classified load instructions based on the region of memory (stack, heap, or global), the kind of reference (array, field, or scalar) and the type of data (pointer or value). They showed that load instructions for certain classes caused more cache misses than others in C and Java programs. Our technique identifies exactly those load and store instructions that tend to cause many cache misses. Panait *et al.* [6] proposed a technique to statically identify the load instructions that cause many cache misses. They call such a load instruction a delinquent load. Their technique focuses on analyzing program binaries to calculate a weight for each load instruction. They estimate the likelihood each load causes cache misses based on criteria such as the number of dereferences and the base register used to calculate the address to be accessed. Our technique identifies more information for compiler optimizations, such as the target classes, rather than just identifying load instructions that cause cache misses. Therefore we focus on analyzing the compiler IR, which includes more information than the binaries. We demonstrated the practical effectiveness of our technique by implementing two types of optimizations in a Java JIT compiler, in contrast to simply identifying the delinquent load instructions.

There are some techniques that use cache miss profiles from HPMs for optimizations in compilers and runtime systems. Adl-Tabatabai *et al*. [1] exploit cache miss statistics in their Java JIT compiler to insert effective prefetch instructions for the Itanium 2 processor. Schneider *et al.* [2] used cache miss statistics from the garbage collector to optimize the placement of objects in the Jikes RVM on the Pentium 4 processor. Serrano and Zhuang [3] also identified opportunities to reduce cache misses by reordering the objects in the garbage collector in the POWER5 and POWER6 processors. Cuthbertson *et al.* [4] exploited the HPM of the Itanium 2 processor for instruction scheduling and object collocation in the garbage collector. In our work, we use alignment [3] and collocation [2-4] to test the effectiveness of our approach, since they are two of the most proven optimization techniques based on cache miss profiles. Both HPM-based and pattern-matching-based optimizations use approaches similar to the previous techniques [2-4], locating load instructions that cause many cache misses, identifying target classes, and then optimizing the object locations to reduce the cache misses. Though we did not study prefetch injection [1] with our heuristic approach, it could be used to identify the targets for prefetching.

```
ClassA objA;
ClassB objB;
while (!end) { // in a hot loop
   …
   // 1) first, load a reference of ClassA
   objA = objB.referenceToClassA;
   …
   // 2) then, access a field of objA
   access to objA.field1;
   …
}
```

a) a pattern for frequent cache misses

```
ClassA objA;
while (!end) { // in a hot loop
   …
   // 1) first, load a reference of ClassA from
   //    a field of 'this' object
   objA = this.referenceToClassA;
   …
   // 2) then, access a field of objA
   access to objA.field1;
   …
}
```

b) anti-pattern

Figure 1. (a) A pattern that tends to cause frequent cache misses and (b) An anti-pattern that does not cause frequent cache misses.

Object placement optimization has a rich history of research and many software-based techniques have been proposed. These techniques use a variety of types of static and dynamic information that can be obtained without special hardware, such as field access profiles at read barriers [7, 8, 9], object lifetimes [10], allocation frequencies for each Java class [11], hints provided by the STL container libraries [12], or static access patterns analyzed at the compilation time [13]. Our heuristic approach is unique in the sense that we try to detect objects and fields that cause many cache misses, not just those that are frequently accessed. In most environments cache misses are transparent to software. Therefore, to predict the number of cache misses we need to use an empirical approach based on pattern matching rather than inserting instrumentation code.

3. Identifying the Instructions and Objects Causing Frequent Cache Misses without an HPM

In this section, we first explain our technique to identify the instructions and objects that frequently cause cache misses to provide effective information for optimizations in the JIT compiler. Our key insight to locate cache misses without using an HPM is that most cache misses identified by an HPM in typical Java workloads are often caused by certain idiomatic code patterns in those programs. This insight allows us to identify the objects that frequently cause cache misses by matching the hot loops with the idiomatic patterns.

Though these patterns look quite simple and standard in many Java programs, we found that the percentage of load instructions selected by matching with this idiom in hot loops was up to 5.9% of the total load and store instructions in the JIT-compiled code (and 2.8% on average). We identified these patterns by investigating the cache miss profiles from the HPM using SPECjbb2005 and SPECjvm2008. As shown later, due to the simplicity and generality of these patterns, pattern matching with them worked well with programs from the DaCapo-9.12 benchmark suite [14]. In Section 4, we also discuss some cache

Table 1. Thresholds for two configurations.

	hot loop threshold			hot method threshold
	hot	veryHot	scorching	
Base	40	20	10	veryHot
Aggressive	16	8	4	hot

misses in multi-threaded programs that we cannot capture with our current technique.

Figure 1(a) shows the most frequently observed pattern that causes many cache misses. In this pattern, there is a load of a reference to `objA` and a following access to the `objA` in one highly iterated loop. Here, the access to the `objA` can be a load or a store to a field of `objA`, or an operation accessing the object header, such as a monitor enter, a monitor exit, a `checkcast`, or an `instanceof` operation. As a variant, a reference to `objA` can be obtained from a return value of a method call instead of loading from a field of `objB`. We observed that a hot loop matching this pattern tends to cause a cache miss in each iteration when accessing the `objA.field1`. Thus we should focus on the objects of `ClassA` to improve the memory system performance.

In addition to this basic pattern, we used an anti-pattern in our analysis. Figure 1(b) shows this anti-pattern. The code sequence is almost identical to the basic pattern shown in Figure 1(a), but a reference to `objA` is loaded from the 'this' object. In this case, the 'this' object is loop invariant and thus `objA` should not cause many cache misses. We observed that this anti-pattern appeared frequently in many programs but rarely caused cache misses. This anti-pattern can be extended to check the loop invariance of the first load in addition to loading from a 'this' object. When we match the pattern after applying the loop optimizations including loop invariant code motion, we do not need to explicitly apply this anti-pattern. We used this approach in our implementation rather than handling 'this' object explicitly.

We do this pattern matching in the JIT compiler. The JIT compiler uses this analysis when it recompiles a method with a higher optimization level than the initial level. We used the execution frequency information obtained by software-based profiling to identify the hot loops for the pattern matching. Many high performance dynamic compilers already provide this information because it is important for many widely used optimizations. We designate a loop as hot if the estimated number of iterations per method invocation (N_{iter}) exceeds a threshold, which we call the *hot loop threshold*. In the current implementation, we adjust the threshold for the hot loops between N_{iter} = 10 and 40 based on the hotness of the compiling method. When the method is compiled with the highest optimization level (*scorching*), such a method is typically consuming more than 10% of the total CPU time, and we use 10 as the threshold. When the method is compiled with the second highest level (*veryHot*), typically consuming more than 3% of the CPU time, we use 20 as the threshold. For other hot methods we use 40.

The heuristics based on the number of iterations give good estimates for the hot loops, but in some cases opportunities are missed. If a method is invoked frequently, loops included in the method are also executed frequently, even though the loops do not meet our criteria. To identify such methods, we do pattern matching for the entire method when there are no loops having N_{iter} larger than 3.0 in the method and the method was compiled with veryHot or scorching levels (for the *hot method threshold*). If there are no highly iterated loops within a very hot method, then the method must be invoked quite frequently, typically from inside a very hot loop. We do not use the anti-pattern shown in Figure 1(b) when we match the pattern for the entire method,

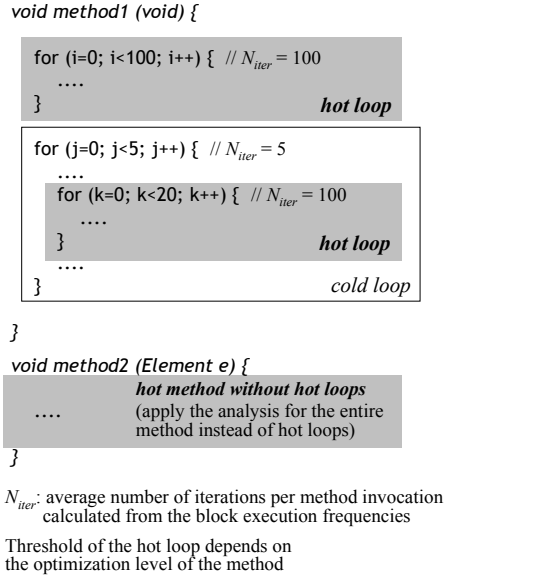

Figure 2. Overview of our hot loop criteria.

because in this case 'this' pointer may change in each invocation of the method. Figure 2 summarizes our hot loop detection methods. We also use the execution frequency to exclude cold blocks when we analyze a loop. There can be cold blocks even inside of a hot loop, such as a rarely executed if block.

These thresholds control the aggressiveness of the identification. Using smaller values for these thresholds increase the number of identified targets. To study the effect of the thresholds, we also evaluated another configuration with lower threshold values to pick more loops for analysis. We call these two threshold configurations *Base* and *Aggressive*. Table 1 summarizes the two configurations.

We do the pattern matching after applying most of the code transformation optmizations including method inlining to simplify the implementation of our analyzer. Small methods in hot loops are inlined by method inlining and so we do not need to implement our analysis as a costly interprocedural analysis. In the current implementation, we did not alter the method inlining policy to cooperate with our analyzer.

4. Experimental Results

This section presents our experimental results to evaluate the accuracy of our pattern-matting-based technique in identifying the instructions that frequently cause cache misses. We used standard benchmarks: SPECpower_ssj2008, SPECjbb2005, SPECjbb2000, SPECjvm2008 (excluding the scimark and crypto benchmarks), and the DaCapo-9.12 benchmark suite.

We ran the benchmarks on an IBM BladeCenter JS22 using 2 POWER6 [15] cores running at 4.0 GHz with 2 SMT threads per core. We implemented our technique in the 32-bit JVM included in the IBM SDK for Java 6 SR2. Each POWER6 core has 64 KB of L1 data cache (L1D), 64 KB of L1 instruction cache, and 4 MB of L2 cache. The cache line size of the POWER6 processor is 128 bytes for both L1 and L2 caches. The size of the Java heap was 2 GB using 16-MB pages. We selected the generational garbage collector so that the Java heap was divided into a nursery space and a survivor space for young objects and a tenured space for older objects. The test system had 16 GB of system memory and used RedHat Enterprise Linux 5.2.

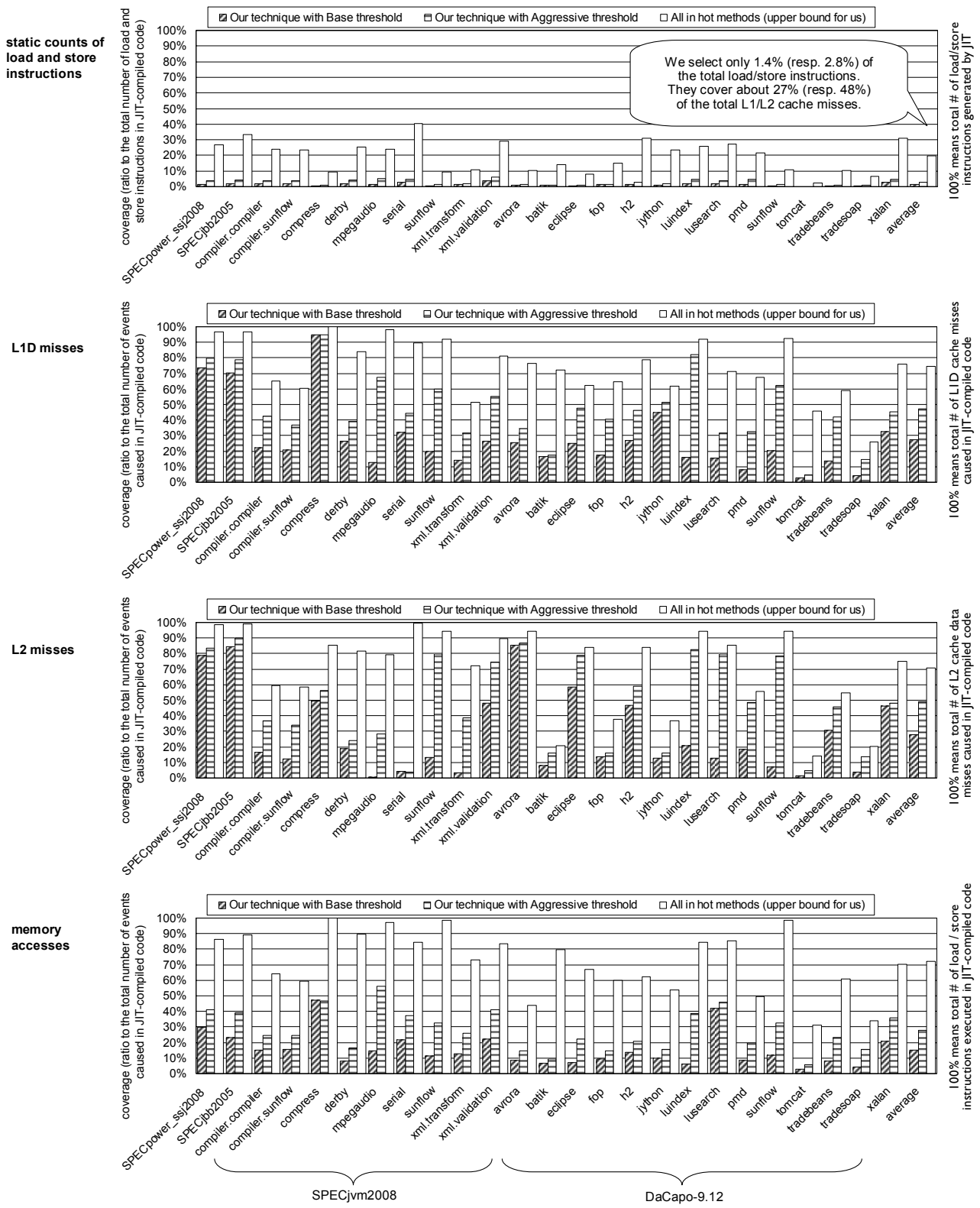

Fig. 3. Coverages of the instructions identified by our technique for the number of L1D cache misses, L2 cache data misses, memory accesses, and the static counts of load and store instructions. Our technique selected only 1.4% and 2.8% of the total load and store instructions, which account for about 27% and 48% of the total L1 cache misses for Base and Aggressive threshold configurations.

Table 2. Summary of the coverage by our technique.

	Base Threshold	Aggressive Threshold
ratio to the total counts in all the JIT-compiled method	coverage: L1D miss 27.3%, L2 miss 27.9% (by selecting 1.4% of load/store instructions)	coverage: L1D miss 47.3%, L2 miss 48.9% (by selecting 2.8% of load/store instructions)
ratio to the total counts in the hot methods that we apply our analysis (against the white bars labeled "All in hot methods" in Figure 3)	coverage: L1D miss 36.7%, L2 miss 39.4% (by selecting 6.9% of load/store instructions)	coverage: L1D miss 63.6%, L2 miss 69.2% (by selecting 14.2% of load/store instructions)

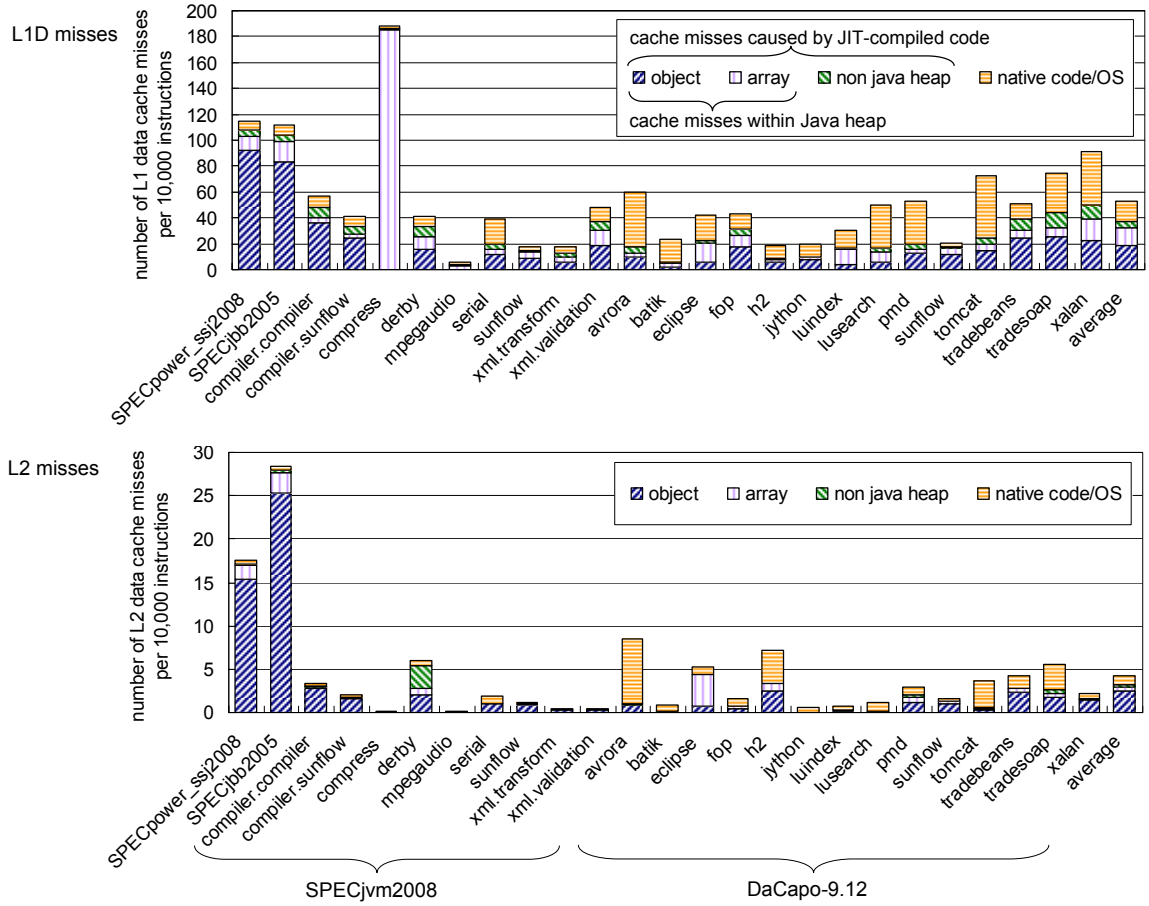

Figure 4. L1 and L2 cache miss ratios and breakdown by code location and access targets. We exclude the cache misses during stop-the-world garbage collection to focus on mutator performance.

4.1 Coverage

The first graph in Figure 3 shows the number of load and store instructions selected by our technique over the total number of load and store instructions generated by the JIT compiler. The other three graphs in Figure 3 show the coverages for the instructions identified by our technique for L1D cache misses, L2 cache data misses and all memory accesses. We measured the number of cache misses and memory accesses by using the HPM. We compare the coverages for our technique using the two threshold configurations (*Base* and *Aggressive*, as shown in Table 1) against the case of selecting all of the load and store instructions in the methods compiled with hot or higher compilation levels (labeled *all in hot methods* in the figure). Here, we focus on the events caused by JIT-compiled code and hence the results do not include the events caused by the Linux kernel or native code in the JVM such as the garbage collector. We performed the measurements 4 times and averaged the results.

From the figure, our technique selected an average of only 1.4% and 2.8% (and up to 3.7% and 5.9%) of the total load and store instructions generated by the JIT compiler for Base and Aggressive threshold configurations, respectively. These instructions accounted for 27.3% and 47.3% of the total L1D cache misses and 27.9% and 48.9% of the total L2 data cache misses (average values). The coverages for the numbers of memory accesses of the instructions selected by our technique were 15.2% and 27.9%, which were smaller than the coverages of the cache misses. This means that the instructions selected by our technique were not only frequently executed but also caused more

Basic pattern for alignment optimization

```
ClassA objA;
ClassB objB;
while (!end) { // in a hot loop
   …
   // 1) first, load a reference of ClassA
   objA = objB.referenceToClassA;
   …
   // 2) then, access at least two different fields of objA
   access to objA.field1;
   …
   access to objA.field2;
   …
}
```

Figure 5. An example of a code sequence for which object alignment optimization is appropriate. Here we select ClassA as a target for alignment.

cache misses per execution than the instructions not selected. For example, the instructions selected by pattern matching caused more than twice as many cache misses per execution as the instructions not selected regardless of the threshold configuration.

When we select all of the load and store instructions in the methods compiled with hot or higher optimization levels, the number of instructions selected was 19.7% of the total of the load and store instructions. Because we apply our analysis only to these hot methods to avoid excessive overhead, the number of total cache misses for the hot methods were the upper bound for our technique. Compared to this upper bound (the white bars labeled "All in hot methods" in Figure 3), our technique achieved 63.6% and 69.2% coverage for the L1 and L2 cache misses with the Aggressive threshold by selecting 14.2% of the load and store instructions in the hot methods. Table 2 summarizes the coverage by our technique. These results show that our technique can cover a large part of the sources of the cache misses without depending on the HPM.

4.2 Discussion

We show the L1 and L2 cache miss statistics for each benchmark in Figure 4. The bar shows the number of cache misses per 10,000 executed instructions and the breakdown by the code location (JIT-compiled code or native code) and the accessed data type (objects, arrays, or non-Java-heap addresses). SPECpower_ssj2008, SPECjbb2005, and compress generated much more frequent cache misses than the other benchmarks in the JIT-compiled code. In these benchmarks, many of these cache misses were caused by only few hot loops. Our technique is effective in identifying such hot loops even with the Base threshold and hence the coverages for these benchmarks are much higher than the average of the other benchmarks. In the other non-cache-miss-intensive programs, no dominant hot loops exist and many code locations were contributing to the cache misses. Hence, the coverages were more dependent on the hot loop thresholds and their coverages were typically smaller than the three cache-miss-intensive programs.

We found one type of frequent cache miss that our technique failed to detect. Such cache misses are caused by conflicting store instructions from multiple threads. For example, the derby benchmark from SPECjvm2008 caused many L2 cache misses when accessing a method's static variable, and our heuristics could not detect this. These cache misses in the derby benchmark were only observed with multi-threaded execution. When we ran the derby benchmark with only one thread, accesses to the static

Additional patterns for alignment optimization

```
ClassA objA;
ClassB objB;
ClassS objS;  // ClassS is a super class of ClassA
while (!end) { // in a hot loop
   …
   // 1) first, load a reference of a super class of ClassA
   objS = objB.referenceToSuperClass;
   …
   // 2) next, cast objC to ClassA
   objA = (ClassA) objS;
   …
   // 3) then, access at least one field of objA
   access to objA.field1;
   …
}
```

```
ClassA objA;
objA = head;
while (objA != null) { // in a hot loop
   …
   // 1) access at least one field of objA
   access to objA.field1;
   …
   // 2) load a reference to ClassA from objA
   objA = objA.next; or objA = objA.child[nextChild];
   …
}
```

Figure 6. Two additional patterns for the object alignment optimization. ClassA is the alignment candidate for both patterns. The first is typical of hashmap or treemap operations. The second appears while traversing a linked list.

variables did not cause frequent cache misses and the coverage of our technique was greatly improved. This type of cache miss is more difficult to capture by static analysis alone. Adding idioms to capture such special cases is our future work. Additional runtime information such as a lock contention profile will potentially help in identifying such cache misses, because objects shared by multiple threads tend to be guarded by monitors.

5. Applications in Runtime Optimization

In this section, we demonstrate the usefulness of our technique to identify the instructions that cause frequent cache misses in compiler optimizations. We evaluated two types of object placement optimizations, an object alignment optimization and an object collocation optimization based on our cache miss identification technique without relying on the HPM. We also describe techniques to exploit the corresponding opportunities based on the accurate cache miss profiles obtained from the HPM to compare with the optimizations based on our technique.

5.1 Our Object Alignment Optimization Without HPM

In the object alignment optimization, we adjust the address of an object to keep two or more hot fields of the object in the same cache line. Here, we describe how we identify the objects that generate frequent cache misses in more than two fields as targets to align based on matching the basic pattern shown in Figure 1. We search for the hot loops with the pattern shown in Figure 5. This pattern is a straightforward extension of the pattern shown in Figure 1(a). In this pattern, there is a load of a reference to `objA` and two following accesses in one loop. We want to adjust the address of the object to keep these two fields in the same cache

Basic pattern for collocation optimization

```
ClassA objA;
ClassB objB;
ClassC objC;
while (!end) { // in a hot loop
   …
   // 1) first, load a reference of ClassA
   objA = objC.referenceToClassA;
   …
   // 2) next, load a reference of ClassB from objA
   objB = objA.referenceToClassB;
   …
   // 3) then, access at least one field of objB
   access to objB.field1;
   …
}
```

Figure 7. An example of a code sequence for the object collocation optimization. The pair of ClassA and ClassB is the target for collocation.

line to reduce the cache misses caused by this code sequence (from two to one). If we find a hot loop that matches this pattern when compiling a method, we can identify the `ClassA` as a target for the object alignment optimization. Based on the anti-pattern shown in Figure 1(b), we do not select `ClassA` as a target if a reference to `objA` is loaded from the 'this' object in the loop.

Figure 6 shows two variant patterns, but still based on the pattern shown in Figure 1, in addition to the most common pattern shown in Figure 5. We use these special patterns to handle collection classes. Because it is known that the objects managed by a collection class, such as hashmaps, frequently cause cache misses and so these patterns are important when identifying likely cache misses [12]. The first pattern in Figure 6 includes a type cast. In this example, `objS` is converted from `ClassS` (a superclass of `ClassA`) to `ClassA`. In this case, we select `ClassA`, but not `ClassS`, as the target. This pattern commonly appears in loops accessing a hashmap (java.util.HashMap) or a treemap (java.util.TreeMap), in which all of the objects are treated as being in java.lang.Object. The second pattern in Figure 6 handles a linked data structure. This code pattern often appears in a loop iterating over all of the objects in a linked list. In this example, a reference to `objA` is loaded in each iteration and then its fields are accessed in the next iteration. We select `objA` as the target even though the load of the reference and the following use belong to different iterations of the loop. In the hot loop, two fields of each object (field1 and either next or child) are accessed. Thus we pick `ClassA` as the target.

After a target is identified, we do the optimizations at both allocation time and GC time. If the objects frequently cause cache misses in the nursery space, we optimize the object locations at allocation time. If they cause cache misses in the survivor or the tenure space, we do the optimization in the garbage collector. From the source code analysis alone, however, we cannot determine where the objects will reside. Hence, we adjust the object locations of each target at both allocation time and GC time.

For the allocation-time optimization, we generate special allocation code in the JIT compiler, which checks the alignment and adds padding before the new object for all of the allocation sites of the class. In the current implementation, all methods having at least one allocation site of the identified target class are recompiled to apply allocation-time optimization.

The generational garbage collector in the JVM copies objects using the parallel hierarchical copying order [16]. When an object to be tenured is marked as a target for the GC-time alignment, we first check whether the size of the remaining cache line is large enough to hold the object. If the remaining space is too small, we add padding and skip to the next cache line to ensure the object fits into one cache line. We did not implement these optimizations for objects in survivor space because additional operations in the frequently executed young GC may impose heavy overhead in the GC pause time.

5.2 Object Collocation Optimization Without HPM

The object collocation optimization collocates two objects that are accessed together into the same cache line. In the current implementation, we do not apply our optimization to array objects nor do we collocate more than two objects at a time. We identify pairs of classes to collocate when the code matches the pattern shown in Figure 7. We selected the pair of `ClassA` and `ClassB` as a target for the object collocation in this example. In this example, first a reference to `objA` is loaded, then a reference to `objB` is loaded from a field of `objA`, and finally a field of `objB` is accessed in the same loop. In such a code sequence, `objA` and `objB` often cause cache misses and so we can reduce two cache misses to one by collocating the two objects into one cache line. Based on the anti-pattern shown in Figure 1(b), we do not select the target if the reference to `objA` is loaded from the 'this' object.

We need the ordering of the two objects to do the allocation-time object collocation. We check their order in the garbage collector. The analyzer sends information on the collocation targets consisting of a referrer class, a referee class, and the field id of the referrer class, which has a reference to the referee. In the example of Figure 7, the referrer is `ClassA` and the referee is `ClassB`. The garbage collector checks the order of the two objects of these respective classes when it finds an object of the referrer class and its specified field holds a valid pointer to an object of the referee class.

We check if the referee object resides in front of the location of the corresponding referrer object. This means that the referee object was allocated before the referrer object was allocated, because we allocate the objects in the Java heap by simply incrementing a pointer that tracks the next location to allocate in our JVM. In this situation, we can generate special allocation code to add a reserved area of the size of the referrer object in the same cache line for all of the allocation sites of the referee objects. Also, we generate special allocation code for the allocation sites of the referrer objects to use the reserved area generated when the referee object was allocated. In the current implementation, all methods having at least one allocation site of the identified target classes are recompiled. When the garbage collector counts objects in the Java heap, we do not consider the class hierarchy. For example, in Figure 7 `objA` is not necessarily an instance of `ClassA`, but it might be an instance of a subclass of `ClassA`. In the current implementation, we skip such cases to avoid excessive profiling overhead.

For the identified targets, we calculate the object creation frequencies as the ratios of the number of objects created for each class to the total number of objects created by counting the number of objects in the nursery area. If this ratio exceeds our threshold, 10%, then we do not use the object collocation optimization for the class. This is because the optimization imposes additional overhead for CPU cycles and space proportional to the number of the created objects. We count the number of objects allocated in Java heap in the garbage collector and use this information to avoid applying allocation-time

```
allocateObject(class) {
   allocateByte = size of the class;
   updatedCursor = allocationCursor + allocateByte;

   if (updatedCursor > end_of_heap)  call allocation helper

   allocationCursor = updatedCursor;
   return allocationCursor;
}
```

(a) original object allocation code

```
allocateObject(class) {
   allocateByte = size of the class;

   if (class is marked to use reserved area for collocation ) {
      if (allocateByte < sizeOfReservedArea) {
         sizeOfReservedArea = 0;
         return reservedArea;
      }
   }

   if (class is marked to generate reserved area for collocation) {
      if ((allocateByte + byteToReserve) > remainingBytesInCacheline)
         allocationCursor += remainingBytesInCacheline;
      sizeOfReservedArea = byteToReserve;
      reservedArea = allocationCursor;
      allocationCursor += byteToReserve;
   }                                          for object collocation

   else if (class is marked for alignment) {
      if (allocateByte > remainingBytesInCacheline)
         allocationCursor += remainingBytesInCacheline;
   }                                          for object alignment

   updatedCursor = allocationCursor + allocateByte;

   if (updatedCursor > end_of_heap)  call allocation helper

   allocationCursor = updatedCursor;
   return allocationCursor;
}
```

(b) object allocation code for the allocation-time object placement optimization

Figure 8. Pseudocode of the object allocation code sequence used for the allocation-time object-placement optimizations.

optimization for frequently instantiated classes. We also apply the criteria for object alignment optimization described in Section 5.1. Figure 8 is pseudocode for the allocation code sequences for both the alignment and collocation optimizations. The current implementation does not collocate more than two objects. We could support more objects per cache line, but at the price of additional instructions in the allocation code.

5.3 Object Placement Optimizations Using HPM

This section describes the object alignment and collocation optimizations based on the accurate L1 and L2 cache miss profiles obtained from the HPM to contrast them against the technique based on our cache miss identification technique. Note that these optimizations themselves are not the primary focus of this paper though our techniques are much simpler than existing techniques, but still effective. For example, our HPM-based techniques do not require additional metadata to translate the instruction addresses into Java bytecode while previous techniques require huge amounts of additional metadata for this purpose. We implemented a framework to obtain HPM profiles from the JVM by adapting the earlier work [17] to obtain L1 and L2 data cache miss profiles at runtime. To focus on the mutator performance, we do not include the cache misses during stop-the-world GC in the profiles.

Table 3. An example of a cache miss profile for a method in which the object alignment optimization is used.

	class	offset	location	number of samples
1	spec/jbb/Stock	0	tenure	5.1%
2	spec/jbb/Stock	32	tenure	2.1%
3	spec/jbb/Orderline	8	nursery	2.1%
4	spec/jbb/Orderline	56	nursery	1.7%
5	java/math/BigDecimal	24	nursery	1.5%

- L2 cache miss profile for *spec/jbb/*CustomerReportTransaction.process
- number of samples shown in the ratio to the total number of samples

Table 4. An example of a cache miss profile for a method in which the object collocation optimization is used.

	class	offset	location	number of samples
1	java/util/TreeMap$Entry	24	nursery	8.4%
2	spec/jbb/History	0	nursery	4.5%
3	spec/jbb/History	24	nursery	3.8%
4	spec/jbb/Order	0	nursery	2.4%

⋮

- L1 cache miss profile for *spec/jbb/DeliveryTransaction.preprocess*
- number of samples shown in the ratio to the total number of samples

5.3.1 Object Alignment Optimization Using HPM

Based on the accurate cache miss profiles, similar object alignment opportunities can be found. We first use the HPM to generate L1 and L2 cache miss profiles for each method. If multiple fields of one class cause many cache misses in one method, that class is a target for the object alignment optimization. Table 3 shows cache miss profiles that include targets. In this example, we picked two classes, `Stock` and `Orderline`, as the targets for the object alignment optimization, because two fields of each class cause cache misses above the threshold. We used 0.5% of the total cache miss samples from the entire program as the threshold for the alignment optimization. Based on our measurements, this simple heuristic identified many targets that did not provide significant cache miss reductions when using a threshold smaller than 0.5%. In the HPM-based optimization, we can get the location of the objects that caused the cache misses directly from the HPM, as shown in Table 3. In the table, the `Stock` class causes cache misses in the tenure space and so we control the location of the `Stock` objects when the garbage collector moves them into the tenure space. The `Orderline` objects cause cache misses in the nursery space and so we optimize at allocation time. We use the same mechanism to control the object location at the allocation time and the GC time.

5.3.2 Object Collocation Optimization Using HPM

For the object collocation optimization, we identify pairs of classes to collocate in a way similar to the alignment optimization. Table 4 shows an example of cache miss profiles that include targets for the object collocation optimization. In this optimization, we count the references from the objects that cause the cache misses for other objects, as well as the cache misses themselves, to select the targets. First we select each pair of two classes that cause more than 0.5% of the total cache misses in one location (such as the nursery). Then we iterate over the objects of those classes that caused the cache misses and count the number of objects that have references to objects in another class. If the

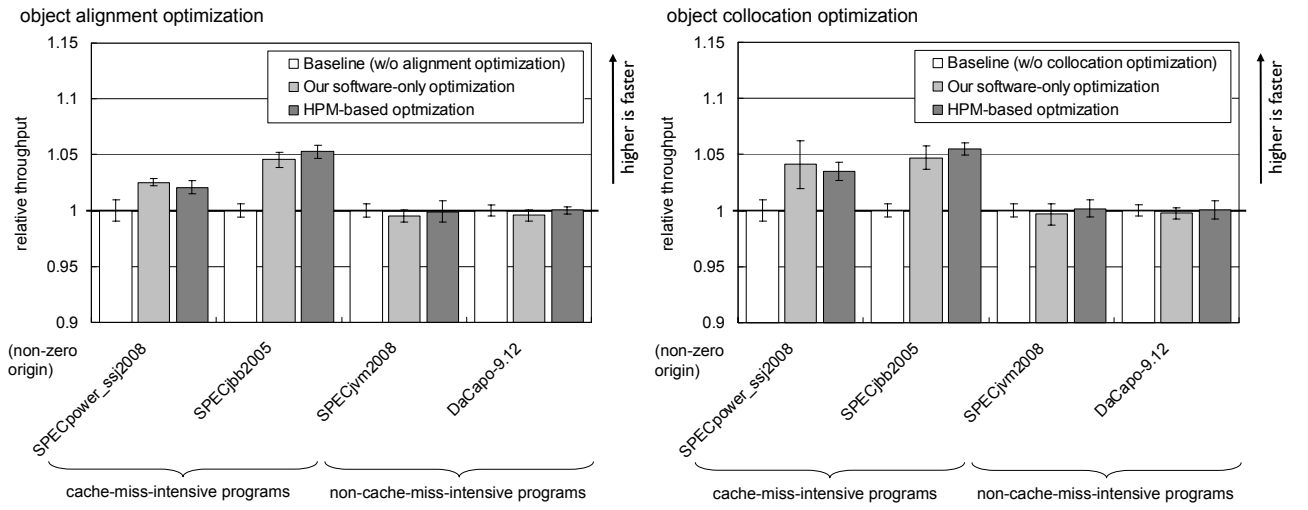

Fig. 9. Performance improvements from object placement optimizations with our approach and with HPM-based approach. The error bars show 95% confidence intervals. We used the Base threshold for pattern matching. Note that the origin of the Y-axis is not zero.

number of objects that have a reference also exceeds 0.5% of the total number of sampled cache misses, then this pair is a target for the object collocation optimization. In Table 4, the `TreeMap$Entry` and `History` classes generate many cache misses and (though the table does not includes this information) many of the `TreeMap$Entry` objects causing cache misses have references to `History` objects, making this pair is a good target. The `TreeMap$Entry` and `History` classes shown in Table 4 both cause cache misses in the nursery space and so we optimize them when they are allocated in the Java heap. If the target objects cause cache misses in the tenure or the survivor space, we collocate the objects in the garbage collector.

For the allocation-time object collocation, we first check the order of the two objects, the `TreeMap$Entry` object that caused the cache miss and the `History` object that is referenced from the `TreeMap$Entry` object as we do in pattern-matching-based approach. To avoid excessive allocation-time overhead, we do not use this object collocation and alignment optimization if the object creation frequency of one of the target classes exceeds 10% based on the object creation profile generated using HPM information [17].

5.4 Performance Improvements by the Optimizations

We implemented the HPM-based and our pattern-matching-based optimizations to compare the two approaches in the object placement optimizations. We implemented both optimizations as online optimizations in Java JIT compiler. We use Base threshold configurations for our pattern-matching used in the evaluations.

Figure 9 shows the performance improvements for SPECpower_ssj2008 and SPECjbb2005 with our pattern-matching-based and the HPM-based optimizations for object alignment and collocation. As shown in Figure 4, the cache miss rates for the other benchmarks are much smaller than these two and hence the effects of the optimizations for the other programs were not significant even when we used the accurate cache miss profile from the HPM. Thus we only show the averages for the SPECjvm2008 and dacapo-9.12 benchmark suites. We ran the performance measurements 8 times and averaged the throughputs. The error bars in the graph show the 95% confidence intervals.

We observed acceleration for SPECpower_ssj2008 and SPECjbb2005. The largest improvements were for SPECjbb2005, with 4.7% for our pattern-matching-based optimization and 5.5% for the HPM-based optimization with collocation. The effects of the optimizations for the non-cache-miss-intensive programs were not significant and the confidence intervals overlap in most cases. On average for the non-cache-miss-intensive programs, we observed small performance degradation with our approach. This performance degradation came from additional runtime overhead caused by additional profiling in garbage collector and also from the extra CPU time for the special allocation code.

We observed significant reduction in both L1 and L2 data cache misses in the two benchmarks that did benefit from our optimizations. In these benchmarks, many of the L1 and L2 cache misses were due to very hot loops, and so it was possible to change the memory access behavior of the entire program by controlling the objects related to these hot loops. In the other programs, many objects were contributing to the data cache misses and thus it was much harder to improve the cache behavior with the object placement changes even when using the precise profiles from the HPM.

These results showed that our pattern-matching-based heuristic approach successfully identified optimization opportunities for object placement optimizations in cache-miss-intensive programs and our techniques achieved similar performance gains by exploiting the same opportunities without depending on the HPM.

5.5 Challenges in Software-Only Optimizations

By comparing the differences between the HPM-based and our pattern-matching-based optimizations in detail, we identified two major remaining challenges in optimizations that do not rely on the HPM.

One obvious advantage is that the HPM can directly identify the location of the objects causing the cache misses. With static analysis alone, we cannot determine the locations of the objects. Hence, for our heuristic approach, we aggressively control the locations of the target objects at both allocation time and GC time. With HPM-based optimizations, we can select the best ways to control the object locations to minimize the additional overhead in CPU time and memory waste. Using the more accurate

statistics gathered in the garbage collector may allow the pattern-matching-based optimizations to be location aware in exchange for the additional runtime profiling overhead.

Another advantage of using the HPM is that the HPM can identify the classes of the objects that cause the cache misses. When a class is identified by pattern matching, the objects at runtime might be instances of subclasses of the identified class. We actually observed such a case with compiler.compiler. The HPM-based collocation identified a target consisting of a pair of `Symbol$MethodSymbol` and `Scope$Entry`. However, the `Symbol$MethodSymbol` objects caused many cache misses when they were accessed as instances of `Symbol`, a superclass of the `Symbol$MethodSymbol`. Our current implementation of the source code analysis and also the profiling facility in the garbage collector do not handle such cases to avoid excessive overhead and thus failed to identify this target. To obtain such information without depending on the hardware, we can generate special code for additional profiling in the identified loop.

Another challenge in the pattern-matching-based optimizations was the criteria to select hot loops. In general, picking more loops for analysis increases the opportunities to reduce the cache misses, but also increases the overhead. When using the Aggressive threshold configuration, the number of identified classes was especially increased for the programs from SPECjvm2008. The thresholds play roles similar to the threshold for the cache miss rate to pick the targets in the HPM-based optimizations (0.5% in our implementation).

Future work for the object placement optimizations will implement more sophisticated profiling techniques, such as software-based sampling techniques to capture object creations [18] or techniques to track the allocation site of each object [19] to obtain more accurate information with smaller overhead.

In summary, our pattern-matching-based heuristic approach successfully identified many of the same opportunities for Java object placement optimization as the HPM-based approach. Though the HPM had some advantages, such as dynamic information on objects that caused the cache misses, our results showed that we can achieve comparable performance gains without using the HPM in SPECjbb2005 and SPECpower_ssj2008.

6. Summary

In this paper, we presented our techniques to identify the instructions and objects that frequently cause cache misses without using the HPM of the processor and then showed its effectiveness in compiler optimization using two examples. Our key insight is that the cache misses are often caused by pointer dereferences in hot loops in the Java programs. Thus we can heuristically identify the targets by finding the hot loops with idiomatic patterns often used in Java programs. We showed that our heuristic technique effectively identified many of the cache misses in a variety of Java programs. As a result, optimizations based on our heuristic approach successfully identified many of the same targets that the HPM-based optimizations identified.

References

[1] A. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney, "Prefetch injection based on hardware monitoring and object metadata", in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 267–276, 2004.

[2] F. T. Schneider, M. Payer, and T. R. Gross, "Online optimizations driven by hardware performance monitoring", in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 373–382, 2007.

[3] M. Serrano and X. Zhuang, "Placement Optimization Using Data Context Collected During Garbage Collection", In *Proceedings of the International Symposium on Memory Management*, pp. 69–78, 2009.

[4] J. Cuthbertson, S. Viswanathan, K. Bobrovsky, A. Astapchuk, E. Kaczmarek, and U. Srinivasan, "A Practical Approach to Hardware Performance Monitoring Based Dynamic Optimizations in a Production JVM", in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 190–199, 2009.

[5] M. Burtscher, A, Diwan and M. Hauswirth, "Static load classification for improving the value predictability of data cache misses" in Proceedings of the ACM Conference on Programming Language Design and Implementation, pp. 222–233, 2002.

[6] V. M. Panait, A. Sasturkar, and W. F. Wong, "Static Identification of Delinquent Loads", in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 303–314, 2004.

[7] T. M. Chilimbi, and J. R. Larus, "Using generational garbage collection to implement cache-conscious data placement", in *Proceedings of the ACM International Symposium on Memory Management*, pp. 37-48, 1998.

[8] T. M. Chilimbi, M. D. Hill, and J. R. Larus, "Cache-conscious structure layout", in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 1–12, 1999.

[9] W. Chen, S. Bhansali, T. M. Chilimbi, X. Gao, and W. Chuang, "Profile-guided proactive garbage collection for locality optimization", in *Proceedings of ACM Conference on Programming Language Design and Implementation*, pp. 332–340, 2006.

[10] M. L. Seidel and B. G. Zorn, "Segregating Heap Objects by Reference Behavior and Lifetime", in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 12–23, 1998.

[11] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh, Exploiting prolific types for memory management and optimizations, in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 295–306, 2002.

[12] A. Jula and L. Rauchwerger, "Two memory allocators that use hints to improve locality", in *Proceedings of the ACM International Symposium on Memory Management*, pp. 109–118, 2009.

[13] J. Jeon, K. Shin, and H. Han, "Layout transformations for heap objects using static access patterns", in *Proceedings of the International Conference on Compiler Construction*, pp. 187–201, 2007.

[14] S. M. Blackburn *et al.,* "The DaCapo Benchmarks: Java Benchmarking Development and Analysis", in *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 169–190, 2006.

[15] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden, "IBM POWER6 microarchitecture", *IBM Journal of Research and Development*, Vol. 51 (6), pp. 639–662, 2007.

[16] D. Siegwart and Martin Hirzel, "Improving locality with parallel hierarchical copying GC", in *Proceedings of the International Symposium on Memory Management*, pp. 52–63, 2006.

[17] H. Inoue and T. Nakatani, "How a Java VM Can Get More from a Hardware Performance Monitor", in *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications*, pp. 137–154, 2009.

[18] M. Jump, S. M. Blackburn, and K. S. McKinley, "Dynamic object sampling for pretenuring", in *Proceedings of the International Symposium on Memory Management*, pp. 152–162, 2004.

[19] R. Odaira, K. Ogata, K. Kawachiya, T. Onodera, and T. Nakatani, "Efficient Runtime Tracking of Allocation Sites in Java", in *Proceedings of the ACM International Conference on Virtual Execution Environments*, pp. 109–120, 2010.