How a Java VM Can Get More from a Hardware Performance Monitor

Hiroshi Inoue and Toshio Nakatani

IBM Research – Tokyo
1623-14, Shimo-tsuruma, Yamato-shi, Kanagawa-ken, 242-8502, Japan
{inouehrs, nakatani}@jp.ibm.com

Abstract

This paper describes our sampling-based profiler that exploits a processor's HPM (Hardware Performance Monitor) to collect information on running Java applications for use by the Java VM. Our profiler provides two novel features: Java-level event profiling and lightweight context-sensitive event profiling. For Java events, we propose new techniques to leverage the sampling facility of the HPM to generate object creation profiles and lock activity profiles. The HPM sampling is the key to achieve a smaller overhead compared to profilers that do not rely on a hardware help. To sample the object creations with the HPM, which can only sample hardware events such as executed instructions or cache misses, we correlate the object creations with the store instructions for Java object headers. For the lock activity profile, we introduce an instrumentation-based technique, called *ProbeNOP*, which uses a special NOP instruction whose executions are counted by the HPM. For the context-sensitive event profiling, we propose a new technique called *CallerChaining*, which detects the calling context of HPM events based on the call stack depth (the value of the stack frame pointer). We show that it can detect the calling contexts in many programs including a large commercial application. Our proposed techniques enable both programmers and runtime systems to get more valuable information from the HPM to understand and optimize the programs without adding significant runtime overhead.

Categories and Subject Descriptors. D.3.4 [**Programming Languages**]: Processors – Run-time environments

http://doi.acm.org/1640089.1640100

General Terms. Measurement, Performance

Keywords. Hardware Performance Monitor; Profiling; Calling Context

1. Introduction

Many modern high-performance processors have an HPM (Hardware Performance Monitor) to count performance-related hardware events and to sample events at specified sampling intervals. Such hardware events include executed instructions, cache misses (at each level of a memory hierarchy), and branch mispredictions. Many profilers are capable of using an HPM to provide programmers with profiles of hardware events.

In this paper, we introduce new techniques in our profiler that extend the scope of HPM-based profilers in two ways. First, our profiler can capture Java-level events, such as object creation or lock activities, by correlating them with the hardware events directly supported by the HPM. Second, we make it possible to detect calling context in addition to the program location (method name and instruction address) for each HPM event with minimal additional runtime overhead in many applications.

As examples of Java-level event profiling, here we study object creation profiling and lock activity profiling. We show how our profiler can derive object creation profiles, including information on both allocated objects and allocation sites, from the store instruction profiles collected by the HPM. For lock activity profiling, which cannot be easily derived from hardware events, we propose a new instrumentation-based technique, called *ProbeNOP*. It uses a special NOP instruction, the *ProbeNOP instruction,* which does not affect the program execution, but whose executions are counted by the HPM. To correlate a piece of code with a hardware event, the JIT compiler generates a ProbeNOP instruction in the code of interest, such as a lock acquisition code sequence. It also encodes information on which register or memory location

to profile at the location of the ProbeNOP instruction within the unused bits of the ProbeNOP instruction. The handler for the HPM interrupt decodes the information encoded in the ProbeNOP instruction and collects the values of the specified targets. Our ProbeNOP technique makes it possible to leverage the sampling facility of the HPM for value profiling in the JVM with low profiling overhead.

Another new feature in our profiler is an efficient context detection technique, *CallerChaining*. It detects the calling context for the HPM event based on the call stack depth, calculated from the value in the stack frame pointer. It does not incur significant additional runtime overhead compared to profiling that is not aware of the context of the events. In our CallerChaining, we first collect quadruples of {caller method, caller call stack depth, callee method, callee call stack depth} using the HPM. We construct a CCT (Calling Context Tree) [1] with a call stack depth in each CCT node by chaining pairs of a caller and callee that have the same call stack depth. We use the call stack depth as a hint to distinguish among the calling contexts that include the same method. While profiling the HPM events, we capture the call stack depth and the instruction address for each event and map the event onto a CCT node using these values. We found that this simple technique works surprisingly well with many tested benchmarks including a large application server workload, though it was not able to uniquely distinguish the full calling context information in those programs which have complicated CCTs.

We implemented our new profiler in the IBM J9/TR Java VM and JIT compiler [2] for Linux running on an IBM POWER6 processor [3]. The runtime overhead of profiling was 2.2% or less when we controlled the sampling interval to generate 16,000 samples/sec. Our techniques do not depend heavily on the specific environment and thus most of the proposed techniques can be applied to other processors and other dynamic language runtimes.

This paper makes the following contributions. (1) We demonstrate that the HPM can be used to profile high-level events in language runtime systems by correlating them with hardware events. We present how our profiler generates object creation profiles and lock activity profiles with low overhead. (2) We introduce a lightweight context detection technique called *CallerChaining*, which detects the calling contexts for events without adding runtime overhead. (3) We present efficient techniques to identify the Java object that caused an HPM event based on a data address captured by the HPM. For example, our technique for identifying objects in the Java heap can avoid the overhead of costly memory scanning [4].

Our proposed techniques enable both programmers and runtime systems to get more valuable information from the HPM to understand and optimize the running programs without adding excess runtime overhead. For example, the lock activity profiles can help optimizing a lock protocol based on its behavior and traits, such as the owner locality [5, 6]. Also, it is known that the calling context of the allocation site is important to predict object lifetime [7], and hence our object creation profiles with the calling context can support adaptive optimizations based on the objects' lifetimes [8].

The rest of the paper is organized as follows. Section 2 gives an overview of the hardware performance monitor of the POWER6 processor. Section 3 discusses related profiling techniques. Section 4 describes our HPM-based sampling profiler. Section 5 illustrates our new techniques to profile high-level events using the HPM. Section 6 uses our technique to detect the calling context using the call stack depth. Section 7 describes the experimental environment and our results. Finally, Section 8 summarizes our new techniques.

2. HPM of the POWER6 Processor

In this paper we use the IBM POWER6 processor [3] to present our new profiling techniques. Like many other processors, POWER6 has a built-in HPM (Hardware Performance Monitor) that can monitor and count various performance-related events, such as the cache misses or instructions executed on the processor. The POWER6 provides four performance counters (programmable from the operating system) so that up to four performance-related events can be counted simultaneously.

An HPM counter can be configured to generate an interrupt when the counter value overflows. To provide detailed information on the HPM event that caused the interrupt, POWER6 has two special purpose registers called SIAR (sampled instruction address register) and SDAR (sampled data address register) [9]. The SIAR contains the instruction address of the sampled instruction and the SDAR contains the data address if the sampled event is a memory-related event[†]. By using the HPM interrupts, we can sample events for specified sampling intervals and obtain instruction and data addresses for each HPM event.

The implementation of an HPM depends on the processor, but most of today's processors have similar features. For example, recent Intel's x86 processors support PEBS (precise event based sampling), which arrows profilers to obtain an architectural state information of the processor when a selected event occurs [11] like the

[†] To have accurate values in the SIAR and SDAR registers, we need to use HPM events named with the prefix *PM_MRK*. The list of the HPM events on POWER6 is available in the Oprofile distribution [10].

SIAR and SDAR of the POWER6. The PEBS is useful to implement our ProbeNOP technique and CallerChaining. However the PEBS does not support the store instruction executed event and thus our object creation profiling is not directly applicable for the PEBS. To implement the ProbeNOP technique with the PEBS, x87 floating-point instructions or SIMD instructions can be used as the ProbeNOP instruction if these instructions are not used in the JVM.

3. Related Work

There are existing profilers, such as Oprofile [10] in Linux and the tprof command in IBM AIX, which can generate hardware events profiles using the HPM. The scope of those existing profilers is, however, limited to the profiles for hardware events directly supported as an event in the HPM, such as cache miss profiles. In contrast, our profiler can provide low-overhead Java-level events profiles, such as object creation profiles and the lock activity profiles, in addition to the hardware event profiles.

There are profilers to capture such Java-level events, which typically use JVM Tool Interface (JVMTI) [12] to communicate with the JVM. However they often incur unacceptable overhead during program execution. For example, HPROF as included in JRE distributions can provide a rich set of useful information including CPU time profiles, object creation profiles, and monitor contention profiles. Because HPROF incurs significant overhead, the performance of profiled applications often degrades by more than a factor of 10. Such overhead may make the CPU time or monitor contention profiles unreliable and different from the unmonitored profiles. Our profiler can provide similar Java-level event profiles with much lower overhead by exploiting the sampling facility of the HPM.

Due to the importance of the object creation profiles, many profilers have been developed to collect various information on object creation. Some profilers insert a hook in the object creation code to identify the allocation site, while others count the number of objects for each class in the Java heap. However a hook incurs significant overhead, while object counting cannot capture the allocation site information accurately. Compared to these techniques, our HPM-based approach can capture information on both the created objects and on their allocation sites with low overhead by exploiting the sampling facility of the HPM. Some software-based sampling techniques have been proposed for the object creation sampling [13, 14]. The overhead of these techniques are typically small, such as less than 3.0% [13]. An advantage of our HPM-based object creation sampling technique over these software-based sampling techniques is that we do not need to generate any additional instructions in the JIT compiler and thus our technique does not impose additional overhead while the HPM is not configured to generate interrupts.

We created ProbeNOP, an instrumentation technique for value profiling, which uses HPM sampling features. In this technique, we insert only a special NOP instruction in the program code and thus the overhead is negligible while the sampling is disabled. In contrast to an existing low-overhead profiling technique [15], which generates duplicated code with instrumentations, our technique does not impose significant space overhead because it only adds one instruction per sampling point. This advantage makes it possible to insert instrumentation code for all locations of interest to generate complete profiles.

Our profiler supports context-sensitive profiling by our *CallerChaining* technique. Profiles with calling context information, consisting of current program locations and sequences of call sites on the stack, are more informative in characterizing the program behaviors compared to profiles with only current locations. Often stack walking is used when a profiler needs to know the current calling context. Alternatively, it is also possible to track the current calling context at each method invocation and exit by inserting the instrumentation code [1, 16]. However both of these techniques are very costly. For example, Oprofile supports context-sensitive profiling by walking the stack at each invocation of the handler for an HPM interrupt, but this is much more work than profiling that ignores the contexts. Our technique avoids this work in the interrupt handler by not using stack walking.

There are advanced techniques for context-aware profiling [17, 18, 19, 20]. Among them, Bond and McKinley's [17] Probabilistic Calling Context (PCC) can be directly applicable for the HPM profiling. The PCC allows for lightweight context-sensitive profiling by maintaining only one value that represents the current calling context. Both the PCC and our CallerChaining are probabilistic approaches to detect the calling context. In contrast to ours, the PCC generates special code and data structure to do context-sensitive profiling and thus imposes overhead in both computation time and memory space in exchange for higher accuracy. Our CallerChaining does not impose visible runtime overhead as long as the HPM is not configured to generate interrupts. This advantage makes our technique more attractive to use as part of the runtime support for adaptive optimizations. Because we only use call stack depth to identify the calling context instead of a special value in PCC, our technique cannot distinguish contexts accurately in programs that have very complicated CCTs. For such programs, we can combine our technique with the PCC to improve the accuracy in profiling with additional overhead.

Recently, Mytkowicz *et al.* [21] proposed a technique called *inferred call path profiling* for C and C++ programs. Although developed independently, their technique also

uses pairs of the current instruction pointer and the call stack depth to identify the calling context. They focused on techniques to disambiguate the calling contexts that had the same call stack depth.

There are some existing techniques that use cache miss profiles obtained by the HPM when applying adaptive optimizations in compilers and runtime systems [4, 22, 23, 24]. For example, Adl-Tabatabai *et al.* [4] exploit cache miss statistics in their Java JIT compiler to insert effective prefetch instructions on the Intel Itenium2 processor. Later, Schneider *et al.* [22] also used cache miss statistics in the garbage collector to optimize the placement of objects in the Jikes RVM on the Intel Pentium4 processor. These techniques identify the instructions and objects that cause many cache misses and exploit the information for optimizations. We seek to provide more information on running programs, including Java-level event profiles and context-sensitive event profiles, without adding significant runtime overhead.

4. Profiling Framework

In this section, we describe the implementation of our profiler to capture the hardware events and generate the profiles for those events. In particular, we focus on our effective translation techniques to identify the Java object that caused an event based on the data address of the event. Our new technique to collect Java-level events using the HPM and our context-sensitive profiling technique will be described in later Sections 5 and 6, respectively.

4.1 Oprofile device driver in the Linux kernel

Our profiler consists of two parts: an operating system device driver (to access the HPM from the user space) and a built-in profiler implemented in the JVM, which accesses the HPM via the interface exported by the device driver. We used Linux as our target operating system and used the existing Oprofile [10] device driver in the Linux kernel as the basis of our device driver.

The Oprofile device driver for the PowerPC architecture provides an interface to set the HPM-related special purpose registers from user space. By using this interface, we can select up to four HPM events to count in the HPM counters and also specify a sampling interval for each counter to generate an interrupt when that counter reaches its sampling interval. We did not modify this interface. Oprofile also provides an OS-space buffer to store the results of the HPM samples. User-space processes can read the OS-space buffer using a `read` system call to a special device file. A user process will block on the system call and return when the OS-space buffer becomes full or is explicitly flushed.

4.2 Overview of our profiler

What we extended in the device driver is the information to capture in the HPM interrupt handler as implemented in the device driver. The original Oprofile only captures the HPM counter ID, to label the source of the interrupt, and the instruction address that caused the event for each HPM sample. Our extended version also captures a data address from the SDAR register (for memory-related events) and the value of the stack frame pointer. We use this frame pointer value in three ways: (1) to identify the software thread that caused the event, (2) to identify the Java object that caused an event if that object is allocated on stack, and (3) to track the calling context for the events as described in Section 6. The JVM has two kinds of stack frame pointers, GPR1 for the system stack and GPR14 for the Java stack. We use GPR14 in our profiler.

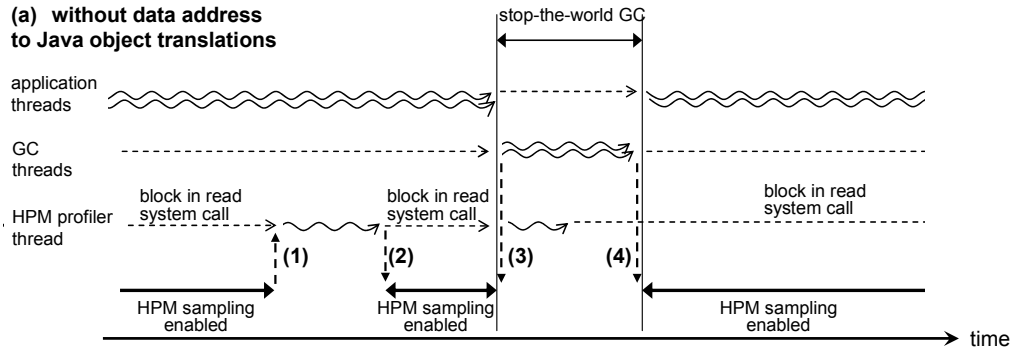
We create a dedicated thread for the HPM profiler in the JVM to read the HPM samples from the OS-space buffer and generate the statistics. Though the JVM has another profiling mechanism based on timer interrupts to control the JIT compilation, our HPM profiler thread only handles the HPM-based profiling and does not affect the timer-based profiling mechanism.

The HPM profiler thread first sets the HPM events to profile, and then synchronously read the OS-space buffer for the HPM samples, so the thread blocks on this `read` system call until the OS-space buffer becomes full. When the buffer becomes full, then the HPM profiler thread returns from the `read` system call and copies the content of the buffer into a user-space buffer.

Once the data has been copied to user space, the HPM profiler thread translates the instruction address of each sample into a Java method, when the address is included in a JIT-compiled method, or into a JVM module, when the address is not in a JIT-compiled method. The JVM module will be, for example, the interpreter, the memory manager, or the JIT compiler. The original JVM already has the data to convert an instruction address of the JIT-compiled method into a Java method, so we do not need a new data structure for that purpose. Most JVM implementations should have similar data structures to identify program locations when exceptions occur in JIT-compiled methods.

For memory-related events, the HPM profiler thread also translates the data address to a Java class, an offset in the object, and the location of the objects (nursery, survivor, tenure, or stack). We give details on the data address to Java object translation process in the next sections.

Focusing on running Java applications, we do not use the HPM sampling during stop-the-world GC. When the Java heap becomes full and stop-the-world GC begins, we stop the HPM sampling and explicitly flush the OS-space buffer. The GC threads must wait for the HPM profiler thread to complete the data address translation. This is

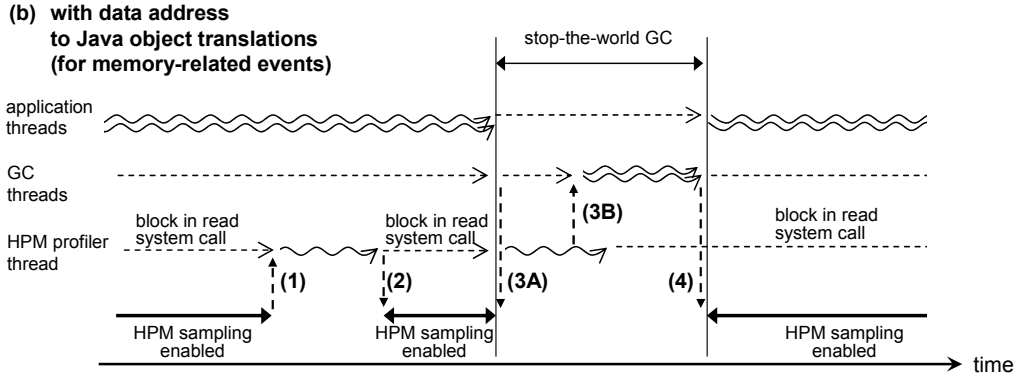


(1) When the buffer for HPM samples in the OS space becomes full, the HPM profiler thread returns from the `read` system call. HPM sampling is disabled.

(2) When the HPM profiler thread finishes processing the samples, HPM sampling is re-enabled.

(3) When a stop-the-world GC starts, the GC threads force flushing the OS-space buffer and the HPM profiler thread returns from the `read` system call.

(4) When the stop-the-world GC ends, GC threads re-enable the HPM sampling. (If the HPM profiler thread has not finished its processing, the GC thread does not re-enable the sampling.)



(3A) When a stop-the-world GC starts, the GC threads force flushing the OS-space buffer and the HPM profiler thread returns from `read` system call. The GC threads wait for the HPM profiler thread to finish translating the data addresses of all samples.

(3B) After the HPM profiler thread completes the translations, it wakes the GC threads and continues running to generate profiles based on the already translated samples.

Figure 1. Schematics of collaboration among an HPM profiler thread, GC threads, and application threads: (a) when the data address to Java object translations are not involved (b) when the translations are involved.

because the HPM profiler thread cannot identify the Java object from the data address if the GC threads move the object in the Java heap. Figure 1 depicts how the HPM profiler thread collaborates with the other threads with and without the data address translation. In the current implementation, we do not parallelize the HPM profiler thread and use only one dedicated thread.

4.3 Identifying objects in the Java heap

This section describes our new techniques to effectively find the object that caused a memory-related event, such as a cache miss, based on a data address obtained from the HPM, when the data address points at an address inside the Java heap. As already mentioned in Section 4.2, we ensure that objects in the Java heap have not been moved by the garbage collector before the data address to Java object translation completes.

Adl-Tabatabai *et al.* [4] do this translation by scanning memory backward from the sampled data address to find the nearest valid object header, starting with a pointer to a virtual function table of the class, in their technique called *Mississippi delta*.

To avoid the large cost of this backward scan, we introduce a new technique, which tries a heuristic before the scan. It first checks the instruction that caused the event, and if the instruction is a load or store instruction that points to an address to access with a value in a register and a constant offset, it is likely that the value in the register is

Existing Technique

```
doBackwordMemoryScan (dataAddress) {
   p = dataAddress;
   count = 0;
   while (count < LIMIT) {
      if (*p contains a valid object header) {
         return p;
      }
      count++;
      p = p – 8;  // assuming object is 8-byte aligned
   }
   return HEADER_NOT_FOUND;
}
```

Our Technique

```
findObjectHeader (dataAddress, instructionAddress) {
   // shortcut path by checking instruction
   instruction = *instructionAddress;
   if (instruction is a load/store addressed by register + constant) {
      if (*(dataAddress – constant) contains valid object header) {
         // successfully identified object
         return dataAddress – constant;
      }
   }
   // fall back to memory scanning
   return doBackwardMemoryScan (dataAddress);
}
```

Figure 2. Pseudocode of the two methods to identify the Java object that caused an event.

a pointer to an object and the constant is an offset within the object. We then test that location to see if it contains a valid object header. Because the IBM JIT compiler generates this form of load and store instructions for field accesses, this works in many cases. If the instruction is not in the form or if the assumed location does not contain a valid object header, we fall back to the memory scan to find a valid object header. Figure 2 shows pseudocode for our technique and the existing memory scanning technique. In SPECjbb2005, we successfully identified the objects using this heuristic in 84.2% and 83.6% of the L1 and L2 cache miss events. This technique cannot be used if the JIT compiler uses optimization techniques that change the object format, such as object inlining [25].

4.4 Identifying stack-allocated objects

This section describes how our profiler identifies a stack-allocated object that generates an event from a data address. We can assure that the location of the object in the Java heap is not changed before the HPM profiler executes the data address to Java object translations by controlling the GC activities. However, it is not possible to retain the state of a stack-allocated object because the stack frame that included the object was discarded when the application thread exited from the method. Thus we need another technique to track stack-allocated objects.

As already described, our profiler captures the value of the stack frame pointer for each HPM sample. We use the frame pointer value to identify the stack-allocated objects. If the sampled data address is in a stack for any Java thread, the profiler calculates the offset of the data address in the stack frame by subtracting the frame pointer value from the data address. Then it looks up a table that contains offsets and sizes of all stack-allocated objects for each Java method. We modified the JIT compiler to maintain this table because the original JVM did not track the information. The space overhead for this additional data structure is small because the stack allocation of local objects based on escape analysis is a costly optimization and is applied only to a few very hot methods.

5. How to Get Java-level Event Profiles Using the HPM

In this section, we give details about our techniques to obtain high-level information in a Java VM using the HPM. First we describe a simple example for an object creation profile. Then we describe our techniques for a more complicated example using lock activity profiling.

5.1 Identifying object creation events by screening store instructions

Here, we show how our profiler derives an object creation profile from the HPM samples. Though what we describe in this section looks simple and naive, it yields accurate statistics, including information for both the created objects and allocation sites.

Our observation is that, in our JVM, the first word in the object header, which is the virtual function table pointer, is not modified once the object is initialized. Based on this observation, we create the object creation profiles by first collecting the store instruction profiles using the HPM, which includes a data address and an instruction address for each sampled store instruction, and then translate the data address of each sample into a Java class and offset in the object. Here we used the PM_MRK_ST_NEST event on POWER6. Next, we filter out any sample whose offset value is not zero. After this filtering, the store instruction profile has become an object creation profile, because a store to the virtual function table pointer appears only when the object is created. Of course, the garbage collector also writes this word in the headers, but we disable the HPM sampling during garbage collection to prevent those store instructions during GC from being included in the profile.

Compared to existing profiling techniques for object creation profiling, our HPM-based approach can capture information on both the created objects and their allocation

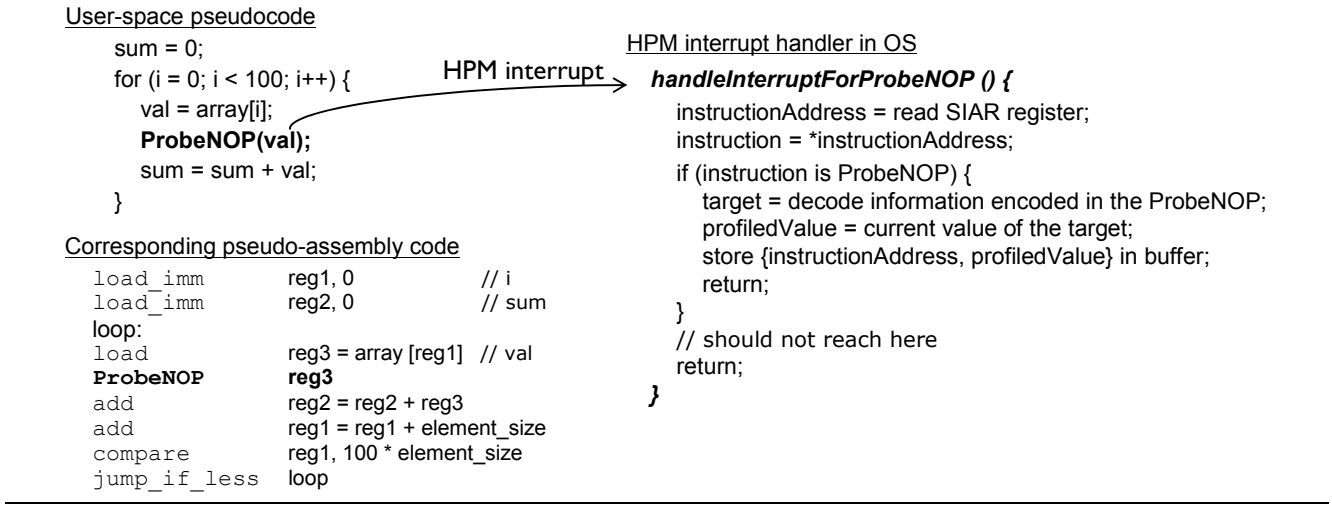


Figure 3. A simple example of ProbeNOP usage.

sites with low overhead by exploiting the sampling facility of the HPM. Our technique does not require any special code for object allocations nor does it impose limitations in compiler optimizations.

By applying a similar technique to a profile of store instructions, we can also create a profile of those objects and methods which invoke a write barrier for the generational garbage collector.

5.2 Identifying Lock activities by inserting ProbeNOP instructions

In this section, we introduce a new instrumentation-based profiling technique using a special NOP instruction to capture Java-level events that cannot be easily associated with hardware events. This technique involves the JIT compiler generating special NOP instructions whose execution is counted by the HPM in the code of interest, such as lock acquisition code sequences.

5.2.1 ProbeNOP

The basic idea of our technique is that we insert a special NOP instruction where we want to probe. This does not affect the program meaning or performance but only increments an HPM counter when encountered. We call this NOP instruction *ProbeNOP*. We use the ProbeNOP instruction to invoke an HPM interrupt and also to send context-dependent information to the HPM interrupt handler by encoding the information in unused fields in the NOP instruction.

We show a simple example of using ProbeNOP in Figure 3. The left side of the figure shows simple pseudocode to calculate the sum of the elements in an array with a ProbeNOP instruction inserted in the loop. This ProbeNOP does not affect the program execution or performance if HPM sampling is not active. If the HPM is configured to count the ProbeNOP executions and to generate an interrupt with a sampling interval of 10, it still does not affect the meaning of the code, but it generates an HPM interrupt once per ten iterations. The pseudocode for the HPM interrupt handler in the OS is shown on the right side of the figure. The handler first identifies the instruction that generates the interrupt and checks if that instruction is a ProbeNOP. Then it decodes the target information encoded in the ProbeNOP instruction's bit pattern. Here the target is the `val` stored in reg3. The handler stores the pair of the instruction address and the value of the target, reg3, in the OS-space buffer for HPM samples instead of the pair of the instruction address and the data address as when the HPM handler is tracking a memory-related event.

The profiler can later determine the value of the target by reading the HPM samples from the OS-space buffer. As

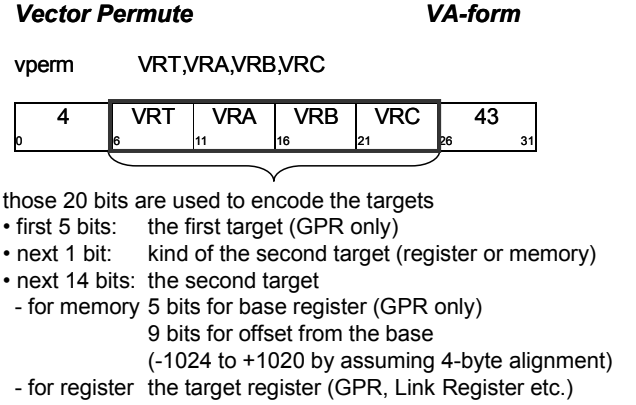


Figure 4. ProbeNOP instruction format on POWER6 using the vector permute instruction [26].

Monitor enter code sequence in JIT-generated code

```
...
ProbeNOP(obj);   // at monitor enter
// monitor enter for obj
if (obj is in inflated mode || tryLock(obj) == FAILED) {
   // tryLock failed if obj is locked by another thread
   // or recursion count overflows
   monitorEnterHelper(obj);
}
// critical section begins
...
```

Monitor enter helper in JVM

```
monitorEnterHelper(obj) {
   caller = read link register;
   ProbeNOP(obj, caller);   // at helper enter
   sync();
   if (obj is in inflated mode) {
      inflatedMonitorEnterHelper(obj);
      return;
   }
   for (i=0; i<loopCount1; i++) {
      for (j=0; j<loopCount2; j++) {
         if (tryLock(obj) == SUCCESS) return;
         ProbeNOP(obj, caller);  // at spin loop
         sync();
         doIdleLoop();  // just consume cycles
      }
      yield_cpu();
   }
   inflateTheObject();  // spin locking failed
   return;
}
```

Figure 5. ProbeNOPs for lock activity profile.

a result, the profiler gets the values of the 10th element, 20th element, .., and the 100th element in the array as a result of the sampling.

This technique using the ProbeNOP enables value profiling of the specified target while minimizing the performance degradation, especially when the HPM sampling is not enabled. Note that the interrupt handler in Figure 3 is simplified for ease of explanation. We will show the pseudocode for the full interrupt handler in Section 6.

5.2.2 Implementation on POWER6

To implement this technique on POWER6, we use the vector permute (`vperm`) instruction of the VMX (Altivec) instruction set as a ProbeNOP [26]. We selected this instruction because our JVM does not use VMX instructions and the HPM of the POWER6 can count the number of `vperm` instructions executed by the PM_MRK_VMX_PERMUTE_ISSUED event. Also, the VMX instructions are executed in a dedicated pipeline and thus do not affect the execution of other instructions significantly. The `vperm` instruction format has 20 bits that we can use to encode the target information, because it uses four registers as operands and each register ID is 5 bits. In the current implementation, we support up to two targets when collecting values in the interrupt handler in one ProbeNOP. The first target is dedicated for general purpose registers and the second target is for one of the general-purpose registers, another register such as the link register, or a memory location referenced by the value in the specified general-purpose register with an offset. If two targets are specified in a ProbeNOP instruction, the interrupt handler stores the instruction address and the values of both targets for each HPM sample. Figure 4 depicts the instruction format of the ProbeNOP.

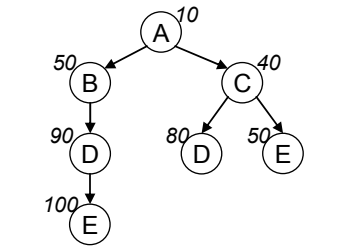
We have a limitation in implementing our technique on POWER6 since the HPM interrupt is not precise; the processor state when an event occurs are not preserved until the HPM interrupt handler invoked. It means the target values, such as those in general purpose registers, may change before the profiler read the values in the interrupt handler. We can still determine the correct instruction address and data address because the POWER6 keeps the information in the SIAR and SDAR registers until the interrupt handler is invoked. To ensure that the interrupt handler reads a valid target value on POWER6, we can add a sync instruction after each ProbeNOP as a workaround[†]. In the next section, we insert a ProbeNOP in each lock acquisition operation to generate lock acquisition profiles. Fortunately, in this case the monitor acquisition operation inherently includes a sync operation and thus we did not need to explicitly add an expensive sync instruction.

5.2.3 Lock activity profile using ProbeNOP

In this section, we use the profiling technique with ProbeNOP to generate a detailed lock activity profile.

[†] We empirically found this workaround worked on the POWER6 processor, although the Power Instruction Set Architecture [26] does not explicitly require this behavior for the sync instruction.

callee method	callee call stack depth	caller method	caller call stack depth	number of samples
B	50	A	10	10
C	40	A	10	10
D	90	B	50	20
D	80	C	40	40
E	50	C	40	10
E	100	D	90	80



constructed calling context tree
(number at each node means call stack depth)

Figure 6. Constructing calling context tree from HPM samples based on call stack depths.

The IBM JVM implements a bimodal locking algorithm [27, 28, 29] in which a lock word in the object header has one of two modes: a *flat* mode for spin locking and an *inflated* mode for suspend locking. Each object starts from the flat mode and enters the inflated mode when a Java thread fails to acquire the lock of the object by spin locking. We focus on profiling the flat mode locks with our techniques. The IBM JVM supports a powerful lock profiling tool named Java Lock Monitor (JLM), included in the Performance Inspector [30], which can provide detailed information on lock contentions. However the scope of JLM is limited to the inflated locks, and it cannot provide information on the program locations that caused the lock contention. Our technique can overcome these limitations, and combining our profiler with the JLM can give a complete picture of the lock activities.

We insert ProbeNOP instructions at the following three kinds of locations:

1) all lock acquisition operations in the JIT-compiled methods (labeled *monitor enter* in the figure)
2) at the entry point of the monitor enter helper function (labeled *helper enter*)
3) in the spin loop for flat locks in the helper function (labeled *spin loop*)

Figure 5 shows the locations of the inserted ProbeNOPs using simplified pseudocode for the JIT-compiled code and the helper function. Each ProbeNOP has a different purpose. The ProbeNOP in the JIT-compiled code is to capture all of the lock acquisitions, including the successes that do not call the monitor enter helper function. The ProbeNOP at the helper entry point is to provide the program location information for the inflated monitor activities captured by the JLM. The ProbeNOP in the spin loop is to identify the locks which consume CPU time in this spin loop. Note that the two ProbeNOPs in the helper code are followed by sync instructions to assure that an HPM interrupt for the ProbeNOP is generated at those places. These sync instructions are in rarely executed paths and they do not incur significant overhead in most cases.

The HPM profiler thread can distinguish among the samples from these three types of ProbeNOPs by the instruction address in each sample. Then it generates high-level lock activity profiles, such as the hot locks that often spent long periods in the spin loop. Programmers can combine such information with the output of the JLM to understand the lock activities.

6. How to Get Context-Sensitive Event Profiles Using HPM

In this section we present our technique, called *CallerChaining*, to provide context-sensitive profiles without walking a stack for each HPM sample.

6.1 Frame-pointer-based Calling Context Tree (CCT) generation

We first explain how we generate a calling context tree with a call stack depth in each CCT node from the HPM samples based on the values in the stack frame pointer. In this technique, we assume that the size of stack frame for each method is a constant during the measurements. This implies we need to stop the JIT compiler during the measurements and this particular technique is only suitable for working with an application in a steady state, but is not suitable for transient states such as those at boot time.

For this technique, we need to collect samples as quadruples of {caller method, caller call stack depth, callee method, callee call stack depth}. The sampling of those values does not require complicated operations and is much less expensive than the stack walking. Figure 6 shows examples of HPM sample profiles and a constructed CCT. In the table, the call stack depths for the callee and caller are shown as offsets from the base of the stack area of the thread. Though the stack grows downward in our JVM, we show the offset value as positive in the figure. We construct the CCT by chaining the callers and callees based

on call stack depths. From the HPM samples shown in the table, it is obvious that Methods B and C are called from Method A. Also, Method D is called from Methods B and C via the paths A-B-D and A-C-D, respectively. The method E is called from Method C and D, but for Method D we cannot tell whether the method E is called via path A-B-D-E or path A-C-D-E from the callee and caller relationship data alone. For such a case, we can use the call stack depth to distinguish the paths. In the example, Method D has a call stack depth of 90 only when it is called from Method B and thus we can determine that method E is called via the path A-B-D-E rather than A-C-D-E. The right side of figure 6 shows the constructed CCT with the call stack depth in each node. Note that because this technique distinguishes among the call sites based on the call stack depth, it cannot distinguish between call sites in the same method if a method has multiple call sites for the same callee.

A good way to collect these quadruples is to insert a ProbeNOP at each method entry point to profile the link register value (caller address) and the call stack depth. On the POWER6 processor, however, we need to add a sync instruction for each ProbeNOP to assure a precise interrupt. Adding a sync instruction for each method entry point incurs unacceptable overhead and so we took a different approach as a workaround. We first configure the HPM to generate an interrupt after a certain number of instructions are executed. In this configuration, interrupts are generated almost randomly throughout the program. If an interrupt happens in a method prologue, while the link register still has the caller address, we capture the link register value (caller address) and the frame pointer value (call stack depth). This approach does not require ProbeNOP and sync instructions for each method, but it requires more samples because we throw away most of the samples that miss the method prologues. That means that we need a long time to generate the CCT.

If the stack frame structure allows the interrupt handler to access the caller address stored in the stack frame, we can use the samples that do not hit the method prologue. In the current stack frame structure in our JVM, however, the location of the stack slot which contains the caller address is different from method to method, so it is difficult to obtain the caller address stored on the stack in the interrupt handler.

Another disadvantage of not using ProbeNOP is that the number of samples cannot always be used as an indicator for the hotness of the edge in the CCT. In Figure 6, for example, the edges A-B and A-C have the same number of samples (10) as shown in the table, but that does not mean that Method A calls Methods B and C with the same frequency, because the size of prologue differs for each methods. However it is still possible to use the number of samples to determine the relative hotness of the edges, as long as those edges point at the same callee. For example, we can estimate that Method D calls Method E eight times more often than Method C based on numbers of samples in the table (10 and 80). We can use these estimates of edge hotness to distribute the HPM events.

Figure 7 illustrates two cases where our technique fails to identify the context based on call stack depths. The first case is one in which some methods, B and F in the figure, have the same stack frame size. This can be generalized as to cases where two call sequences, each consisting of multiple methods, have the same total stack frame size. The other case is when some methods, B and C here, appear in the calling context in different orders. In both cases, we add a node as needed in this phase.

6.2 Mapping HPM events on Calling Context Tree

In this section, we map the HPM events, such as cache misses, onto the generated CCT. Because our profiler collects a value in the stack frame pointer for each sample, it is mostly straightforward to map the events based on the instruction addresses and the call stack depth. For example, if Method E caused a cache miss when the call stack depth was 100, the event is mapped on the node of Method E on the path A-B-D-E.

When there are multiple nodes having the same pair of a method and call stack depth, such as the cases shown in Figure 7, we distribute the HPM events based on hotness of the edges as many existing profilers, such as gprof [31], do when distributing the events or execution times for the callers. In the example of Figure 7(A), cache miss events generated by Method D are distributed into two paths A-B-D and A-F-D in proportion to the estimated hotnesses of the edges B-D and F-D, which we have already calculated when we built the CCT. If this uncertainty affects the overall profile too greatly, we can change the stack frame size of a method by simply adding padding in the stack frame and retry the profiling [21].

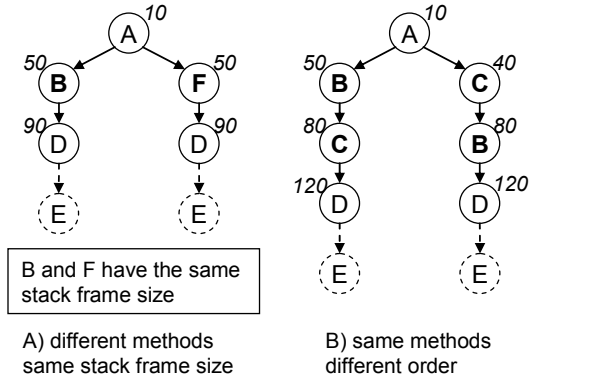


Figure 7. Two examples of calling context trees for which our techniques cannot uniquely identify the calling context of the Method E.

```
handleInterrupt () {
  for each HPM counter which overflows {
    instructionAddress = read SIAR register;                    // instruction address which caused the interrupt
    if (running in Oprofile compatible mode) {                  // to allow the original Oprofile running with this driver
      store {instructionAddress, counterID} in buffer;
    }
    else if (the counter is counting ProbeNOPs) {               // handler for the ProbeNOP profiling (see section 4)
      instruction = *instructionAddress;
      if (the instruction is a ProbeNOP) {                      // to wipe out irregular samples
        targets = decode information encoded in the ProbeNOP;   // one ProbeNOP includes up to two targets
        profiledValue1 = current value of the target1;
        profiledValue2 = current value of the target2;
        store {instructionAddress, profiledValue1, profiledValue2, framePointerValue, counterID} in buffer;
      }
    }
    else if (generating a calling context tree) {               // handler for the CCT reconstruction (see section 5)
      currentInstAddr = current instruction address             // to get the address consistent with the frame pointer
      store {currentInstAddr, linkRegisterValue, framePointerValue, counterID} in buffer;
    }
    else if (SDAR register contains a valid value) {            // by checking a flag in a special purpose register
      dataAddress = read SDAR register;                         // handler for memory-related events
      store {instructionAddress, dataAddress, framePointerValue, counterID} in buffer;
    }
    else {                                                      // handler for non-memory events
      store {instructionAddress, framePointerValue, counterID} in buffer;
    }
    reset the counter according to the sampling interval;
  }
}
```

Figure 8. Pseudocode for the HPM interrupt handler.

The lack of precision of the HPM interrupts in POWER6 is a more severe problem. As already mentioned, we can get the exact instruction and data addresses that caused an event, but the frame pointer value (call stack depth) at the time of the event may have already changed when the interrupt handler is invoked. Due to this problem, some HPM events do not have a corresponding node in the CCT. We try to find the correct nodes for such HPM events by changing the call stack depth to that of a possible caller or callee in the CCT. In the CCT of Figure 6, for example, a cache miss generated by Method E with a call stack depth of 40 is mapped to the node of Method E in the path A-C-E, because the node's caller also has the call stack depth of 40. This can happen when Method E causes a cache miss but the HPM interrupt for the cache miss occurs after returning to the caller, method C. With this fitting process, we successfully found the corresponding nodes for most of the samples, even in the very large application.

Complete pseudocode for the entire HPM interrupt handler appears in Figure 8.

7. Experimental Results

This section evaluates our profiling techniques by using standard benchmarks and a very large Web application server, the IBM WebSphere Application Server.

7.1 Profiling overhead

We evaluate our profiler using SPECjbb2005 [32], SPECjvm2008 (compiler.compiler, derby, sunflow, xml.validation, serial, and mpegaudio) [33] and DayTrader 2.0 [34] running on IBM WebSphere Application Server version 7.0 [35]. We implemented our profiler in the 32-bit JVM included in the IBM SDK for Java 6 SR2. We ran the benchmarks on an IBM BladeCenter JS22 using 2 cores of the 4.0-GHz POWER6 processors with 2 SMT threads per core. This means SPECjbb2005 and SPECjvm2008 were configured to run with 4 threads. Each core has 64 KB of L1 data cache and 64 KB of L1 instruction cache and 4 MB of L2 cache. The size of the Java heap was 2 GB using 16-MB large pages and the generational garbage collector was selected. The full JVM command line options were -Xgcpolicy:gencon -Xms2000m -Xmx2000m -Xmo400m -Xgcthreads4 -Xlp. The system has 16 GB of system memory and runs RedHat Enterprise Linux 5.2. For DayTrader, the DB2 database server and the client emulator ran on separate machines.

Figure 9 compares the performance of programs with and without the profiler attached to determine the overhead of the profiler. The overhead of the profiling depends on the sampling interval. Shorter intervals (higher interrupt frequencies) give higher accuracies and larger overheads. We controlled the sampling interval to generate 8,000 samples/sec for each event (one sample per 500 μsec per HW thread, or per 2-million CPU cycles) or 2,000

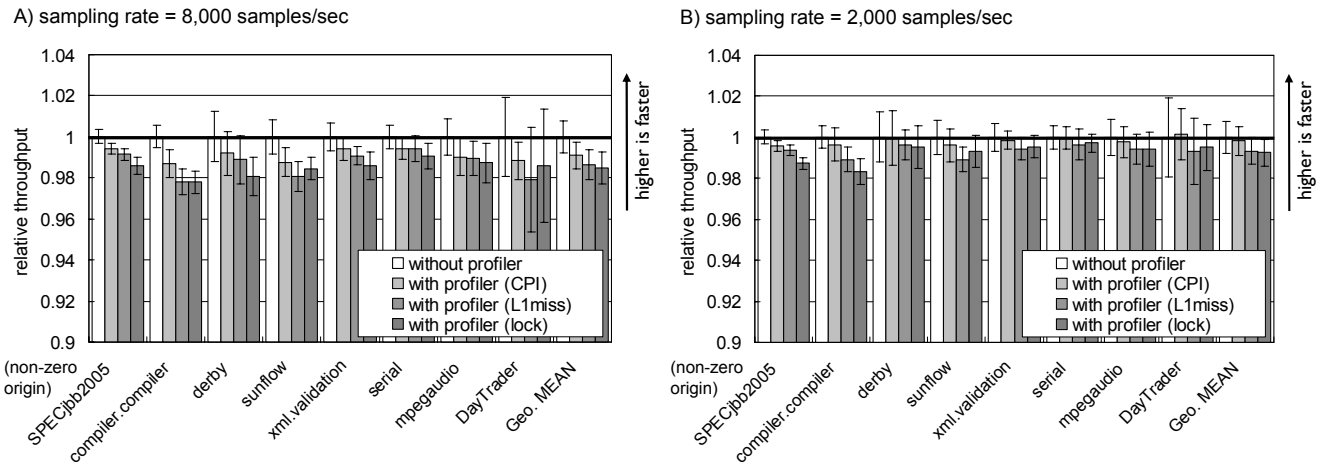


Figure 9. Average throughputs without profiler and with three profiler configurations. The error bars show 95% confidence intervals.

samples/sec for each event. The figure shows the overhead of three different configurations for the profiler. In the first configuration (labeled *CPI*), the profiler counts the active CPU cycles (PM_RUN_CYC event) and the instructions executed (PM_INST_CMPL event) simultaneously and calculates the CPI (cycles per instruction) for each method. The profiler generated a CPI value for each method every 30 sec. In the second configuration (labeled *L1miss*), the profiler counts the L1 cache misses (PM_MRK_ LD_MISS_L1 event) and the instructions executed (PM_MRK_INST_FIN event). This configuration involved the expensive data address to Java object translation for the cache miss events, which requires stopping the GC threads during the translation and consumed more CPU time in the HPM profiler. Then the profiler generated a summary of the cache miss ratios and a sorted list of the objects and fields that caused the cache misses for each method. The additional statistics led to more overhead compared to the first configuration. The overhead to generate an object creation profile is almost the same as the overhead of the second configuration because the HPM profiler thread executes similar operations. In the third configuration (labeled *lock*), we inserted ProbeNOPs to monitor the lock activities as described in Section 5 and the profiler counts the ProbeNOP instructions executed and all of the instructions executed in the same interval. In this configuration, the overhead due to the HPM profiler thread was similar to the second configuration, confirming that the inserted ProbeNOPs (`vperm` instructions) do not significantly affect the performance. For SPECjbb2005 and SPECjvm2008, we ran the performance measurements 24 times with four iterations each and averaged the best score of each run. For DayTrader, we ran and averaged four measurements. We also show a 95% confidence interval for each data in the figure. Examples for each type of the profiles appears in appendix.

From Figure 9, the overhead of the profiler is generally small, within 2.2%, even for the second and the third configurations that involve the costly data address to Java object translations. The difference between the first configuration and the other configurations mostly comes from the overhead of the translation. For the third configuration, the inserted ProbeNOPs did not affect the performance of the tested programs. The increase in the code size of JIT-compiled code due to the ProbeNOP, one instruction per lock acquisition operation, was fairly small and smaller than the fluctuations due to the dynamic nature of the JIT compiler. In some programs, the overhead of the third configuration was slightly smaller than the second configuration because the data address of each HPM sample, as captured by the ProbeNOP technique, always pointed at the Java object header, and the overhead of the data address to Java object translation was smaller than the translation in the second configuration.

The sampling frequencies in the configurations were high enough to generate accurate profiles in most cases. Actually much lower sampling frequencies, which would impose much smaller overheads, would provide sufficiently accurate profiles for most purposes. For example, Schneider *et al.* [22] proposed 200 samples/sec as a reasonable choice on a single-core Pentium4 processor for their optimizations, while we used 8,000 samples/sec on the two POWER6 cores. We used a higher sampling frequency here because some of our advanced techniques, such as creating the object creation profiles, require a large number of samples compared to simple cache miss profiles.

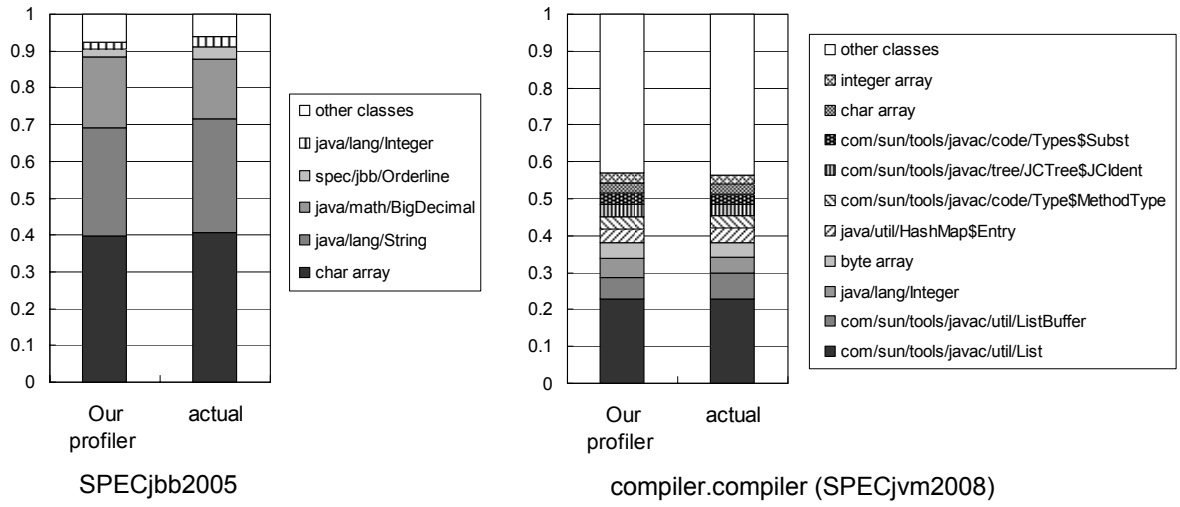


Figure 10. Breakdown of created objects by numbers of objects, as obtained from the HPM.

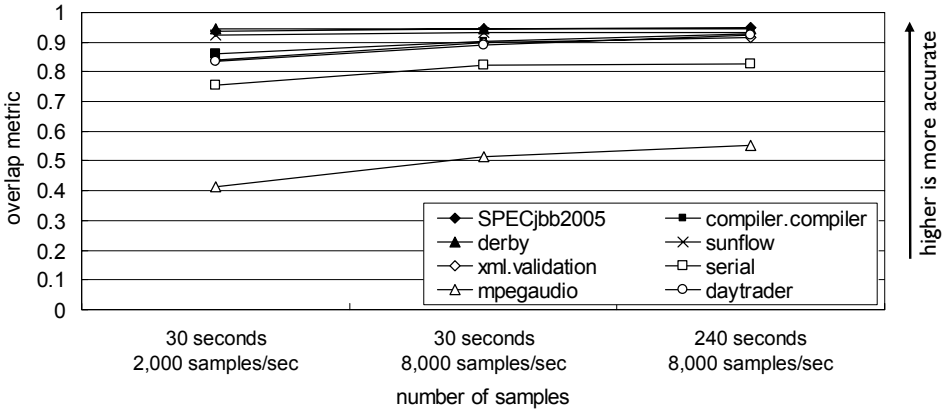


Figure 11. Accuracy of the object creation profiles as measured by the overlap metric.

7.2 Accuracy of the object creation profiling

To confirm the accuracy of our object creation profiling technique, Figure 10 compares the breakdowns of the numbers of objects created in the Java heap as measured by our profiler for two benchmarks, SPECjbb2005 and compiler.compiler, to the numbers counted in the garbage collector. The figure shows that the object creation profiles from our profiler were consistent with those generated by the more accurate method.

To quantitatively evaluate the accuracy of our technique, we use the *overlap* metric [15] as calculated by the following formula:

$$overlap(profile1, profile2) = \sum_{class \in profiles} min(ratio(class)\ in\ profile1, ratio(class)\ in\ profile2)$$

where

$$ratio(class) = number\_of\_samples(class) / total\_number\_of\_samples$$

The overlap metric shows how large portion of the samples are included in both profiles. This metric becomes 100% for a pair of identical profiles. Figure 11 show the overlap metric of the profiles generated by our profiler and the profiles generated by counting objects in the garbage collector. We show the average overlap metric calculated from samples gathered in 30 seconds of measurements with the sampling rates of 8,000 samples/sec and 2,000 samples/sec. We also show the overlap metric calculated from 4 minutes measurements with the 8,000 samples/sec sampling rate. The accuracy was limited for mpegaudio because the number of newly created objects was so small

Table 1. Statistics for the calling context trees for each benchmark. Our technique cannot uniquely identify the locations of the children of those nodes in a CCT which have multiple callers with the same frame pointer values, such as Method D in Figure 7.

benchmark	number of nodes in CCT			number of methods in CCT	average number of nodes per method	ratio of L1 cache miss events	
	total	having multiple callers with the same call stack depth				whose callers were uniquely identified for at least one level	whose callers were uniquely identified for at least three levels
SPECjbb2005	346	10	(2.9%)	122	2.8	99.5%	99.2%
compiler.compiler	78,264	20,184	(25.8%)	1,477	53.0	71.0%	52.5%
derby	1,092	8	(0.7%)	432	2.5	98.9%	98.9%
sunflow	1,039	57	(5.5%)	104	10.0	97.6%	94.6%
xml.validation	1,745	59	(3.4%)	442	3.9	97.9%	95.6%
serial	1,813	36	(2.0%)	218	8.3	90.8%	80.2%
mpegaudio	54	0	(0.0%)	50	1.1	99.2%	99.2%
DayTrader	37,616	2,962	(7.9%)	2,983	12.6	91.9%	86.4%

✓ The ratios in parenthesis show the ratio to the total number of nodes.

that it was not possible to sample a lot of events required to generate an accurate profile.

These results show our simple technique to derive the object creation profile from the store instruction profile is an effective and accurate way to capture the object creation behavior with very low overhead. Note that, though Figure 10 shows only the breakdown of object classes, our profiler got information on the allocation sites for every sampled object at the same time as shown in an example in appendix.

7.3 Accuracy of the context-sensitive profiling

To evaluate how accurately our CallerChaining technique can generate the calling context tree (CCT) based on the call stack depths, we compare the number of nodes in the generated CCT and the number of nodes that have multiple callers with the same call stack depth for each benchmark in the first two columns of Table 1. As described in Section 5, our technique cannot uniquely identify the locations of the children of such nodes in a CCT and we distribute the HPM events based on the edge hotness in the CCT for the calling contexts that include such nodes. From the table, at most 7.9% of the nodes were problematic for the benchmarks other than compiler.compiler. For compiler.compiler, more than 25% of the nodes have multiple callers with the same call stack depth, and thus our technique has limited accuracy for the calling context in each sample. The compiler.compiler has a very complicated CCT consisting of the largest number of nodes among the benchmarks even though the number of methods involved in the CCT is smaller compared to DayTrader, as shown in the next two columns of Table 1. This means that each method was called from a variety of calling contexts, which led to frequent conflicts of the call stack depths in the different calling contexts. Note that we enabled the HPM sampling for five minutes to generate the CCT. This measurement time can be much shorter if the processor supports precise HPM interrupts so we could use the ProbeNOP technique for this purpose.

To evaluate how accurately our technique can identify the calling context of each HPM event, we show the number of L1 cache miss events for which we can track their calling contexts accurately against the total cache miss events occurring in the JIT-compiled code in the last two columns of Table 1. For the programs other than compiler.compiler, we can identify the call sites for at least one level in more than 90% of the cache miss events, and for at least three levels in more than 80% of the cache miss events without ambiguity. Jones and Ryder [7] reported that only one level of the calling context improved the prediction of object lifetime though predictions based on allocation site alone were not accurate enough. Because of its low overhead and good accuracy, we believe that our CallerChaining is an attractive way to provide valuable information to use with adaptive optimizations. For mpegaudio, we failed to identify the calling contexts for 0.8% of the cache miss events, even though the benchmark did not suffer from any call stack depth conflicts. These failures were caused by the imprecise HPM interrupts that returned mismatched pairs of an instruction address and a call stack depth. For benchmarks other than compiler.compiler, the ratios of such failures were 0.1% to 1.3%, while it was 4.4% for compiler.compiler.

Figure 12 shows the L1 cache miss profiles mapped for the calling contexts for SPECjbb2005. Each line of the profile shows a method signature, the call stack depth shown in the offset from the stack base of the thread, the number of samples in the method (labeled *s:*), the ratio of the total samples, the number and ratio for all descendants and itself (labeled *d:*). We only show those nodes whose

```
+-spec/jbb/TransactionManager.go()V:0x160 s:0(0.0%) d:221721(91.6%)
  +-spec/jbb/TransactionManager.goManual(ILspec/jbb/TimerData;)J:0x210 s:2953(1.2%) d:221376(91.5%)
    +-spec/jbb/CustomerReportTransaction.process()Z:0x2d0 s:74758(30.9%) d:75946(31.4%)
      +-spec/jbb/Company.getCustomerByLastName(SBLjava/lang/String;)Lspec/jbb/Customer;:0x340 s:454(0.2%) d:1188(0.5%)
    +-spec/jbb/CustomerReportTransaction.processTransactionLog()V:0x2d8 s:1581(0.7%) d:6002(2.5%)
      +-spec/jbb/infra/Util/XMLTransactionLog.clear()V:0x370 s:669(0.3%) d:985(0.4%)
      +-spec/jbb/infra/Util/XMLTransactionLog.populateXML(Lspec/jbb/infra/Util/TransactionLogBuffer;)V:0x360 s:1432(0.6%) d:1533(0.6%)
    +-spec/jbb/DeliveryTransaction.process()Z:0x278 s:122(0.1%) d:77080(31.9%)
      +-spec/jbb/DeliveryTransaction.preprocess()Z:0x3b0 s:74438(30.8%) d:76904(31.8%)
        +-spec/jbb/District.removeOldNewOrders(I)V:0x418 s:780(0.3%) d:883(0.4%)
        +-spec/jbb/District.removeOldOrders(I)V:0x420 s:691(0.3%) d:990(0.4%)
    +-spec/jbb/NewOrderTransaction.init()V:0x248 s:777(0.3%) d:860(0.4%)
    +-spec/jbb/NewOrderTransaction.process()Z:0x330 s:1486(0.6%) d:20621(8.5%)
      +-java/util/TreeMap.rbInsert(Ljava/lang/Object;)Ljava/util/TreeMap$Entry;:0x360 s:2092(0.9%) d:2821(1.2%)
      +-spec/jbb/Company.getCustomer(JZ)Lspec/jbb/Customer;:0x388 s:1004(0.4%) d:1004(0.4%)
      +-spec/jbb/Order.processLines(Lspec/jbb/Warehouse;SZ)Z:0x4b8 s:7935(3.3%) d:15310(6.3%)
        +-spec/jbb/Orderline.<init>(Lspec/jbb/Company;IBSSSZ)V:0x500 s:1393(0.6%) d:1405(0.6%)
        +-spec/jbb/Orderline.process(Lspec/jbb/Item;Lspec/jbb/Stock;)V:0x5d8 s:3162(1.3%) d:5232(2.2%)
          +-java/math/BigDecimal.multiply(Ljava/math/BigDecimal;)Ljava/math/BigDecimal;:0x640 s:1513(0.6%) d:1513(0.6%)
    +-spec/jbb/NewOrderTransaction.processTransactionLog()V:0x2c8 s:3754(1.6%) d:13269(5.5%)
      +-spec/jbb/infra/Util/TransactionLogBuffer.putDollars(Ljava/math/BigDecimal;III)V:0x318 s:504(0.2%) d:1040(0.4%)
      +-spec/jbb/infra/Util/TransactionLogBuffer.putText(Ljava/lang/String;III)V:0x308 s:1793(0.7%) d:1793(0.7%)
      +-spec/jbb/infra/Util/XMLTransactionLog.clear()V:0x360 s:1236(0.5%) d:1630(0.7%)
      +-spec/jbb/infra/Util/XMLTransactionLog.populateXML(Lspec/jbb/infra/Util/TransactionLogBuffer;)V:0x350 s:3115(1.3%) d:3482(1.4%)
    +-spec/jbb/OrderStatusTransaction.processTransactionLog()V:0x268 s:89(0.0%) d:935(0.4%)
    +-spec/jbb/PaymentTransaction.init()V:0x250 s:696(0.3%) d:1078(0.4%)
    +-spec/jbb/PaymentTransaction.process()Z:0x2c0 s:1058(0.4%) d:7627(3.2%)
      +-spec/jbb/Company.getCustomerByLastName(SBLjava/lang/String;)Lspec/jbb/Customer;:0x330 s:840(0.3%) d:2227(0.9%)
        +-spec/jbb/TreeMapDataStorage.getMedianValue(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;:0x390 s:28(0.0%) d:1238(0.5%)
          +-java/util/TreeMap$AbstractSubMapIterator.<init>(Ljava/util/TreeMap$NavigableSubMap;)V:0x3b0 s:3(0.0%) d:1207(0.5%)
            +-java/util/TreeMap$AscendingSubMapIterator.getBoundaryNode()Ljava/util/TreeMap$Entry;:0x3c8 s:0(0.0%) d:1120(0.5%)
              +-java/util/TreeMap$NavigableSubMap.smallerEntry(Ljava/lang/Object;)Ljava/util/TreeMap$Entry;:0x3f0 s:1(0.0%) d:1120(0.5%)
                +-java/util/TreeMap$NavigableSubMap.findLowerEntry(Ljava/lang/Object;)Ljava/util/TreeMap$Entry;:0x428 s:2(0.0%) d:1119(0.5%)
                  +-java/util/TreeMap$NavigableSubMap.findEndNode()Ljava/util/TreeMap$Entry;:0x498 s:1068(0.4%) d:1068(0.4%)
      +-spec/jbb/Warehouse.removeOldestHistory()V:0x300 s:49(0.0%) d:1121(0.5%)
        +-java/util/TreeMap.find(Ljava/lang/Object;)Ljava/util/TreeMap$Entry;:0x330 s:654(0.3%) d:814(0.3%)
      +-spec/jbb/Warehouse.updateHistory(Lspec/jbb/History;)V:0x300 s:321(0.1%) d:2433(1.0%)
        +-java/util/TreeMap.put(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;:0x320 s:0(0.0%) d:2112(0.9%)
          +-java/util/TreeMap.rbInsert(Ljava/lang/Object;)Ljava/util/TreeMap$Entry;:0x350 s:1568(0.6%) d:2112(0.9%)
    +-spec/jbb/PaymentTransaction.processTransactionLog()V:0x2e8 s:2242(0.9%) d:7964(3.3%)
      +-spec/jbb/infra/Util/XMLTransactionLog.clear()V:0x380 s:1045(0.4%) d:1303(0.5%)
      +-spec/jbb/infra/Util/XMLTransactionLog.populateXML(Lspec/jbb/infra/Util/TransactionLogBuffer;)V:0x370 s:1959(0.8%) d:2126(0.9%)
    +-spec/jbb/StockLevelTransaction.process()Z:0x2b8 s:4127(1.7%) d:4127(1.7%)
```

Figure 12. An example of HPM events mapped to the calling context trees (L1 data cache miss profile for SPECjbb2005). We show only those nodes whose descendants generated at least 0.3% of the total events.

descendants generated at least 0.3% of the total cache misses. Also, we do not show inlined methods as separate nodes in the CCT. In the CCT, the root of all of the calling contexts is the `TransactionManager.go` method. However this method is not the true root in the program and it has a caller. However the Java threads did not exit and reenter the `TransactionManager.go` method during the measurement period and so the profiler was not able to identify the caller of this method. By combing the results of a few stack-walk-based stack traces with our profiler, the results give a much better picture of the calling context tree.

Note that these tree-structured profiles were constructed from the outputs of the HPM profiler in the post-processing phase, and the profiler did not generate the trees at runtime. This post-processing took only a few seconds using a non-optimized Perl script. An entire tree-structured profile is not required for the adaptive optimizations, such as when identifying the callers of selected nodes that cause many events.

While investigating various workloads with our profiler, we often observed that collection classes such as hashmap generated many cache misses. In such cases, more valuable than the program locations was information about the calling context of the hashmap functions. The information about the callers of the hashmap functions is not satisfactory because programmers often wrap a hashmap with their own class and all callers of the hashmap functions become the same. Our technique can provide enough information to identify the callers of such wrapped hashmaps, even though it cannot uniquely distinguish the full calling context information.

When more exact calling context information is required, we could use the PCC [17], which maintains a value to identify the current calling context at each method entry and exit. To read the PCC value from the interrupt handler, we would need to put the value in a location that the handler can easily find, such as a slot in the stack with a constant offset from the frame pointer.

8. Summary

In this paper, we described our sampling-based profiler that exploits a hardware performance monitor (HPM) available in the processor to collect information on running Java applications for use by the Java VM. Our profiler provides two novel features: Java-level events profiling and lightweight context-sensitive event profiling. For the former feature, we showed that the HPM can be used to profile high-level events in language runtime systems by correlating them with hardware events. We showed how our profiler generates object creation profiles and lock activity profiles with low overhead. For lock activity, we

presented a lightweight context detection technique called *CallerChaining*, which detects the calling context for events. Our proposed techniques enable both programmers and runtime systems to get valuable information from the HPM to understand and optimize running programs without adding major overhead.

Based on the insights in this paper, we hope that future processors support the precise HPM interrupts that allow the profiling tools to obtain the detailed processor states at the time of HPM events. Also NOP instructions whose execution can be counted by the HPM can provide more freedom for programmers to exploit the HPM sampling facility and thus offers another interesting extended use of the HPM.

Acknowledgments

We are grateful to the anonymous reviewers for their valuable comments and suggestions. We thank Mauricio Serrano and Peter F. Sweeney for their useful feedback and detailed comments on earlier drafts.

Appendix

This appendix shows examples of various profiles to show what kind of information is included in the profiles.

(a) An example of CPI profile (for SPECjbb2005).

```
RUN CYCLE        INSTRUCTION     RATIO      METHOD
==============  ============== =======  ================================================
  %    samples    %    samples
32.2%  (77637)   9.8%  (23753) 1131.31% spec/jbb/DeliveryTransaction.preprocess()Z
13.9%  (33441)   9.4%  (22828) 507.04%  spec/jbb/CustomerReportTransaction.process()Z
 4.3%  (10255)   9.6%  (23263) 152.58%  spec/jbb/infra/Util/XMLTransactionLog.populateXML(
 2.5%   (6049)   1.1%   (2572) 814.04%  spec/jbb/StockLevelTransaction.process()Z
 2.4%   (5677)   3.5%   (8456) 232.37%  spec/jbb/infra/Util/TransactionLogBuffer.putText(L
...
```

(b) An example of cache miss profile (L1 data cache miss profile for SPECjbb2005).

```
L1D$ MISS MRKD  INSTRUCTION     RATIO    METHOD
==============  ============== =======  ================================================
  %    samples    %    samples
30.6%  (74107)  11.2%  (15992)   2.93%  spec/jbb/CustomerReportTransaction.process()Z
30.1%  (72905)  11.4%  (15178)   2.83%  spec/jbb/DeliveryTransaction.preprocess()Z
 4.7%  (11480)   2.8%   (4371)   1.84%  spec/jbb/Order.processLines(Lspec/jbb/Warehouse;SZ
 3.0%   (7251)   8.7%  (15535)   0.37%  spec/jbb/infra/Util/XMLTransactionLog.populateXML(
...

spec/jbb/DeliveryTransaction.preprocess()Z  30.1%  (72905)
L1D$ MISS MRKD  LOCATION  OFFSET  CLASS
==============  ========  ======  ================================
  %    samples
 4.0%   (9720)  tenure       0    spec/jbb/Stock
 2.5%   (6108)  tenure      32    spec/jbb/Stock
 2.0%   (4847)  nursery       8    spec/jbb/Orderline
 1.9%   (4552)  nursery       0    java/math/BigDecimal
...
```

(c) An example of lock activity profile (for DayTrader).

```
MONITOR ENTER   INSTRUCTION     RATIO    METHOD
==============  ============== =======  ================================================
  %    samples    %    samples
 9.0%  (22629)   0.5%   (1258)   1.29%  org/apache/openjpa/jdbc/sql/SQLBuffer.append(Ljava
 4.5%  (11287)   0.5%   (1143)   0.71%  org/apache/openjpa/jdbc/meta/strats/HandlerFieldSt
 4.3%  (10778)   0.6%   (1399)   0.55%  org/apache/openjpa/jdbc/sql/SelectImpl.getTableInd
 2.9%   (7397)   0.3%    (606)   0.88%  org/apache/openjpa/jdbc/sql/SelectImpl$SelectResul
...

org/apache/openjpa/jdbc/sql/SQLBuffer.append(Ljava  9.0%  (22629)
MONITOR ENTER   LOCATION  CLASS
==============  ========  ================================
  %    samples
 9.0%  (22625)  nursery   java/lang/StringBuffer
...
```

```
SPIN LOOP       INSTRUCTION    RATIO    METHOD
==============  ============== =======  ================================================
  %    samples    %    samples
36.9%     (75)   0.0%     (90)   0.06%  com/ibm/ejs/ras/Tr.register(Ljava/lang/Class;Ljava
14.3%     (29)   0.1%    (187)   0.01%  org/apache/openjpa/meta/MetaDataRepository.getMeta
 6.4%     (13)   0.2%    (455)   0.00%  com/ibm/io/async/ResultHandler.runEventProcessingL
 3.0%      (6)   0.0%     (58)   0.01%  com/ibm/ws/persistence/EntityManagerImpl.createNam
...

com/ibm/ejs/ras/Tr.register(Ljava/lang/Class;Ljava  36.9%  (75)
SPIN LOOP       LOCATION  CLASS
==============  ========  ==============================
  %    samples
36.9%     (75)  tenure    com/ibm/ws/bootstrap/WsLogManager
...


HELPER ENTER    INSTRUCTION    RATIO    METHOD
==============  ============== =======  ================================================
  %    samples    %    samples
44.9%     (48)   0.1%    (185)   0.01%  org/apache/openjpa/meta/MetaDataRepository.getMeta
13.1%     (14)   0.3%    (650)   0.00%  java/util/Hashtable.put(Ljava/lang/Object;Ljava/la
 6.5%      (7)   0.0%      (5)   0.08%  com/ibm/ws/util/BoundedBuffer.waitGet_(J)V
 5.6%      (6)   0.2%    (377)   0.00%  java/util/Hashtable.get(Ljava/lang/Object;)Ljava/l
...

org/apache/openjpa/meta/MetaDataRepository.getMeta  44.9%  (48)
HELPER ENTER    LOCATION  CLASS
==============  ========  ==============================
  %    samples
44.9%     (48)  tenure    org/apache/openjpa/jdbc/meta/MappingRepository
...
```

(d) An example of object creation profile (for SPECjbb2005).

```
OBJ. CREATION   LOCATION  CLASS
==============  ========  ==============================
  %    samples
36.4%  (40780)  nursery   [C
27.5%  (30820)  nursery   java/lang/String
16.7%  (18726)  nursery   java/math/BigDecimal
 4.1%   (4595)  stack     java/lang/Integer
 1.8%   (2013)  nursery   java/lang/Integer
...

spec/jbb/infra/Util/XMLTransactionLog.populateXML(  31.9%  (35628)
OBJ. CREATION   LOCATION  CLASS
==============  ========  ==============================
  %    samples
15.8%  (17695)  nursery   [C
14.9%  (16734)  nursery   java/lang/String
 0.9%   (1014)  stack     java/lang/String
...
```

References

[1] G. Ammons, T. Ball, and J. R. Larus. “Exploiting hardware performance counters with flow and context sensitive profiling”. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 85–96, 1997.

[2] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. “Java just-in-time compiler and virtual machine improvements for server and middleware applications”. In *Proceedings of the USENIX Virtual Machine Research and Technology Symposium*, pp. 151–162, 2004.

[3] 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.

[4] 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.

[5] T. Ogasawara, H. Komatsu, and T. Nakatani. “To-lock: Removing lock overhead using the owners’ temporal locality”. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, pp. 255-266, 2004.

[6] K. Kawachiya, A. Koseki, and T. Onodera. “Lock reservation: Java locks can mostly do without atomic operations”. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 292–310, 2002.

[7] R. Jones and C. Ryder. “A Study of Java Object Demographics”. In *Proceedings of the ACM International Symposium on Memory Management*, pp. 121-130, 2008.

[8] M. L. Seidl and B. G. Zorn. “Segregating heap objects by reference behavior and lifetime”. In *Proceedings of the eighth Architectural Support for Programming Languages and Operating Systems*, pp 12-23, 1998.

[9] F. E. Levine. “A programmer's view of performance monitoring in the PowerPC microprocessor”. IBM Journal of Research and Development, Vol 41 (3), pp. 345-356, 1997.

[10] OProfile - A System Profiler for Linux. http://oprofile.sourceforge.net/news/

[11] Intel Corp. *IA-32 Intel Architecture Software Developer's Manual*.

[12] JVM Tool Interface version 1.0. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html

[13] 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.

[14] M. Hauswirth and T. M. Chilimbi. “Low-overhead memory leak detection using adaptive statistical profiling”, in Proceedings of the international conference on Architectural support for programming languages and operating systems table of contents, pp. 156–164, 2004.

[15] M. Arnold, and B. G. Ryder. “A framework for reducing the cost of instrumented code”. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 168–179, 2001.

[16] J. M. Spivey. “Fast, Accurate Call Graph Profiling”. *Software: Practice and Experience*, Vol. 34 (3), pp. 249–264, 2004.

[17] M. D. Bond, and K. S. McKinley. “Probabilistic Calling Context”. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications*, pp. 97–112, 2007.

[18] X. Zhuang, M. J. Serrano, H. W. Cain, and J Choi. “Accurate, efficient, and adaptive calling context profiling”. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 263–271, 2006.

[19] M. Arnold and P. F. Sweeney. “Approximating the calling context tree via sampling”. *IBM Research Report*, 2000.

[20] J. Whaley. “A portable sampling-based profiler for java virtualmachines”. In *Proceedings of ACM Java Grande*, pp. 78–87, 2000.

[21] T. Mytkowicz, D. Coughlin, and A. Diwan. “Inferred Call Path Profiling”, In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 175–189, 2009.

[22] 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.

[23] 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.

[24] 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.

[25] J. Dolby. “Automatic Inline Allocation of Objects”, In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp 7–17, 1997.

[26] Power.org, Power Instruction Set Architecture Version 2.05. http://www.power.org/resources/reading/PowerISA_V2.05.pdf

[27] N. Grcevski, “Effective method for Java Lock Reservation for Java Virtual Machines that Have Cooperative Multithreading” *6th Workshop on Compiler-Driven Performance,* 2007.

[28] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. “Thin Locks: Featherweight Synchronization for Java”. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 258–268, 1998.

[29] T. Onodera and K. Kawachiya. “A study of locking objects with bimodal fields”. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications*, pp. 223–237, 1999.

[30] Performance Inspector, http://perfinsp.sourceforge.net/

[31] S. L. Graham, P. B. Kessler, and M K. McKusick. “An execution profiler for modular programs”. *Software: Practice and Experience*, Vol. 13 (8), pp. 671–685, 1983.

[32] Standard Performance Evaluation Corporation. SPECjbb2005. http://www.spec.org/jbb2005/

[33] Standard Performance Evaluation Corporation. SPECjvm2008. http://www.spec.org/jvm2008/

[34] The Apache Software Foundation. DayTrader. http://cwiki.apache.org/GMOxDOC20/daytrader.html

[35] IBM Corporation. WebSphere Application Server. http://www-01.ibm.com/software/webservers/appserv/was/