A Portable Sampling-Based Profiler for Java Virtual Machines

John Whaley
IBM Tokyo Research Laboratory
Network Computing Platform
1623-14 Shimotsuruma
Yamato-shi, Kanagawa-ken 242-8502 Japan
jwhaley@alum.mit.edu

ABSTRACT

This paper describes a portable and efficient sampling-based online measurement system for production-level Java virtual machines. This system is designed to provide continuous real time system performance measurements to a dynamic compiler, which can use these measurements to target frequently executed and time-consuming code for optimization and to make more informed optimization decisions. Because the system has very low overhead, it can be run continuously, providing a feedback mechanism to the dynamic compiler. This system utilizes a novel data structure, the partial calling context tree (PCCT), which allows the efficient encoding of approximate context-sensitive profile information. The PCCT is organized such that both incremental updates and extracting the important information are efficient operations.

This online measurement system has been implemented in a cross-platform industry-leading Java Just-In-Time compiler. We present detailed performance results on a variety of platforms that show that the system is not only efficient enough to be used continuously in a production environment (2-4% slowdown for most applications) but is also surprisingly accurate in the data that it collects (typically ≥90% accurate).

1. INTRODUCTION

A Java Just-In-Time compiler must walk a delicate line. Because compilation occurs at run time, it must be very selective in what it decides to compile and how it decides to compile it. A dynamic compiler should only spend extra time analyzing and compiling a piece of code if there is a reasonable chance that the extra time spent on compilation will be made up in run time. However, if the dynamic compiler is too timid in its decision making, it may miss good opportunities for optimization and lead to overall slow performance. This tradeoff between compilation cost and run time benefit is the central issue in dynamic compilers.

Java 2000 San Francisco CA USA

A useful metric in dynamic compilation systems is the *compilation cost versus run time benefit* equation. See Figure 1 below.

$$\Delta T_{overall} \approx T_{compile} - N * (T_{unopt} - T_{opt})$$

Figure 1: Compilation cost versus run time benefit equation.

In this equation, $\Delta T_{overall}$ refers to the change in total run time, $T_{compile}$ refers to the amount of time it takes to compile and maintain the new version, N refers to the number of times that the new version is executed, T_{unopt} refers to the average execution time of the code before compilation, and T_{opt} refers to the average execution time of the newly compiled version. When $\Delta T_{overall}$ is negative, the overall execution time of the program is reduced and therefore the decision to compile was a good one. However, when $\Delta T_{overall}$ is positive, overall time is increased, and therefore it would have been better to not have attempted the compilation in the first place.

Thus, a dynamic compiler should attempt to compile only those methods for which it is possible to considerably reduce their absolute total run time. Therefore, it should focus on methods whose total run time is significant, *i.e. long-running* (T_{unopt} is large) or *frequently-executed* (N is large) methods, because even a slight speedup in those methods can lead to a substantial speedup overall.

A dynamic compiler can use online profile information to help guide decisions about what to compile and how to compile it. Using such information, an aggressive dynamic compiler even has the ability to beat the best static compilers because information about the actual run time performance of the system is available, and it can specialize the code and data to suit the situation [15, 10].

However, the tractability of using dynamic profile information depends greatly on how efficiently the profile data can be collected and organized. When using dynamic profile information, the equation from Figure 1 above turns into that of Figure 2:

$$\Delta T_{overall} \approx (T_{measure} + T_{compile}) - N * (T_{unopt} - T_{opt})$$

Figure 2: *Measurement and compilation cost versus run time benefit* equation.

where $T_{measure}$ refers to the time spent collecting, organizing, and analyzing the profile data. If dynamic profile information is too expensive to collect, then a static compilation policy heuristic might be more appropriate. Therefore, for an online profile-directed dynamic compiler to be feasible, it must be possible to collect, organize, and analyze online profile information efficiently.

This paper describes a portable and efficient sampling-based online measurement system for production-level Java virtual machines. This system provides continuous real time system performance measurements to the dynamic compiler for use in making compilation policy decisions and for optimizing code to suit the actual run time program characteristics.

One rather unique feature of this online measurement system is that it is sampling-based, rather than instrumentation-based. Instrumentation-based profilers can give exact profile information on the number of times a method is executed or a branch is taken. However, they have a number of problems that make them ill-suited for online use in a Java virtual machine; including high overhead, difficulty in adjusting precision, unintended effects on target code, and problems in multithreaded contexts. By using a sampling-based profiler, we can avoid many of these problems at the cost of exactness of profile information. Because the profile information is simply used as inputs to compiler policy decision heuristics, approximate profile information is sufficient. Furthermore, sampling can be surprisingly accurate, as we show in Section 6. This paper also describes a data structure, the partial calling context tree (PCCT), which is used to efficiently encode approximate context-sensitive profile information.

1.1 Contributions

This paper makes the following contributions:

- **Data structures:** It presents the partial calling context tree (PCCT), a data structure for efficiently encoding approximate context-sensitive profile information. It also presents algorithms to update and traverse this data structure.

- **Profiling techniques:** It presents a profile sampling technique that estimates not only the time spent in a method, but also the context in which it occurred. Our technique is also able to distinguish between *new* and *old* stack frames and thereby discover frequently-executed edges without code instrumentation. It is able to correctly identify the complete calling context without traversing the entire stack.

- **Evaluation of the overhead and accuracy of sample profiling:** It presents a detailed performance evaluation of the overhead and accuracy of our sampling-based profiler on a variety of benchmarks and systems.

The remainder of the paper is organized as follows. Section 2 compares our system to prior and related systems. Section 3 gives an overview of our dynamic compilation system. Section 4 describes the sampling profiler. In Section 5, we describe the partial calling context tree (PCCT) data structure. Section 6 presents detailed performance results. We conclude in Section 7.

2. RELATED WORK

This section discusses prior profiling systems and other related work.

2.1 Profiling systems

Prior profiling systems can be roughly broken down into two categories: those which collect their data via instrumentation code, which are a majority of the systems, and those which collect their data via sampling, like our system. We will discuss each category in turn.

2.1.1 Instrumentation-based profiling systems

Almost all existing profiling systems are *instrumentation-based*, that is, they insert *instrumentation code* into the program [1, 11, 5, 2]. When the instrumentation code executes, it records profile data, for example, by incrementing a counter or recording a time.

A fundamental problem with instrumentation-based profiling is that its overhead and accuracy are determined by the dynamic run time program profile. Controlling the tradeoff between accuracy and efficiency is very difficult because the frequency with which the code executes is determined entirely by the dynamic profile of the program. Furthermore, it may not be feasible to transform the code to collect profile data at the desired rate. For example, placing instrumentation code inside of a tight loop may cause it to execute at far too high of a frequency, but the alternative, breaking out the iterations of the loop, destroys the loop structure. Adjusting the profile frequency requires recompiling or rewriting the code, a process that is time-consuming and complicated by multithreading. On the other hand, in a sampling-based profiler the tradeoff can be controlled easily and precisely by varying the frequency of the sampling and the data collected on each sample. The ability to precisely control the tradeoff between accuracy and efficiency is essential to an adaptive dynamic compiler.

Another problem with instrumentation-based profiling is that it is too heavyweight for general use. For example, a well-optimized system that collects context-sensitive profile information has an overhead of about 70%, even though significant static analysis is performed on the entire program to minimize the profiling overhead; a luxury that is not possible for a dynamic compiler [1]. Also, they are often less precise, recording at the method or basic block level, while a sampling profiler can tell precisely which instructions are taking the most time. Performing timing measurements is also problematic because correctly factoring out the run time overhead of the instrumentation code is difficult or impossible [16]. There are also issues in using instrumentation to update the profile data structure correctly in the context of multiple threads — either synchronization or thread-local copies must be used. This makes many operations on the data structure more difficult, *e.g.* finding the most-frequently-executed methods.

Arnold and Sweeney described a unique hybrid profile technique for building approximate calling context information [2]. They instrument all method entries to increment and check a global counter. When the counter overflows, they call a routine to sample the current thread's stack. This has lower overhead than most other instrumentation techniques and it also allows some amount of control over the rate at which profiling occurs.

2.1.2 *Sampling-based profiling systems*

The DIGITAL Continuous Profiling Infrastructure, like our infrastructure, is a sampling-based profiling system [18]. It uses hardware performance counters to attain high-frequency sampling (~5200 samples/sec) with fairly low overhead (1-3% slowdown for most applications). They make use of other hardware performance counters to obtain very precise cache miss and branch prediction information at an instruction-level granularity.

We do not rely on hardware performance counters to perform the high-frequency sampling for a number of reasons. First, our system is designed to be portable to multiple operating systems and microprocessors, and using hardware performance counters requires on operating system modifications and special chip features. Second, because the sampling interrupts occur at the highest possible priority, the time spent in the interrupt routine must be very very short. Therefore, building a data structure such as the PCCT would be intractable; the DIGITAL system simply records the program counter. Third, on an SMP machine, using hardware performance counters on each processor requires synchronization or per-processor versions of the data structure. Because of these reasons, we use a different approach to generate the timer events necessary for sampling; see Section 6.4.

The HPROF profiler is a profiling tool for the Sun Java Development Kit (JDK) 1.2 [14]. It uses a variety of techniques to profile all aspects of the running virtual machine, including sample-based CPU time profiling. It supports full context sensitivity. However, their profiler is rather inefficient and does not scale due to the fact that they suspend all threads during sampling and they walk the entire stack on every sample. Even with a single thread, the overhead of their implementation while sampling every 1 *ms* is about 20%, far higher than ours.

Some other sampling profilers use hardware performance counters, but require special hardware modifications [8, 23]. Hall and Goldberg use process sampling to record some context-sensitive metrics for Unix processes [11]. Unlike our method, they must walk the entire stack on every sample, and the size of their data structure is unbounded because it stores a copy of the entire call stack with every sample. Other sampling profilers do not record calling context [4, 5] or record only a single level of context [9].

2.2 Context-sensitive data representations

The partial calling context tree was originally introduced in a slightly different form[1] as the primary data structure to organize profile data in a dynamic compilation system that automatically identifies and exploits opportunities for dynamic specialization [24]. This work extends the previous published work by more precisely defining the data structure and the algorithms used to traverse and update it, and by providing experimental results on its accuracy and performance.

The PCCT is based on an earlier representation called the calling context tree, or CCT [1]. The CCT differs from the PCCT in that the CCT assumes complete calling context information. Therefore, the algorithms to build the CCT involve instrumenting all procedure entries, exits, and calls in the program to update the tree structure. Because they use instrumentation, they suffer from all of the problems discussed in the previous section. The PCCT, on the other hand, supports incomplete information and can therefore operate in the context of periodic sampling and incomplete stack traversals. Another structural difference is in the treatment of recursive methods. To bound the size, the CCT combines nodes for recursive methods, losing some of the context — we keep the full context for recursive methods because we have different size restrictions. Because the PCCT does not have to contain complete call graph information, it is also much smaller; even if we keep information over an entire program run, the PCCT occupies on the order of kilobytes, while the CCT can take megabytes or more [1].

Arnold, *et al.* use a dynamic call graph with edge weights (DCG-E) to explore the effectiveness of various inlining decision heuristics [3]. Because they use a DCG as the basis of their graph and they use weighted edges, they in essence use a single level of calling context. Their construction algorithm uses post-mortem traces of method entries and exits.

Arnold and Sweeney also presented the approximate calling context graph, or ACCG [2]. The ACCG, like the PCCT, contains weighted edges; however only the last edge is incremented per sample, rather than all new edges as in the PCCT. Like the PCCT construction algorithm, their algorithm is able to avoid walking the entire stack by marking stack frames.

Jerding, Stasko, and Ball describe another way to encode dynamic call tree information, a compacted dynamic call tree [13]. Their construction algorithm uses complete traces of all calls made during the execution of the program to generate a dag structure in a bottom-up fashion. They use hash-consing and a pattern-recognition algorithm to combine subtrees, and therefore, like the PCCT, can have multiple nodes that correspond to a single context.

3. OVERVIEW OF DYNAMIC COMPILATION SYSTEM

First, we will give a brief overview of the dynamic compilation system in our Java virtual machine. See Figure 3. The Online Controller subsystem is the "brain" of the system. It makes decisions about which methods to compile and how to compile them, which it sends to the Just-In-Time compiler in the form of *compile commands*. It also controls the Online Measurement subsystem through *profile commands*. The Just-In-Time compiler outputs machine code, which Java threads execute. The Online Measurement subsystem consists of the profile data repository, stored in the form of a partial calling context tree (PCCT), and a sampling profiler, which periodically samples the running Java threads and updates the PCCT. Information in the PCCT is also used by the Just-In-Time compiler to make better optimization decisions, and as a feedback mechanism to the Online Controller for compilation and profile decisions.

This paper focuses on the Online Measurement subsystem, contained in the dashed box. Other publications cover the other subsystems in more detail [24, 20, 21]. The Online Measurement system consists of the sampling profiler, described in the next section, and the partial calling context tree, described in Section 5.

[1]The original version was called a "weighted calling context graph" (WCCG). It allowed back edges for recursive methods, and the construction algorithm was time-limited rather than depth-limited.

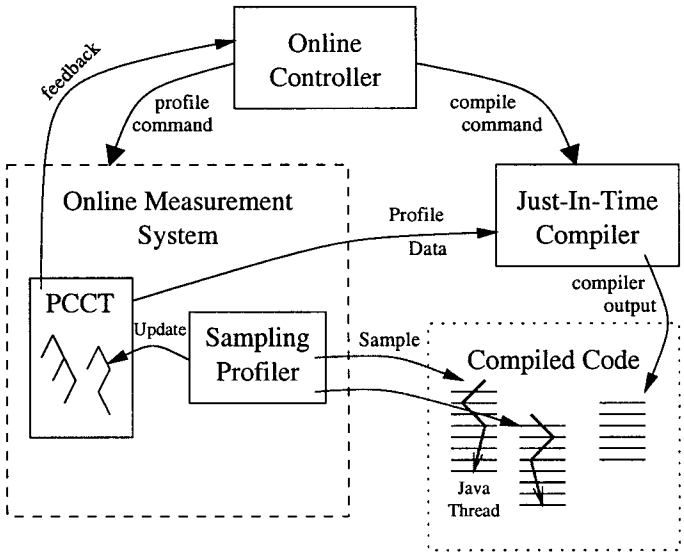

Figure 3: Overview of the dynamic compilation system in a Java virtual machine. This paper focuses on the Online Measurement System, in the dashed box.

4. SAMPLING PROFILER

One of the primary concerns of the Online Measurement subsystem is efficiency. If the performance gains garnered by having the profile data are outweighed by the cost of collecting the data, then collecting the profile data was pointless. Furthermore, the existence of the profiler should not adversely effect the peak performance of executable code. Because of these reasons, we opted to use a sampling profiler to gather the profile data.

The purpose of the sampling profiler is to periodically sample the running threads and update the data in the profile repository. Pseudocode for the sampling profiler can be found in Figure 4.

```
while (suspend/exit command not received)
   {
   if (timer tick received)
      {
      for each thread t in local thread queue
         {
         get t's program counter, stack pointer,
             and CPU time
         update profile data in repository
         }
(SMP only) signal profiler thread on next processor
      }
   }
```

Figure 4: Basic operation of sampling profiler.

The sampling profiler first waits for a time out period to elapse. We use a busy-wait timer as described in Section 6.4. After the timer expires, the sampling profiler traverses the local thread queue, and for each active Java thread, it retrieves its register state (program counter and stack pointer) and its CPU time. It then uses these to update the profile data in the repository (typically the PCCT, however other representations are possible — see Section 6.5.) On SMP systems that use multiple thread queues to perform M:N threading, it signals the profiler thread on the next processor to begin profiling.

Note that on systems that use multiple OS threads for the virtual machine, there is a lock associated with the active thread queue that insures thread safety when adding/removing threads from the queue. However, in the case of the sampling profiler, we found that acquiring and releasing the lock on every sample was too time consuming. Therefore, there is a possible race condition during sampling as threads are added or removed from the queue. By making a slight modification[2] to the thread queue code, we insured that the race condition could, at worst, cause some threads to be skipped or counted multiple times. Because the chance that a race condition occurs is very low and the profile data is inexact anyway, this is acceptable behavior.

4.1 Complications when using OS thread scheduling

Some of the virtual machines on our target platforms do not use their own threads package and therefore do not have explicit control over the thread scheduling. In these systems, each Java thread corresponds to an OS thread, and therefore the OS handles the thread scheduling. The fact that Java threads can run as they are being sampled complicates matters somewhat.[3]

[2]Whenever another OS thread can remove a thread from the current thread queue (on termination or work-stealing), it suspends the profiler before making the modification. Threads are rarely removed from the queue, so this is acceptable.

[3]This is not a problem in the M:N threading case because a sampling profiler only inspects threads that run on the same processor, and the profiler is suspended whenever threads

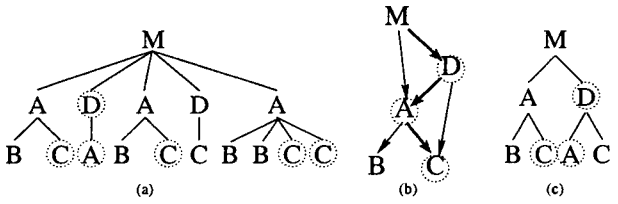

Figure 5: (a) A dynamic call tree, with hot nodes circled, (b) its corresponding dynamic call graph, and (c) its corresponding calling context tree.

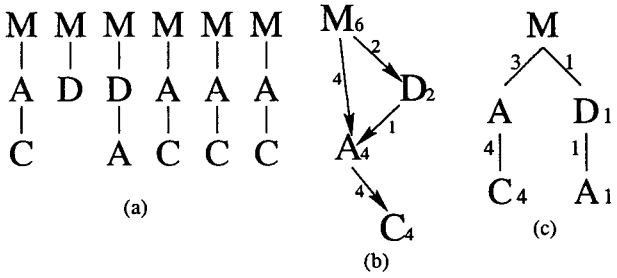

Figure 6: (a) A sequence of call stack samples, (b) its corresponding dynamic call graph with node and edge weights, and (c) its corresponding partial calling context tree.

When there is a one-to-one mapping between Java threads and OS threads we must use an system call to retrieve the thread register state. This call returns an error if the named thread is currently executing. On an SMP system, the failed calls are most likely from computationally-intensive threads — exactly those threads that we want to record. Furthermore, if OS threads are used, the stacks of the sampled threads may change as they are being inspected. In the case of the PCCT, the stack traversal algorithm writes a marker bit into the stack frames that it visits (see Section 5.2) which has disasterous effects if the thread is running.[4] Therefore, when using OS thread scheduling and a non-thread-safe method of profile data acquisition, the sampling profiler suspends each thread before getting its register state.

5. PARTIAL CALLING CONTEXT TREE

This section describes the data structure we use to store and organize the profile information — the partial calling context tree, or PCCT. The PCCT is a compact way to represent context-sensitive profile information. We give an example of the PCCT and compare it with other representations, and we present an algorithm for constructing the PCCT in a sampling profiler.

5.1 Overview and Example

The partial calling context tree is partial in two senses. First, it is partial in the sense that, because it is constructed via data from a sampling profiler, it only includes call paths that are active at the time of a sample. Second, the contexts that it stores may only be partially accurate, because of excessively deep stacks.

The PCCT, like its predecessor the CCT, offers an efficient intermediate point in the spectrum of run-time call graph representations. At one end of the spectrum, there is the dynamic call tree, or DCT. The DCT records the full context for all method invokations that occur during the life of the program; see Figure 5(a) for an example.[5] Each node in the tree represents a single method activation; each edge in the tree corresponds to a single method invokation. The sample figure has been annotated with some profile information — the circled nodes are the most time-consuming.

In a DCT, you can precisely distinguish the full calling context and even distinguish among successive calls to the same method. In our example, the DCT can tell us that the first call to D is time-consuming, while the second one isn't.

At the other end of the spectrum is the dynamic call graph (DCG), which very compactly represents the method calls that occur in the program. See Figure 5(b). In the DCG, there is a single node for each invoked method which represents all activations of that method. An edge exists between two nodes if there was ever a call between those two nodes. This is a compact representation, but is also very imprecise because information for methods from multiple calling contexts are conglomerated. For example, the information that the path $D \rightarrow A \rightarrow B$ and $D \rightarrow A \rightarrow C$ are never executed is lost.

We have also encoded edge weight information in the example DCG in Figure 5(b). Bold edges are *hot* edges — that is, they refer to method calls in which the target method was time-consuming. If the dynamic compiler were to use this DCG graph to make inlining decisions, it would most likely inline C into A into D into M, even though the path $D \rightarrow A \rightarrow C$ is never executed. Furthermore, it would probably miss the potentially-profitable C into A into M inlining decision. Defining edge weights as number of calls between methods doesn't do much better; the most likely unprofitable B into A inlining decision would have the highest edge weight.[6]

The calling context tree (CCT) is an intermediate point between the DCT and the DCG [1]. It represents all calling contexts that occur in the DCT, but combines nodes that have the same calling context. The CCT for our example is contained in Figure 5(c). Notice that it successfully represents the two different calling contexts associated with A ($M \rightarrow A$ and $M \rightarrow D \rightarrow A$) and with C ($M \rightarrow A \rightarrow C$ and $M \rightarrow D \rightarrow C$).

Up until this point, we have only mentioned exact dynamic call graph representations, such as those that could be constructed by an instrumentation-based profiler. Now, let's consider how context-sensitive data from a sampling profiler might be encoded. One straightforward representation is to record a copy of the call stack at every sample. This is similar to the representation used by Hall and Goldberg [11]. For example, see Figure 6(a). These are six hypothetical

migrate between processors.

[4]We have a thread-safe implementation of the PCCT stack traversal algorithm that does not mark stack frames and therefore does not distinguish between new and old activations of identical methods. It is not covered in this paper.

[5]For ease of comparison, this paper uses the same example as the CCT paper [1].

[6]B into A is unprofitable for a dynamic compiler because neither B, nor A when it calls B, are time consuming.

samples of the call stack in our program with DCT in Figure 5(a). The obvious problem with such a representation is space — this representation takes up space proportional to the depth of the stack times the number of samples. Another problem with this representation is the fact that the salient information is not readily available — to find the hot methods and good candidates for inlining requires traversing the data structure and keeping track of a large number of sums. Finally, information about whether calls are new since the last sample is not encoded, so for example, it is impossible to tell whether the last three samples came from seperate invokations of C or the same invokation of C. Thus, it cannot distinguish whether C is a time-consuming method or a frequently-invoked method.

Another possible representation of the profile data is to use a single node for the set of all invoked activations of a method, à la the DCG, and annotate nodes with profile counters. See Figure 6(b) for an example of this sample-constructed DCG. Notice that, like the DCG, the fact that the path $D \rightarrow A \rightarrow C$ is never executed is lost.

The partial calling context tree (PCCT) is an attempt to use stack trace samples to discover the important nodes and edges in the DCT, combining nodes that have the same context like the CCT, and keeping the important profile information readily available. An example of a PCCT is contained in Figure 6(c). This is the PCCT that would be constructed assuming that the samples occurred at the same times as the samples in Figure 6(a). (The numbers in the PCCT are weights and will be explained below, in Section 5.2.) Notice that like the DCG, the multiple calls from M to A and from A to D share a single edge, but that there are two distinct nodes for A — one from being called by M and the other from being called by D. Notice also that there is no node for B in the tree, because it never appeared in a sample and therefore doesn't appear in the tree.

The construction algorithm for the PCCT also distinguishes between calls that existed during the last sample and calls that are newly created from the last sample. This gives the PCCT the ability to distinguish whether a method occurs in a sample because it is time-consuming or because it is frequency-executed.

5.2 Construction algorithm

The PCCT is constructed by periodically sampling the call stacks and CPU times of the running Java threads and incrementally updating the data structure [7]. The algorithm is presented in a C-like code syntax in Figure 7.

The algorithm proceeds in two phases. The upward phase begins by looking up the method corresponding to the program counter address. (In our system, this information is stored in a binary search tree.) It stores the method in a buffer, then reads the word in the current stack frame containing the return address. We use a bit in the return address as a marker.[8] The profiler sets the marker bits as stack frames are visited. When a new stack frame is constructed, the return address word is overwritten and so the bit is reset. Therefore, we can distinguish between new and old stack frames with no change to the code and zero runtime cost. If the stack frame was not visited, we set the visited bit and continue to the next stack frame. The traversal continues until it either reaches a visited stack frame or the buffer overflows.

```
void update_pcct(void* PC, void* SP, int cputime)
{
  Method* buffer[BUFSIZE], *m = lookup(PC);
  int ind=0; Node* n1=0;
  /*** UPWARD PHASE ***/
  while (ind < BUFSIZE) {
    buffer[ind++] = m;
    WORD rt = *(SP-4); m = lookup(rt);
    if (rt&VISITED_BIT) {
      n1 = get_node(m, rt);
      break;
    }
    *(SP-4) = (rt | VISITED_BIT); SP = (*SP);
  }
  if (!n1) n1 = allocate_node(m);
  /*** DOWNWARD PHASE ***/
  while (ind-- > 0) {
    m = buffer[ind];
    Node* n2; Edge* e = get_edge(n1, m);
    if (!e) {
      n2 = allocate_node(m);
      e = allocate_edge(n1, n2);
    } else n2 = e.to;
    e.time += cputime; update_sorted_pos(e);
    n1 = n2;
  }
  n1.time += cputime; update_sorted_pos(n1);
}
```

Figure 7: Code to update the PCCT when a new sample is encountered.

If the traversal ends with a visited stack frame, we obtain the node for that stack frame from the method, otherwise we create a new node for that method. We then enter the downward phase of the algorithm. The downward phase retrieves the method for the next stack frame from the buffer, and searches for an edge from the current node to a node marked with the named method. If it exists, the edge weight is incremented and the target node becomes the current node, otherwise a new edge and node are created. This process continues for all methods in the buffer. When the last method is reached, the weight of the final node is incremented, and the sample is complete.

The actual implementation differs slightly from the code in Figure 7. First, it distinguishes between different call sites of the same method. Second, it stops searching for edges from nodes once a new edge is created. For clarity, we did not include these in the code in Figure 7.

5.3 Data structure definitions

The data structure definition is contained in Figure 8. Nodes are represented by the `PCCT_Node` structure. They contain a pointer to the method identifier for the node, a CPU time, a pointer to their incoming edge, a pointer to a list of outgoing

[7] On systems that do not support obtaining CPU times on a thread-by-thread basis, we make the assumption that all threads take approximately the same amount of CPU time. Other approximations are also possible [14].

[8] Word addressed systems use a low order bit because it is ignored. Byte addressed systems use a high order bit, and high memory is mapped to be aliased to low memory.

```
struct PCCT_Node {
  Method* method;          // method identifier
  PCCT_Edge *parent;       // incoming edge
  PCCT_Edge *children;     // list of out edges
  PCCT_Node *prev, *next; // for sorted node list
  short time;              // CPU time
}
struct PCCT_Edge {
  PCCT_Node *from, to;     // edge source/target
  PCCT_Edge *next_edge;    // next with same source
  PCCT_Edge *prev, *next; // for sorted edge list
  short offset;            // call site location
  short weight;            // edge weight
}
```

Figure 8: Data structure definitions for the PCCT_Node and PCCT_Edge types.

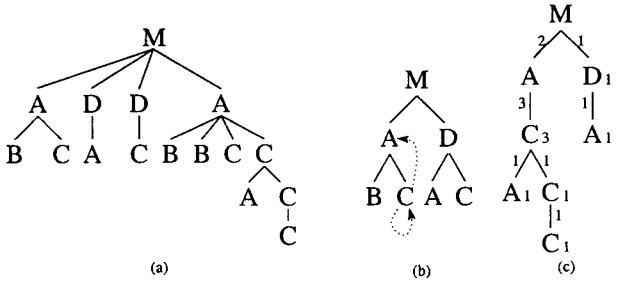

Figure 9: (a) A dynamic call tree containing recursive calls (b) its corresponding calling context tree (with lost context), and (c) its corresponding partial calling context tree, with full context.

edges, and previous and next pointers for the sorted list of nodes. Edges are represented by the PCCT_Edge structure, which contains the location of the call site, source and target nodes, weight, link to the next edge with the same source, and links for the sorted edge list.

All PCCT_Node and PCCT_Edge objects are allocated out of a single memory buffer to reduce allocation overhead. To purge the profile data, we simply clear the buffer. Because we periodically purge the profile buffer, the buffer is small, giving us good spacial locality between nodes and edges.

5.4 Recursion

It is worth noting that recursion is handled slightly differently in the PCCT as compared to the CCT. For recursive methods, the CCT combines nodes with separate contexts into a single node, creating a cycle or backedge in the tree [1]. This loses valuable context information for recursive methods; however it is necessary in order to bound the depth of the CCT in the presence of recursion. See Figure 9(b) for an example. The PCCT, on the other hand, has other mechanisms to bound the size of the data structure; for example, only traversing a certain number of stack frames and periodically purging data. The PCCT for the program in Figure 9(a) would be as shown in Figure 9(c).

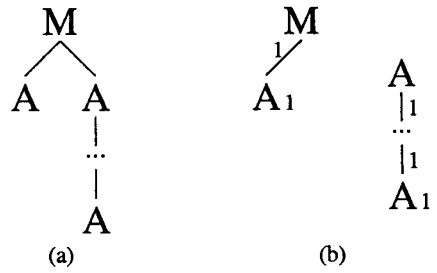

Figure 10: (a) A dynamic call tree with a very deep sequence of calls. (b) In the case where the first sample was on the left branch of (a) and the second sample was at the end of the right branch of (a), the long string is not correctly attached to the calling context.

5.4.1 *Imprecision due to limited number of stack frames visited on backward pass*

The construction algorithm limits the number of stack frames that it can visit on any given sample. This is to insure that the sampling profiler can complete quickly, even in the presence of very deep stacks.[9] This means that if a very deep sequence of calls occurs between two samples, the algorithm may not be able to identify where to place the new profile data point. In this case, the profile point is not attached to the original tree, as shown in Figure 10.

6. RESULTS

This section presents some experimental results showing the accuracy and efficiency of our profiling system. We implemented this profiler on the various platforms supported by the IBM Developer Kit, Java(TM) Technology Edition [12, 21, 19], an industry-leading[10] Java virtual machine and Just-In-Time compiler. The current version of the profiler is written in portable C code, and most of the profiler code base is shared between the different platforms.

6.1 Benchmarking methodology

Due to the variability of benchmarking, especially in the case of a timer-based profiler where times may vary by less than 1%, we ran each test multiple times to establish confidence intervals. Because the distribution was approximately normal and the number of samples was small, we used a Student's t distribution calculation with to establish the confidence intervals. We ran one warmup run, and then we repeated the tests until the 95% confidence interval converged to within ±5% of the run time value.

For the tests that collect profiled data, we attempt to simulate how the profile data is actually used by the dynamic compiler. The dynamic compiler operates by collecting profile data until some threshold is reached, at which time it performs recompilations and purges all old profile data. We modeled this behavior by reseting the profile data set every 10 seconds.

[9]It is worth noting that the CCT construction algorithm does not pay attention to the stack depth while searching the stack, which it does once for *every edge in the CCT.*

[10]As measured by the VolanoMark [22] and SPECjvm98 on Intel [7] benchmarks.

Name	Processor(s)	Memory	Operating System
P3-550-S	Pentium III 550 mhz	256 MB	Windows NT 4 SP5
P3-500-D	Dual Pentium III 500 mhz	256 MB	Windows NT 4 SP5
P-120-S	Pentium 150 mhz	48 MB	Windows 98 SP1
P604e-D	Dual PowerPC 604e 233 mhz	256 MB	AIX 4.3.2
P3-D	Dual Power3 200 mhz	256 MB	AIX 4.3.2

Figure 11: Benchmark machine descriptions

We also include information about the precision of the sample profiling. We cannot use another instrumentation-based profiler to accurately measure what the "correct" program profile is because introducing any instrumentation code disrupts the program profile. Therefore, we measured the accuracy of the measurements made by the sampling profiler through running the benchmark multiple times and evaluating the correlation between successive runs. (It is assumed that successive benchmark runs have identical program profiles.) We measured the correlation between sets using correlation coefficients: the covariances divided by the product of the standard deviations. These numbers range from -1 to 1. A correlation of 1 means identical distributed, 0 means no correlation, and -1 means opposite correlation. We measured the correlation between each 10 second profile data set independently so that a benchmark's length did not effect the correlation coefficient. The given precision numbers are the geometric means of the correlation coefficients.

6.2 Benchmark machines

We used a total of five different benchmark machines. See Figure 11. All of these systems were running an internal build of IBM Developer Kit, Java(TM) Technology Edition version 1.2 with 1:1 Java-to-OS-thread mapping.

6.3 Benchmark applications

We used some industry standard benchmarks to test the performance of our sampling profiler. `spec` is the set of SPECjvm98 benchmarks run in noninteractive mode with the default large `-s100` input size [7]. `cm30` is the set of CaffeineMark benchmarks version 3.0 [17]. `pbob` is Portable BOB version 2.0b, a multithreaded business transaction benchmark for Java [6]. We ran it in `-auto` mode with a 128MB heap. Note that `cm30` is single threaded, `spec` contains some use of threads, and `pbob` contains extensive use of multithreading.

6.4 Timer Interrupt Mechanism

There are a number of different mechanisms of obtaining periodic timer interrupts, ranging from process level interrupts to processor hardware performance counters. After evaluating each of them, we found that a simple busy loop with a call to the operating system `sleep()` function had acceptable precision and minimal overhead. All results presented here use this busy-wait timer.

6.5 Hot method counters

We evaluated the performance of our sampling profiler infrastructure in collecting context-insensitive profile information by implementing a "hot method count" profiler. See Figure 12 for the code. The operation of this profiler is straightforward — it simply takes the program counter from each of the threads, finds which method contains that address, and increments a counter associated with the method. The counters are kept in a sorted linked list so that we can quickly extract the hottest methods. (This can be thought of as equivalent to the PCCT construction algorithm with a maximum depth of zero and only a single node per method.)

```
void update_hm(void* PC, int cputime)
{
  Method* m = lookup(PC);
  m.time += cputime;
  update_pos_in_sorted_node_list(m);
}
```

Figure 12: Code to update the hot method counters.

The performance of this profiler on the different machines, benchmarks, and sampling frequencies can be found in Figure 13. For each machine/benchmark/frequency combination, we measured the increase in run time from enabling the profiler. We also measured the precision of the profile by measuring the correlation between the sets of method hits found on successive benchmark runs.

As we see in Figure 13, even the at fastest level of sampling (once every $1ms$), the overhead is typically around 1.5-2% on the Windows machines and 2-2.5% on the Unix machines. (`cm30` on P-150-S is a special case; see below.) However, as shown by the correlation scores, the sampling can be significantly slower without losing much accuracy. Even only sampling once every $50ms$ we can obtain very accurate results. In general, the multiprocessor machines have slightly less overhead but also worse precision than the single processor machines, especially for complex multithreaded benchmarks such as `pbob`.

The `cm30` benchmark shows a very dramatic slowdown on the P-150-S at high sample frequencies. This is because CaffeineMark is a set of very small microbenchmarks, and so the code can fit within the Pentium's L1 cache. Activating the profiler causes other memory to be touched and therefore the microbenchmark code is cast out to memory, increasing the stalls due to instruction cache misses. The Pentium III, with its larger cache, does not exhibit this behavior. Note that this case is the pathological worst case for a profiler, but such code is rare outside of microbenchmarks. Note also that an instrumentation-based profiler would have an even larger performance degradation on such microbenchmarks — not only from the overhead of the instrumentation code on every iteration of the tight loop, but also from the instruction cache misses that would likely occur on every iteration, rather than once per sample like with the sampling profiler.

Machine	Benchmark	1 *ms*		10 *ms*		50 *ms*	
		Overhead	Precision	Overhead	Precision	Overhead	Precision
P3-550-S	spec	102.52%	.9892	100.55%	.9611	100.36%	.9113
	cm30	103.71%	.9995	100.29%	.9991	99.45%	.9973
	pbob	101.34%	.9987	100.88%	.9959	100.19%	.9904
P3-500-D	spec	101.10%	.9962	100.46%	.9617	100.30%	.8934
	cm30	101.65%	.9999	100.40%	.9990	100.48%	.9983
	pbob	101.03%	.9955	100.90%	.9952	100.46%	.9453
P-150-S	spec	101.45%	.9895	100.74%	.9928	100.35%	.9846
	cm30	133.15%	.9998	102.49%	.9993	100.65%	.9989
	pbob	101.14%	.9945	100.60%	.9920	100.20%	.9850
P604e-D	spec	103.80%	.9140	103.81%	.9210	101.76%	.8818
	cm30	101.07%	.9979	101.26%	.9976	100.18%	.9945
	pbob	103.40%	.9495	103.10%	.9480	101.05%	.8742
P3-D	spec	103.53%	.9201	103.09%	.9096	100.69%	.8718
	cm30	101.34%	.9994	100.98%	.9969	100.15%	.9954
	pbob	104.18%	.9520	104.01%	.9535	101.98%	.8954

Figure 13: Performance of the hot method counting sampling profiler on various machines, benchmarks, and profile frequencies. The overhead is benchmark performance versus running without profiling. The precision is the correlation in the sets of method hits between corresponding 10 second segments of different benchmark runs; a value of 1 means exact correlation.

6.6 PCCT

Next, we present the performance of the PCCT. We used a slightly optimized version of the construction algorithm from Figure 7. The method buffer size was 16. See Figure 14 for the results. We measured the precision using a similar technique as for the hot methods algorithm; however, in this case the data set was a combination of the set of node counts and edge counts. Two nodes or edges from two PCCTs were considered equivalent for correlation purposes if their contexts were textual identical; that is, the paths from that node/edge to the root had nodes with the same method names in the same order.

As we can see from these results, the PCCT is slightly more expensive to build than simply keeping track of hot methods, but the costs are still very reasonable — around 2-4% for most applications. Furthermore, even with the inclusion of context information, the precision stays high, around the 90% level. The accuracy of the profile seemed to be benchmark dependent. Thus, it may be beneficial for the controller to dynamically adjust the profile period depending on the application profile.

7. CONCLUSION

We presented a portable and efficient sampling-based online measurement system, implemented in a cross-platform industry-leading Java Just-In-Time compiler. This online measurement system is used to provide real-time profile direction and feedback to the dynamic compiler. We presented a new sample profiling technique that is able to correctly identify calling context without walking the entire stack and is able to distinguish between frequently-executed and long-running methods. We also presented a new data structure, the PCCT, for efficient storage of context-sensitive profile information.

Our experimental results are extremely encouraging. Despite the fact that the current implementation is written to be portable to a wide variety of machines, we were able to attain full context profiling at a very minimal performance cost (2-4% slowdown for most applications, which is nearly below measurable levels). Even more surprisingly, the profile data is extremely accurate. We observed typical correlations of $\geq$90% between separate profile segments. These results show that accurate, context-sensitive profile data can be obtained cheaply and effectively, even in a highly-optimized production-level system. This opens the door for a wide variety of new and exciting uses for dynamic profile data in the areas of compilation strategies, dynamic optimizations, and beyond.

Acknowledgements

Many of the ideas for online measurement and the forerunner to the PCCT came about during discussions with Peter Sweeney and others. I would also like to acknowledge all of the people who have worked on the IBM Java virtual machine and Just-In-Time compiler for providing the structure in which this online measurement system is based.

REFERENCES

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

[2] M. Arnold. Real-time call stack sampling. Presentation at IBM T. J. Watson Research Center, June 1999.

[3] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney. A comparative study of static and dynamic heuristics for inlining. In *Dynamo '00 workshop, Held in conjunction with POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2000.

[4] J. J. Barton and J. Whaley. A real-time performance visualizer for Java. *Dr. Dobb's Journal of Software Tools*, 23(3):44, 46–48, 105, March 1998.

Machine	Benchmark	1 *ms*		10 *ms*		50 *ms*	
		Overhead	Precision	Overhead	Precision	Overhead	Precision
P3-550-S	spec	104.71%	.9347	103.74%	.8715	103.46%	.8422
	cm30	105.90%	.9989	102.25%	.9950	101.42%	.9887
	pbob	105.83%	.9146	104.37%	.8832	102.24%	.8428
P3-500-D	spec	103.13%	.9368	102.28%	.8765	102.01%	.8336
	cm30	102.25%	.9998	101.56%	.9930	101.12%	.9858
	pbob	103.89%	.8802	102.49%	.8696	102.62%	.8201
P-150-S	spec	104.04%	.9105	103.26%	.8948	103.08%	.8809
	cm30	141.92%	.9997	106.19%	.9916	102.04%	.9881
	pbob	105.81%	.9091	104.81%	.8911	103.94%	.8249
P604e-D	spec	106.36%	.8691	105.99%	.8689	104.45%	.8170
	cm30	103.05%	.9946	103.14%	.9963	102.20%	.9803
	pbob	106.51%	.8697	106.81%	.8561	104.53%	.7800
P3-D	spec	106.81%	.8589	104.55%	.8541	104.74%	.8112
	cm30	103.10%	.9995	103.91%	.9901	102.08%	.9892
	pbob	106.09%	.8603	106.14%	.8599	103.23%	.7847

Figure 14: Performance of the partial calling context tree on various machines, benchmarks, and profile frequencies.

[5] J. L. Bentley. Software exploratorium: A pride of profilers. *UNIX Review*, 10(10):89–98, October 1992.

[6] Business object benchmark for java. http://www.as400.ibm.com/developer/performance/bob/jbob400paper.pdf.

[7] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks, 1998. http://www.spec.org/osg/jvm98/.

[8] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 292–302, 1997.

[9] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. *ACM SIGPLAN Notices*, 17(6):120–126, June 1982.

[10] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. Technical Report TR 97-03-03, University of Washington, March 1997.

[11] R. J. Hall and A. J. Goldberg. Call path profiling of monotonic program resources in UNIX. In *Proceedings of the Summer 1993 USENIX Conference*, pages 1–13, 1993.

[12] International Business Machines. IBM Developer Kit, Java(tm) Technology Edition, 1999. http://www.ibm.com/java/jdk/index.html.

[13] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 360–371, May 1997.

[14] S. Liang and D. Viswanathan. Comprehensive profiling support in the Java virtual machine. In *Proceedings of the Fifth USENIX Conference on Object-Oriented Technologies and Systems*, pages 229–240, May 3–7 1999.

[15] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, pages 109–121, 1997.

[16] C. Ponder and R. J. Fateman. Inaccuracies in program profilers. *Software Practice and Experience*, 18(5):459–467, May 1988.

[17] Pendragon Software. CaffeineMark Benchmark, 1999. http://www.webfayre.com/pendragon/cm3/.

[18] J. M. Anderson *et al.* Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.

[19] K. Ishizaki *et al.* Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM SIGPLAN '99 Java Grande Conference*, pages 119–128, June 12–14, 1999.

[20] M. G. Burke *et al.* The Jalapeño dynamic optimizing compiler for Java. In *Proceedings of the ACM SIGPLAN '99 Java Grande Conference*, June 12–14, 1999.

[21] T. Suganuma *et al.* Overview of the IBM Java Just-in-time compiler. *IBM Systems Journal, Java Performance Issue*, 39(1), 2000.

[22] Volano LLC. The Volano Report, 1999. http://www.volano.com/report.html.

[23] D. W. Westcott and V. White. Instruction sampling instrumentation. U.S. Patent #5,151,981, assigned to IBM Corporation, Sept. 1992.

[24] J. Whaley. Dynamic optimization thru the use of automatic runtime specialization. Master's thesis, Massachusetts Institute of Technology, May 1999.