RC 21789 (98099) 7 July 2000 ComputerSciences/Mathematics

IBM Research Report

Approximating the Calling Context Tree via Sampling

Matthew Arnold
Department of Computer Science
Rutgers, The State University of NJ
Piscataway, NJ 08854
Email: marnold@cs.rutgers.edu

Peter F. Sweeney
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
Email: pfs@us.ibm.com

Research Division
Almaden - Austin - Beijing - Haifa - T.J. Watson - Tokyo - Zurich

Approximating the Calling Context Tree via Sampling*

Matthew Arnold
Department of Computer Science
Rutgers, The State University of NJ
Piscataway, NJ 08854
Email: marnold@cs.rutgers.edu

Peter F. Sweeney
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
Email: pfs@us.ibm.com

Abstract

The calling context tree (CCT) is a data structure for recording context-sensitive profiling information. The CCT can be expensive to construct since it requires instrumenting all procedure entries and exits. This paper presents an algorithm for constructing an approximate CCT (ACCT) by performing periodic call-stack sampling. Results from a preliminary implementation are reported validating the accuracy of our technique.

1 Introduction

The *calling context tree* [1], or CCT, is a data structure for recording context-sensitive profiling information. It contains more information than a *dynamic call graph*, since it creates separate nodes for calls that occur in different contexts. However, it is more compact than the *dynamic call tree* (the complete record of all calls made in the program) since the CCT uses one node to represent all calls made in the same calling context.
Constructing a CCT requires instrumenting all procedure entries and exits, potentially introducing substantial time overhead, particularly for programs containing frequent calls to short-running methods, as is generally the case for object-oriented programs. The CCT can also become quite large: the number of nodes can grow to N^2 the number of procedures in the program.[1] These time and space overheads may prohibit the use of the CCT in situations where time is critical. For example, its use in a Just-In-Time compiler. To reduce the cost of building a CCT, we proposed using runtime call stack sampling to construct an *approximate CCT*, which we refer to as an *ACCT*. The ACCT is cheaper to build than a CCT, since work is performed only when a sample is taken, rather than

*The work described in this paper was completed during the first author's 1999 summer internship at IBM.

[1] If recursion is not explicitly checked, the size of the CCT is potentially unbounded.

on every procedure entry and exit. The worst-case space required by the ACCT is the same as the CCT, however the ACCT will require less space in practice, since infrequently executed methods are unlikely to be sampled, and thus unlikely to be included in the CCT. Furthermore, Whaley [6] has shown that the time overhead to construct an ACCT is negligible.
In this paper we present our algorithm for constructing the ACCT via sampling. We also present results obtained from a prototype implementation used to validate the ACCT construction algorithm by comparing the accuracy of the ACCT to the CCT.

2 Background

A call stack is a thread-specific runtime data structure that keeps track of the methods that are currently active in a thread. Each active method is represented by a frame in the call stack. In this description we assume the stack grows in a downward fashion, i.e., the most recent frame is at the bottom of the stack and least recent frame is at the top. For example, if method A calls method B which calls method C, the call stack before C returns will contain (from top to bottom) a stack frame for A, a stack frame for B, and a stack frame for C.
For each stack frame, a stack frame pointer and a return address can be computed. The stack frame pointer points to the previous stack frame, which is the stack frame of the calling method. In the example above, the stack frame for C would have a stack frame pointer to B's stack frame, and B would have a stack frame pointer to A's stack frame. Most processors have a dedicated stack frame register which points to the last stack frame that was created. In our example above, during the execution of method C, the stack frame register would be pointing to C's stack frame.
The return address points to an instruction in the calling method that occurs after the branch to the callee. The return address uniquely identifies a call site. The return address points to the next instruction that is execute after a method terminates, resulting in execution continuing in the caller.

2.1 Review of CCT Construction

The following is a review the exhaustive CCT construction algorithm. A more detailed description can be found in [1].
The CCT is constructed by performing updates on all procedure entries and exits. The CCT data structure maintains a pointer P to the node N_1 that represents most recently called method (M_1). During the execution of M_1, M_1 either calls another method or returns to its caller. If it returns to its caller, P is adjusted to point to the N_1's parent. If M_1 calls another method, M_2, at call site s, then either M_1 has called M_2 in this context previously or this is the first time. If M_2 was previously called from M_1 in this context, then P is adjusted to point to the existing node, N_2^{old} that corresponds to M_2 in this

context. Otherwise, a new node, N_2^{new}, is created to represent M_2, an edge is added from N_1 to N_2^{new} at call site s, and P is updated to point to N_2^{new}.
Any path length in the CCT can be bounded by the number of methods in the application by handling recursion explicitly. For example, before creating a new node for method M, determine if there is an ancestor node of P that corresponds to M. If there is, add a back edge from P to this ancestor. Of course adding back edges changes a CCT from a tree to a graph.

3 ACCT construction

This section describes our technique for building the ACCT via sampling. When a sample occurs, there are two steps. First, the call stack is examined to identify stack frames which have been created since the last sample. Second, the new information is spliced into the ACCT. Both steps are described below.

Examining the current call stack: Starting from the stack frame of the most recently called method (SF_{last}), the call stack is walked upward. It is not necessary to examine the entire call stack since we only need to identify those frames which have been created since the last sample was taken. To identify these frames, a bit (called a *sampled-bit*) is added in each stack frame. The sampled-bit is turned off when the stack frame is created, and turned on when its stack frame is visited during a sample. Thus, when a sample is taken, the algorithm walks up the call stack until it reaches a stack frame (SF_{old}) for which the sampled-bit is turned on. The stack walk results in a sequence of stack frames starting with SF_{last}, and ending with SF_{old}.
One way of implementing the sampled-bit is to use the low order bit of the return address. Since return addresses are on even boundaries, the low order bit is not used. This avoids the cost of clearing the bit every time a method is called, the bit will be off by default.
The same effect can be achieved by manipulating the return address in a different way. When the stack frame is sampled, the sampled-bit is set by recording the return address and replacing it with the return address of a known location. Thus, a frame has been visited if and only if its return address contains this known location. To ensure that the code executes properly, the code at this known location keeps a list of the true return addresses, and ensures that the code returns to the proper location. There is no overhead for calling and returning from methods which are not sampled. [2]

Splicing the new information into the ACCT: Once identified, the new stack frames need to be spliced into the ACCT. The first step is to identify where in the ACCT the new frames should be spliced. Recall that the stack was walked upward until a previously sampled stack frame was found (SF_{old}). Since SF_{old} was previously sampled, there must be a corresponding node N_{old} already existing in the ACCT.

[2]Thanks to Craig Chambers for suggesting this idea.

To find N_{old} efficiently, the ACCT algorithm maintains a pointer, P_{acct}, to the node, N_{last}, in the ACCT that corresponds to the last method that was called when the previous sample was taken. Since SF_{old} was sampled previously and still remains on the call stack, N_{old} must be an ancestor of N_{last}. Therefore, N_{old} can be found by recursively following parent pointers upward from N_{last}.
Once N_{old} is found in the ACCT, the sampled stack frames starting from SF_{old} are spliced into the ACCT by setting P_{acct} to point to N_{old} and treating the ACCT as a PCCT; that is, simulating a call for each new stack frame encountered when walking down the stack from SF_{old} to SF_{last}. When finished, P_{acct} will point to node which corresponds to SF_{last}.

3.1 Edge weight update strategies

The CCT and ACCT are data structures which allow various context-sensitive profiling information to be recorded. The above ACCT construction algorithm is not specific to any one particular type of profiling information which could be collected.
As a first step, we chose context-sensitive invocation counts as the type of profiling information to be approximated in our ACCT. Each edge in the CCT (or ACCT) connecting nodes A and B contains a weight which represents the number of calls from node A to B.
The following two strategies for incrementing the edge weights were tried while building the ACCT.

1. **increment-all-new-edges:** When a sample is taken, increment ACCT edge weights of all newly sampled methods. More specifically, increment the weights of all ACCT edges connecting the nodes corresponding to path from SF_{old} and SF_{last}.

2. **increment-the-last-edge-only:** When a sample is taken, increment the edge corresponding to the most recent method invocation; that is, the edge that corresponds to the call between the stack frame before SF_{last} and SF_{last}.

Section 4 compares the effectiveness of both of the above strategies.

3.2 Trigger mechanism

The ACCT construction algorithm above describes the actions which need to be taken when a sample occurs, and is independent of the trigger mechanism that signals when a sample should be taken. There are several possible trigger mechanism that could be used, all having different advantages and disadvantages. Three possibilities are:

1. **Time-Based:** Sample every 'n' clock ticks. Contains no runtime overhead.

2. **Global Method Counter:** Sample every 'n' clock ticks. A global counter is incremented on every call, and a sample is taken when the counter exceeds a threshold. Incrementing and checking the counter incurs overhead on every call.

3. **Method Counters:** Each method contains a counter which is incremented and checked on entry to the method. If that counter exceeds a threshold, a sample is taken. Incrementing and checking the counters requires time overhead on each call, and space to store the counters. Method counter implementations [4] typically involve a decay mechanism.

4 Experimental Results

We have implemented a prototype sampler to gain insight into the accuracy and correctness of the sampling system described in Section 3.
Our implementation used a global method counter (as described in Section 3.2) as a mechanism to trigger a sample being taken. On every call a global counter is decremented. When the counter reaches zero, a sample is taken and the counter is reset to its threshold value. The sampled-bit in each stack frame (for determining whether the frame was previously sampled) was implemented by adding an extra word to every frame.
This study uses `javap` (Java Parser) as a benchmark, run with a medium sized input (parsing about 10,000 lines of Java source code). ACCT's were collected for a variety of sampling frequencies, using both counter strategies presented in Section 3.1. Each sampled ACCT is compared to a perfect, or *exhaustive* CCT, which we refer to simply as the "CCT".
Figures 1–5 allow a visual comparison of the accuracy of the ACCT compared to the CCT. The x-axis shows the top 40 edges of the CCT, sorted by "hotness", with the hottest edges on the left. The y-axis shows the hotness of each edge (in terms of number of invocations), shown as a percentage of the hottest edge. There are two bars for each CCT edge, one light and one dark. The light edge represents the weight of the edge in the CCT, while the dark edge represents the weight of the edge in the ACCT.
Figure 1 illustrates a perfect correlation between the ACCT and the CCT. The light and dark bars are the same height for each CCT edge.
Figure 2 shows the ACCT which results from sampling every 59th call, using the increment-all-new-edges strategy. The number 59 was arbitrarily chosen as a small prime number. As shown by the chart, there is no apparent correlation between the light and dark bars, thus arguing that this ACCT is a very poor approximation of the CCT.
Figure 3 shows the ACCT which results from the same sample rate (every 59th call) but using the increment-the-last-edge-only strategy. This graph shows a *much* stronger correlation, with the bars deviating only a small amount from a perfect correlation. Figures 4 and 5 show ACCTs collected sampling every 307th and 1559th call respectively, using the increment-the-last-edge-only strategy. As the sample frequency decreases, the correlation between the ACCT and the CCT degrades; however, these approximations are still

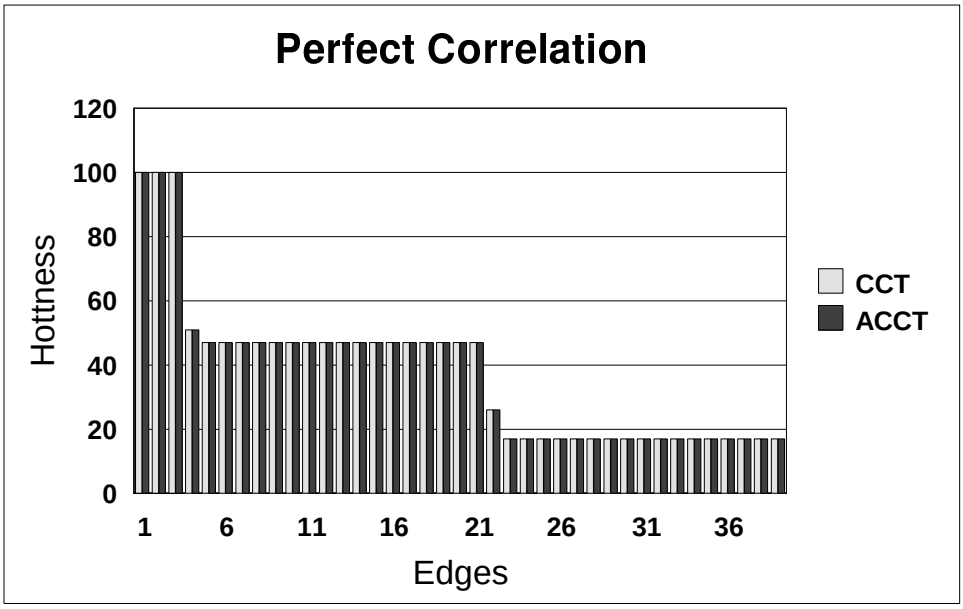

Figure 1: Sample every call. Represents a perfect correlation between the CCT, and the ACCT.

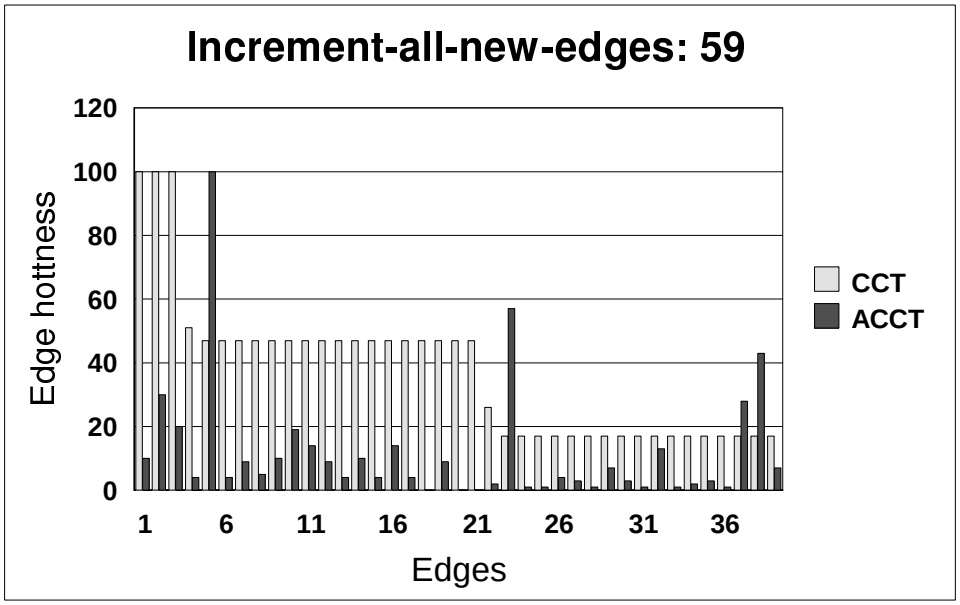

Figure 2: Use the increment-all-new-edges strategy and sample every 59th call.

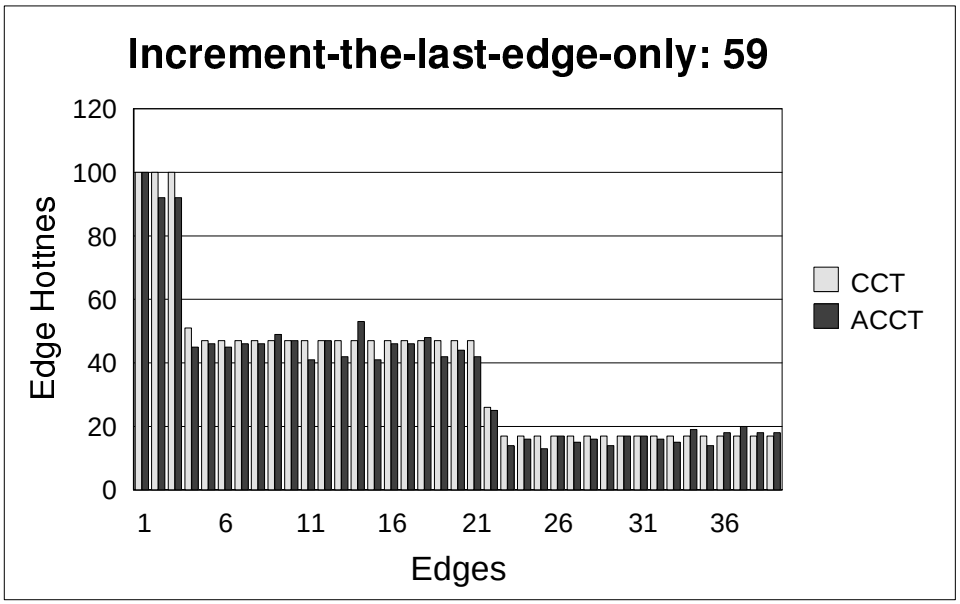

Figure 3: Use the increment-the-last-edge-only strategy and sample every 59th call.

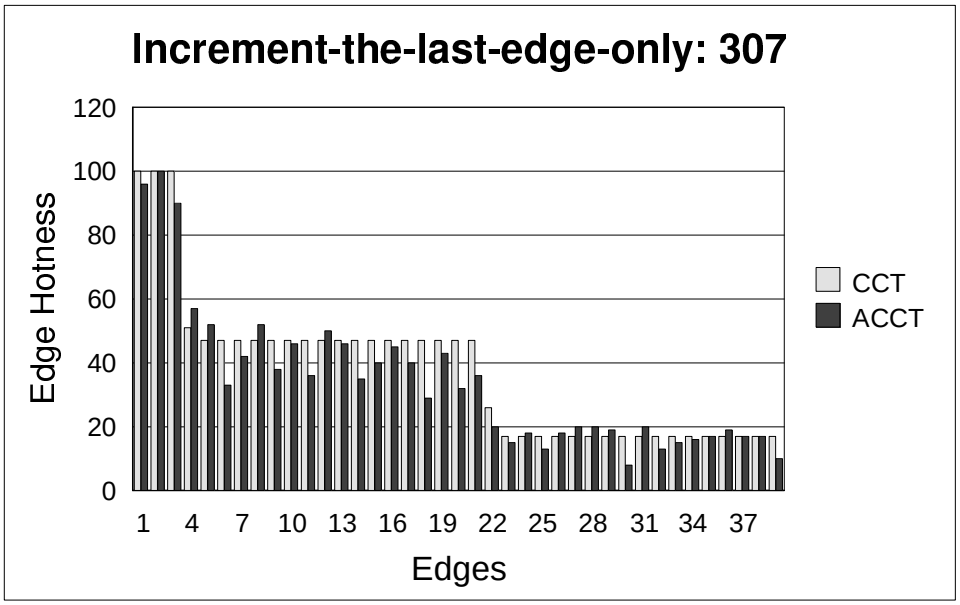

Figure 4: Use the increment-the-last-edge-only and sample every 307th call.

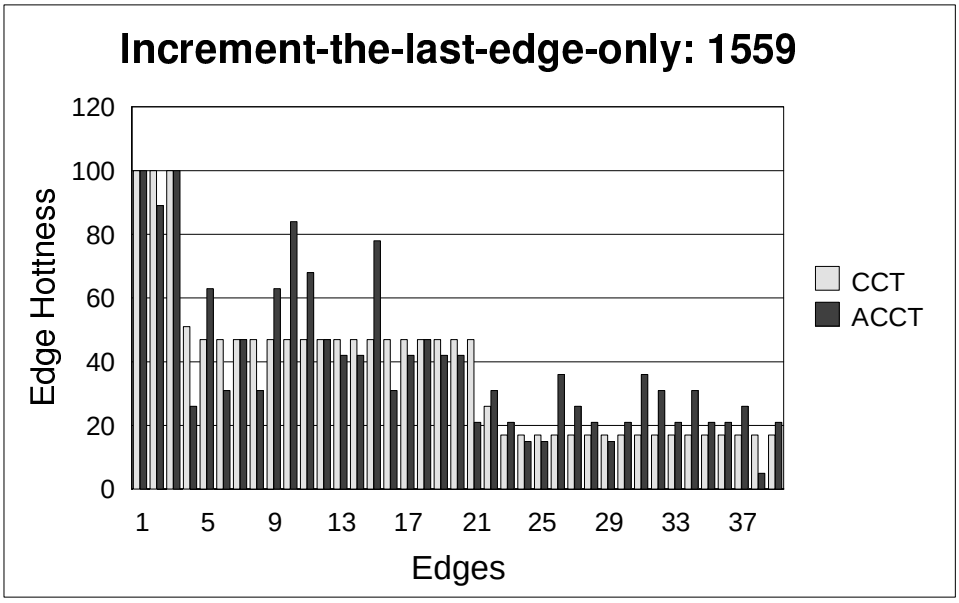

Figure 5: Use the increment-the-last-edge-only strategy and sample every 1559th call.

acceptable when compared to the higher sample rate with the increment-all-new-edges strategy.

Although the increment-all-new-edges strategy seemed intuitive at first, it is fundamentally incorrect. Consider the example program in Figure 6. As shown by the CCT, Method `A` calls method `B` 10 times, which calls Method `C` 1000 times.

The ACCT shown is the ACCT which would be constructed by sampling every 100th call when using the increment-all-new-edges strategy. It is clearly incorrect, demonstrating this strategy's shortcoming. For any sample rate over 100, both `B` and `C` will be new on every sample, and thus both edges will always have the same weight. It is important to note that this error a fundamental error in the edge incrementing strategy, and has nothing to do the sample rate being a multiple of the loop bound. Adding a degree of randomness to the sampler does not help the problem, nor does using a different trigger mechanism, such as time-based or method counters.

Using the increment-the-last-edge-only strategy, however, *does* solve the problem. Probabilistically, the call from `B` to `C` will trigger a sample 10 times more frequently than the call from `A` to `B`, therefore incrementing only the last edge will produce desired edge weights for ACCT. Incrementing the edge from `A` to `B` when `C` triggers a sample simply adds error to the ACCT by over-inflating the value of this edge.

5 Related Work

Whaley [6] describes a technique for using time-based sampling to construct a *partial calling context tree*, or PCCT. Although developed independently, Whaley's construction

Program: **CCT:** **ACCT:**

```
a() {
  for(i=0;i<10;i++)
    b();
}

b() {
  for(i=0;i<100;i++
     c();
}

c() {
  count++;
}
```

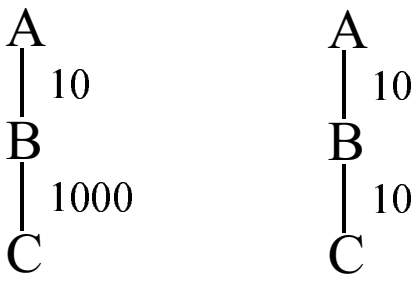

Figure 6: Sample code demonstrating the fundamental inaccuracies of the increment-all-new-edges update strategy.

algorithm is similar to ours, and he demonstrates that a PCCT can be constructed with negligible time overhead: less than 3% of an application's execution time.

Nonetheless, a difference between the PCCT and our ACCT is that the PCCT has the option to place a limit K on the stack traversals. When sampling the stack, only K stack frames are examined, regardless of whether a previously sampled stack frame is reached. If a previously sampled stack frame is not reached, it is unclear from Whaley's paper what is the corresponding node (if any) in the PCCT at which the sample should be spliced, and what effect this choice has on the accuracy and usefulness of the PCCT.

In addition, Whaley uses the increment-all-new-edges strategy. He evaluates the PCCT by running each benchmark multiple times and computing the correlation between the runs. Although this metric shows stability, it does *not* show the accuracy of the PCCT compared to the exhaustive CCT. Because of this, his work did not discover the fundamental inaccuracies in the edge weight increment strategy.

Several other works use low overhead sampling to improve program performance. Such systems include Jalapeño [3], Digital FX!32 [5], *Morph* [7], and *DCPI* [2]. In contrast, this paper focuses on a particular algorithm for using sampling to construct an approximation of the CCT.

6 Conclusions and Future Work

In this work we have presented an algorithm for constructing an approximate CCT (ACCT) using periodic call-stack sampling. This technique allows the the ACCT to be used in cases where time and space constraints do not allow the construction of an exhaustive CCT. We have reported preliminary results from a prototype implementation which validates the accuracy of the ACCT. Furthermore, we have shown that the strategy that is used to increment edge counters can have a significant impact the accuracy of an ACCT. In particular, we have shown that the increment-all-new-edges strategy is fundamentally incorrect, and inaccurately approximates the CCT. We have also shown that the increment-the-last-edge-only strategy is a significantly better approximation, making the ACCT a promising alternative to the CCT.

Acknowledgments

We would like to thank Michael Burke, Barbara Ryder, and Vivek Sarkar for their support of this work.

References

[1] G. Ammons, T. Ball, and J.R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN '97 Conf. on Programming Language Design and Implementation*, 1997.

[2] Jennifer M. Andersen, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? Technical Note 1997-016a, Digital Systems Research Center, www.research.digital.com/SRC, September 1997.

[3] Matthew Arnold, David Grove, Michael Hind, Stephen Fink, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 2000.

[4] Urs Holzle and David Ungar. A third generation self implementation: Reconciling responsiveness with performance. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 229–243, 1994.

[5] Raymond J. Hookway and Mark A. Herdeg. Digital FX!32: Combining emulation and binary transslation. *Digital Technical Journal*, 9(1):3–12, January 1997.

[6] John Whaley. A portable sampling-based profiler for Java virtual machines. In *ACM 2000 Java Grande Conference*, June 2000.

[7] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System support for automated profiling and optimization. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, Operating Systems Review, 31(5), pages 15–26, October 5–8 1997.