

Accurate, Efficient, and Adaptive Calling Context Profiling

Xiaotong Zhuang*
Georgia Institute of Technology
xt2000@cc.gatech.edu

Mauricio J. Serrano Harold W. Cain
Jong-Deok Choi
IBM T.J. Watson Research Center
{mserrano,tcain,jdchoi}@us.ibm.com

Abstract

Calling context profiles are used in many inter-procedural code optimizations and in overall program understanding. Unfortunately, the collection of profile information is highly intrusive due to the high frequency of method calls in most applications. Previously proposed calling-context profiling mechanisms consequently suffer from either low accuracy, high overhead, or both.

We have developed a new approach for building the calling context tree at runtime, called *adaptive bursting*. By selectively inhibiting redundant profiling, this approach dramatically reduces overhead while preserving profile accuracy. We first demonstrate the drawbacks of previously proposed calling context profiling mechanisms. We show that a low-overhead solution using sampled stack-walking alone is less than 50% accurate, based on degree of overlap with a complete calling-context tree. We also show that a static bursting approach collects a highly accurate profile, but causes an unacceptable application slowdown. Our adaptive solution achieves 85% degree of overlap and provides an 88% hot-edge coverage when using a 0.1 hot-edge threshold, while dramatically reducing overhead compared to the static bursting approach.

Categories and Subject Descriptors D2.3.4 [Programming Languages]: Processors—compilers

General Terms Algorithms, Measurement, Performance

Keywords Profiling, Call Graph, Calling Context, Calling Context Tree, Adaptive, Java Virtual Machine

1. Introduction

Many compiler optimizations benefit from accurate profile information, but this information must be collected carefully. A profiling mechanism must not significantly affect the performance of an application, otherwise the measured profile may not reflect the application's true behavior. Unfortunately, it is difficult to collect the fine-grained events necessary for creating an accurate profile without perturbing the application, due to the intrusiveness of collection. Constructing accurate profiles while minimizing overhead

is particularly important for just-in-time compilers, since any slowdown due to profiling must be compensated by the performance gains obtained from the profile information.

Inter-procedural optimizations often depend upon accurate calling-context profiles; however, creating these profiles is complicated by common object-oriented programming styles and language features. Due to small average method sizes, calls and returns are frequently made, leading to high overheads if calling context information is collected on a per-method-call basis. Frequent calls may also lead to deep stack depths, resulting in a high-cost operation to determine the calling context within the stack. Additionally, virtual function calls obscure the caller-callee relationship, making static construction of the profile very difficult.

Call graphs are a commonly used, succinct representation of method invocation behavior, in which nodes representing methods are connected by directed edges representing caller-callee relationships. For some applications, the usefulness of call graphs is limited due to a lack of detailed context information. For example, an effective inlining may require inlining not just a single-level call site, but a sequence of calls. A call graph does not provide enough information for this purpose, because it represents each method using a single node, regardless of the number of paths on which the method appears. Also, understanding the cause(s) of events such as cache misses or synchronizations frequently requires identifying not just the method where such events occur, but the overall context in which these methods execute. The importance of additional context information has been demonstrated in much prior work [3, 4, 6, 9, 21, 23, 24, 26]. Furthermore, as mentioned by others [3, 23], the standard profiling tools *gprof* [22] and *gpt* [7] cannot properly apportion the cost of a procedure to its callers, leading to an inaccurate profile.

The calling context was previously defined as the chain of method calls that are concurrently active on the stack [3, 23]. Naturally, a tree may be used to represent an execution's contexts, which reduces the duplication of common methods at the bottom of the stack (assume stack grows from bottom to top). Two commonly used representations for calling contexts are *Call Trees (CT)* and *Calling Context Trees (CCT)* [3]. In a CT, a new node is added for each method invocation, attached to the parent node (i.e. the caller method), whereas a CCT merges identical child nodes (methods) of the same parent node. Call trees and calling context trees will be described in more detail in Section 2 and Figure 1.

We present a sampling-based profiler for building a CCT that is both space efficient and highly accurate. At every sampling point, the profiler walks the stack from the top of the stack to the root of the stack and identifies the full calling context at that point. The stack-walking is immediately followed by a short period, called a *burst*, during which the profiler traces each and every method call and return.

This profiling mechanism is refined through *adaptive sampling*. Adaptive sampling predicts the method calls and returns made

*This research was performed while the author was an intern at IBM Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI '06 June 10–16, 2006, Ottawa, Ontario, Canada
Copyright © 2006 ACM 1-59593-320-4/06/0006...\$5.00.

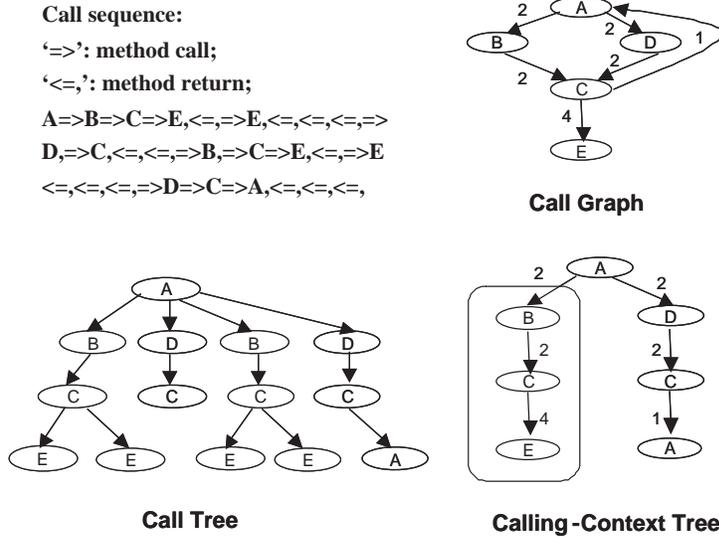


Figure 1. Examples of call graph, call tree and calling context tree (CCT)

during a burst using a history-based predictor, and disables certain bursts that are deemed to be redundant. This prediction is based on the observation that bursts of stack-walks that begin from the same calling context tend to contain similar sequences of method calls and returns. Predicting and disabling certain bursts greatly reduces the profiling overhead, but suffers from mispredictions. To remedy this, we *stochastically* re-enable a small portion of predicted-and-disabled bursts, based on a predetermined re-enable ratio, called the *adaptive re-enable ratio (RR)*. In order to maintain the relative balance of edge-weights, the profiler then adjusts the counts of the calls of the bursts that are re-enabled, again, using RR (more precisely, using $1/RR$).

We show that this adaptive mechanism approaches the accuracy of an exhaustively constructed CCT, while keeping profiling overheads within reason.

The main contributions of this paper are:

1. A novel profiling technique for efficient construction of highly accurate CCT.
 - The runtime overhead of the technique is much smaller than exhaustive calling-context profiling techniques, while maintaining high accuracy in the range of 80% to 90%.
 - Our implementation is based on a commodity profiling interface, JVMPI, and does not require compiler-specific instrumentation. We expect the overhead to come down significantly with instrumentation techniques more efficient than JVMPI.
2. Formal comparison of two metrics for evaluating CCT accuracy: degree of overlap [5, 6, 14], and hot-edge coverage, which is our new definition.
3. Extensive measurements of the efficiency and accuracy of our technique, and extensive comparison with previous techniques, using a large number of benchmark programs, including a very large commercial J2EE Java application.

The rest of the paper is organized as follows: Section 2 introduces the calling context and the CCT. Section 3 describes metrics to measure the accuracy of the CCT. Section 4 presents previous approaches for constructing the CCT. Section 5 details our adaptive sampling approach. Section 6 presents experimental results and dis-

ussion. Section 7 discusses related work, and Section 8 concludes the paper.

2. Calling Context and the Calling Context Tree

The (dynamic) calling context of a method invocation is the sequence of un-returned method calls from the program’s root method to the method invocation. This calling context can be determined by walking the call stack backwards from the top-of-stack to the root of the stack, recording each method on the stack along the way. Although call graphs, call trees, calling contexts, and calling context trees may be (partially) constructed statically, we focus on dynamic program behavior, and therefore further discussion will assume the dynamic construction of these structures unless explicitly stated otherwise.

Figure 1 shows an example call sequence, along with the call graph, the call tree (CT), and the calling context tree (CCT), all corresponding to the call sequence. The weight of each call-graph edge represents the total number of calls of the target method (of the edge) by the source method (of the edge). The CCT is a succinct summary of the call tree, built recursively from the root by merging each node’s children that correspond to the same method. The edge weight of a CCT represents the number of nodes merged into the single target node of the edge. The edge weight also represents the number of calls (i.e. calling frequency) of the callee by the caller within the particular calling context of the caller.

Although a CT captures more complete information, its space overhead is much higher than a CCT. The CT size is proportional to the total number of calls in an execution since each call must generate a new node, and it therefore has the same asymptotic complexity as the entire call trace. The space requirement of a CCT can also be large in the presence of recursion, but in practice is much smaller than that of a CT. For this reason, we focus on the CCT representation.

Note that in the example, all calls of E by C happen when C is called by method B, i.e. when the calling context of method C is “ $A \Rightarrow B \Rightarrow C$ ”. C does not call E at all when its calling context is “ $A \Rightarrow D \Rightarrow C$ ”. This information is absent from the call graph: both edges onto C have the same edge weight (i.e. counter value of two). This information, however, is clear from both the CT and the CCT. A CCT, like the one in the figure, whose

$$\text{hotcover}(CCT_1, CCT_2, T) = \frac{|\{e : e \in CCT_1, e_{\text{weight}} \geq T * \text{hottest}_1\} \cap \{e : e \in CCT_2, e_{\text{weight}} \geq T * \text{hottest}_2\}|}{|\{e : e \in CCT_2, e_{\text{weight}} \geq T * \text{hottest}_2\}|} \times 100\%$$

Figure 2. Hot-edge coverage of CCT_1 over CCT_2 with threshold T ($0\% \leq T \leq 100\%$)

edge weights represent their calling frequencies could be used for context sensitive inlining as described in prior work [19]. However, depending on the purpose of the calling context information, edge weights may represent different metrics such as the number of cache misses incurred within a particular context.

A more precise representation of the CCT should also distinguish calls made from different call sites. That is, the children of a parent node may be distinguished by both the method and its call site in the caller/parent method. We omit this information for now, due to a lack of support in our experimental infrastructure (JVMPi).

If a CCT captures all caller/callee pairs during an execution, we call it a *Complete CCT*. It is noteworthy that the completeness of a CCT is only with respect to a particular execution. Edges that are statically possible but do not occur during an execution will be missing from the complete CCT. Due to the incomplete nature of sampling-based profiling mechanisms, we call a CCT built from a sampling-based approach, including ours, an *approximate calling context tree* (ACCT)[6]. Informally, the accuracy of an ACCT of an execution is the measure of how “faithfully” the ACCT represents the Complete CCT of the same execution.

For comparison of two CCTs, we must know whether two edges or nodes from the two CCTs are equivalent. We recursively define the *equivalence of nodes and edges* as follows:

DEFINITION 1. Equivalence of Nodes and Edges:

1. Two root nodes n_1 and n_2 that correspond to the same static method are equivalent.
2. Node n_1 on CCT_1 and node n_2 on CCT_2 are equivalent if they correspond to the same static method, and their parent nodes are equivalent.¹
3. An edge e_1 on CCT_1 is equivalent to an edge e_2 on CCT_2 if the source nodes of the edges are equivalent and the target nodes of the edges are equivalent. If two edges or nodes are equivalent, we simply say the edge or node is part of both CCTs.

3. Measuring CCT Accuracy

To fairly evaluate and compare different approaches for CCT construction, we must define proper metrics so that the effectiveness of each approach may be compared. To compare profiler overhead, we measure the execution slowdown caused by the profiling mechanism. Because some approaches reduce the overhead at the expense of accuracy, we must define metrics to compare the quality of an ACCT relative to a complete CCT.

We present two metrics to measure how accurately one CCT matches another CCT: (1) degree of overlap, and (2) hot-edge coverage.

3.1 Degree of Overlap

The degree of overlap metric is used to judge the completeness of one CCT with respect to another, and has been used in several other research papers [5, 6, 14]. The definition is as follows:

$$\text{overlap}(CCT_1, CCT_2) = \sum_{e \in CCT_1 \cap CCT_2} \min(\text{pweight}(e, CCT_1), \text{pweight}(e, CCT_2))$$

¹If call sites are distinguished through the addition of labeled edges, the call-site labels of the edges connecting to their parents must also be equivalent

where $\text{pweight}(e, CCT)$ is defined as the percentage of CCT 's total edge weights represented by the edge weight on e . Only edges on both CCT_1 and CCT_2 are counted. The degree of overlap indicates how CCT_2 overlaps with CCT_1 or how CCT_2 is covered by CCT_1 . The degree of overlap range is from 0% to 100%.

3.2 Hot-edge Coverage

While degree of overlap measures the similarity of two CCTs in their entirety, the hot-edge coverage metric focuses on the similarity of edges in each graph having relatively high edge weights (so-called “hot edges”). This metric is useful because hot edges are usually the focus of performance optimization. Accurately collecting information about these edges may be more useful than accurately constructing an entire CCT that includes rarely called paths.

As defined in Figure 2, hot-edge coverage measures the percentage of hot edges of one CCT that are covered by another CCT, where an edge-weight threshold criteria is used to determine hotness. In this definition, hottest_1 and hottest_2 represent the weights of the hottest edges of CCT_1 and CCT_2 , and $0 \leq T \leq 100\%$.

A higher value of the hot-edge coverage indicates that CCT_1 more accurately captures the hot edges from CCT_2 . The hot-edge coverage range is from 0% to 100%.

For example, a hot-edge coverage threshold of 0.95 will use all the edges whose weights are within 5% of the hottest CCT edge in computing the coverage. It should be noted that this is very different than simply choosing the top 5% hottest edges, or edges whose weights represent 5% of the cumulative CCT edge weights, which are two other measures of hotness.

Hot-edge coverage approximately captures how well the ordering of the edges in CCT_2 , with respect to their hotness, matches that in CCT_1 . Intuitively, if

$$\text{hotcover}(CCT_1, CCT_2, T_i) \gg \text{hotcover}(CCT_1, CCT_2, T_j),$$

for $100\% \geq T_i > T_j \geq 0\%$, then the set of edges in CCT_2 whose hotness is between $\text{hottest}_2 * T_i$ and $\text{hottest}_2 * T_j$ is quite different from that of the corresponding edges in CCT_1 .

4. Conventional Approaches to Calling Context Tree Construction

In this section, we compare and contrast two profiling mechanisms that have been previously used to collect method invocation profiles: the *exhaustive approach* and *sampled stack-walking*. Each of these proposals has downsides that are addressed by our adaptive profiling mechanism described in Section 5.

4.1 The Exhaustive Approach

Given a trace containing all method calls and returns, calling context tree construction is straightforward. Initially, a root node is added to the tree, and a cursor pointer is maintained that points to the current method context, initialized to the root node. If a method call is encountered, the node’s children are compared to the new callee. If a matching child is found, the weight of the edge onto the child is incremented. If no child matches the callee, a new child is created. The cursor is then moved to the callee method. If a return is seen, the cursor is moved back one level to the parent. In the case of multi-threaded applications, a cursor is needed per thread.

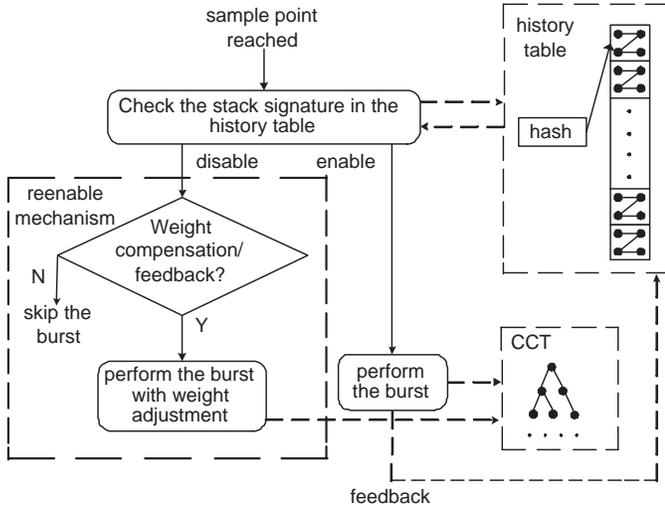


Figure 3. Overview of the adaptive bursting mechanism

This exhaustive approach builds a Complete CCT, but obviously suffers from severe performance degradation due to the tracing overhead. Our experiments indicate that tracing overhead can cause over 50 times slowdown using our JVMPI-based infrastructure, since each and every method call and return must be instrumented. Prior work has also reported considerable slowdown, despite using a more efficient instrumentation mechanism [3].

4.2 Sampled Stack-Walking

Sampled stack-walking is one alternative to the exhaustive approach, which was used in previous experiments by Arnold and Sweeney [6]. Since a cursor pointer cannot be maintained across samples, the current context is determined at each sampling point by performing a stack walk from the current method to the root method, and adding this path to the CCT if necessary. If the CCT already contains the path, the edge weight between the top two methods on the stack is incremented.

Since the sampling rate may be controlled, profiling overhead can be easily minimized at the cost of accuracy. However, the accuracy of the sampled stack-walking approach suffers for two main reasons. First, because individual method calls are not observed but inferred, the collected CCT results may be inaccurate and misleading. For example, a program may spend most of its time executing within a single method. However, the sampled stack-walking approach will assume that the method’s caller is making frequent calls to the method because it is always on the top of the stack. Consequently, the CCT obtained with this approach reflects execution time spent in each context more than the method invocation frequency of each context. Second, increasing accuracy by increasing the sampling rate can be costly because of the generally high overhead of the interrupt mechanism to trigger a sampled stack-walking. Furthermore, supporting high-sampling rates may not even be feasible on some systems whose timer-resolution is limited. As will be shown in our results section, the degree of overlap and hot-edge coverage for sampled stack walking are typically below 50%.

5. CCT Profiling using Static and Adaptive Bursting

Because neither the sampling-based nor exhaustive profiling mechanism provides both high-accuracy and low-overhead, we have built a profiling mechanism called *static bursting* that combines

the best features of each approach. We have experimentally found the static bursting profiler to provide many advantages over the exhaustive and sampling-based profiling mechanisms. Static bursting, however, still causes significant performance degradation, which we address with a mechanism called *adaptive bursting*. Adaptive bursting dynamically disables profile collection for previously observed calling contexts. In the next two subsections, we describe these approaches. Because application behavior changes over time, it is useful to periodically re-enable profile collection for disabled calling contexts. We describe this re-enablement mechanism in the third subsection.

5.1 Static Bursting

Like the sampling-based approach, a bursting profiler allows the application to run unhindered between sampling points. At each sampling point, the stack is walked to determine the current calling context. Instead of incrementing an edge weight based on this stack sample (which may not reflect actual method invocation), we revert to the exhaustive approach and collect a “burst” of call/return samples for an interval whose length we refer to as *burst length*. Performing bursting alone at each sampling point, without a stack walk, would result in a low accuracy CCT. Because the calling context at the beginning of the burst would be unknown, it would be difficult to determine where to update the CCT.

A similar approach was previously used for a different purpose: the collection of temporal information for cache references [20]. Although CCT accuracy is dramatically improved when using the static bursting approach, it still introduces significant overheads due to the intrusiveness of each burst, which leads to our next profiling mechanism: *adaptive bursting*, a profiling mechanism that achieves the accuracy of static bursting while minimizing overhead.

5.2 Adaptive Bursting

Because application control flow is highly repetitive, it is no surprise that static bursting collects much redundant information. Adaptive bursting reduces this overhead by selectively disabling bursts for previously sampled calling-contexts, thus reducing redundant samples due to repetitive execution sequences. Unfortunately, permanently disabling bursting for certain contexts leads to two problems: 1) as runtime behaviors periodically change, new calling patterns will be lost from the calling context tree, and 2) by disabling bursting for a common calling context, CCT edge-weights, for example, may become skewed as hot call-return sequences are sampled with the same frequency as rare call-return sequences. Our adaptive mechanism addresses these two problems through *probabilistic burst re-enablement* and *edge-weight compensation*, described below. Figure 3 illustrates the overall flow of the adaptive bursting mechanism.

5.3 Adaptive Bursting with Re-enablement

We adaptively disable bursting based on history information stored in a software-implemented history table and the adaptive re-enable ratio (RR). At each sampling point, this table is indexed using a signature constructed using the run-time stack. If there is no table entry with the matching signature, a new entry is created, and a burst is initiated. However, if a matching entry is found, a random number $0 \leq n \leq 1$ is generated. If $n \leq RR$, a burst is performed. Otherwise, no burst is performed.

Intuitively, the runtime stack contains all methods that are currently on the stack, all parameters being passed, and values of local variables. This information can give us a great deal of information regarding the current state of the program’s execution. This information, however, must be distilled into a concise signature that can be computed at low cost while also being well distributed. We use

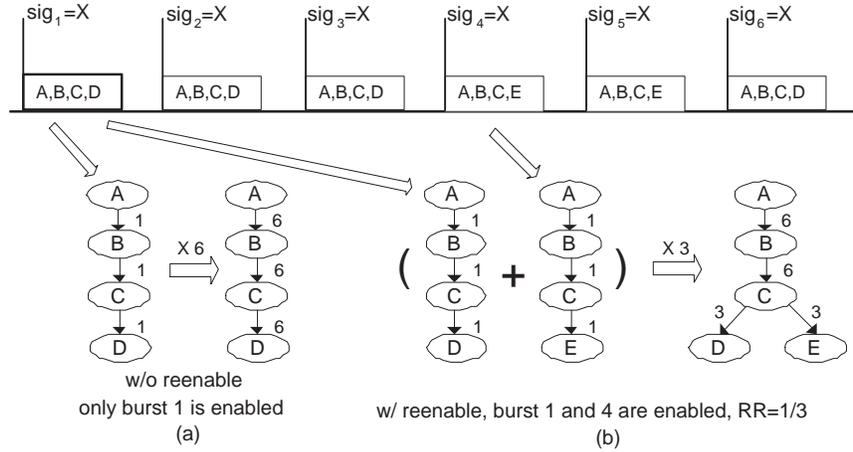


Figure 4. Illustration for weight compensation

NAME	DESCRIPTION	PLAT FORM	Call Graph		CCT
			# nodes	# edges	# nodes
checkit	jvm98 - check program	x86	988	1827	9115
compress	jvm98 - Modified Lempel -Ziv method	x86	721	1227	7581
db	jvm98 - database simulation	x86	744	1310	8666
ipsixql	Persistent XML -database	x86	802	1330	9439
jack	jvm98 - Java Parser Generator	x86	987	1996	58422
javac	jvm98 - java compiler	x86	1505	4144	917986
specjbb	Java business application	x86	2467	5368	66792
jess	jvm98 - Expert Shell System	x86	1101	2106	24194
kawa	Java -based Scheme system	x86	2454	5496	430557
mpegaudio	jvm98 - decompress audio files	x86	898	1516	14019
JAS	SpecJAppServer2004:3 tier java server	AIX	6918	14597	256189

Table 1. Benchmarks evaluated

a variation of the CRC [10], computed using the address of each method and call site on the stack.

Although we could control overhead by manipulating sampling rate, the advantage added by this history-based mechanism is its favoritism of bursting for those calling contexts whose signatures have not been previously observed. Given a fixed bursting rate, this favoritism increases the number of unique calling contexts for which bursting will be enabled.

As mentioned earlier, permanently disabling bursting for a particular calling context is undesirable, because this skews CCT edge-weights by decreasing the relative weight of hot edges and increasing the relative weight of cold edges. Also, because the calling context signature is approximate, it is possible that the same signature may represent two different calling contexts.

In order to maintain an accurate distribution of edge-weights, we again use RR for weight compensation. As described before, at each sampling point, we generate a random number $0 \leq n \leq 1$ to determine whether bursting need be re-enabled (i.e. whether $n \leq RR$). If it is determined that a burst should be re-enabled, we use RR to perform weight compensation, by multiplying every counter value added to the CCT by $1/RR$. For example, if RR is set to 0.25, 25% of the bursts are selectively enabled whose history table entries indicate that they should be disabled. Consequently the edge weights incremented on the CCT during each enabled

burst are multiplied by 4. Intuitively, we only enable one burst for every four bursts that are skipped. Therefore the weights should be four times larger. As an example in Figure 4.b, we set $RR = 1/3$. Thus burst 4 is re-enabled and we can capture the call from C to E . The CCT in Figure 4(b) is therefore more accurate than the CCT in Figure 4(a). For those calling context signatures that are not disabled according to the history table, this multiplication by $1/RR$ is not performed, because their weights do not need to be compensated.

The value of RR reflects the trade-off between accuracy and overhead. A higher RR causes more bursts to be re-enabled, adding higher overhead, whereas a smaller RR could miss more bursts and lowers the quality of the CCT. In Section 6.3, we report the sensitivity of our results to this important parameter.

6. Experimental Evaluation

In this section, we present accuracy and performance data for each of the profiling mechanisms. We empirically show that the static bursting profiler yields the most accurate calling context-tree, but causes unacceptable performance overhead. Using adaptive bursting with re-enablement, we are able to approach the accuracy of static bursting, while incurring much less overhead. This approach also provides much greater accuracy than both the previously pro-

Sampling Interval	10ms
Burst Length	0.2ms
Re-enable Ratio	5%
History Table	2048 entries

Table 2. Default parameters

posed sampled stack-walking technique, and the adaptive-bursting profiler without re-enablement. We also present a study testing the sensitivity of the adaptive profilers to the parameters chosen to control burst length and re-enable ratio, followed by a study of calling-context tree accuracy as a function of time. First, we present the details of our experimental methodology.

6.1 Methodology

Our profiler implementation uses the industry-standard JVMPI instrumentation interface. We obtained results for two different platforms, primarily experimenting using Sun Microsystem’s 1.3.2_08 JVM, running on a 1.8 GHZ Pentium 4/Windows XP system configured with 1GB of memory. We additionally evaluated a Java server middleware application on a 1.6 GHZ, 4-processor (each 2-way SMT), IBM Power5 system configured with 16GB of memory, running IBM’s J9 production virtual machine [17].

Table 1 shows the benchmarks evaluated. On the x86/Windows platform, for comparison we use the eight SPECjvm benchmarks that are also evaluated in [4]. In addition, *kawa* exercises a Java-based Scheme system [2]. *Ipsixql* is a benchmark of persistent XML database services [1]. From SPECjvm98 [12], we did not include *mtrt*, which is a multithreaded application. Instead, we used SPECjbb2000, and SPECjAppServer2004 (*JAS*) [13]. *JAS* is a heavily multithreaded J2EE benchmark program, running on PowerPC/AIX in a three-tier configuration using WebSphere Application Server 6.0, with a DB2 8.2 back-end tier and a client/driver front-end tier. We report results for the middle-tier only, which is running the WebSphere application server.

The last three columns of Table 1 show the number of nodes and edges for the call graph and CCT. (CCT being a tree, its number of edges is always smaller than its number of nodes by one.) This data illustrates the size of the CCT relative to a call graph; on average, the number of nodes (edges) on a CCT is 29 (14) times more than that of the corresponding call graph.

Due to the non-deterministic nature of the multithreaded benchmarks (SPECjbb, and *JAS*) significant variation may exist across runs. To eliminate this potential source of experimental error, we first record a timestamped trace for each of these benchmarks, including all method calls and returns during one execution. We use this trace for off-line simulation of each profiling approach so that we can perform a deterministic comparison. For benchmarks run on the x86 (excluding SPECjbb), we evaluate the slowdown by measuring real elapsed time.

For the initial experiments presented in the next subsection, we use the parameters listed in Table 2. A fixed sampling rate of approximately 10 ms was used for our experiments. Depending on the algorithm, a burst of length 0.2 ms may be initiated. Our adaptive bursting profiler with re-enablement may enable bursting with a default re-enable ratio of RR=5%. These experiments are followed by a sensitivity analysis of the parameter values.

6.2 Results

In this section, we compare the accuracy and overhead of the four profiling mechanisms: sampled stack walking (as described in Sec-

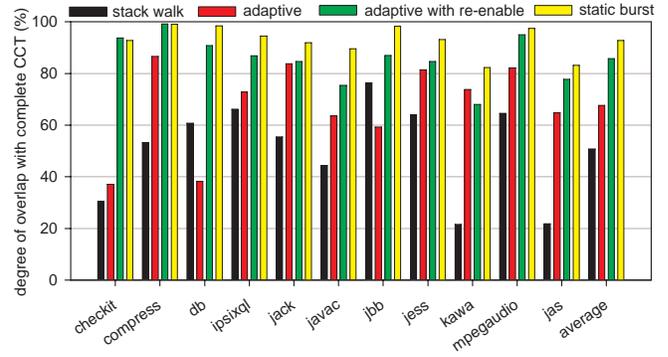


Figure 5. Comparison of the degree of overlap

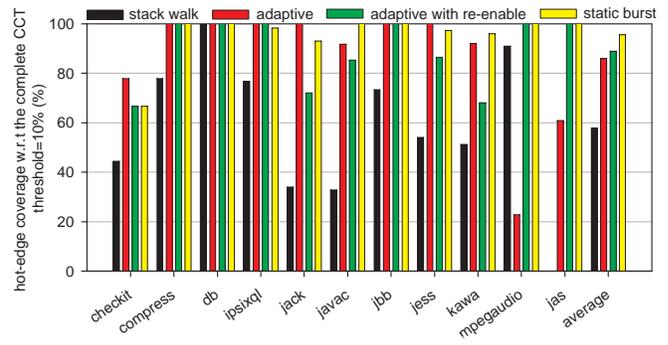


Figure 6. Comparison for hot-edge coverage with threshold=0.1

tion 4), adaptive bursting, adaptive bursting with re-enablement, and static bursting. Figure 5 shows the degree of overlap for all benchmarks, calculated against the complete CCT. In general, sampled stack walking is the least accurate, significantly lower than the bursting mechanisms. Also, its effectiveness varies widely across our benchmarks; the degree of overlap is especially low for benchmarks with a relatively flat method execution profile. Static bursting achieves the highest degree of overlap for most applications, while the adaptive mechanism with re-enabling performs nearly as well. The average values are 49.8%, 68.8%, 85.2%, 91.4% for the four schemes, respectively. Due to the probabilistic nature of each of these sampling-based profilers, it is possible for non-representative samples to skew the profiler’s accuracy. We believe this is the case for *checkit*, and *kawa*, in which profiling mechanisms that collect more profiling information actually yield slightly less accurate results.

Figure 6 compares the accuracy of each profiler using the hot-edge coverage metric, using a threshold of 0.1. This means we are only interested in the edges with weight between $0.1 \times \text{hottest weight}$ and hottest weight . Notice that this range typically consists of very few edges (7 out of 7580 for *compress*, which contains the fewest edges, and 24 out of 917985 for *javac*, which contains the most edges). One clear difference from the previous figure is that the adaptive profiler performs much better when using the hot-edge coverage accuracy metric. This may be because the hottest edges will probabilistically be the first-sampled edges with a particular calling context signature, and therefore do not require further sampling based on re-enabling.

Among all profilers, adaptive with re-enablement is the most stable across all benchmarks, while the simple stack-walking profiler performs inconsistently. The results are especially poor on ap-

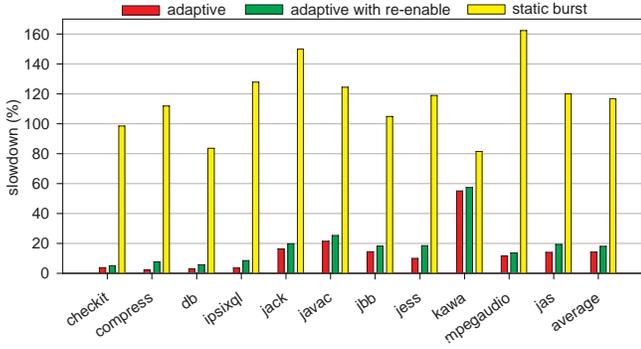


Figure 7. Comparison of slowdown

plications with very flat method profiles, e.g. *checkit*, *kawa*, *javac*, and *JAS*.

The difference in hotness between an edge meeting the hot-edge threshold and an edge missing the threshold is quite small for many applications, resulting in susceptibility to variation for some applications (e.g. *mpegaudio*, *kawa*). In summary, the average hot-edge coverage of each profiler is 52.9%, 79.1%, 88.2%, and 88.1%, respectively.

Finally, we compare the overhead caused by each profiling mechanism in Figure 7. Across all benchmarks, slowdown is a function of the frequency of bursting. Because the static bursting profiler performs the most bursts, its slowdown is naturally greatest, at 117%. As shown in Figure 8, the adaptive with re-enablement profiler performs more bursting than the basic adaptive profiler, and consequently, its slowdown is slightly higher, on average 18.8% compared to 14.8%. The slowdown for sampled stack walking (not shown in this figure) is negligibly small (less than 1%).

In summary, evaluation based upon these parameters shows that with a modest slowdown (around 20%), the JVMPI-based adaptive bursting approach can achieve 85% degree of overlap, and cover 88% of the hottest edges with $threshold = 0.1$. This is significantly more accurate than the previously proposed stack sampling mechanism [6]. The slowdown is incurred partly because of the inefficient JVMPI profiler interface. We expect this to be further reduced with a more efficient instrumentation mechanism.

6.3 Sensitivity Study

In this subsection, we evaluate the sensitivity of the profiling mechanisms to the choice of the burst length, and re-enable ratio, and also investigate each profiler’s performance using the hot-edge coverage metric for varying hotness thresholds.

In Figure 9, we present profiling accuracy while altering two parameters: the burst length and the re-enable ratio. In each small figure, there are four curves corresponding to four burst length: 0.1 ms, 0.2 ms, 0.5 ms and 1.0 ms, with a 10ms sampling rate (denoted $burstlength/samplingrate$). The x -axis consists of up to 5 cases: adaptive, adaptive with $RR = 5\%$, adaptive with $RR = 10\%$, adaptive with $RR = 20\%$ and static bursting. All numbers are averaged across all benchmarks.

As in the previous results, we observe the same trends of increased overlap in Figure 9(a) as we move from the adaptive profiler to the static bursting profiler. As one might expect, as the burst length increases, overlap increases for all schemes because more profiling data is being collected.

The second sensitivity test, shown in Figure 9(b), examines the percentage of bursts that are disabled as a function of re-enable ratio. The four curves show almost no difference because whether or not a burst will be enabled is a function of re-enable ratio,

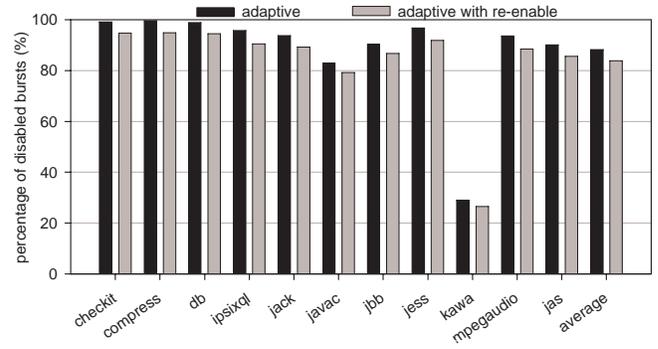


Figure 8. Comparison of the percentage of disabled bursts

not burst length. As one would expect, the percentage of disabled bursts drops dramatically as a function of the re-enable ratio, almost linearly.

Figure 9(c) compares slowdown as this re-enable ratio is adjusted. Of course, because raising the re-enable ratio incurs more bursting, more overhead is created causing a larger slowdown. Comparing this data with Figure 9 (a), we conclude that if we take a modest burst length such as 0.2 ms, and choose a small re-enable ratio, the adaptive profiler with re-enablement offers the best trade-off between accuracy and overhead. We do not show the results for hot edge coverage, because they only vary slightly across configurations. This is perhaps due to the fact that the adaptive approaches always get high coverage and are close to saturation.

In Figure 10, we show how hot edge coverage varies depending on chosen threshold values. The threshold (x -axis) is varied from 0.95 to 0.05 (the higher the threshold, the fewer hot edges included in the range). Due to space limitations, we report this result for SPECjbb alone. The static burst and adaptive profilers perform well across all of these threshold values. We find that the relative performance of the profilers depicted here applies for most applications, although absolute numbers are quite different depending on application behavior (mostly as a function of the flatness of the profile). In general, coverage increases as the threshold decreases, because as the number of edges meeting a threshold grows, the probability that a profiler will correctly discover the edges grows. The coverage dips at certain points, because as the threshold is lowered, the number of methods meeting this threshold jumps abruptly. When a large jump occurs, there is a probability that different profilers will find different hot-edges, depending on variations in sampling timing.

6.4 Degree of Overlap Over Time

As shown in Figure 11, our final experiment tests how well the adaptive profiler (0.2/10, no re-enablement) tracks the complete CCT over time, with respect to degree of overlap. In this figure, the x -axis represents the number of method calls and returns. Because the complete CCT changes over time (we compare the ACCT to the complete CCT at identical points in time), the difference between the two is not necessarily convergent. The two may match perfectly at one point in time. However, a program’s behavior may change (e.g. due to a phase change), causing the adaptive profiler to play catch-up to the complete CCT.

The figure contains two curves, showing the average degree of overlap across the x86 benchmarks (averaged), and *JAS*, respectively. Horizontal lines corresponding to each curve are also included indicating the degree of overlap observed at the end of profiling for that application. Each curve initially increases steeply, then increases more slowly towards their ultimate values. The ini-

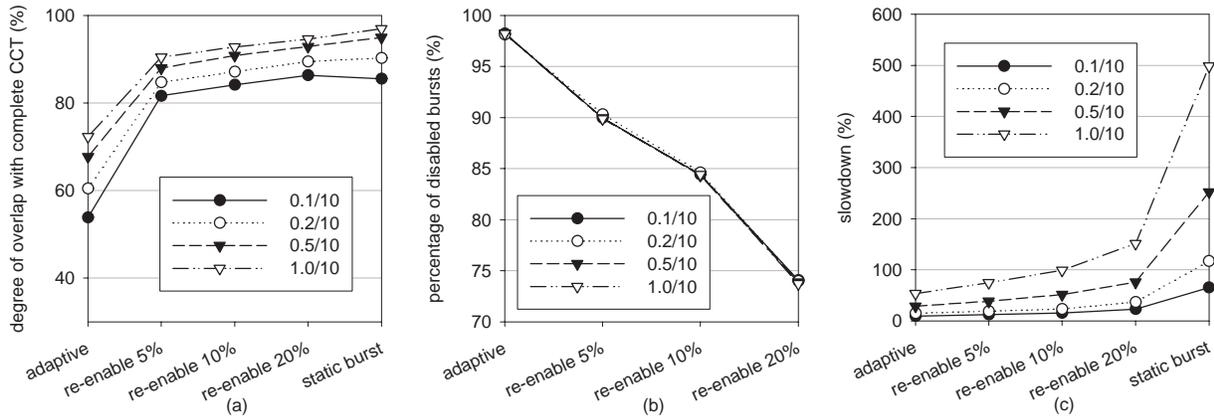


Figure 9. Sensitivity test for a variety of re-enable ratios and bursting lengths

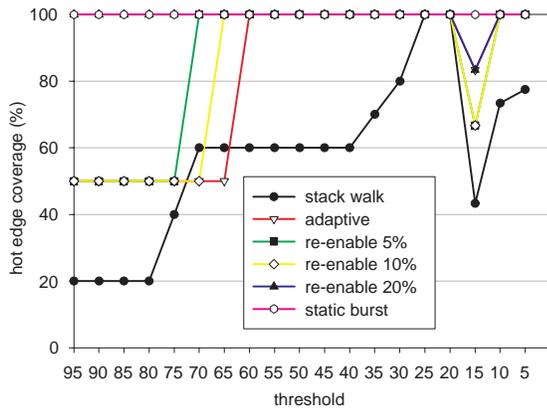


Figure 10. Hot-edge coverage while varying threshold

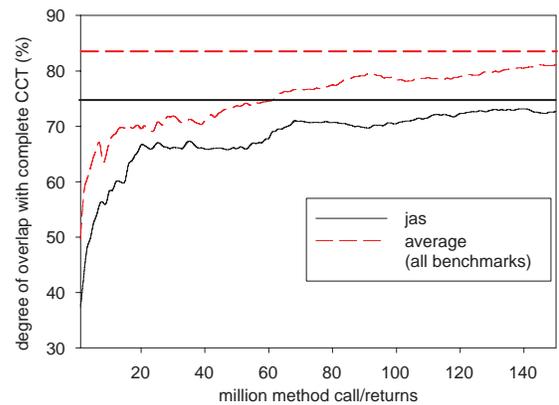


Figure 11. Convergence of the Degree of Overlap

tial jump is mostly within the first second of execution (roughly translated from the number of method calls and returns), and approaches saturation within a few seconds. In future work, we plan to use this convergence phenomenon to further reduce overhead; once CCT updates become rare, profiling can be reduced without sacrificing accuracy.

7. Related Work

There has been an abundance of research related to call graphs, call trees, and calling context trees. For brevity, we discuss only those that are most related to our approach.

Calling context tree profiling has been previously described by many researchers. Both Ammons et al. [3] and Spivey [23] proposed construction of the CCT through instrumentation of all method calls and returns, leading to a large slowdown. Several authors have used a sampling-based approach to obtain an ACCT (also called a *PCCT*) [6, 15, 25]. Although it has low overhead, the sampling-based approach sometimes causes significant loss of accuracy compared with the complete CCT, as shown here. For some applications, accuracy may be high, but its inconsistent results yield low confidence.

There has been extensive research constructing call graph profiles based on sampling, bursting, or a mixture of the two. Call

graphs are relatively easy to capture with high accuracy. Even with sampling based approaches, a high degree of overlap may be achieved. Because the CCT is inherently more valuable than the call graph (e.g. as shown in prior work [19]), a low-overhead profiling mechanism that can generate CCT should be preferable to a call graph profiler of comparable overhead.

Bernat and Miller [8] propose an incremental call path profiling mechanism, which reduces overhead by allowing users to choose methods of interest and generate path profiles including those methods. Aside from targeting a different application (call path profiling or manually selected methods), their instrumentation approach is orthogonal to our approach. We rely on the heavyweight JVMPi interface, but our adaptive-based profiler can tolerate its overheads reasonably well. Combining their lightweight instrumentation with the adaptive approach should strengthen both approaches.

Arnold and Grove [4] present an approach that combines timer-based and counter-based sampling. A timer interrupt triggers the collection of a bursty (potentially non-contiguous, stride-based) sequence of n samples, where each sample records a method call. These n samples are used to construct an approximate call graph used for guiding inlining decisions. A key result of this work is its evidence that timer-based sampling alone produces inaccurate results that are reflected in poor inlining decisions, which has been

one of the biggest motivations for our work. Their paper also describes a clever way to reuse existing method entry checks, so that there is no additional overhead when the instrumentation is disabled. This approach will also be useful in future implementations of our adaptive profiling mechanism.

8. Conclusions

This paper describes and evaluates an adaptive bursting approach to building the calling context tree at runtime. It selectively inhibits redundant bursts, and re-enables a certain percentage of bursts to react to changing application phases, while performing compensation to maintain the relative weights of edges. We have shown that our adaptive approach dramatically reduces profiling overhead compared to the exhaustive and static bursting approaches, while still creating a high accuracy calling context tree. We also demonstrate that a pure stack-walking approach, although quite inexpensive, yields inconsistent results with large variations in accuracy across applications.

Our results show that a JVMPI-based adaptive bursting approach achieves 85% degree of overlap and cover 88% when using 0.1 hot-edge threshold with a modest slowdown (around 20%). We can expect the slowdown to further go down with more efficient instrumentation. In contrast, the overlapping of sampled stack walking is below 50%, and the exhaustive approach incurs 50 times slowdown with its accuracy comparable with our approach.

Despite its simplicity, our adaptive mechanism has been shown to be very effective. We claim that with the use of a more efficient instrumentation mechanism, the overheads of our adaptive profiler will be negligibly small. This will be shown in future work. We also plan to further investigate adaptive mechanisms that make a more informed decision on when to enable and disable bursting, in addition to using the current RR-based stochastic approach. For example, heuristics such as the number of nodes added to the CCT during a burst for a particular stack signature may be a good indicator that some stack signatures should initiate bursts more or less frequently than others. We also believe that our adaptive approach may be applicable to other types of profilers (e.g. path profiling), and plan to investigate its use for other types of collection.

Acknowledgments

We thank Matthew Arnold for donating parts of the code for stack sampling and for extensive conversations about his adaptive scheme. Many thanks to Kristis Makris and Kyung Ryu for the scripts to run SPECjAppServer2004, and to the anonymous reviewers for their invaluable comments.

References

- [1] Colorado bench. http://www-plan.cs.colorado.edu/henkel/projects/colorado_bench.
- [2] Kawa, the java-based scheme system. <http://www.gnu.org/software/kawa>.
- [3] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, 1997.
- [4] M. Arnold and D. Grove. Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines. In *International Symposium on Code Generation and Optimization*, 2005.
- [5] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.
- [6] M. Arnold and P. F. Sweeney. Approximating the calling context tree via sampling. IBM Research Report, July 2000.
- [7] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.
- [8] A. R. Bernat and B. P. Miller. Incremental call-path profiling. Technical report, University of Wisconsin, 2004.
- [9] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W.M. Hwu. Profiled-guided automatic inline expansion for c programs. *Software-Practice and Experience*, 22(5):349–369, 1992.
- [10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge MA and McGraw-Hill, 1990.
- [11] Standard Performance Evaluation Corporation. Specjbb2000 java business benchmark. <http://www.spec.org/jbb2000>.
- [12] Standard Performance Evaluation Corporation. Specjvm98 benchmarks. <http://www.spec.org/jvm98>.
- [13] Standard Performance Evaluation Corporation. SPECjAppServer2004. <http://www.spec.org/jAppServer2004>.
- [14] P. T. Feller. Value profiling for instructions and memory locations. *Masters Thesis CS98-581, University of California San Diego*, April 1998.
- [15] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 81–90, 2005.
- [16] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proceedings OOPSLA '97*, pages 108–124, 1997.
- [17] 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 *Usenix 3rd Virtual Machine Research and Technology Symposium (VM'04)*, 2004.
- [18] R. J. Hall. Call path profiling. In *Proceedings of the 14th International Conference on Software Engineering*, IEEE Computer Society 1992.
- [19] K. Hazelwood and D. Grove. Adaptive Online Context Sensitive Inlining. In *International Symposium on Code Generation and Optimization*, pages 253–264, 2003.
- [20] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.
- [21] U. Holzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–336, 1994.
- [22] P. Kessler S. Graham and M. McKusick. An execution profiler for modular programs. *Software-Practice and Experience*, 13(8):671–685, 1983.
- [23] J. M. Pivey. Fast, accurate call graph profiling. *Software-Practice and Experience*, 34(3):249–264, 2004.
- [24] T. Suganuma, T. Yasue, and T. Nakatani. An empirical study of method inlining for a java just-in-time compiler. In *JVM-02 Java Virtual Machine Research and Technology Symposium*, pages 91–104, 2002.
- [25] J. Whaley. A portable sampling-based profiler for java virtual machines. In *Java Grande*, pages 78–87, 2000.
- [26] T. Yasue, T. Suganuma, H. Komatsu, and T. Nakatani. An efficient online path profiling framework for java just-in-time compilers. In *PACT Conference on Parallel Architectures and Compilation Techniques*, pages 148–158, 2003.