Probabilistic Calling Context *

Michael D. Bond   Kathryn S. McKinley

Department of Computer Sciences
The University of Texas at Austin
{mikebond,mckinley}@cs.utexas.edu

Abstract

Calling context enhances program understanding and dynamic analyses by providing a rich representation of program location. Compared to imperative programs, object-oriented programs use more interprocedural and less intraprocedural control flow, increasing the importance of context sensitivity for analysis. However, prior online methods for computing calling context, such as stack-walking or maintaining the current location in a calling context tree, are expensive in time and space. This paper introduces a new online approach called *probabilistic calling context* (PCC) that continuously maintains a probabilistically unique value representing the current calling context. For millions of unique contexts, a 32-bit PCC value has few conflicts. Computing the PCC value adds 3% average overhead to a Java virtual machine. PCC is well-suited to clients that detect new or anomalous behavior since PCC values from training and production runs can be compared easily to detect new context-sensitive behavior; clients that query the PCC value at every system call, Java utility call, and Java API call add 0-9% overhead on average. PCC adds space overhead proportional to the distinct contexts stored by the client (one word per context). Our results indicate PCC is efficient and accurate enough to use in deployed software for residual testing, bug detection, and intrusion detection.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Monitors, Testing tools

General Terms Reliability, Security, Performance, Experimentation

* This work is supported by an Intel fellowship, NSF CCF-0429859, NSF CCR-0311829, NSF EIA-0303609, DARPA F33615-03-C-4106, Intel, IBM, and Microsoft. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

Keywords Calling Context, Dynamic Context Sensitivity, Probabilistic, Residual Testing, Anomaly-Based Bug Detection, Intrusion Detection, Managed Languages

1. Introduction

Several trends are making it harder for developers to understand, test, debug, and optimize applications. Due to feature demand, one trend is that applications are larger, more complicated, and often combine modules from disparate sources. Another trend is that more software is written in managed languages, such as Java and C# [46]. Development practices in these languages divide functionality into small methods and thus express context as interprocedural control flow rather than intraprocedural control flow. Together the consequences of these trends are that (1) programmers have more difficulty understanding an entire application, (2) exhaustive testing is infeasible and even attaining high testing coverage is challenging, and (3) static analyses are less effective.

As a result, developers are turning to dynamic tools to understand, test, and optimize their programs. *Dynamic calling context* is the sequence of active method invocations that lead to a program location. Previous work in testing [9, 11, 16, 20, 35, 39], debugging and error reporting [18, 31, 36, 41], and security [14, 23] demonstrates its utility. Calling context is powerful because it captures interprocedural behavior and yet is easy for programmers to understand. For example, programmers frequently examine calling context, in the form of error stack traces, during debugging. Untested behavior such as unexercised calling contexts are called *residuals* [37]. If residual calling contexts are observed in deployed software, they are clues to unmet test coverage obligations and potential bugs. Anomalous sequences of calling contexts at system calls can reveal security vulnerabilities [14, 23].

Computing calling context cheaply is a challenge in non-object-oriented languages such as C, and it is even more challenging in object-oriented languages. Compared with C programs, Java programs exacerbate this problem because they generally express more control flow interprocedurally in the call graph, rather than intraprocedurally in the control flow graph. Our results show that Java has more distinct contexts than comparable C programs [2, 42]. For example, large C programs such as GCC and 147.vortex have 57,777 and 257,710 distinct calling contexts respectively [2, 42], but the remaining six SPEC CPU C programs in Ammons et al.'s workload have fewer than 5,000 contexts. In contrast, we find that 5 of 11 DaCapo Java benchmarks [10] contain more than 1,000,000 distinct calling contexts, and 5 others contain more than 100,000 (Section 5).

The simplest method for capturing the current calling context is walking the stack. For example, Valgrind walks the stack at each memory allocation to record its context-sensitive program location, and reports this information in the event of a bug [36, 41]. If the client of calling contexts very rarely needs to know the context, then the high overhead of stack-walking is easily tolerated. An alternative to walking the stack is to build a calling context tree (CCT) dynamically and to track continuously the program's position in the CCT [2, 42]. Unfortunately, tracking the program's current position in a CCT adds a factor of 2 to 4 to program runtimes. These overheads are unacceptable for most deployed systems. Recent work samples hot calling contexts to reduce overhead for optimizations [52]. However, sampling is not appropriate for testing, debugging, or checking security violations since these applications need coverage of both hot and cold contexts.

This paper introduces an approach called *probabilistic calling context* (PCC) that continuously maintains a value that represents the current calling context with very low overhead. PCC computes this value by evaluating a function at each call site. To differentiate calling contexts that include the same methods in a different order, we require a function that is non-commutative. To optimize a sequence of inlined method calls into a single operation, we prefer a function whose composition is cheap to compute. We show the computation $V \leftarrow 3 \times V + cs$ has these properties, where V is the current value of the calling context, and cs is a hash value for the current call site. We show in theory and practice that this function produces a unique value for up to millions of contexts with relatively few conflicts (false negatives). If necessary, a 64-bit PCC value can probabilistically differentiate billions of unique calling contexts.

PCC is well suited to adding context sensitivity to dynamic analyses that detect new or anomalous program behavior such as coverage testing, residual testing, anomaly-based bug detection, and intrusion detection. These clients naturally have a *training* phase, which collects program behavior, and a *production* phase, which compares behavior against training behavior. Calling contexts across runs can be compared easily by comparing PCC values: two different PCC values definitely represent different contexts. Although a new PCC value indicates a new context, the context is not determinable from the value, so PCC walks the stack when it encounters anomalous behavior to report the calling context.

Using Jikes RVM [1, 26], we demonstrate on the DaCapo Benchmarks, SPEC JBB2000, and SPEC JVM98 that continuously computing a 32-bit PCC value adds on average 3% overhead. Clients add additional overhead to query the PCC value at client-specific program points. We approximate the overhead of querying the PCC value by looking up the value in a hash table on each query. Querying at every call in the application increases execution times by an average of 49% and thus is probably only practical at testing time. In several interesting production scenarios, we demonstrate that querying the PCC value frequently is feasible: querying at every system call adds no measurable overhead, at every `java.util` call adds 3% overhead; and examining it at every Java API call adds 9% overhead. Computing the PCC value adds no space overhead, but clients add space overhead proportional to the number of distinct contexts they store (one word per context), which is millions in some cases but still much smaller than all statically possible contexts. In contrast, other approaches use space proportional to all contexts and/or use many words per context. To our knowledge, PCC is the first approach to achieve low-overhead and always-available calling context. We believe this functionality can enable new online client analyses that improve program correctness, reliability, and security.

2. Motivation

This section motivates efficient tracking of calling context for improving testing, debugging, and security. Some previous work shows dynamic context sensitivity helps these tasks [9, 14, 31, 23]. However, most prior work uses intraprocedural paths or no control-flow sensitivity for these tasks [3, 18, 20, 24, 47, 48] since paths are often good enough for capturing program behavior and calling context is too expensive to compute. Because developers more often now choose object-oriented, managed languages such as Java and C# [46], calling context is growing in importance for these tasks. In essence, Java programs use more method invocations (i.e., interprocedural control flow) and fewer control flow paths (i.e., intraprocedural control flow) compared with C programs. This paper seeks to help enable the switch to dynamic context-sensitivity analyses by making them efficient enough for deployed systems.

Testing. Half of application development time is spent in testing [6, 35]. A key part of testing is coverage, and one metric of coverage is exercising unique statements, paths, calling contexts [9], and calling sequences that include the order of calls and returns [20, 39]. *Residual testing* identifies untested coverage, such as paths, that occur at production time but were not observed during testing [37, 47]. PCC is well-suited to context-sensitive residual testing since it identifies new contexts with high probability while adding low enough overhead for deployed software.

Debugging. Identifying program behavior correlated with incorrect execution often helps programmers find bugs. Previous work in *anomaly-based bug detection* (also called *invariant-based bug detection* and *statistical bug isolation*) tracks program behavior such as variables' values across multiple runs to identify behavior that is well-correlated with errors [19, 30, 32]. We are not aware of work that uses calling context for anomaly-based bug detection, although the high time and space overhead may be a factor. Some previous work uses a limited amount of calling context in features in bug detection. Liu et al. use *behavior graphs*, which include call relationships (essentially one level of context sensitivity), to help identify call chains correlated with bugs [31]. *Clarify* uses *call-tree profiling*, which measures two levels of context sensitivity as well as the order of calls, to classify program executions for better error reporting, a task similar to bug-finding [18]. We note that programmers already appreciate the usefulness of calling context in debugging tasks. For example, developers typically start with an error stack trace to diagnose a crash and *Valgrind*, a testing-time tool, reports context-sensitive allocation sites for heap blocks involved in errors [36].

Artemis provides a framework for selectively sampling bug detection instrumentation to keep overhead low [13]. The key idea is to track contexts and to avoid sampling contexts that have already been sampled. Artemis's definition of context includes values of local and global variables but does not include calling context, possibly because of the cost of computing it. PCC makes it viable to add calling context to Artemis.

Security. *Anomaly-based intrusion detection* seeks to detect new attacks by identifying anomalous (i.e., previously unseen) program behavior [14, 24, 48]. Existing approaches typically keep track of system calls and flag system call sequences that deviate from previously-observed behavior and may indicate an attacker has hijacked the application. Wagner and Soto show that attackers can circumvent these approaches by *mimicking* normal application behavior while still accomplishing attacks [48]. Adding context sensitivity to the model of acceptable behavior constrains what attackers can do without getting caught, and recent work on intrusion detection uses calling context to identify program control-flow hijacking [14, 23, 51]. Inoue on page 109 in his dissertation writes the following [23]:

> *Adding context by increasing the number of observed stack frames can make some attacks significantly more difficult. So-called "mimicry" attacks take advantage of the inner workings of applications to attack while still behaving similarly to the attacked application. Adding context makes this more difficult because it restricts the attacker to using only methods usually invoked from within the enclosing method that the exploit attacks, instead of any method invoked by the entire application.*

Zhang et al. show that k-length interprocedural paths gathered with hardware reveal possible security violations [51]. Feng et al. utilize a single level of context sensitivity by including each system call's return address in the sequence of system calls, constraining possible attacks [14].

We show that the expense of walking the stack stands in the way of deployed use of context-sensitive system calls but that PCC permits cheap computation of context sensitivity (Section 5). An intrusion detection system could use PCC to record the calling context for each system call in sequences of system calls. Because PCC is probabilistic, it may incur false negatives if it misses anomalous calling contexts that map to the same value as an already-seen calling context. However, the conflict rate is very low, 0.1% or less for up to 10 million contexts with 32-bit values, and 64-bit values provide even fewer conflicts. A determined attacker with knowledge of PCC could potentially engineer an attack using an anomalous calling context with a conflicting PCC value. We believe randomizing call site values on the host would make a "conflict attack" virtually impossible, although we do not prove it.

In summary, existing work shows dynamic calling context is useful for residual testing, anomaly-based bug detection, and intrusion detection. Trends toward managed languages and more complex applications are likely to make dynamic context sensitivity more essential, and PCC has the potential to help make it feasible.

3. Probabilistic Calling Context

This section describes our approach for efficiently computing a value that represents the current calling context and is unique with high probability.

3.1 Calling Context

The current program location (method and line number) and the active call sites on the stack define dynamic calling context. For example, the first line below is the current program location, and the remaining lines are the active call sites:

```
at com.mckoi.database.jdbcserver.JDBCDatabaseInterface.
      execQuery():213
at com.mckoi.database.jdbc.MConnection.
      executeQuery():348
at com.mckoi.database.jdbc.MStatement.
      executeQuery():110
at com.mckoi.database.jdbc.MStatement.
      executeQuery():127
at Test.main():48
```

3.2 Probabilistic Approach

Probabilistic calling context (PCC) keeps track of an integer value, V, that represents the current calling context. Our goal is to compute random, independent values for each context. To determine the feasibility of this approach, we assume a random number generator and use the following formula to determine the number of expected conflicts given population size n and 32- or 64-bit values [34]:

Random values	Expected conflicts 32-bit values	64-bit values
1,000	0 (0.0%)	0 (0.0%)
10,000	0 (0.0%)	0 (0.0%)
100,000	1 (0.0%)	0 (0.0%)
1,000,000	116 (0.0%)	0 (0.0%)
10,000,000	11,632 (0.1%)	0 (0.0%)
100,000,000	1,155,170 (1.2%)	0 (0.0%)
1,000,000,000	107,882,641 (10.8%)	0 (0.0%)
10,000,000,000	6,123,623,065 (61.2%)	3 (0.0%)

Table 1. Expected conflicts for various populations of random numbers using 32-bit and 64-bit values.

$$E[\mathit{conflicts}] := n - m + m\left(\frac{m-1}{m}\right)^n$$

where m is the size of the value range (e.g., $m = 2^{32}$ for 32-bit values). Table 1 shows the expected number of conflicts for populations ranging in size from one thousand to ten billion. For example, if we choose 10 million random 32-bit numbers, we can expect 11,632 conflicts on average. Applied to the calling context problem, if a program executes 10 million distinct calling contexts, we expect to miss contexts at a rate of just over over 0.1%, which is likely good enough for many clients.

The programs we evaluate execute fewer than 10 million distinct calling contexts (except `eclipse` with the *large* input; Section 5.1). For programs with many more distinct calling contexts, or for clients that need greater probability guarantees, 64-bit values should suffice. For example, one can expect only a handful of conflicts for as many as 10 billion distinct calling contexts.

3.3 Computing Calling Context Values

The previous section shows that assigning randomly-chosen PCC values results in an acceptably small level of conflicts (i.e., distinct calling contexts with the same value). This section introduces an online approach for computing a PCC value that has the following properties:

- PCC values must be distributed roughly randomly so that the number of value conflicts is close to the ideal.
- The PCC value must be deterministic, i.e., a given calling context always computes the same value.
- Computing the next PCC value from the current PCC value must be efficient.

We use a function

$$f(V, cs)$$

where V is the current calling context value and *cs* is the call site at which the function is evaluated. We add instrumentation that computes the new value of V at each call site by applying f as follows:

```
method() {
       int temp = V;      // ADDED: load PCC value
       ...
       V = f(temp, cs_1); // ADDED: compute new value
  cs_1: calleeA(...);      // call site 1
       ...
       V = f(temp, cs_2); // ADDED: compute new value
  cs_2: calleeB(...);      // call site 2
       ...
}
```

We have two requirements for this function: non-commutativity and efficient composability.

Non-commutativity. We have found that our benchmarks contain many distinct calling contexts that differ only in the order of call sites. For example, we want to differentiate calling context ABC from CAB. We therefore require a function that is *non-commutative* and thus computes a distinct value when call sites occur in different orders.

Efficient composability. We want to handle method inlining efficiently and gracefully because of its widespread use in high-performance static and dynamic compilers. For example, suppose method A calls B calls C calls D. If the compiler inlines B and C into A, now A calls D. We want to avoid evaluating f three times before the inlined call to D. By choosing a function whose composition can be computed efficiently ahead-of-time, we can statically compute the *inlined* call site value that represents the sequence of call sites B, C, D.

We use the following non-commutative but efficiently composable function to compute PCC values:

$$f(V, cs) := 3 \times V + cs$$

where $\times$ is multiplication (modulo 2^{32}), and $+$ is addition (modulo 2^{32}). We statically compute *cs* for a call site with a hash of the method and line number.

The function is non-commutative because evaluating call sites in different orders do not give the same value in general:

$$\begin{array}{rcl} f(f(V, cs_A), cs_B) & = & 9 \times V + (3 \times cs_A) + cs_B) \\ \neq \quad f(f(V, cs_B), cs_A) & = & 9 \times V + (3 \times cs_B) + cs_A) \end{array}$$

since in general

$$(3 \times cs_A) + cs_B \quad \neq \quad (3 \times cs_B) + cs_A$$

Non-commutativity is a result of mixing addition and multiplication (which are commutative operations by themselves). At the same time, the function's composition is efficient because addition and multiplication are distributive with respect to each other:

$$\begin{array}{l} f(f(V, cs_A), cs_B) = \\ \quad 3 \times (3 \times V + cs_A) + cs_B = \\ \quad 9 \times V + (3 \times cs_A + cs_B) \end{array}$$

Note that $(3 \times cs_A + cs_B)$ is a compile-time constant, so the composition is as efficient to compute as f.

Gropp and Langou et al. use similar functions to compute hashes for Message Passing Interface (MPI) data types [17, 28]. We experimented with these and other related functions. For example, multiplying by 2 is attractive because it is equivalent to bitwise shift, but bits for methods low on the stack are lost as they are pushed off to the left. Circular shift (equivalent to multiplication by 2 modulo $2^{32}-1$) solves this problem, but when combined with addition modulo $2^{32} - 1$ (necessary to keep efficient composability), we lose information about multiple consecutive recursive calls; i.e., 32 consecutive recursive calls computes $f^{32}(V, cs)$, which for this function is simply V for any V and *cs*.

3.4 Querying Calling Context Values

This section describes how clients can query PCC values at program points. In any given method, V represents the current dynamic context, *except for the position in the currently executing method*. To check V at a given program point, we simply apply f to V using the value of *cs* for the current site (not necessary a call site), i.e., current local method and line number:

```
method() {
     ...
 cs: query(f(V, cs));       // ADDED: query PCC value
     statement_of_interest; // application code
     ...
}
```

PCC is most applicable to clients that detect new or anomalous behavior, which naturally tend to have two modes, *training* and *production*. In training, clients can query PCC values and store them. In production, clients query PCC values and determine if they represent anomalous behavior; if so, PCC walks the stack to determine the calling context represented by the anomalous PCC value. Many anomalous contexts in production could add high overhead because each new context requires walking the stack. However, this case should be uncommon for a well-trained application.

4. Implementation

PCC's approach is suitable for implementation in ahead-of-time or dynamic compilation systems. This section describes how we implement PCC in Jikes RVM 2.4.6, a high-performance Java-in-Java virtual machine [4, 26]. PCC is publicly available on the Jikes RVM Research Archive [27].

Jikes RVM uses two compilers at run time. When a method first executes, Jikes RVM compiles it with a non-optimizing *baseline* compiler. When a method becomes hot, Jikes RVM recompiles it with an *optimizing* compiler at successively higher levels of optimization. We modify both compilers to insert PCC instrumentation.

Computing the PCC value. PCC adds instrumentation to maintain V that computes $f(V, cs)$ at each call site, where *cs* is an integer representing the call site. PCC could assign each call site a random integer using a lookup table, but this approach adds space overhead and complicates comparing PCC values across runs. Instead, PCC computes a hash of the call site's method name, declaring class name, descriptor, and line number. This computation is efficient because it occurs once at compile time and produces the same results across multiple program executions.

Implicitly, V is a global variable modified at each call site. To implement PCC in the context of multiple threads and processors, we use per-thread PCC values. Since multiple threads map to a single processor, each processor keeps track of the PCC value for the current thread. When a processor switches threads, it stores the PCC value to the outgoing thread and loads the PCC value from the incoming thread. Accessing the PCC value is efficient in Jikes RVM because it reserves a register for per-processor storage. In systems without efficient access to per-processor storage, an implementation could modify the calling conventions to add the PCC value as an implicit parameter to every method. While this alternative approach is elegant, we did not implement it because it would require pervasive changes to Jikes RVM.

To compute PCC values, the compiler adds instrumentation that (1) at the beginning of each method, loads V into a local variable, (2) at each call site, computes the next calling context with f and updates the global V, and (3) at the method return, stores the local copy back to the global V (this redundancy is helpful for correctly maintaining V in the face of exception control flow). At inlined call sites, the compiler combines multiple call site values ahead-of-time into a single value and inserts a function that is an efficient composition of multiple instances of f (Section 3.3).

Querying the PCC value. Clients may query PCC values at different program points, and they may use PCC values differently. For example, an intrusion detection client might query the PCC value at each system call, recording sequences of consecutive context-sensitive program locations (in the form of PCC values) during training, then detecting anomalous sequences during production. A client performs work per query that is likely to be similar to hash table lookup, so our implementation looks up the PCC value in a global hash table at each query point. The hash table implements *open-address hashing* and *double hashing* [12] using an array of 2^k 32-bit slots. To look up a PCC value, the *query* indexes the array using the low k bits of V, and checks if the indexed slot contains V. In the common case, the slot contains V, and no further action is needed. In the uncommon case, either (1) the slot is empty (contains zero), in which case PCC stores V in the slot; or (2) the slot holds another PCC value, in which case the *query* performs secondary hashing by advancing $s + 1$ slots where s is the high $32 - k$ bits of V. Secondary hashing tries three times to find a non-conflicting slot, then gives up.

For efficiency, we inline the common case into hot, optimized code. For simplicity in our prototype implementation, we use a fixed-size array with $2^{20} = 1,048,576$ elements (4 MB of space), but a more flexible implementation would adjust the size to accommodate the number of stored PCC values collected during training (e.g., intrusion detection clients could use much less space since there are relatively few distinct contexts at system calls). The hash table approach is efficient as long as the table size is roughly at least twice the size of the number of entries in the table, or table conflicts will lead to high overhead. Of our benchmarks, *pmd* queries the most distinct PCC values, over 800,000 at Java API calls (Table 3), for which a hash table with a million elements is probably not quite large enough for good performance. We also measure the overhead of querying the PCC value at every call as an upper bound for a PCC client (Figure 1); for several benchmarks with millions of distinct contexts, the hash table is not large enough, resulting in many hash table lookup failures, but the time overhead should still be a representative upper bound.

Defining calling context. Our implementation distinguishes between VM methods (defined in Jikes RVM classes), Java library methods (`java.*` classes), and application classes (all other classes). The implementation does not consider VM and library call sites to be part of calling context, since call sites in these methods are probably not interesting to developers and are often considered "black boxes." All application methods on the stack are considered part of the calling context, even if VM or library methods are above them. For example, container classes often access application-defined `equals()` and `hashCode()` methods:

```
at result.Value.equals():164
at java.util.LinkedList.indexOf():406
at java.util.LinkedList.contains():176
at option.BenchOption.getFormalName():80
at task.ManyTask.main():46
```

Our implementation considers this context to be simply

```
at result.Value.equals():164
at option.BenchOption.getFormalName():80
at task.ManyTask.main():46
```

Similarly, sometimes the application triggers the VM, which calls the application, such as for class initialization:

```
at dacapo.TestHarness.<clinit>():57
at com.ibm.JikesRVM.classloader.VM_Class.
     initialize():1689
at com.ibm.JikesRVM.VM_Runtime.
     initializeClassForDynamicLink():545
at com.ibm.JikesRVM.classloader.
     VM_TableBasedDynamicLinker.resolveMember():65
at com.ibm.JikesRVM.classloader.
     VM_TableBasedDynamicLinker.resolveMember():54
at Harness.main():5
```

Our implementation considers this context to be

```
at dacapo.TestHarness.<clinit>():57
at Harness.main():5
```

PCC implements this definition of calling context. PCC instruments application methods only, and in these methods it instruments call sites to application and library methods. In cases where the application calls the VM directly, and the VM then invokes the application (e.g., for class initialization), PCC walks the stack to determine the correct value of V, which is feasible because it happens infrequently.

5. Results

This section evaluates the performance and accuracy of probabilistic calling context (PCC). The methodology subsection first describes *deterministic* calling context profiling, which we use to measure the accuracy of PCC, and experimental configurations, benchmarks, and platform. Then we present the query points we evaluate, which correspond to potential clients of PCC. Next we evaluate PCC's accuracy at identifying new contexts at these query points, then measure PCC's time and space performance and compare it to walking the stack. Finally we evaluate PCC's ability to identify new contexts not observed in a previous run and the power of calling context to detect new program behavior not detectable with context-insensitive profiling.

5.1 Methodology

Deterministic calling context profiling. To evaluate the accuracy of PCC and to collect other statistics, we also implement *deterministic* calling context profiling. Our implementation constructs a calling context tree (CCT) and maintains the current position in the CCT throughout execution. Our implementation is probably less time and space efficient than the prior work (Section 6) because (1) it collects per-node statistics during execution, and (2) for simplicity, we modify only the non-optimizing baseline compiler and disable the optimizing compiler for these experiments only. Since we only use it to collect statistics, we are not concerned with its performance.

VM configurations. Jikes RVM runs by default using *adaptive* methodology. Initially it uses a baseline non-optimizing compiler. Then it dynamically identifies frequently-executed methods and recompiles them at higher optimization levels. Because Jikes RVM uses timer-based sampling to detect hot methods, the adaptive compiler is nondeterministic. To measure performance, we use *replay compilation* methodology, which is deterministic [22]. Replay compilation forces Jikes RVM to compile the same methods in the same order at the same point in execution on different executions and thus avoids high variability due to sample-driven compilation.

Replay compilation uses *advice files* produced by a previous well-performing adaptive run (best of five). The advice files specify (1) the optimization level for compiling each method, (2) the dynamic call graph profile, and (3) the edge profile. Fixing these inputs, we execute two consecutive iterations of the application. During the first iteration, Jikes

RVM optimizes code using the advice files. The second iteration executes only the application with a realistic mix of unoptimized and optimized code.

We execute performance results using a generational mark-sweep collector and a fixed heap size of three times the minimum for each benchmark. We report the minimum of five trials since it represents the deterministic run least perturbed by external effects.

Benchmarks. We use the DaCapo benchmarks (version 2006-10), a fixed-workload version of SPEC JBB2000 called `pseudojbb`, and SPEC JVM98 [10, 43, 44]. We exclude `xalan` from performance results because we could not get it to run correctly with replay compilation, with or without PCC. We use the *large* input size for all performance and statistics runs, except we use *medium* for `eclipse`'s statistics runs since with *large* our deterministic calling context implementation runs out of memory: `eclipse`'s *large* input executes at least 41 million distinct contexts.

Platform. We perform experiments on a 3.6 GHz Pentium 4 with a 64-byte L1 and L2 cache line size, a 16KB 8-way set associative L1 data cache, a 12Kμops L1 instruction trace cache, a 2MB unified 8-way set associative L2 on-chip cache, and 2 GB main memory, running Linux 2.6.12.

5.2 Potential PCC Clients

PCC continuously keeps track of a probabilistically unique value that represents the current dynamic calling context. To evaluate PCC's use in several potential clients, we query the PCC value at various program points corresponding to these clients' needs.

System calls. Anomaly-based security intrusion detection typically collects sequences of system calls, and adding context-sensitivity can strengthen detection (Section 2). To explore this potential client, we add a call to PCC's *query* method before each system call, i.e., each call that can potentially throw a Java security exception. The callees roughly correspond to operations that can affect external state, e.g., file system I/O and network access. Our benchmarks range in behavior from very few to many system calls. Programs most prone to security intrusions, such as web servers, are likely to have many system calls.

Java utility calls. Residual testing seeks to determine whether behavior seen at production time deviates from behavior seen at testing time [37, 47]. Residual testing of a software component at production time detects if the component is called from a new, untested context. These contexts may indicate errors in the application or poor test coverage. While application developers often perform residual testing on a component of their own application, we use the Java *utility* libraries as a surrogate for exploring residual testing on a component library. These libraries provide functionality such as container classes, time and date conversions, and random numbers. This choice is justified because these libraries are heavily used by our benchmarks and other programs. At each call to a `java.util.*` method, instrumentation queries the PCC value.

Java API calls. We also explore residual testing using the Java *API* libraries as a surrogate. This library is a superset of `java.util`. We add instrumentation to query the PCC value at each call to a method in `java.*`. This simulates residual testing of a larger component, since calls to `java.*` methods, especially `java.lang.*` methods, are extremely frequent in most Java programs (e.g., all `String` operations are in the API libraries). The results for Java API calls show that PCC scales well to a frequently-used component invoked from many distinct contexts.

All calls. In addition, we evaluate querying the PCC value at every call site. This configuration would be useful for measuring code coverage and generating tests with good code coverage [9, 20, 39], and it represents an upper bound on overhead for possible PCC clients. We find querying PCC values at every call is too expensive for deployed use but can speed up testing time compared with walking the stack.

5.3 PCC Accuracy

Table 2 shows calling context statistics for the first three potential clients from the previous section. *Dynamic* is the number of dynamic calls to *query*. For example, for system calls, *Dynamic* is the dynamic number of system calls. *Distinct* is the number of distinct calling contexts that occur at query points. *Conf.* is the number of PCC value conflicts that occur for these calling contexts. Conflicts indicate when PCC maps two or more distinct calling contexts to the same value (k contexts mapping to the same value count as $k - 1$ conflicts). We summarize the dynamic and distinct counts using geometric mean.

The benchmarks show a wide range of behavior with respect to system calls. Seven benchmarks perform more than 1,000 *dynamic* system calls, and two benchmarks (`antlr`, `jython`) exercise more than 1,000 distinct contexts at system calls. No PCC value conflicts occur between contexts.

As expected, the programs make significantly more calls into the utility libraries and the entire Java API. For the utility libraries, *dynamic* calls range from about a thousand for several SPEC JVM98 benchmarks to a billion for `bloat`, and the number of unique contexts ranges from 176 to 442,845. For the Java API, the *dynamic* calls are up to 2 billion for `xalan`, and distinct contexts range from 905 to 847,108. These potential clients will therefore require many PCC value queries, but as we show in the next section, PCC is efficient even with this high load. The numerous zero entries in the Conf. columns show that PCC is completely accurate in many cases. The conflicts are low—at most 79 for `pmd`'s 847,108 distinct contexts at API calls—and are consistent with the ideal values from Table 1.

Table 3 presents calling context statistics for *all* executed contexts, as well as average and maximum call depth. To

Program	System calls Dynamic	Distinct	Conf.	Java utility calls Dynamic	Distinct	Conf.	Java API calls Dynamic	Distinct	Conf.
antlr	211,490	1,567	0	698,810	8,010	0	24,422,013	128,627	3
bloat	12	10	0	1,030,955,346	143,587	3	1,159,281,573	600,947	40
chart	63	62	0	43,345,653	44,502	0	258,891,525	202,603	4
eclipse	14,110	197	0	3,958,510	54,175	0	132,507,343	226,020	5
fop	18	17	0	5,737,083	25,528	0	9,918,275	37,710	0
hsqldb	12	12	0	90,324	267	0	81,161,541	16,050	0
jython	5,929	4,289	0	76,150,625	131,992	2	543,845,772	628,048	48
luindex	2,615	14	0	5,437,548	1,024	0	39,733,214	102,556	0
lusearch	141	11	0	23,183,861	176	0	113,511,311	905	0
pmd	1,045	25	0	372,159,946	442,845	24	537,017,118	847,108	79
xalan	137,895	59	0	744,311,518	6,896	0	2,105,838,670	17,905	0
DaCapo geo	843	60		19,667,815	12,689		163,072,787	85,963	
pseudojbb	507,326	145	0	18,944,200	475	0	30,340,974	3,410	0
compress	7	5	0	1,018	682	0	8,138	1,081	0
jess	50	6	0	4,851,299	2,061	0	16,487,052	5,240	0
raytrace	7	5	0	1,078	684	0	5,331,338	3,383	0
db	7	5	0	65,911,710	767	0	90,130,132	1,439	0
javac	7	5	0	6,499,455	55,994	0	24,677,625	255,334	4
mpegaudio	7	5	0	874	682	0	7,575,084	1,668	0
mtrt	7	5	0	880	682	0	5,573,455	3,366	0
jack	7	5	0	14,987,342	14,718	0	21,771,285	29,461	0
SPEC geo	30	7		199,386	1,724		7,074,200	5,410	
Geomean	188	23		2,491,316	5,168		39,734,213	24,764	

Table 2. Statistics for calling contexts at several subsets of call sites. Dynamic and distinct contexts, and PCC value conflicts, for (1) system calls, (2) Java utility calls, and (3) Java API calls.

Program	All contexts Dynamic	Distinct	Conf.	Call depth Avg	Max
antlr	490,363,211	1,006,578	118	21.5	164
bloat	6,276,446,059	1,980,205	453	30.6	167
chart	908,459,469	845,432	91	16.6	29
eclipse	1,266,810,504	4,815,901	2,652	15.0	102
fop	44,200,446	174,955	2	22.4	49
hsqldb	877,680,667	110,795	1	19.3	36
jython	5,326,949,158	3,859,545	1,738	58.3	223
luindex	740,053,104	374,201	12	19.4	34
lusearch	1,439,034,336	6,039	0	15.2	24
pmd	2,726,876,957	8,043,096	7,653	28.9	416
xalan	10,083,858,546	163,205	6	19.4	63
DaCapo geo	1,321,327,982	562,992		22.3	78
pseudojbb	186,015,473	19,709	0	7.1	25
compress	451,867,672	1,518	0	13.6	17
jess	198,606,454	18,021	0	43.1	83
raytrace	557,951,542	21,047	0	6.7	18
db	91,794,359	2,118	0	13.0	18
javac	135,968,813	2,202,223	544	29.5	122
mpegaudio	218,003,466	7,576	0	21.9	26
mtrt	564,072,400	21,040	0	6.7	18
jack	35,879,204	82,514	1	22.3	49
SPEC geo	200,039,740	20,695		14.8	32
Geomean	565,012,654	127,324		18.6	52

Table 3. Statistics for every calling context executed. Dynamic and distinct contexts, PCC value conflicts, and average and maximum size (call depth) of dynamic contexts.

profile every context, we query calling context at every call site, as well as every method prologue in order to capture leaf calls. The table shows six programs execute over one million distinct contexts and another five over one hundred thousand contexts. The last two columns show that programs spend a lot of time in fairly deep call chains: average call depth is almost 20, and maximum call depth is over 100 for several benchmarks.

5.4 PCC Performance

This section evaluates PCC's run-time performance. We evaluate PCC alone without a client and also measure the additional cost of using PCC with four sets of query points corresponding to potential clients (Section 3.4). These experiments report application time only using replay compilation, which produces a deterministic measurement.

Figure 1 shows the run-time overhead of PCC, normalized to *Base*, which represents execution without any instrumentation. *PCC* is the execution time of PCC alone: instrumentation keeps track of the PCC value throughout execution but does not use it. The final four bars show the execution time of examining the PCC value at call sites correspond to potential clients: system calls, Java utility calls, Java API calls, and all calls. PCC actually improves `chart`'s performance, but this anomaly is most likely because of architectural sensitivities due code modifications that may affect the trace cache and branch predictor.

PCC by itself adds only 3% on average and 9% at most (for `hsqldb`). Since system calls are relatively rare, checking the context at each one adds negligible overhead on average. PCC value checking at Java utility and API calls adds 2% and 9% on average over PCC tracking, respectively, which is interesting given the high frequency of these calls (Table 2). The highest overhead is 47%, for `bloat`'s API calls.

Compilation overhead. By adding instrumentation to application code, PCC increases compilation time. We measure compilation overhead by measuring time spent in the baseline and optimizing compilers during the first iteration of replay compilation. Figure 2 shows compilation time overhead. PCC instrumentation alone adds 18% compilation overhead on average. Adding instrumentation to query the PCC value increases compilation time by an additional 0-31% for system, utility, and API calls, and up to 150% for all calls, although this overhead could be reduced by not inlining the query method. Per-phase compiler timings show that most of the compilation overhead comes from compiler phases downstream from PCC instrumentation, due to the bloated intermediate representation (IR). Although PCC adds a greater percentage to compilation than application time, many adaptive optimizers control the fraction of time spent in compilation. Therefore by design, compilation time is a small fraction of overall execution time. In our experiments, the time spent in the application is on average 20 times greater than the time spent in compilation, for each of the PCC configurations shown in Figure 2.

Space overhead. Computing the PCC does not add space overhead to keep track of the PCC value, but of course the clients use space proportional to the number of PCC values they store. Our experiments that test potential clients simply use a fixed-size hash table with $2^{20} = 1,048,576$ slots (4 MB), as described in Section 4, but real clients would use space proportional to their needs. Clients storing PCC values in a large data structure could potentially hurt execution time due to poor access locality.

PCC also adds space overhead by increasing the size of generated machine code. We find that on average, PCC instrumentation adds 18% to code size. Adding instrumentation to query the PCC value at system calls, utility calls, API calls, and all calls adds an additional 0%, 2%, 6%, and 14%, respectively.

Comparison with stack-walking. An alternative to PCC is to walk the stack at each query point (Section 6). We evaluate here how well stack-walking performs for the call sites corresponding to potential clients. We implement stack-walking by simply calling a method that walks the entire stack at each query point; we do not add any PCC instrumentation for these runs. Stack-walking implementations would typically look up a unique identifier for the current context [36], and they could save time by walking only the subset of calls occurring since the last walk [49], but we do not model these costs here.

Figure 3 shows the execution time overhead of walking the stack at various points corresponding to three potential clients: system calls, Java utility calls, and Java API calls (we omit "all calls" because its overhead is greater than for Java API calls, which is very high). Since most benchmarks have few dynamic system calls, stack-walking adds negligible overhead at these calls. However, for the two benchmarks with more than 200,000 dynamic system calls, `antlr` and `pseudojbb`, stack-walking adds 67% and 62% overhead, respectively. These results show the substantial cost of walking the stack even for something as infrequent as system calls. Applications prone to security attacks such as web servers are likely to have many system calls.

5.5 Comparing Calling Contexts Between Runs

The previous sections demonstrate that PCC is an efficient mechanism for identifying new context-sensitive behavior in a variety of potential clients. This section explores the sensitivity of calling context behavior to different program inputs by comparing calling context profiles between two runs of each benchmark, the *large* and *medium* input sets. We note that while some benchmarks execute many new distinct contexts with the large input not seen with medium (which is not surprising since it is a larger workload by design), we do not expect so much new behavior in production mode on well-trained applications. Nonetheless, the results are

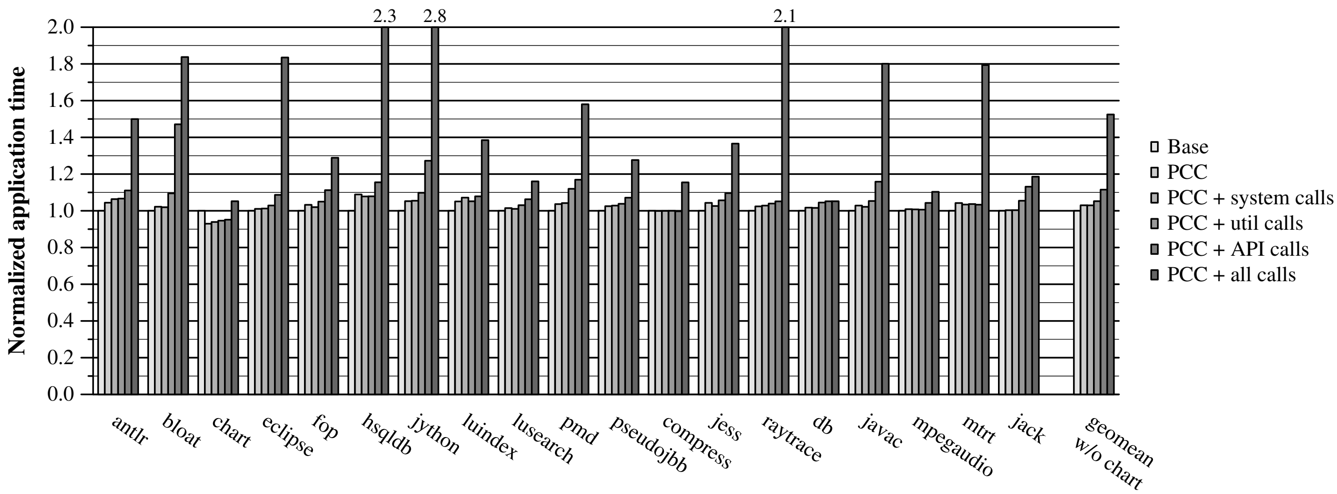

Figure 1. Application execution time overhead of maintaining the PCC value and querying it at (1) system calls, (2) Java utility calls, (3) Java API calls, and (4) all calls.

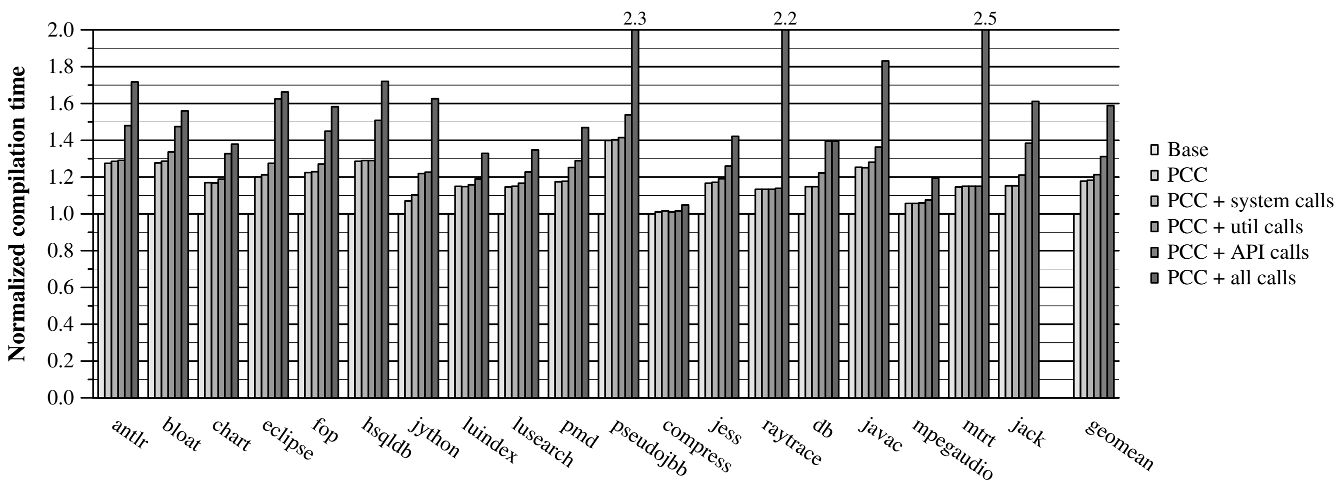

Figure 2. Compilation time overhead due to adding instrumentation to maintain the PCC value and query it at (1) system calls, (2) Java utility calls, (3) Java API calls, and (4) all calls.

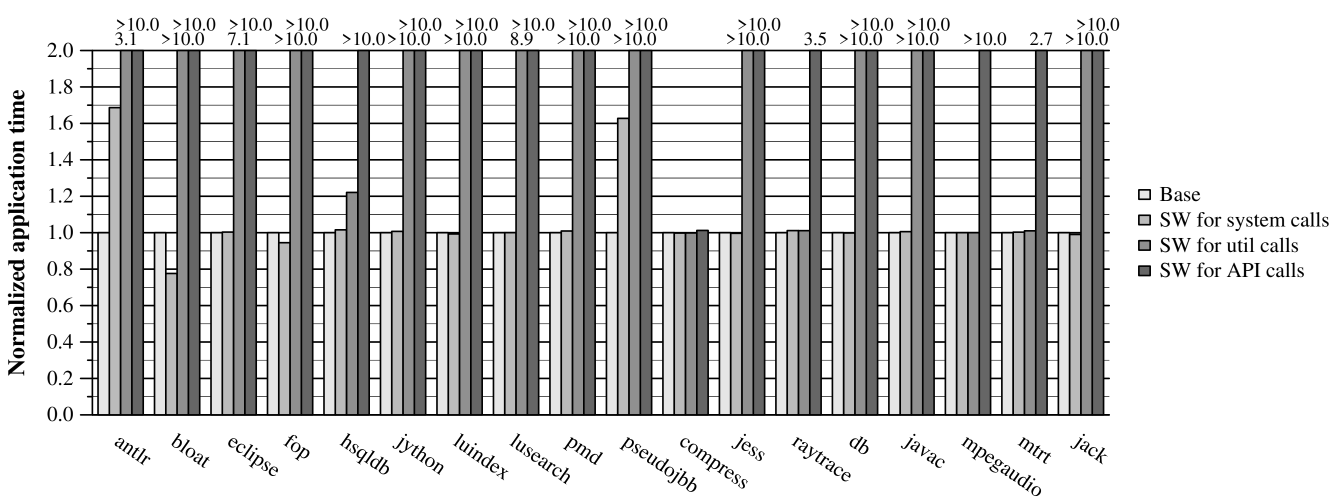

Figure 3. Application execution time overhead of walking the stack at (1) system calls, (2) Java utility calls, and (3) Java API calls.

Program	Java utility calls: Dyn.	Java utility calls: New distinct	Java utility calls: Conf.	Java API contexts: Dyn.	Java API contexts: New distinct	Java API contexts: Conf.
	Relative increase of large input compared to medium input					
antlr	2.5x	0 (0.0%)	0	2.5x	8 (0.0%)	0
bloat	11.8x	74,536 (51.9%)	3	10.2x	320,864 (53.4%)	33
chart	2.3x	31,419 (70.6%)	0	2.5x	139,599 (68.9%)	4
eclipse*	4.2x	15,114 (27.9%)	0	5.8x	121,939 (54.0%)	4
fop	1.0x	0 (0.0%)	0	1.0x	0 (0.0%)	0
hsqldb	6.0x	0 (0.0%)	0	2.5x	13 (0.1%)	0
jython	7.5x	12,705 (9.6%)	0	7.3x	59,202 (9.4%)	5
luindex	1.0x	0 (0.0%)	0	1.0x	7,398 (7.2%)	0
lusearch	2.0x	0 (0.0%)	0	2.0x	0 (0.0%)	0
pmd	4.4x	368,862 (83.3%)	24	4.4x	711,223 (84.0%)	79
xalan	10.0x	0 (0.0%)	0	10.0x	15 (0.1%)	0
Dacapo geo	3.5x			3.3x		
compress	1.1x	0 (0.0%)	0	2.2x	0 (0.0%)	0
jess	61.5x	530 (25.7%)	0	56.7x	1,827 (34.9%)	0
raytrace	1.0x	0 (0.0%)	0	11.0x	25 (0.7%)	0
db	71.6x	25 (3.3%)	0	61.4x	72 (5.0%)	0
javac	39.9x	36,419 (65.0%)	0	36.9x	163,916 (64.2%)	6
mpegaudio	1.0x	0 (0.0%)	0	10.9x	32 (1.9%)	0
mtrt	1.0x	0 (0.0%)	0	7.7x	25 (0.7%)	0
jack	8.5x	0 (0.0%)	0	8.5x	0 (0.0%)	0
SPEC geo	6.0x			14.7x		
Geomean	4.4x			6.2x		

Table 4. Comparing calling contexts at API calls between large and medium inputs. *Medium vs. small for eclipse.

interesting because they give an indication of how calling context behavior differs from one run to the next and how well PCC identifies new context-sensitive behavior.

Table 4 compares the calling contexts at Java utility and Java API calls for runs with the medium and large inputs. We do not show data for system calls since they vary very little between medium and large inputs: only five benchmarks execute new calling contexts at system calls (`chart` executes the most, 38). We omit `pseudojbb` since it has only one input size. We use small and medium for `eclipse` (Section 5.1).

In the table, *Dyn.* is the factor increase of dynamic calling contexts in the large run vs. the medium run. All DaCapo programs except `fop` and `luindex` exercise substantially more dynamic calls to the utility libraries and to the whole Java API. *New distinct* is the number of new distinct calling contexts occurring in the large but not in the medium run, and the percentage is the value relative to all distinct calling contexts from the large run. *Conf.* is the number of PCC value conflicts that occur when adding the new PCC values seen only in the large run, to the set of PCC values seen in the medium run. For example, `pmd` executes more than four times as many dynamic Java API calls in the large as in the medium run, and 711,223 distinct contexts occur in the large run that were not observed in the medium run. These contexts account for 84% of the large run's distinct contexts; and PCC values for these new calling contexts have 79 conflicts when added to the PCC values from the medium run, so the probability of a particular new calling context not being identified as new is about 1 in 10,000. Eleven of the programs execute few if any new distinct calling contexts, even if they execute many times more dynamic calls to Java utility and API methods, while five benchmarks execute hundreds of thousands of new distinct calling contexts. The programs generate about 1 conflict for every 10,000 new contexts at worst, which should be a reasonable false negative rate for most clients.

5.6 Evaluating Context Sensitivity

This section compares calling context profiling to *call site profiling*, which is context *insensitive*, to evaluate whether calling context detects significantly more previously unobserved behavior than call sites alone. Table 5 compares calling contexts and call sites. The first two columns are counts of distinct calling contexts and call sites for calls to Java API methods (the calling context figures are the same as in Table 2). For most programs, there are many more calling contexts than call sites, which indicates that call sites are invoked from multiple calling contexts. The first two columns show that thousands of call sites generate hundreds of thousands of calling contexts.

Program	Large input		Large-medium diff		Contexts w/ new
	Contexts	Call sites	Contexts	Call sites	call sites
antlr	128,627	4,184	8	0	0
bloat	600,947	3,306	320,864	82	1,002
chart	202,603	2,335	139,599	379	9,112
eclipse*	226,020	9,611	121,939	1,240	46,206
fop	37,710	2,225	0	0	0
hsqldb	16,050	947	13	0	0
jython	628,048	1,830	59,202	1	1
luindex	102,556	654	7,398	0	0
lusearch	905	507	0	0	0
pmd	847,108	1,890	711,223	48	388
xalan	17,905	1,530	15	2	2
Dacapo geo	85,963	1,897			
pseudojbb	3,410	846	17	0	0
compress	1,081	1,017	0	0	0
jess	5,240	1,363	1,827	22	22
raytrace	3,383	1,215	25	2	5
db	1,439	1,105	72	4	4
javac	255,334	1,610	163,916	9	201
mpegaudio	1,668	1,072	32	1	4
mtrt	3,366	1,190	25	2	5
jack	29,461	2,173	0	0	0
SPEC geo	5,410	1,242			
Geomean	24,764	1,568			

Table 5. Comparing call site profiles with calling context on Java API calls. *Medium vs. small inputs for eclipse.

Finally, we consider the power of residual calling context profiling compared to residual call site profiling on the medium versus the large inputs. Columns under *Large-medium diff* count the distinct calling contexts and call sites seen in a large run but not a medium run. In several programs many new distinct calling contexts occur, but many fewer new call sites occur, and `luindex` in particular executes 7,398 new contexts without executing any new call sites. The final column shows the number of new, distinct calling contexts that correspond to the new call sites in the large run. This column shows how well residual call site profiling would do at identifying new calling context behavior. If every new call site (i.e., call site seen in large but not medium run) triggered stack-walking, call site profiling would identify only a small fraction of the new calling contexts for most programs.

6. Related Work

This section discusses related work in calling context profiling. It first considers stack-walking, then heavyweight approaches that construct a calling context tree (CCT), and finally sampling-based approaches. We also consider related forms of profiling.

Walking the stack. One approach for identifying the current calling context is to walk the program stack, then look up the corresponding calling context identifier in a calling context tree (CCT) [36, 41]. Unfortunately, walking the stack more than very infrequently is too expensive for production environments, as shown in Section 5.4.

Calling context tree. An alternative approach to walking the stack is to build a dynamic calling context tree (CCT) where each node in the CCT is a context, and during execution maintain the current position in the CCT [2, 42]. This instrumentation slows C programs down by a factor of 2 to 4. The larger number of contexts in Java programs and the compile-time uncertainty of virtual dispatch further increase CCT time and space overheads. The size of CCT nodes are 100 to 500 bytes in previous work, whereas PCC values are very compact in comparison, since each one only needs 32 or 64 bits, and storing them in a half-full hash table achieves good run-time performance, as shown in Section 5.4.

Sampling-based approaches. Sampling-based and truncation approaches keep overhead low by identifying the calling context infrequently [8, 15, 21, 49, 52]. Clients use hot context information for optimizations such as context-sensitive inlining [21] and context-sensitive allocation sites for better object lifetime prediction and region-based allocation [25, 40]. Hazelwood and Grove sample the stack periodically to collect contexts to drive context-sensitive inlining [21]. Zhuang et al. improve on sampling-based stack-walking by performing *bursty* profiling after walking the stack, since it is relatively cheap to update the current position in the CCT on each call and return for a short time [52]. Bernat and Miller limit profiling to a subset of methods [8]. Froyd et al. use unmodified binaries and achieve extremely low overhead through stack sampling [15]. Sampling is useful for identifying hot calling contexts, but it is not suitable for clients such as testing, security, and debugging because sampling sacrifices coverage, which is key for these clients.

Although PCC primarily targets clients requiring high coverage, it could potentially improve the accuracy-overhead trade-off of sampling-based approaches. Zhuang et al.'s calling context profiling approach avoids performing bursty sampling at already-sampled calling contexts [52]. Currently they walk the stack to determine if the current context has been sampled before, but instead they could use PCC to quickly determine, with high probability, if they have already sampled a calling context.

Dynamic call graph profiling. Dynamic optimizers often profile call edges to construct a dynamic call graph (DCG) [5, 29, 38, 45], which informs optimizations such as inlining. DCGs lack context sensitivity and thus provide less information than calling context profiles.

Path profiling. Ball-Larus path profiling computes a unique number through each possible path in the control flow graph [7]. An intriguing idea is applying path profiling instrumentation to the dynamic call graph and computing a unique number for each possible context. However, this approach is problematic because call graphs, which have thousands of nodes for our benchmarks, are typically much larger than control flow graphs (CFGs). The number of possible paths both through CFGs and call graphs is exponential in the size of the graph in practice, so the statically possible contexts cannot be assigned unique 32- or even 64-bit values. Other challenges include: (1) recursion leads to cyclic graphs; (2) dynamic class loading modifies the graph at run time; and (3) virtual dispatch obscures call targets and complicates call edge instrumentation. Wiedermann computes a unique number per context at run time by applying Ball-Larus path numbering to the call graph, but does not evaluate whether large programs can be numbered uniquely [50]. His approach uses C programs, avoiding the challenges of dynamic class loading and virtual dispatch, and handles recursion by collapsing strongly-connected components in the call graph. Melski and Reps present *interprocedural path profiling* that captures both inter- and intraprocedural control flow, but their approach does not scale because it adds complex call edge instrumentation, and there are too many statically possible paths for nontrivial programs [33].

As Section 2 points out, much prior work uses path profiling to understand dynamic behavior in testing, debugging, and security, but dynamic object-oriented languages need calling context, too, since it captures important behavior. Paths and calling contexts are largely orthogonal since paths capture *intraprocedural* control flow while calling context provides *interprocedural* control flow. One could imagine combining PCC and path profiling for best-of-both-worlds approaches in residual testing, anomaly-based bug detection, and intrusion detection.

7. Conclusion

Complex object-oriented programs motivate calling context as a program behavior indicator in residual testing, anomaly-based bug detection, and security intrusion detection. Previous techniques are too expensive for use in production environments. We present a probabilistic calling context (PCC) approach suited to detecting new behavior that is efficient enough to use in deployed systems. PCC maintains a value representing the current calling context in a probabilistically unique value. PCC adds just 3% overhead on average to a Java VM. Querying the PCC value at points corresponding to testing and security clients adds 0 to 9% additional overhead, and querying at every call adds 49%, while missing relatively few new contexts due to conflicts (0.1% at worst). These results show that PCC is efficient and accurate enough to add context sensitivity to dynamic analyses that detect new or anomalous program behavior.

Acknowledgments

Many thanks to Vitaly Shmatikov for feedback and advice regarding security applications. Thanks to Samuel Guyer, Jungwoo Ha, Nicholas Nethercote, Ben Wiedermann, and Emmett Witchel for helpful discussions. Thanks to Jungwoo Ha, Calvin Lin, Ben Wiedermann, and the anonymous reviewers for valuable feedback on the paper.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *ACM Conference on Programming Language Design and Implementation*, pages 85–96, Las Vegas, NV, 1997.

[3] T. Apiwattanapong and M. J. Harrold. Selective Path Profiling. In *ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 35–42, 2002.

[4] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, 2000.

[5] M. Arnold, M. Hind, and B. G. Ryder. An Empirical Study of Selective Optimization. In *International Workshop on*

Languages and Compilers for Parallel Computing, pages 49–67, London, UK, 2001. Springer-Verlag.

[6] T. Ball. The SLAM Toolkit: Debugging System Software via Static Analysis, 2001.

[7] T. Ball and J. R. Larus. Efficient Path Profiling. In *IEEE/ACM International Symposium on Microarchitecture*, pages 46–57, 1996.

[8] A. R. Bernat and B. P. Miller. Incremental Call-Path Profiling. *Concurrency and Computation: Practice and Experience*, 2006.

[9] D. Binkley. Semantics Guided Regression Test Cost Reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, 1997.

[10] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, 2006.

[11] A. Chakrabarti and P. Godefroid. Software Partitioning for Effective Automated Unit Testing. In *ACM & IEEE International Conference on Embedded Software*, pages 262–271, 2006.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 11. The MIT Press, McGraw-Hill Book Company, 2nd edition, 2001.

[13] L. Fei and S. P. Midkiff. Artemis: Practical Runtime Monitoring of Applications for Execution Anomalies. In *ACM Conference on Programming Language Design and Implementation*, pages 84–95, 2006.

[14] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly Detection Using Call Stack Information. In *IEEE Symposium on Security and Privacy*, page 62. IEEE Computer Society, 2003.

[15] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-Overhead Call Path Profiling of Unmodified, Optimized Code. In *ACM International Conference on Supercomputing*, pages 81–90, 2005.

[16] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *ACM Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[17] W. Gropp. Runtime Checking of Datatype Signatures in MPI. In *European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 160–167, London, UK, 2000. Springer-Verlag.

[18] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved Error Reporting for Software that Uses Black Box Components. In *ACM Conference on Programming Language Design and Implementation*, pages 101–111, 2007.

[19] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *ACM International Conference on Software Engineering*, pages 291–301, 2002.

[20] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An Empirical Investigation of the Relationship Between Spectra Differences and Regression Faults. *Software Testing, Verification & Reliability*, 10(3):171–194, 2000.

[21] K. Hazelwood and D. Grove. Adaptive Online Context-Sensitive Inlining. In *IEEE/ACM International Symposium on Code Generation and Optimization*, pages 253–264, 2003.

[22] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The Garbage Collection Advantage: Improving Program Locality. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 69–80, 2004.

[23] H. Inoue. *Anomaly Detection in Dynamic Execution Environments*. PhD thesis, University of New Mexico, 2005.

[24] H. Inoue and S. Forrest. Anomaly Intrusion Detection in Dynamic Execution Environments. In *Workshop on New Security Paradigms*, pages 52–60, 2002.

[25] H. Inoue, D. Stefanović, and S. Forrest. On the Prediction of Java Object Liftimes. *ACM Transactions on Computer Systems*, 55(7):880–892, 2006.

[26] Jikes RVM. http://www.jikesrvm.org.

[27] Jikes RVM Research Archive. http://www.jikesrvm.org/-Research+Archive.

[28] J. Langou, G. Bosilca, G. Fagg, and J. Dongarra. Hash Functions for Datatype Signatures in MPI. In *European Parallel Virtual Machine and Message Passing Interface Conference*, pages 76–83, 2005.

[29] B. Lee, K. Resnick, M. D. Bond, and K. S. McKinley. Correcting the Dynamic Call Graph Using Control Flow Constraints. In *International Conference on Compiler Construction*, 2007.

[30] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable Statistical Bug Isolation. In *ACM Conference on Programming Language Design and Implementation*, pages 15–26, 2005.

[31] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu. Mining Behavior Graphs for Backtrace of Noncrashing Bugs. In *SIAM International Converence on Data Mining*, pages 286–297, 2005.

[32] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

[33] D. Melski and T. Reps. Interprocedural Path Profiling. In *International Conference on Compiler Construction*, pages 47–62, 1999.

[34] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.

[35] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[36] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[37] C. Pavlopoulou and M. Young. Residual test coverage montoring. In *ACM International Conference on Software Engineering*, pages 277–284, May 1999.

[38] F. Qian and L. Hendren. Towards Dynamic Interprocedural Analysis in JVMs. In *USENIX Symposium on Virtual Machine Research and Technology*, pages 139–150, 2004.

[39] A. Rountev, S. Kagan, and J. Sawin. Coverage Criteria for Testing of Object Interactions in Sequence Diagrams. In *Fundamental Approaches to Software Engineering*, LNCS 3442, pages 282–297, 2005.

[40] M. L. Seidl and B. G. Zorn. Segregating Heap Objects by Reference Behavior and Lifetime. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, 1998.

[41] J. Seward and N. Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *USENIX Annual Technical Conference*, pages 17–30, 2005.

[42] J. M. Spivey. Fast, Accurate Call Graph Profiling. *Softw. Pract. Exper.*, 34(3):249–264, 2004.

[43] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, 1999.

[44] Standard Performance Evaluation Corporation. *SPECjbb2000 Documentation*, release 1.01 edition, 2001.

[45] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A Dynamic Optimization Framework for a Java Just-in-Time Compiler. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 180–195, 2001.

[46] TIOBE Software. TIOBE programming community index, 2007. http://tiobe.com.tpci.html.

[47] K. Vaswani, A. V. Nori, and T. M. Chilimbi. Preferential Path Profiling: Compactly Numbering Interesting Paths. In *ACM Symposium on Principles of Programming Languages*, pages 351–362, 2007.

[48] D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *ACM Conference on Computer and Communications Security*, pages 255–264. ACM Press, 2002.

[49] J. Whaley. A Portable Sampling-Based Profiler for Java Virtual Machines. In *ACM Conference on Java Grande*, pages 78–87. ACM Press, 2000.

[50] B. Wiedermann. Know your Place: Selectively Executing Statements Based on Context. Technical Report TR-07-38, University of Texas at Austin, 2007.

[51] T. Zhang, X. Zhuang, S. Pande, and W. Lee. Anomalous Path Detection with Hardware Support. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 43–54, 2005.

[52] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, Efficient, and Adaptive Calling Context Profiling. In *ACM Conference on Programming Language Design and Implementation*, pages 263–271, 2006.