Breadcrumbs: Efficient Context Sensitivity for Dynamic Bug Detection Analyses [*]

Michael D. Bond
University of Texas at Austin
mikebond@cs.utexas.edu

Graham Z. Baker
MIT Lincoln Laboratory
gzbaker@ll.mit.edu

Samuel Z. Guyer
Tufts University
sguyer@cs.tufts.edu

UT Austin Computer Sciences Technical Report TR-09-36. Under double-blind submission to PLDI 2010.

Abstract

Calling context—the set of active methods on the stack—is critical for understanding the dynamic behavior of large programs. Dynamic program analysis tools, however, are almost exclusively context insensitive because of the prohibitive cost of representing calling contexts at run time. Deployable dynamic analyses, in particular, are limited to reporting only static program locations.

This paper presents Breadcrumbs, an efficient technique for recording and reporting dynamic calling contexts. It builds on an existing technique for computing a compact (one word) encoding of each calling context that client analyses can use in place of a program location. The key feature of our system is a search algorithm that can reconstruct a calling context from its encoding using only a static call graph and a small amount of dynamic information collected in cold methods. Breadcrumbs requires no offline training or program modifications, and handles all language features, including dynamic class loading. On average, it adds 10% to 20% overhead to existing dynamic analyses, depending on how much additional information it collects: more information slows down execution, but improves the decoding algorithm.

We use Breadcrumbs to add context sensitivity to two dynamic analyses: a race detector and an analysis that identifies the origins of null pointer exceptions. Our system can reconstruct nearly all of the contexts for the reported bugs in a few seconds. These calling contexts are non-trivial, and they significantly improve both the precision of the analyses and the quality of the bug reports.

1. Introduction

A stack trace is one of the most useful pieces of information to have when debugging a program failure. A stack trace captures the full dynamic calling context of the code that failed, not just its static program location. This level of detail is crucial for debugging object-oriented programs, which have many small methods and a high degree of reuse. Producing a stack trace is straightforward at the point of failure: the program is halted, and a debugging tool can inspect the program's stack and emit a sequence of methods or call sites for the programmer to inspect. The cost of computing or storing the stack trace is largely unimportant.

Recently, more sophisticated dynamic debugging techniques have focused on identifying the root causes of bugs, not just the immediate circumstances of failures. To do this, they record a lot of information that *might* be useful in explaining a failure, if one occurs. Dynamic race detection tools, for example, record all memory accesses; when a race is detected they provide information about both the access that triggered the detector and the earlier access with which it conflicts. Ideally, these tools would report multiple stack traces: one for the point of failure, and one for each event leading up to it. Without knowing which stack traces will be needed, however, most tools are limited to recording just static program locations (e.g., method and bytecode index). The naive approach of walking the stack and recording the explicit calling context for every event—for example, every memory access—is prohibitively expensive.

This paper presents *Breadcrumbs*, an efficient runtime mechanism for recording and reporting dynamic calling contexts. Our work builds on an existing mechanism called *probabilistic calling context*: during execution our system computes a probabilistically unique ID (called a *PCC value*) for each calling context [Bond and McKinley 2007]. Dynamic debugging tools can be made context sensitive by using the PCC value to tag events or data wherever they would have used simple program locations. Unlike the original PCC work, though, our system can decode a PCC value back into its original sequence of calls for error reporting. In addition, unlike recent related techniques, our system does not require any ahead-of-time training, data collection, or program analysis. All information needed to encode and decode PCC values is computed online. Our technique is suitable for use by debugging and analysis tools for deployed software.

The key problem we solve is collecting enough extra information at runtime to allow accurate decoding of PCC values, without incurring a large overhead. Since PCC values are computed top-down during execution (that is, the PCC value in a method is computed as a function of the PCC value in its caller and a callsite ID), the decoding algorithm is naturally a bottom-up search: starting at the most recent method, we invert the computation at each call site, moving from callees to callers until we reach "main". While decoding occurs offline, a blind search of the PCC space is much too expensive. To constrain the search Breadcrumbs collects two additional kinds of information at runtime: (1) a static call graph, which constrains which potential callers it must consider, and (2) the set of PCC values observed at a callsite. Since method calls are extremely frequent, however, we can only build these sets for code that is relatively cold. This threshold is a tunable parameter that trades online performance for reconstruction accuracy.

We implement Breadcrumbs in Jikes RVM, a high-performance Java-in-Java virtual machine [Alpern et al. 1999], and integrate it with two real dynamic debugging tools: a dynamic race detector based on the FastTrack algorithm [Flanagan and Freund 2009], and origin tracking, an analysis for diagnosing null pointer exceptions [Bond et al. 2007]. Breadcrumbs is able to reconstruct almost all calling contexts for the bugs reported with overheads around 10% to 20% on average (depending on the hotness threshold). The resulting bug reports provide much more information than

[*] This material is based upon work supported by the National Science Foundation under Grant No. CCF-0916810

the context-insensitive versions. With Breadcrumbs, origin tracking reports a full stack trace for both the exception and the origin of the null value. The race detector reports the full calling context of most conflicting memory accesses, and separates buggy from non-buggy uses of common code.

The rest of this paper is organized as follows. First, we discuss in more detail the benefits of context sensitivity for dynamic analysis, and the class of dynamic analysis tools that can benefit from this technique. In Section 3 we present the Breadcrumbs decoding algorithm and associated runtime support. In Section 4 we describe our results—both performance and accuracy—using Breadcrumbs to make two dynamic bug detectors context sensitive.

2. Motivation

Dynamic analysis has emerged as an important technique for understanding program behavior, and in particular, detecting programming errors. Catching bugs at runtime has a number of advantages over other techniques, such as static analysis and testing. It works on all inputs, easily handles language features like dynamic class loading and bytecode rewriting, and, in many cases, produces no false positives. In addition, it is effective for catching difficult-to-reproduce errors, such as race conditions.

Monitoring programs at runtime, however, imposes significant constraints on the analysis algorithm, both in terms of time and space. Deployable dynamic analyses, in particular, cannot significantly degrade program performance. As a result, most dynamic bug detectors are context insensitive: they analyze and report bugs strictly in terms of static program locations.

For modern object-oriented programs, however, static program locations are often insufficient to explain program behavior. These programs exhibit complex patterns of code reuse and delegation that make it hard to understand how execution arrived at a particular point. In addition, these programs consist of many layers of software, assembled using components and application frameworks. Context sensitivity is critical for understanding the circumstances of an error.

The goal of Breadcrumbs is to provide a general mechanism for making deployable dynamic bug detectors context sensitive. In this section we discuss the benefits and challenges of context sensitivity in a dynamic setting, and we describe the class of applications that will benefit from our approach.

2.1 Why context sensitivity?

Context sensitivity is crucial for understanding the behavior of large object-oriented programs [Inoue and Nakatani 2009; Lhoták and Hendren 2008]. Unlike static analysis, however, prior work on dynamic analysis has largely avoided context sensitivity because no efficient technique was known. Dynamical analysis tools, such as bug detectors, however, stand to benefit considerably from context sensitivity, which improves both the precision of the analysis and the quality of the bug reports.

Context sensitivity improves precision by separating buggy from non-buggy uses of the same code. Modern software is assembled from class libraries, application frameworks, and other reusable components. Failures in common code might occur in some contexts of use and not in others. In our race detection experiments, for example, we discovered that one static program location is actually involved in several dynamically distinct races.

Context sensitivity improves bug reporting by providing more information about the circumstances of relevant program events. State-of-the-art memory leak detectors, for example, tag each object with its allocation site [Chilimbi and Hauswirth 2004]. If a leak is detected, the allocation site is used to identify the objects involved. With factory methods, however, objects of a particular class are all generated at a single allocation site. Using only static program locations, this information is essentially useless for debugging. Tagging objects with the full calling context of their allocation solves this problem by revealing the context in which the factory method was called.

In addition, when the origin of a bug and the point of failure are far apart in the program, it is not always obvious how they are connected. With the full calling context for both it is much easier to see how control flows from one to the other.

The main challenge, for both static and dynamic analyses, is the shear number of calling contexts. In theory, this number is exponential in the size of the program. In practice, even relatively small programs can have millions of calling contexts. In the presence of recursion, the number of possible calling contexts is unbounded.

2.2 Target clientele

Not all client analyses need the techniques described in this paper. Breadcrumbs represents a specific tradeoff between cost and precision that is targeted at deployable bug detectors. These clients represent a broad class of dynamic analysis with the following properties:

- (1) Correct execution must be fast: our goal is to provide context sensitivity in a deployed setting.
- (2) Tracked events are numerous: the client analysis records many context-sensitive events online, not knowing which ones might later be relevant for error reporting.
- (3) Reconstruction of calling contexts occurs offline: since the search space is large, and both the encoding and decoding are probabilistic, Breadcrumbs allows the reconstruction algorithm to run for several seconds before returning its best candidate, if any.

Correct execution must be fast. Deployable bug detectors have become increasingly important for languages like Java, which include many features that hamper other methods of error detection. Unlike static analysis, dynamic analysis operates on the concrete program rather than an abstract approximation, often catching all the real errors with no false positives. Dynamic analysis also works naturally in the presence of dynamic class loading and bytecode rewriting. Deployable bug detectors also catch difficult-to-reproduce errors, such as race conditions, which are often missed during routine testing. To be deployable, however, a bug detector must avoid slowing the program down by a significant amount. Breadcrumbs keeps time and space overheads low, and includes a tunable parameter (see Section 3) that controls the amount of overhead versus the accuracy of decoding.

Other kinds of dynamic analysis do not require high performance. Using Valgrind, for example, can slow programs down by a factor of 30 or more [Nethercote and Seward 2007]. For these applications, the additional expense of walking the stack or building a calling context tree (see below) is not significant. Breadcrumbs might still be useful, however, since it requires few changes to the analysis algorithm.

Events are numerous. Providing context sensitivity in a deployable setting is particularly challenging when events that need calling context information occur very frequently. The dynamic analysis clients presenting in Section 4, for example, record every memory access (for race detection), and every null pointer (for origin tracking). Existing techniques, which represent calling contexts explicitly, would be much too expensive. These techniques fall into roughly two categories: explicit stack traces, and building the dynamic calling context tree.

Creating an explicit stack trace is relatively slow and occupies space proportional to the depth of the calling context (e.g., as an array of call sites) [Nethercote and Seward 2007; Seward and

Nethercote 2005]. For infrequent events, however, this cost is easily hidden; for example, for a dynamic analysis that records the calling context at each new thread start. For frequent events, however, both the time and space costs are much too high for use in deployed software. Walking the stack at every system call or at every object allocation, for example, slows execution dramatically [Bond and McKinley 2007].

An alternative is to build a dynamic calling context tree (similar to a dynamic call graph), in which each node represents a unique calling context [Ammons et al. 1997; Spivey 2004]. Per event recorded, the time and space costs are very low: a calling context is identified by a pointer into the tree. Building and maintaining the tree, however, requires every method call to check the set of children contexts and create a new node if necessary. In addition, the tree can become very large: the eclipse benchmark, for example, executes 35,000 static callsites resulting in 10 million unique calling contexts (see Table 1 in Section 4).

Reconstruction occurs offline The techniques we use to keep overhead low at runtime make reconstructing calling contexts more difficult. First, since the underlying PCC values are only probabilistically unique, the reconstruction algorithm can find multiple calling contexts for a single PCC value. Second, we collect a limited amount of information to guide the search algorithm; depending on the quality of this information the search space might still be very large. Breadcrumbs uses an iterative deepening algorithm with a fixed time budget (five seconds in our experiments), after which it returns the best solution found so far, if there is one. If none is found, the reconstruction fails. As a result, our system is most suitable for error reporting and logging. More work is needed to determine if Breadcrumbs could be used to answer context-sensitive questions online.

3. Breadcrumbs algorithm

Breadcrumbs builds on *probabilistic calling context* (PCC) [Bond and McKinley 2007], an online technique for computing a probabilistically unique ID (called a *PCC value*) for each dynamic calling context. While PCC computes an efficient and compact encoding of calling contexts, it provides no way to decode a PCC value for use in bug reporting. This section describes our calling context reconstruction algorithm and the additional runtime support needed to make this decoding possible.

3.1 The PCC decoding problem

Each PCC value is essentially a hash of the sequence of callsites that compose a calling context. PCC values are computed continuously during execution so that dynamic analyses can obtain a representation of the current calling context at any time. At every method call, a PCC value for the new context is computed as a function of the PCC value in the caller, plus an identifier representing the callsite. Specifically, given a PCC value p for the current calling context, and a callsite ID c, it computes the new calling context in the callee as:

$$f(p, c) = (3p + c) \bmod 2^{32}$$

The initial PCC value representing the top-most context (the `main` method) is zero; c_0 represents a call site in `main` and c_n represents the most recent call. Computing PCC values during execution, shown graphically in Figure 1, proceeds as follows:

$p_0 = 0$ {the `main` calling context}
$p_1 = f(p_0, c_0)$ {context in callee invoked at c_0 in `main`}
...

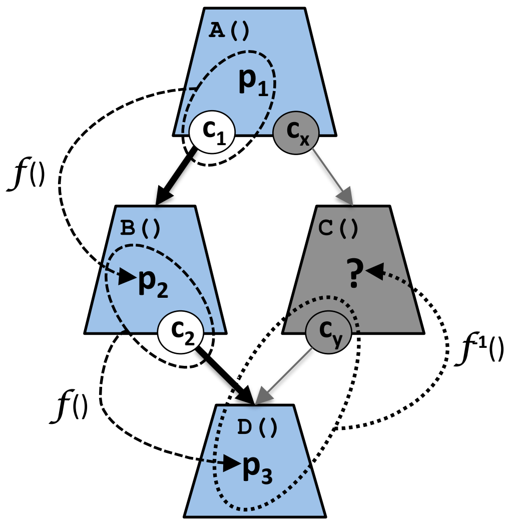

Figure 1. Calling context encoding and decoding. Method A() calls method B() at callsite c_1; the child context in B() is computed as $p_2 = f(p_1, c_1)$. Likewise, when B() calls D(), we compute $p_3 = f(p_2, c_2)$. The inverse function, f^{-1}, takes a PCC value and a callsite and computes the parent context. The challenge is choosing the right callsite: choosing correctly, we compute $p_2 = f^{-1}(p_3, c_2)$ and $p_1 = f^{-1}(p_2, c_1)$. Choosing callsite c_y, however, leads to a PCC value that never occurred.

$p_i = f(p_{i-1}, c_{i-1})$
...
$p_n = f(p_{n-1}, c_{n-1})$

The PCC value *decoding* problem can be described abstractly as follows: given a PCC value p_n, find a sequence of callsite IDs c_0, c_1, ..., c_{n-1} such that

$$p_n = f(f(...f(0, c_0)..., c_{n-2}), c_{n-1})$$

One key observation is that the single-step PCC computation f is invertible, in spite of its use of modular arithmetic, because 3 and 2^{32} are relatively prime. Given a particular p' and c, there is only one p that satisfies $p' = (3p + c) \bmod 2^{32}$. This property allows us to recast the problem as a backward search: starting at the last callsite, we can invert the PCC computation one callsite at a time until we reach `main`.

Given p_n,
choose c_{n-1} and compute $p_{n-1} = f^{-1}(p_n, c_{n-1})$
...
choose c_{i-1} and compute $p_{i-1} = f^{-1}(p_i, c_{i-1})$
...
choose c_0 such that $f'(p_0, c_0) = 0$

Without any additional information, this search problem is intractable. Even relatively small programs contain 1000s of call sites and calling contexts 10s of levels deep, leading to more than $1000^{10} = 10^{30}$ combinations. In addition, because PCC values are only probabilistically unique, considering all possible combinations of call sites will discover many spurious paths that represent hash collisions. In the presence of recursion, the search is actually unbounded because the PCC value does not directly encode the length of the calling context.

3.2 Pruning the search

In order to constrain the search, Breadcrumbs collects two kinds of additional information at run time, one static and the other dynamic.

First, decoding uses a *static call graph* to prune out callsites that cannot invoke the current method. Assuming it has decoded the path up to call site c_i in method m_i, it only considers callsites c_{i-1} that could invoke m_i. This constraint significantly reduces the fan-out of the search, and thus dramatically reduces the size of the search space.

Breadcrumbs computes the static call graph during just-in-time compilation using a traditional type-based algorithm for approximating possible callees. It handles dynamic class loading naturally by adding call graph edges as the VM loads, resolves, and compiles new methods. The resulting data structure is small and does not significantly impact compilation time, but it is typically incomplete, however, due to calls in and out of native methods and the virtual machine.

Second, Breadcrumbs collects dynamic information to prune call paths that never occurred during execution. For real programs, the number of calling contexts that *actually* occur in a particular run is much, much smaller than the number of calling contexts that could *possibly* occur in any run.

Breadcrumbs modifies the dynamic compiler to insert instrumentation at every application call site that collects *per-callsite PCC values*. The per-callsite PCC values for callsite c_{i-1} are all PCC values p_i that PCC instrumentation computed at c_{i-1}. That is, the per-callsite values for a callsite are all the PCC values computed at the callsite.

```
$p' \leftarrow f(p, c)$                                    {Original PCC computation}
$\mathit{values}_c \leftarrow \mathit{values}_c \cup \{p'\}$     {Record per-callsite PCC value}
$c : \mathit{foo}(\ldots)$                                {Original application call site}
```

Collisions among these per-callsite PCC values are highly unlikely, and they prune the search to practically one candidate. Unfortunately, even with careful engineering, updating the per-callsite values at every method call is too expensive, both in time and space, for deployed software.

3.3 Trading accuracy for performance

To control the costs of recording every per-callsite PCC value, Breadcrumbs stops recording per-callsite PCC values at *hot* callsites, which are callsites whose frequency exceeds a custom threshold *hotThreshold*. The instrumentation at per-callsite values tracks the callsite's execution frequency. If the frequency exceeds *hotThreshold*, the instrumentation discards any per-callsite values recorded so far, and it does not record per-callsite values in the future. In our implementation, if the dynamic optimizing compiler recompiles a hot callsite, it can forgo adding instrumentation (including the threshold check) that records per-callsite PCC values.

The *hotThreshold* is a key parameter to Breadcrumbs that significantly affects its performance and reconstruction accuracy. In future work, we plan to explore more sophisticated metrics for selecting hot callsites, such as avoiding marking many callsites hot in one part of the static call graph, a situation that often leads to the reconstruction algorithm being unable to reconstruct values.

3.4 Client sites

As described so far, Breadcrumbs' instrumentation computes PCC values, records per-callsite PCC values, and builds a static call graph. Its purpose, however, is to serve a *client analysis*, such as a dynamic bug detection analysis, that stores program locations.

Each client analysis defines some notion of *client sites*, which are program locations of interest to the analysis. At any program point, however, the PCC value only represents the calling context at the last caller's callsite. When the client requests a PCC value, Breadcrumbs needs to modify the PCC value to include the client site. Breadcrumbs computes the PCC value for client sites as follows:

```
$p' \leftarrow f(p, c_{\mathit{client}})$
{Give $p'$ to client analysis as current program location}
```

Breadcrumbs also records the set of all client sites, which is needed during the first step of the reconstruction algorithm. During or at the end of the run, the client analysis asks Breadcrumbs to decode one or more PCC values (e.g., potential bug locations).

3.5 Reconstruction algorithm

Algorithm 1 shows the complete reconstruction algorithm, which uses the static and dynamic information described above. The algorithm uses iterative deepening to find the most likely calling context for a given PCC value.

Each iteration is a depth-limited backwards search of the PCC space, following potential call edges from callees back to callsites in their callers, continuing until it reaches the value 0. Not all edges are equal, however: when static and dynamic information is available, the resulting edges are much more likely to be part of the correct calling context. To account for this difference, the algorithm computes a depth metric for each edge, called the *blow-up factor* (described in detail below), which estimates the size of the search space based on the fan-out of previous search steps. A large blow-up factor results from many blind or semi-blind search steps; it represents a low-confidence path and indicates a region of a subgraph that will take a long time to explore. Each iteration of the search has a set *blow-up limit*, and the algorithm cuts off any path that exceeds the limit. The main search loop, procedure *decode*() in Algorithm 1, successively increases the blow-up limit looking for a solution with minimum blow-up.

Procedure *decodeWithLimit*() performs a single step of the search given three pieces of information: the current PCC value to decode p_i (p' in the algorithm), the last callsite identified c_i (*site* in the algorithm), and the current blow-up factor. If p_i is zero, we have a potential solution, which we record along with a confidence value (similar to blow-up). Otherwise, we select a set of candidate callsites for c_{i-1} by looking at the static call graph and the dynamic per-callsite PCC sets, yielding two sets of candidates and four possible types of sites to consider:

Static and dynamic: a site that is in both the static and dynamic sets; it is very likely to be part of the correct calling context.

Static only: a site in the static set but not the dynamic set. The algorithm considers *hot* sites since these are the only sites that can be callers but not be in the dynamic set.

Dynamic only: a site in the dynamic set but not the static set. These sites occur when the application calls into the VM or class libraries, which in turn call back into the application. Since we do not analyze this system code, the static call graph will be missing these edges.

No static, no dynamic: a site in neither set. The algorithm must consider *all* hot sites as potential callers.

In each case, the resulting set of candidate callsites is assigned a blow-up factor, which is computed from three variables:

searchSpace Estimates the total branching factor of the region of the call graph being explored with static but no dynamic infor-

Algorithm 1 Decoding calling context from a PCC value

```
globals
{Current blow-up cutoff}
$blowupLimit \leftarrow 2$
{For sites indicated by dynamic information only, what's the probability that a match is not a conflict?}
$probNoConflict \leftarrow (1 - \frac{numValues}{2^{32}})^{|allCallSites|}$ where $numValues = \sum_{s \in allCallSites} |s.perCallSiteValues|$

procedure $decode(p')$
  $solutions \leftarrow \emptyset$
  {Iterative deepening}
  repeat
    $decodeWithLimit(p')$
    $blowupLimit \leftarrow 2 \times blowupLimit$
    $sort(solutions)$                                    {Sort by increasing blow-up}
  until $solutions \neq \emptyset \vee timed\ out$

procedure $decodeWithLimit(p')$
  {If client site known, instead just call $decodeFromSite()$ directly}
  for all $clientSite \in allClientSites$ do
    $p \leftarrow f^{-1}(p', clientSite)$
    decodeFromSite(p, clientSite, {1, 1, 0})
  end for

procedure $decodeFromSite(p', site, \{searchSpace, permProb, blindDepth\})$
  if p' = 0 then
    {Found a possible solution. Assign a blow-up for sorting.}
    $solutions \leftarrow solutions \cup \{contextOnStack, \frac{searchSpace}{permProb}\}$
  end if

  $staticSites \leftarrow getStaticallyPossibleSites(p')$          {Use static call graph}
  $dynamicSites \leftarrow getDynamicallyPossibleSites(p')$        {Use per-call site values}

  if $staticSites \neq \emptyset$ then
    {Try sites indicated by both static and dynamic information}
    $decodeCallers(p', staticSites \cap dynamicSites, \{1, permProb, 0\})$
    {Try sites indicated by static information but with dynamic information discarded}
    $decodeCallers(p', \{site \in staticSites$ s.t. $site.perCallSiteValues = \textbf{null}\}, \{searchSpace \times |callerSites|, permProb, blindDepth + 1\})$
  else
    {Try sites indicated by dynamic information only}
    $decodeCallers(p', dynamicSites, \{searchSpace, permProb \times ProbNoConflict, 0\})$
    {Try sites not indicated by static or dynamic information}
    $decodeCallers(p', allRemovedCallSites, \{searchSpace \times |allRemovedCallSites|, permProb \times ProbNoConflict, blindDepth + 1\})$
  end if

procedure $decodeCallers(p', callerSites, \{searchSpace, permProb, blindDepth\})$
  $blowupFactor \leftarrow \frac{searchSpace + blindDepth}{permProb}$
  if $blowupFactor \leq blowupLimit$ then
    for all $callerSite \in callerSites$ do
      $p \leftarrow f^{-1}(p', callerSite)$
      $decodeFromSite(p, callerSite, \{searchSpace, permProb, blindDepth\})$
    end for
  end if
```

mation. The value is reset to 1 on steps that use dynamic information, and otherwise accumulates the product of the number of possible callers at each step.

blindDepth Estimates the *depth* of the region of the callgraph being explored with static but no dynamic information. The *searchSpace* variable tracks the fan-out of this subgraph, but it does not change when there is just one statically possible caller. Adding in the depth, which only grows by one, captures this small, but significant, unit of work.

permProb If a site is indicated by dynamic but not static information, there is some probability that it represents a conflict in the PCC space. *permProb* accounts for this factor by accumulating the product of the probability of such a collision for each dynamic-only callsite along the search path.

Procedure *decodeCallers*() takes the PCC value, the set of candidate callsites, and the blow-up factor, and cuts off the search if the blow-up exceeds the limit. If not, for each candidate callsite c_{i-1} it inverts the PCC computation to obtain p_{i-1} and calls *decodeFromSite*() recursively.

Procedure *decodeWithLimit*() is the main entry point for a single iteration of the search. Since PCC values do not encode a program location directly, the first step of the search must consider all possible client sites (all places where the dynamic analysis client requested a PCC value). The fan-out of this first search step can be significant. One possible solution is to give the client a 64-bit value for each context identifier: 32 bits for the PCC value and 32 bits for the client site.

4. Results

This section evaluates the accuracy and overhead of Breadcrumbs. We first evaluate Breadcrumbs without a client. We then demonstrate its utility by adding context sensitivity to two existing dynamic bug detectors, one for detecting races and the other for identifying the origins of null pointers. It required modest effort to integrate these analyses with Breadcrumbs. The decoding algorithm can reconstruct the majority of calling contexts for these two clients, within a time limit of five seconds per context. It fails to reconstruct contexts that involve long, uninterrupted hot sites; dynamic information is unavailable for these contexts, and the blow-up in possible static callers becomes too large to search within the time limit. The calling contexts that it successfully reconstructs are long and nontrivial. These bug reports are much more informative than the context-insensitive information provided by the unmodified systems.

4.1 Methodology

Platform. We execute all experiments on a Core 2 Quad 2.4 GHz system with 2 GB of main memory running Linux 2.6.20.3. Each of two cores has two processors, a 64-byte L1 and L2 cache line size, and an 8-way 32-KB L1 data/instruction cache; and each pair of cores shares a 4-MB 16-way L2 on-chip cache.

Benchmarks. We evaluate Breadcrumbs on the DaCapo benchmarks version 2006-10-MR1 [Blackburn et al. 2006] and a fixed-workload version of SPECjbb2000 [Sta 2001] called pseudojbb. We exclude the benchmarks bloat because its performance is erratic and lusearch because it does not execute correctly in Jikes RVM 3.1.0 in our environment, with or without Breadcrumbs.

4.2 Breadcrumbs without a client

We first evaluate characteristics of Breadcrumbs without a client. Figure 2 shows the overhead of several Breadcrumbs configurations compared to unmodified Jikes RVM. Each sub-bar is the median 10 trials. The *PCC only* configuration computes the PCC value all the time but does not store per-callsite PCC values. The other configurations store PCC values for callsites whose execution frequency is below the *hotThreshold*. With no threshold at all, the system records per-callsite PCC sets at all callsites, adding almost 100% overhead. This option makes the reconstruction algorithm fast and very accurate, but it is probably too slow for deployed software. Thresholds of 1,000 and 10,000 add about 10% and 20% overhead, respectively, which is low enough for many deployed settings. The *t = 100* configuration adds about 5% overhead over PCC, mainly due to instrumentation in baseline-compiled methods and the baseline compiler adding static call graph edges.

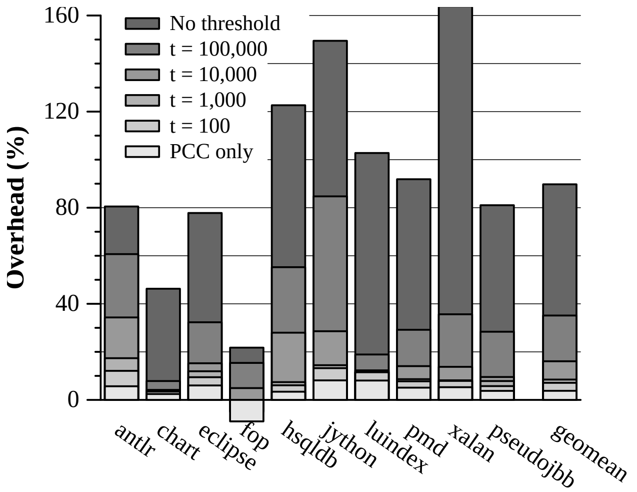

Figure 2. Overhead of Breadcrumbs for various hot callsite thresholds.

Table 1 presents statistics about dynamic and distinct per-callsite PCC values it stores, and executed and hot call sites. Each number is the average from 10 trials. The *PCC values stored* shows how many values Breadcrumbs stores in per-callsite PCC values. *Dynamic values* is the total number of hash table lookups to add per-callsite values. It grows dramatically with the hot threshold, which explains the significant performance impact from high hot thresholds. *Distinct values* is the number of distinct PCC values that Breadcrumbs stores. It increases with the hot threshold but not as drastically, and it shows that using a low threshold lowers Breadcrumbs' memory footprint in addition to its time overhead.

The *Static callsites* columns show the size of the static call graph. *Exec.* is the total number of callsites in the static call graph that the application actually executed. Breadcrumbs is only concerned with executed sites, which it computes easily because callsite instrumentation already tracks execution frequency in order to determine if the site is hot. Breadcrumbs marks a significant proportion of sites as *Hot* for low thresholds. Of course, in the case of no threshold, no site becomes hot.

In summary, the table shows that the hot threshold significantly affects the characteristics of Breadcrumbs. Higher thresholds increase its performance impact, but lower thresholds record a lot less information. We find that thresholds of 10,000 and 100,000 provide an acceptable balance: low enough overheads for most deployed software, but still able to reconstruct most real context-sensitive program locations.

4.2.1 Origin tracking

Origin tracking is a dynamic analysis for identifying the causes of null pointer exceptions [Bond et al. 2007]. It tracks the origin of *every* null value, which it stores *in place* of the null value. These

		PCC values stored		Static callsites	
Program	Threshold	Dynamic	Distinct	Exec.	Hot
	100	834,998	6,152		6,836
	1,000	4,608,868	38,143		3,078
antlr	10,000	17,808,051	121,711	11,105	907
	100,000	69,635,283	413,218		352
	∞	528,695,364	466,492		0
	100	276,828	93,227		1,764
	1,000	1,264,545	213,146		846
chart	10,000	5,452,412	314,987	5,874	381
	100,000	32,020,085	329,185		246
	∞	201,127,995	344,824		0
	100	1,804,807	86,188		14,644
	1,000	10,982,483	319,873		8,570
eclipse	10,000	64,718,116	1,520,876	34,834	4,568
	100,000	259,995,599	4,480,888		1,202
	∞	857,238,160	10,535,356		0
	100	236,824	12,193		1,703
	1,000	1,343,523	15,338		1,030
fop	10,000	5,405,674	27,630	8,059	206
	100,000	14,343,170	53,985		51
	∞	21,143,486	87,320		0
	100	163,775	9,591		1,490
	1,000	1,278,734	22,065		1,151
hsqldb	10,000	10,074,484	30,749	4,079	897
	100,000	34,616,207	47,312		120
	∞	158,805,788	52,797		0
	100	1,865,469	62,256		17,460
	1,000	16,479,966	197,996		15,699
jython	10,000	148,352,609	606,760	32,853	13,951
	100,000	675,365,567	1,660,706		1,294
	∞	3,624,874,761	2,010,286		0
	100	152,502	1,239		1,480
	1,000	1,335,305	2,651		1,267
luindex	10,000	8,261,057	7,141	2,167	642
	100,000	46,989,738	65,117		361
	∞	217,577,829	83,163		0
	100	400,366	17,061		3,573
	1,000	2,730,737	67,794		2,164
pmd	10,000	17,566,711	408,631	7,181	1,230
	100,000	62,704,566	1,928,866		214
	∞	270,967,921	2,628,090		0
	100	462,369	9,309		4,283
	1,000	3,946,777	18,055		3,646
xalan	10,000	31,886,716	35,225	7,674	2,685
	100,000	126,698,291	97,318		559
	∞	738,467,992	128,095		0
	100	139,777	2,998		1,179
	1,000	1,092,692	4,238		1,022
pseudojbb	10,000	8,981,094	5,822	3,033	811
	100,000	44,543,902	10,094		258
	∞	137,258,103	14,171		0

Table 1. Breadcrumbs characteristics without any client, for a variety of hot thresholds. Callsite instrumentation records many more dynamic and distinct PCC values at high thresholds, and it also marks many fewer methods as hot.

special null values flow through the program normally. If a null pointer exception occurs, it reports the origin of the null, as well as a context-sensitive stack trace for the *current* program location. The *origin* is context *insensitive* because origin tracking encodes static program locations for each origin. We modify origin tracking to use PCC values in place of program locations, making the origins context *sensitive*. With Breadcrumbs, the system can report two full stack traces: one for the exception and one for the origin.

Origin tracking is challenging to make context sensitive because there are so many null values at runtime, most of which never cause a exception. As such, it is well suited for Breadcrumbs.

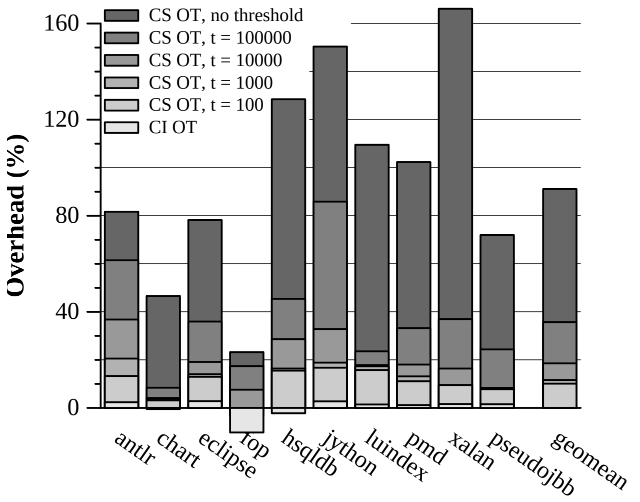

Figure 3. Overhead of origin tracking with and without context sensitivity (i.e., with and without Breadcrumbs) for various hot callsite thresholds.

Origin tracking can only use 27 bits for origins, since the other 5 are needed to mark the value as a null origin, so we modify Breadcrumbs in this case to compute 27-bit PCC values (instead of the usual 32-bit PCC values in our implementation).

Figure 3 shows the overhead of origin tracking with several Breadcrumbs configurations and without Breadcrumbs, compared to unmodified Jikes RVM. Each sub-bar is the median of 10 trials. The *OT only* configuration is the overhead of origin tracking alone, which does not use Breadcrumbs and reports only context-insensitive origins for null pointer exceptions. The other configurations use Breadcrumbs with various hot callsite thresholds. These results mirror Breadcrumbs-only results in Figure 2. That is, Breadcrumbs adds about the same overhead with or without a client.

The original origin tracking paper [Bond et al. 2007] evaluated origin tracking on 12 real null pointer exceptions, which the authors made available publicly as the *Bad Apples Suite*.[1] The suite provides 12 null pointer exception-inducing inputs and programs. We evaluate Breadcrumbs-enabled origin tracking with 10 of 12 exceptions from the Bad Apples Suite. We do not evaluate the 2 Eclipse NPEs because Jikes RVM 3.1.0 does not execute the Eclipse GUI correctly. (The Jikes RVM development head fixes the problem, so we plan to port to it for the final paper.)

Table 2 shows how well Breadcrumbs reconstructs calling contexts for null pointer origins from the Bad Apples Suite, in the same order as the original paper, which describes these exceptions in detail and explains if and how each origin is useful [Bond et al. 2007]. Our system is able to reconstruct all but one of the calling contexts, regardless of the hotness threshold, suggesting that this dynamic analysis could use the t = *100* threshold with the lowest overhead. On the other hand, many of the inputs expose null pointer exceptions almost immediately, so few methods are hot. In contrast, the other client we evaluate, race detection, reconstructs contexts from throughout execution, and unsurprisingly its accuracy degrades as the hot threshold increases (Section 4.3).

The resulting origin stack traces are nontrivial, ranging in length from 2 to 19 stack levels. For example, it reports the following context-sensitive origin for the Jython #2 exception:

```
  at org.python.core.PyObject.fastGetDict():2723
  at org.python.core.PyObject.getDoc():360
```

[1] `http://www.cs.utexas.edu/~mikebond/bad-apples-suite`

Program	Succeeds	Depth
Mckoi SQL DB	Always	6
FreeMarker #1	Always	12
JFreeChart #1	Always	2
JRefactory #1	Always	8
Checkstyle	Always	19
JODE	Never*	?
Jython #1	Always	11
JFreeChart #2	Always	4
Jython #2	Always	14
JRefactory #2	Always	10

Table 2. Origin tracking statistics for the Bad Apples Suite. The threshold does not affect reconstruction accuracy for these inputs.

```
at org.python.core.PyGetSetDescr.__get__():55
at org.python.core.PyObject.object___findattr__():2770
at org.python.core.PyObject.__findattr__():1044
at org.python.core.PyObject.__getattr__():1081
at org.python.pycode._pyx0.f$0():1
at org.python.pycode._pyx0.call_function():0
at org.python.core.PyTableCode.call():213
at org.python.core.PyCode.call():14
at org.python.core.Py.runCode():1182
at org.python.core.__builtin__.execfile_flags():315
at org.python.util.PythonInterpreter.execfile():158
at org.python.util.jython.main():186
```

Context-insensitive origin tracking of course reports only the first line of the context.

Breadcrumbs cannot reconstruct the calling context of the origin in JODE due to technical difficulties. At the higher thresholds, Breadcrumbs completes an exhaustive search without finding a matching context, indicating a bug in our implementation. We plan to investigate for the final paper.

4.3 Race Detection

We implemented a dynamic race detector in Jikes RVM based on the sound and precise *FastTrack* algorithm [Flanagan and Freund 2009]. Our race detector is not highly optimized, and on average it slows program execution by 12x. (The implementation in the FastTrack paper slows programs by 8x on average.) While such an expensive race detector probably does not require Breadcrumbs—it might as well pay the additional 2-4x and use a calling context tree (CCT) [Ammons et al. 1997; Spivey 2004]—Breadcrumbs could be applied to a less expensive analysis, such as sampling-based race detection [Marino et al. 2009]. Here we use a fully sound and precise race detector to evaluate how accurately Breadcrumbs can reconstruct buggy, nontrivial calling contexts. For benchmarks we use four multithreaded programs: DaCapo's eclipse, hsqldb, and xalan; and SPEC pseudojbb.

Figure 4 shows the overhead of race detection with several Breadcrumbs configurations and without Breadcrumbs, compared to unmodified Jikes RVM. Each sub-bar is the median of five trials (since the experiments are time-consuming; we plan to run more trials). The *RD only* configuration is the overhead of our implementation of FastTrack, which does not use Breadcrumbs and reports only context-insensitive locations for the first access of a data race. The other configurations use Breadcrumbs with various hot callsite thresholds. Breadcrumbs adds somewhat more overhead with this client than standalone or with origin tracking. While in theory Breadcrumbs does no additional work, both systems increase various loads in ways that add superlinearly, e.g., at high hot thresholds, they both add significant memory overhead and stress the memory subsystem and garbage collector.

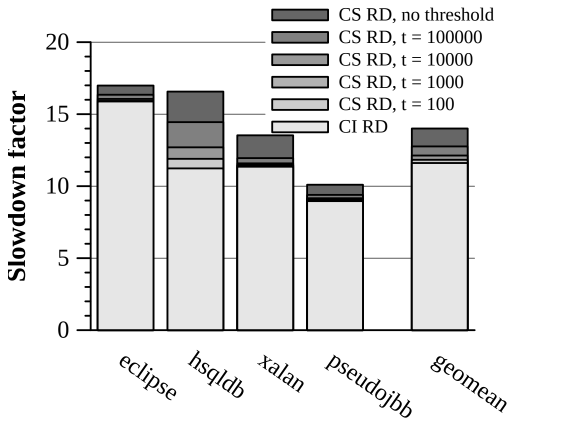

Figure 4. Overhead of Race detection with and without context sensitivity for various hot callsite thresholds.

Table 3 shows how well Breadcrumbs reconstructs calling contexts for the first access of racy access pairs. Except for the number of client sites, all the stats are **sums** across all five trials. In this analysis the client sites are the subset of memory accesses that could be involved in a race. For each combination of benchmark and threshold we report the number of calling context reconstructions that succeeded and failed. Breadcrumbs is able to reconstruct most of the calling contexts regardless of the hotness threshold. For eclipse and xalan, however, we start to see the effects of having less dynamic information at the lower thresholds. In these cases, the search algorithm is running out of time before finding the right path, which in both cases are quite deep. With no threshold, eclipse runs out of memory because Breadcrumbs with no threshold and race detection both use a lot of memory.

Race detection using Breadcrumbs is also more precise: particularly for hsqldb, the context-sensitive analysis (column labeled "CS") finds more races than either a completely context-insensitive analysis (labeled "CI") or a partially context-sensitive analysis, which reports a stack trace for the last memory access (labeled "Part CS"). The contexts involved can be quite long, as shown in the columns labeled "Context Depth".

In some cases we help decoding algorithm succeed by providing the program location of the client site. With the client site information the first step of the search does not need to be completely blind (see *decodeWithThreshold*() in Algorithm 1). FastTrack already stores the program locations, so this information is readily available. In addition, we could use 64 bits for calling contexts, allowing us to include both the PCC value and the static program location. The *Unknown* column shows the number of contexts that Breadcrumbs could not reconstruct without a client site but could reconstruct with a client site. Unfortunately, we cannot say for certain if these are the correct contexts, or if they are false matches that happen to start with the given client site. We are working on a methodology that will allow the system to evaluate whether these are contexts are correct. The *Failures* column shows the remaining contexts, which Breadcrumbs was unable to reconstruct, with or without client information.

The following context from xalan shows the difficulty of decoding contexts without sufficient dynamic information (package names omitted for clarity):

```
1 ElemNumber.getFormattedNumber():1375
1 ElemNumber.formatNumberList():1309
1 ElemNumber.getCountString():880
1 ElemNumber.execute():605
2 ElemApplyTemplates.transformSelectedNodes():425
```

Program	Client sites	Threshold	Context reconstructions			Context depth		Races		
			Successes	Unknown	Failures	Avg	Range	CI	Part. CS	CS
eclipse	65,993	100	97	13	13	15±14	[1, 45]	138	2,220	2,477
		1,000	112	14	12	19±15	[1, 49]	132	2,191	2,554
		10,000	98	6	2	12±12	[1, 42]	131	2,067	2,081
		100,000	117	8	9	19±16	[1, 51]	134	2,443	2,640
		∞	Out of memory							
hsqldb	9,148	100	112	1	0	7±4	[1, 13]	109	276	642
		1,000	95	10	0	7±4	[1, 13]	105	264	574
		10,000	106	6	0	7±4	[1, 13]	106	272	634
		100,000	110	0	0	7±4	[1, 12]	109	257	554
		∞	71	0	0	7±4	[1, 13]	72	173	342
xalan	21,474	100	35	0	41	12±11	[5, 30]	76	91	99
		1,000	31	8	24	19±16	[5, 44]	66	74	79
		10,000	36	23	15	15±12	[5, 35]	78	90	92
		100,000	42	36	3	14±10	[5, 29]	92	103	106
		∞	69	0	5	15±9	[5, 28]	71	84	92
pseudojbb	5,602	100	36	0	0	3±1	[2, 4]	56	56	61
		1,000	35	0	0	3±1	[2, 4]	55	55	60
		10,000	37	0	0	3±1	[2, 4]	57	57	62
		100,000	34	0	0	3±1	[2, 4]	54	54	59
		∞	39	0	0	3±1	[2, 4]	59	59	64

Table 3. Race detection performance versus accuracy tradeoff: with a high enough threshold Breadcrumbs reconstructs most calling contexts. The columns show accuracy results for a range of hotness thresholds. Context reconstruction can either succeed, fail, or produce an unknown context; Context depth shows the average depth and standard deviation, and the range of depths; the last three columns show the number of races found using context insensitive analysis (CI), partially context sensitive analysis (Part CS), and fully context sensitivity(CS)

```
2 ElemApplyTemplates.execute()V:216
2 TransformerImpl.executeChildTemplates():2339
8 ElemLiteralResult.execute():710
4 ElemApplyTemplates.transformSelectedNodes():425
2 ElemApplyTemplates.execute():216
2 TransformerImpl.executeChildTemplates():2339
8 ElemLiteralResult.execute():710
4 ElemApplyTemplates.transformSelectedNodes():425
2 ElemApplyTemplates.execute():216
2 TransformerImpl.executeChildTemplates():2339
8 TransformerImpl.applyTemplateToNode():2160
1 TransformerImpl.transformNode():1213
1 TransformerImpl.transform():668
1 TransformerImpl.transform():1129
1 TransformerImpl.transform():1107
  dacapo.xalan.XalanHarness$XalanWorker.run():93
```

Breadcrumbs finds this context, but only when given the initial client site, `ElemNumber.getFormattedNumber():1375`. This context was manually verified as correct. Even with the hotness threshold at 10,000, *every* callsite in the context is hot, resulting in long chains of methods with no dynamic information. As a result, the algorithm falls back on static information: the number to the left of each callsite is the number of possible caller callsites according to the static call graph. While the individual numbers are small, the size of the search space is the product: 1,048,576. This result suggests that future work should consider strategies for avoiding throwing out dynamic instrumentation in long chains of method calls. This problem is particularly challenging for long chains of recursive calls.

5. Related work

Our work falls into a broad category of techniques in program analysis, both static and dynamic, that associate information with calling contexts.

Our work is most closely related to *inferred call path profiling*, which uses the program counter and stack depth (64 bits together) to identify a calling context [Mytkowicz et al. 2009]. The advantage of this approach is that it has essentially no runtime overhead. The downside is that stack depths have very little entropy, resulting in many ambiguous context IDs. To address this problem, this system first modifies the program, padding activation records to help disambiguate contexts. Second, it relies on training runs to build an offline mapping from context IDs to their full calling contexts. Any new contexts observed online cannot be decoded. Our approach imposes a slightly higher overhead, but reconstructs calling contexts using only information collected online. In addition, it computes context IDs using a hash-like function, which has a much lower probability of producing collisions.

Inoue and Nakatani present a technique similar to inferred call path profiling that reconstructs contexts using program counters and stack depths sampled by a hardware performance monitor [Inoue and Nakatani 2009]. Because contexts are sampled, the number of distinct contexts is significantly smaller than in our work. Even for small numbers of contexts, however, accuracy suffers because there are not many bits of entropy in the values representing a context (program counter and stack depth).

Context sensitivity in static analysis. Context sensitivity has been implemented in a number of static analysis algorithms, most notable for pointer analysis. In many cases, these algorithms use explicit call strings or a calling context tree, since the time and space requirements are not as constrained [Lattner et al. 2007; Lhoták and Hendren 2008; Sridharan and Bodík 2006]. Other analyses use a customized calling context numbering to improve BDD compactness [Whaley and Lam 2004]. Computing this numbering, however, relies on analyzing the entire call graph ahead of time.

Context sensitivity in dynamic analysis. Prior dynamic analyses have used either a calling context tree [Ammons et al. 1997; Spivey 2004; Zhuang et al. 2006] or stack walking [Froyd et al. 2005; Nethercote and Seward 2007; Seward and Nethercote 2005] to implement context sensitivity, neither of which is efficient enough for deployed software.

Ball-Larus path profiling inserts instrumentation that computes a unique number for each possible intraprocedural path [Ball and Larus 1996]. An appealing idea is to apply path profiling to the *dynamic call graph* and compute a unique number for each possible context. This approach is problematic because (1) call graphs are typically much larger than control-flow graphs, and possible paths are typically exponential in the size of the graph; (2) dynamic class loading modifies the graph at run time; and (3) recursion leads to cyclic graphs. Wiedermann applies Ball-Larus path numbering to the call graph but avoids the challenges of dynamic class loading and virtual dispatch, avoids recursion by collapsing strongly connected components, and does not evaluate whether large programs can be numbered uniquely [Wiedermann 2007]. *Interprocedural path profiling* captures both inter- and intraprocedural control flow, but it adds complex call edge instrumentation and does not scale to large programs [Melski and Reps 1999].

Sampling-based approaches keep overhead low by profiling the calling context infrequently [Hazelwood and Grove 2003; Whaley 2000; Zhuang et al. 2006]. While these approaches are good at identifying hot calling contexts, they are not suitable for bug-finding clients that need coverage both in cold and hot code.

6. Conclusions

Programmers need context sensitivity to understand the behavior of large, complex programs. Online dynamic analyses that help programmers find bugs have been context insensitive because prior techniques for context sensitivity have been too expensive to deploy. This paper introduced Breadcrumbs, which enables dynamic bug detection analyses to record contexts inexpensively and later recover bug-causing contexts probabilistically. The key to our approach is combining the static call graph with limited dynamic information collected at cold call sites, and using a backwards heuristic search to find potential contexts that match the calling context value. The result is a system that can be applied to a wide variety of existing analyses to help programmers diagnose hard-to-reproduce errors in deployed software.

Acknowledgments

We would like to thank Kathryn McKinley and Ben Wiedermann for support and helpful discussions.

References

B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 314–324, Denver, CO, 1999.

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.

T. Ball and J. R. Larus. Efficient Path Profiling. In *IEEE/ACM International Symposium on Microarchitecture*, pages 46–57, 1996. URL `citeseer.nj.nec.com/ball96efficient.html`.

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.

M. D. Bond and K. S. McKinley. Probabilistic Calling Context. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 97–112, 2007.

M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking Bad Apples: Reporting the Origin of Null and Undefined Value Errors. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 405–422, 2007.

T. M. Chilimbi and M. Hauswirth. Low-Overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, 2004.

C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *ACM Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

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, New York, NY, USA, 2005.

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

H. Inoue and T. Nakatani. How a Java VM Can Get More from a Hardware Performance Monitor. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2009.

C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *ACM Conference on Programming Language Design and Implementation*, pages 278–289, New York, NY, USA, 2007.

O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Transactions on Software Engineering and Methodology*, 18(1):1–53, 2008. ISSN 1049-331X.

D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *ACM Conference on Programming Language Design and Implementation*, pages 134–143, 2009.

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

T. Mytkowicz, D. Coughlin, and A. Diwan. Inferred Call Path Profiling. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2009.

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.

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

J. M. Spivey. Fast, Accurate Call Graph Profiling. *Softw. Pract. Exper.*, 34 (3):249–264, 2004. ISSN 0038-0644.

M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for java. In *ACM Conference on Programming Language Design and Implementation*, pages 387–400, New York, NY, USA, 2006.

SPECjbb2000 Documentation. Standard Performance Evaluation Corporation, release 1.01 edition, 2001.

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

J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM Conference on Programming Language Design and Implementation*, pages 131–144, New York, NY, USA, 2004.

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

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.