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
Tufts University
gzbaker@ll.mit.edu

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

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, have been 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 at cold (infrequently executed) callsites. Breadcrumbs requires no offline training or program modifications, and handles all language features, including dynamic class loading.

We use Breadcrumbs to add context sensitivity to two dynamic analyses: a data-race detector and an analysis for diagnosing null pointer exceptions. On average, it adds 10% to 20% runtime overhead, depending on a tunable parameter that controls how much dynamic information is collected. Collecting less information lowers the overhead, but can result in a search space explosion. In some cases this causes reconstruction to fail, but in most cases Breadcrumbs produces non-trivial calling contexts that have the potential to significantly improve both the precision of the analyses and the quality of the bug reports.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Reliability, Statistical Methods

General Terms Reliability, Performance, Experimentation

Keywords Context sensitivity, dynamic analysis, bug detection

* This material is based on work supported by the National Science Foundation under grants CCF-0916810 and SHF-0910818, by IBM, by the Lincoln Scholars Program at MIT Lincoln Laboratory, and by the Department of the Air Force under contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.

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 at the point of failure is straightforward: with the program halted, a debugging tool can inspect the program's stack directly 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. During normal execution they record extra information that *might* be useful in explaining a failure, if one occurs. Dynamic race detection tools, for example, record information about all memory accesses; when a race is detected they report both the access that triggered the detector and the earlier access with which it conflicts. Ideally, such a bug report would include multiple stack traces: one for the point of failure, and one for each event leading up to it. For these bug detectors the naïve approach of walking the stack and recording an explicit calling context at every event—for example, every memory access—is simply too expensive. As a result, most tools are limited to recording just static program locations (e.g., method and bytecode index), which can be quickly obtained and compactly stored.

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]. A dynamic debugging tool can be made context sensitive by using the PCC value to tag program events or analysis information wherever it would have used a simple program location. The main limitation of the original PCC work is that it provides no way to retrieve information about the calling context that a PCC value represents. As a result, calling contexts can be *distinguished*, but not *identified* and reported to the programmer.

Breadcrumbs collects a small amount of additional information at run time that allows it to decode a PCC value back into its original sequence of calls. Decoding occurs offline (for example, when a bug report is needed) because it involves a potentially expensive search of the PCC value space. The online component, however, consists of lightweight instrumentation, which keeps the overhead during correct execution low enough for use in debugging and analysis tools for deployed software. In addition, unlike recent related techniques, our system does not require any ahead-of-time training, data collection, or program analysis.

The key problem we solve is collecting enough extra information at run time 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, it inverts the computation at each callsite, moving from callees to callers until it reaches `main`. Even though decoding occurs offline, a blind search of the PCC space is much too expensive and results in many false matches because many *statically* possible contexts map to the same PCC value. To constrain the search, Breadcrumbs collects two additional kinds of information at run time: (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 (executed infrequently). This threshold is a tunable parameter that trades online performance for reconstruction accuracy. In a sense, Breadcrumbs computes a probabilistic calling context tree online for the cold code, and then searches the PCC space offline, constrained by the static call graph where available, to fill in the gaps for the hot code.

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: (1) a dynamic race detector based on the *FastTrack* algorithm [Flanagan and Freund 2009] and (2) origin tracking, an analysis for diagnosing null pointer exceptions [Bond et al. 2007]. Our system is able to reconstruct almost all calling contexts for the bugs reported with overheads of 10% to 20% on average (depending on the hotness threshold). 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. For one program, however, reconstruction fails with all but the highest hot callsite thresholds because the contexts involve long chains of hot callsites with multiple statically possible callers. The resulting bug reports provide much more information than context-insensitive static program location, although we have not qualitatively evaluated the usefulness of the reported contexts.

The rest of this paper is organized as follows. First, we discuss the benefits of context sensitivity for dynamic analysis, and the class of dynamic analysis tools that can benefit from this technique. Section 3 presents the Breadcrumbs decoding algorithm and associated runtime support. In Section 4 we evaluate Breadcrumbs by making 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 run time has a number of advantages over other techniques, such as static analysis and testing. It works on all inputs, easily handles language features such as 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 run time, 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 an efficient and 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. Dynamic 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 the 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

Breadcrumbs makes a tradeoff between cost and precision that is targeted at deployable bug detectors. These clients represent a broad class of dynamic analyses with the following properties:

1. Correct execution must be fast: our goal is to provide context sensitivity in a deployed or field-testing environment.
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 context information can occur offline: while correct execution is fast, reconstructing calling contexts for use in bug reports can be expensive because it involves exploring a potentially large search space.

4. Imprecision is acceptable: our system trades precision for performance and may fail to reconstruct a calling context from its encoding or even report an incorrect calling context.

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 that occur in an execution, without 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 significantly. 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.

Events are numerous. Providing context sensitivity in a deployed setting is particularly challenging when events that need calling context information occur very frequently. The dynamic analysis clients presented in Section 4, for example, record every memory access (for race detection), and every null pointer assignment (for origin tracking). The context-insensitive versions of these analyses record the static program location of each event, which is cheap to compute and store. To make them context sensitive, they need to record the calling context of each event. Using an explicit representation, such as a stack trace or a dynamic calling context tree (Section 5), would be much too expensive, as the next two paragraphs explain.

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 callsites) [Nethercote and Seward 2007; Seward and Nethercote 2005]. For infrequent events, however, this cost is easily hidden; for example, a dynamic analysis that records the calling context at each new thread start can afford to simply walk the stack every time. For frequent events, however, both the time and space costs are much too high for use in deployed software [Bond and McKinley 2007; Bond et al. 2009].

An alternative is to build a dynamic *calling context tree* (CCT) 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 CCT, 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. In order to control overhead during correct execution, our system collects only a limited amount of information for use in decoding PCC values. As a result, reconstruction is more difficult and often too expensive to perform online. The information we collect is essentially pieces of the dynamic CCT, constructed only for cold code. In order to decode an arbitrary PCC value, Breadcrumbs must fill in the missing gaps for the hot code, which involves searching the space of PCC values. This search space can be large, especially for long sequences of hot method calls. Therefore, our technique is most suitable for error reporting and logging, where the cost of reconstruction is paid offline.

Imprecision is acceptable. A second consequence of collecting limited information is that Breadcrumbs cannot always successfully decode a PCC value. Due to the large search space, Breadcrumbs reconstructs calling contexts using an iterative deepening algorithm with a fixed time budget (five seconds in our experiments). In some cases, Breadcrumbs fails to reconstruct the correct calling context within the allotted time. In addition, because many statically possible contexts map to the same PCC value, Breadcrumbs may report an incorrect calling context. Eliminating both of these kinds of failure cases is an ongoing part of our research.

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.

Figure 1 contains Java code that will serve as a running example in the discussion below. This program contains a bug: it throws a null pointer exception at the call to `println()` on line 27 (suppose that calling `println(null)` results in a null pointer exception). In Section 4 we describe how origin tracking is used to identify the origin of the null (a call to `clearMsg()`), and how we use Breadcrumbs to augment origin tracking so that it can report the full calling context of the origin, which identifies the *specific* call to `clearMsg()` that causes the bug. Figure 2 shows the call graph for the example, with callsites labeled according to the line number of the call in the code (e.g., "$c_{@23}$" refers to the callsite on line 23.)

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 callsite in `main` and c_n represents the most recent call. Computing PCC values during execution proceeds as follows:

$$\begin{aligned} &p_0 = 0 && \{\text{the } \texttt{main} \text{ calling context}\} \\ &p_1 = f(p_0, c_0) && \{\text{context in callee invoked at } c_0 \text{ in } \texttt{main}\} \\ &... \\ &p_i = f(p_{i-1}, c_{i-1}) \\ &... \\ &p_n = f(p_{n-1}, c_{n-1}) \end{aligned}$$

In Figure 2, for example, the PCC value for the calling context `main()`→`B.virtualMethod()`→`clearMsg()`→`setMsg` would be computed as $f(f(f(0, c_{@24}), c_{@13}), c_{@6})$. Dynamic analysis tools can label analysis information with this PCC value to indicate it was collected in this specific calling context of `setMsg()`.

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

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

```
1  abstract class A {
2    static String message;
3    static { clearMsg(); }
4
5    void setMsg(String val) { message = val; }
6    void clearMsg() { setMsg(null); }
7    String getMsg() { return message; }
8
9    abstract void virtualMethod();
10 }
11
12 class B extends A {
13   void virtualMethod() { clearMsg(); }
14 }
15
16 class C extends A {
17   void virtualMethod() { setMsg("World"); }
18 }
19
20 class Main {
21   public static void main(String [] args) {
22     A obj = new B();
23     obj.setMsg("hello");
24     obj.virtualMethod();
25     // -- Null pointer exception...
26     //    (who caused it?)
27     System.out.println(A.getMsg());
28   }
29 }
```

Figure 1. Example buggy program. This program throws a null pointer exception on line 27. Without Breadcrumbs, the origin tracking dynamic bug detector identifies the call to `setMsg(null)` in `clearMsg()` as the cause. With Breadcrumbs, origin tracking reports the full calling context: `main()` → `B.virtualMethod()` → `clearMsg()`. Notice that `setMsg()` is called in two other contexts: `main()` → `newB` / `JVM` / `static` → `clearMsg()` → `setMsg()`, and `main()` → `C.virtualMethod()` → `setMsg()`.

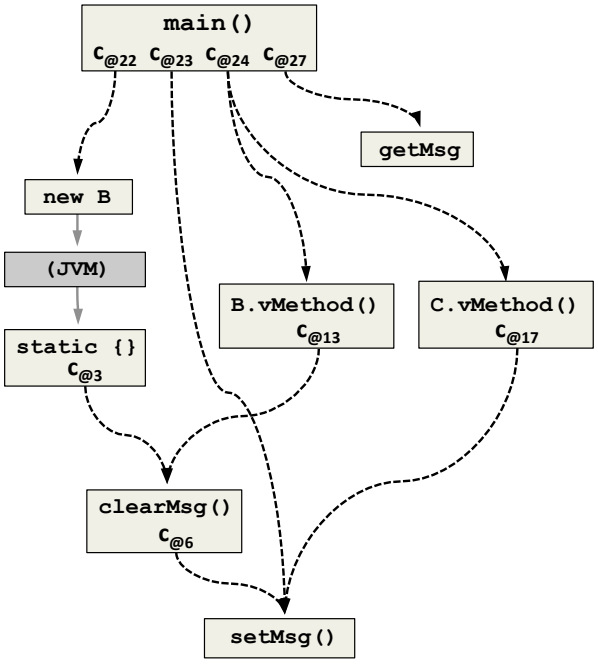

Figure 2. Static call graph for Figure 1. Callsites are labeled with their line number in the code. Gray edges represents calls into and out of the JVM. Although these edges are not present in the call graph that Breadcrumbs computes, it is able to fully reconstruct these calling contexts using dynamic information. The three contexts for `setMsg()` are represented by the PCC values (a) $f(f(f(0, c_{@24}), c_{@13}), c_{@6})$, (b) $f(f(f(0, c_{@22}), c_{@3}), c_{@6})$, and (c) $f(f(0, c_{@24}), c_{@17})$.

One helpful 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 callsites and calling contexts 10s of levels deep, leading to more than $1000^{10} = 10^{30}$ combinations. Because PCC values are only probabilistically unique, considering all possible combinations of callsites 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 one dynamic.

Static information. Decoding uses a *static call graph* (e.g., Figure 2), to prune out callsites that cannot invoke the current method. Assuming it has decoded the path up to callsite 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. It is typically incomplete, however, due to calls in and out of library methods and the virtual machine. In the example in Figure 1, the allocation of a new `B` object in the `main()` method causes the JVM to load classes `A` and `B`, and run the static initializer in class `A`, which invokes `clearMsg()`. When reconstructing this calling context, Breadcrumbs uses the static call graph edge from `clearMsg()` back to callsite $c_{@3}$ in the static initializer, but then finds no incoming edges to this code. Our algorithm can still reconstruct this calling context, but relies on dynamic information (see below) to discover that the static initializer is invoked as a result of callsite $c_{@22}$ in `main()`.

Dynamic information. 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.

Our solution uses the dynamic compiler to add instrumentation at every application callsite that records the set of PCC values generated at that callsite. Specifically, the *per-callsite PCC values* for callsite c_{i-1} are all PCC values p_i that PCC instrumentation computed at c_{i-1}. The resulting instrumentation computes the following for each callsite:

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

These per-callsite PCC values dramatically improve the efficiency of the search: to find the next callsite in the decoding of PCC value p', we just have to find the callsite whose set $values_c$ contains p'. We then invert f using c to obtain p, and continue up. Collisions among these per-callsite PCC values are highly unlikely, because the sets are relatively small, and a collision would only occur if two different contexts ending at the same callsite coincidentally get the same PCC value. With high probability, each PCC value observed will reside in a single per-callsite PCC set, pruning the search to a single candidate callsite at each step.

This information is also helpful in cases where the static call graph is incomplete. For example, when reconstructing the calling context of the static initializer in Figure 1 (line 3), the static call graph (Figure 2) contains no edge back to the callsite $c_{@22}$ (since control passes through the JVM), but the PCC value for this calling context will reside in the per-callsite PCC set for callsite $c_{@22}$, allowing successful reconstruction.

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. Our solution to this problem is discussed next.

3.3 Trading accuracy for performance

To control the costs of recording every per-callsite PCC value, Breadcrumbs stops recording per-callsite PCC values in code that is frequently executed, and therefore likely to impact performance. To implement this throttle, we add code to the per-callsite instrumentation that tracks each callsite's execution frequency (in addition to recording per-callsite PCC values). If the frequency of a callsite exceeds a tunable *hotThreshold*, Breadcrumbs discards any per-callsite values recorded so far for that callsite, and disables its per-callsite instrumentation. In our implementation, we can completely remove per-callsite instrumentation (including the frequency computation and check) if and when the dynamic optimizing compiler recompiles the method containing the callsite.

The *hotThreshold* is a key parameter to Breadcrumbs that significantly affects its performance and reconstruction accuracy. Section 4 explores this tradeoff empirically. 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 computes a decodable identifier for each dynamic calling context. Notice, though, that these calling contexts consist of sequences of callsites only, and cannot identify arbitrary program points. For example, in the code in Figure 1 the PCC value $f(f(f(0, c_{@24}), c_{@13}), c_{@6})$ represents the entire call to the method `setMsg()` in that calling context. Many client analyses, however, need to be able to identify program points at a finer grain. A race detector, for example, will want to use PCC values as context-sensitive identifiers for individual memory reads and writes.

One way to solve this problem is for clients to record a pair of values: the PCC value for the calling context, plus the static location of the event within the current method. The downside of this approach is that it requires two words to fully represent a context-sensitive program location.

The solution we adopt is to perform one more step of the PCC computation at the point the client requests a context sensitive program location. We use the current location as the c value, regardless of whether or not it is actually a callsite. We call these pseudo-callsites *client sites*, since they serve as sites in the encoding and decoding, but are chosen by the client analysis. Breadcrumbs computes the PCC value for client sites as follows:

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

Breadcrumbs 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 in an effort to find the most likely calling context for a given PCC value, in reasonable time.

Each iteration is a depth-limited backward search of the PCC space, following potential call edges from callees back to the callsites in their callers, continuing until it finds the PCC value 0, indicating `main`. Not all edges are equal, however: when static and dynamic information are 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:

(1) 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.

(2) 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.

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 is 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-callsite 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* $=$ **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

(3) 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. In Figure 2, for example, reconstructing the calling context of the static block requires Breadcrumbs to inspect all per-callsite PCC sets in order to identify $c_{@22}$ as the proper one.

(4) No static, no dynamic: a site in neither set. The algorithm must consider *all* hot sites as potential callers. This case occurs when there is a callsite that calls into the VM or libraries *and* the callsite becomes hot, so the dynamic per-callsite PCC values are discarded.

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 not dynamic information. 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 call graph 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. An alternative we evaluate is to give the client 64 bits 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—these experiments measure the overhead of computing PCC values, building the static call graph, and collecting dynamic per-callsite PCC values (as described in Section 3.2). We then demonstrate its utility by adding context sensitivity to two existing dynamic bug detectors, one for detecting races and the other for debugging null pointer exceptions. It required modest effort to integrate these analyses with Breadcrumbs.

The decoding algorithm can reconstruct many but not all calling contexts for these two clients within a time limit of five seconds per context. Accuracy varies significantly depending on the hot callsite threshold, the client, and the program. These calling contexts are often deep and nontrivial, so the resulting bug reports offer significantly more information than the context-insensitive information provided by the unmodified systems, but we have not qualitatively evaluated the usefulness of these contexts. The cases for which reconstruction fails involve calling contexts with long, uninterrupted chains of hot callsites (often in recursive or mutually recursive methods) that have multiple hot callers. In these cases, dynamic information is unavailable, and the blow-up in possible static callers becomes too large. As a result, reconstruction exceeds the time limit, reports incorrect contexts, or both.

4.1 Methodology

Implementation. We implemented Breadcrumbs in Jikes RVM 3.1.0. Our implementation is publicly available on the Jikes RVM Research Archive.[1]

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. 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 two 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 3 shows the overhead of several Breadcrumbs configurations compared to unmodified Jikes RVM. Each sub-bar is the median of 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. This option makes the reconstruction algorithm fast and very accurate, but the instrumentation adds almost 100% overhead on average. 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.

Table 1 shows the relationship between the hot callsite threshold, the number of callsites qualifying as hot, and the number of PCC values recorded as a consequence. With larger hot thresholds, fewer callsites reach the threshold, and therefore more callsites retain the instrumentation that collects per-callsite information. Each number in the table is computed as the average over 10 trials.

The *Static callsites* columns show the total number of callsites in each program, and the number of these that qualify as hot under each threshold. *Executed* is the total number of callsites in the static call graph, but only includes callsites that the application actually executed. Breadcrumbs is only concerned with executed sites, which it can determine easily because callsite instrumentation already tracks execution frequency in order to determine if the site is hot. The *Hot* column shows the number of callsites that reach the given hot threshold. Of course, in the case of no threshold, no site becomes hot.

The *PCC values stored* columns show how many values Breadcrumbs stores in per-callsite PCC sets. Since hot callsites have their instrumentation removed, as the hot threshold goes up, the number of uninstrumented callsites goes down, resulting in more per-callsite PCC values stored and higher overheads. *Dynamic values*

[1] http://www.jikesrvm.org/Research+Archive

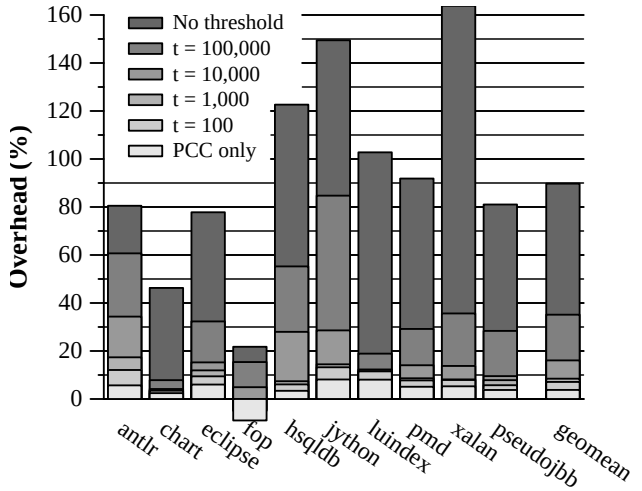

Figure 3. Overhead of Breadcrumbs for various hot callsite thresholds (measured as number of callsite invocations). Recording per-callsite PCC values for all callsites ("No threshold") adds about 90% overhead, but results in fast and accurate reconstruction. Hot thresholds of 100 to 1000 offer a compromise, enabling reconstruction of most real-world contexts, while adding only 10% to 20% overhead.

is the total number of times the instrumentation accesses each per-callsite PCC set (a hash table operation). It grows dramatically with the hot threshold, which accounts for 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.

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 1,000 and 10,000 provide an acceptable balance: low enough overheads for most deployed software, but still able to reconstruct many 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]. At run time, it tags every null value with the program location of its origin, and tracks it through the program (through assignments, method calls, and field loads and stores). If a null pointer exception occurs, it can report the origin of the null that caused it, in addition to the location of the exception itself, which is already reported by the VM.

The key idea that makes this analysis efficient is that origins are encoded directly in the null values themselves. Null references do not need a full 32 bits; they just need to be distinguishable from non-null references. Origin tracking exploits this fact by using just the five high bits of every reference to indicate null: if these bits are zero, the reference is null. It can then use the remaining 27 bits to encode a program location.

In the prior work, origins are represented as a method and bytecode index. In many cases, though, this information is not sufficient to debug the program. In the code in Figure 1, for example, origin tracking would report the null value from the call to `setMsg()` in method `clearMsg()` as the cause of the null pointer exception. Since there are multiple calls to `clearMsg()`, however, and its *purpose* is to null the reference, this information is of little value for debugging. What we want is the full calling context of the specific call to `clearMsg()` that causes the problem.

Origin tracking is challenging to make context sensitive because there are so many null values assigned at run time, most of which never cause an exception. As such, it is well suited for Breadcrumbs. Instead of encoding a static program location in each null value, we store the PCC value computed at that point. Since origins only get 27 bits, we modified Breadcrumbs to compute 27-bit PCC values (instead of the usual 32-bit PCC values in our implementation) for this client. With Breadcrumbs, origin tracking can provide two complete stack traces for a null pointer exception: one for the

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

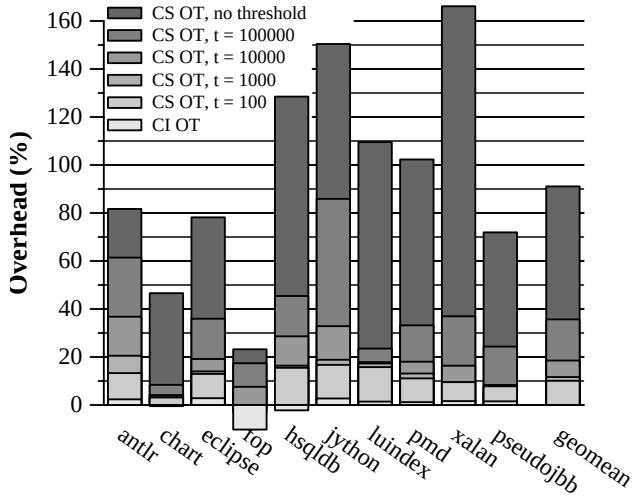

Figure 4. Overhead of origin tracking with and without context sensitivity (i.e., with and without Breadcrumbs) for various hot callsite thresholds.

point of failure, and one for the origin of the null value that caused it.

Figure 4 shows the overhead of origin tracking with several Breadcrumbs configurations, and one without Breadcrumbs, compared to unmodified Jikes RVM. Each sub-bar is the median of 10 trials. The *CI OT* configuration is the overhead of origin tracking alone, which does not use Breadcrumbs and reports only context-insensitive origins. The other configurations use Breadcrumbs with various hot callsite thresholds. These results mirror Breadcrumbs-only results in Figure 3. 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 (NPEs), which are available publicly as the *Bad Apples Suite*.[2] The suite provides 12 NPE scenarios (each one consists of a program and a NPE-inducing input set). We evaluate Breadcrumbs-enabled origin tracking with 10 of 12 exceptions from the Bad Apples Suite. We do not evaluate the two Eclipse NPEs because Jikes RVM 3.1.0 does not execute the Eclipse GUI correctly. The Jikes RVM development trunk fixes the problem, so porting the Breadcrumbs implementation to the trunk should allow these NPEs to be evaluated (we ran out of time to try it).

Table 2 shows how well Breadcrumbs reconstructs calling contexts for null pointer origins for the 10 NPEs (see the original paper for details of the exceptions and origins [Bond et al. 2007]). Our system is able to reconstruct all but one of the calling contexts, regardless of the hot 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 NPEs almost immediately, so few callsites 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 callsites. For example, Breadcrumbs reports the following context-sensitive origin for the Jython #2 exception:

[2] `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 null pointer exceptions from the Bad Apples Suite. Reconstruction of these contexts is not sensitive to hot threshold (because these NPEs occur early in execution).

```
at org.python.core.PyObject.fastGetDict():2723
at org.python.core.PyObject.getDoc():360
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 (the deepest point in the stack trace).

The Breadcrumbs implementation cannot reconstruct the calling context of the origin in JODE. At the higher thresholds, Breadcrumbs completes an exhaustive search without finding a matching context, indicating a bug in our implementation, which we have not been able to find or fix.

4.3 Race detection

In these experiments, we use a dynamically sound and precise race detector to evaluate how accurately Breadcrumbs can reconstruct nontrivial, buggy calling contexts. We implemented a dynamic race detector in Jikes RVM based on the sound and precise *FastTrack* algorithm [Flanagan and Freund 2009], which adds time and space overhead too high for production. Breadcrumbs would be better suited to deployable race detection algorithms based on sampling [Bond et al. 2010; Marino et al. 2009]. We have not combined the *Pacer* [Bond et al. 2010] and Breadcrumbs implementations. The race detector we evaluate here is a preliminary implementation of Pacer that only works correctly at 100% sampling rates; it is functionally equivalent to FastTrack. Our evaluation uses four multithreaded programs: DaCapo's eclipse, hsqldb, and xalan; and SPEC pseudojbb.

Each data race consists of a pair of memory accesses: the current access (at the point the race is detected) and the prior access (an earlier access with which the current access conflicts). In order to be useful, bug reports need to include information about both accesses. The challenge in making race detection context sensitive is in providing information about the earlier accesses: since *any* memory access could later turn out to be the first of the pair in a race, the race detector must record the calling contexts of all memory operations. For this reason, existing race detectors report only context-insensitive program locations for the first access.

Program	Client sites	Threshold	Context reconstructions Successes	(using client site)	Failures	Context depth Avg	Range	Races CI	Part. CS	CS
eclipse	67,783	100	239	(46)	30					
		1,000	257	(21)	12					
		10,000	257	(13)	12					
		100,000	261	(0)	8					
		500,000*	261	(0)	8	15±13	[2, 47]	275	5,847	6,131
hsqldb	9,343	100	223	(18)	0					
		1,000	223	(15)	0					
		10,000	223	(9)	0					
		100,000*	223	(0)	0	7±4	[1, 13]	215	559	1,214
xalan	22,647	100	68	(0)	76					
		1,000	68	(0)	76					
		10,000	68	(0)	76					
		100,000	118	(50)	26					
		∞	137	(0)	7	15±10	[5, 29]	152	184	198
pseudojbb	5,624	100	70	(0)	0					
		1,000	70	(0)	0					
		10,000	70	(0)	0					
		100,000	70	(0)	0					
		∞	70	(0)	0	3±1	[2, 4]	110	110	120

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). *Both eclipse and hsqldb run out of memory with an the threshold = ∞, so we use finite thresholds that are large without running out of memory (obtained by trial and error) in order to obtain a baseline for comparison.

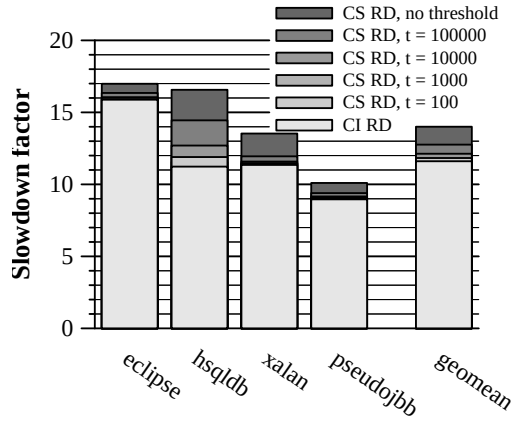

Figure 5. Overhead of race detection with and without context sensitivity for various hot callsite thresholds.

Overhead. Figure 5 shows the overhead of race detection with several Breadcrumbs configurations, and one without Breadcrumbs, compared to unmodified Jikes RVM. Each sub-bar is the median of five 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 in expensive clients, the race detector and Breadcrumbs (at high hot thresholds) both add high memory overhead. We believe these effects combine superlinearly, mainly by triggering garbage collection much more frequently.

Accuracy. Table 3 shows how well Breadcrumbs reconstructs calling contexts for the first access of racy access pairs. For this analysis, the *Client sites* are the subset of memory accesses that could be involved in a race. The other statistics are all *sums* across 10 trials. Under the heading *Context reconstructions*, we report the number of calling context reconstructions that succeeded and failed for each combination of benchmark and threshold. The contexts involved can be quite long, as shown in the columns labeled *Context Depth*.

Breadcrumbs is able to reconstruct most of the calling contexts regardless of the hot threshold. In larger, more complex benchmarks, such as eclipse and xalan, however, we start to see the effects of having less dynamic information at the lower thresholds. (With no threshold, eclipse and hsqldb run out of memory because Breadcrumbs *with no threshold* and race detection each use a lot of memory.) The column labeled *(using client site)* shows the number of contexts that were *only* reconstructed successfully by providing the initial client site to the search. With the client site, the first step of the search is not blind (see *decodeWithThreshold*() in Algorithm 1). Subtracting the *(using client site)* column from the *Successes* column yields the number of contexts reconstructed correctly *without* using the initial client site.

Race detection results using Breadcrumbs are also more precise than context-insensitive race detection results, in the sense that they differentiate the same static callsites participating in different context-sensitive races. Context-sensitive analysis (column labeled *CS*) finds more races than either a completely context-insensitive analysis (labeled *CI*) because it can observe the same static program location participating in different races in different calling contexts. The *Part. CS* column shows results for partially context-sensitive analysis, which reports a stack trace for the second memory access only.

Reconstruction failures. 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
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
  XalanHarness$XalanWorker.run():93
```

Even with the hot 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 *hot* 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.

Breadcrumbs 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 (program counter and stack depth) representing a context.

Sumner et al. present *precise calling context encoding* (PCCE), which encodes calling contexts for C programs [Sumner et al. 2010] using techniques from path profiling [Ball and Larus 1996]. Compared to Breadcrumbs, PCCE offers a non-probabilistic approach, very low overhead, and a simpler decoding scheme. Unlike Breadcrumbs, however, the technique would not work in the presence of dynamic class loading and virtual methods, and the encoding often does not fit in a single integer.

Context sensitivity in static analysis. Context sensitivity has been implemented in a number of static analysis algorithms, most notably 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.

Interprocedural path profiling captures both inter- and intraprocedural control flow, but it adds complex call edge instrumentation and does not scale well 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 callsites, and using a backward 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 are grateful to Kathryn McKinley for generous support and feedback. We would also like to thank Ben Wiedermann and Xiangyu Zhang for helpful discussions, and Todd Mytkowicz and the anonymous reviewers for valuable feedback on the text.

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, 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.

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.

M. D. Bond, V. Srivastava, K. S. McKinley, and V. Shmatikov. Efficient, Context-Sensitive Detection of Semantic Attacks. Technical Report TR-09-14, The University of Texas at Austin, 2009.

M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional Detection of Data Races. In *ACM Conference on Programming Language Design and Implementation*, 2010.

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, 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*, pages 137–154, 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, 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.

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*, pages 175–190, 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.

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, 2006.

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

W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise Calling Context Encoding. In *ACM International Conference on Software Engineering*, 2010.

J. Whaley. A Portable Sampling-Based Profiler for Java Virtual Machines. In *ACM Conference on Java Grande*, pages 78–87, 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, 2004.

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.