Efficient, Context-Sensitive Dynamic Analysis via Calling Context *Up*trees

Ohio State CSE technical report #OSU-CISRC-7/12-TR14, July 2012

Jipeng Huang
Ohio State University
huangjip@cse.ohio-state.edu

Michael D. Bond
Ohio State University
mikebond@cse.ohio-state.edu

Abstract

State-of-the-art dynamic bug detectors such as data race and memory leak detectors report program locations that are likely causes of bugs. However, *static* program location is not enough for developers to understand the behavior of increasingly complex and concurrent software. *Dynamic calling context* provides additional information, but it is expensive to record calling context frequently, e.g., at every read and write. Context-sensitive dynamic analyses can build and maintain a *calling context tree* to track calling context, but in order to reuse existing nodes, CCT-based approaches require an expensive lookup.

This paper introduces the *calling context uptree* (CCU), a new data structure in which each node points "up" to its parent instead of "down" to its children. A CCU-based approach adds low time overhead because it can allocate new nodes quickly, but it adds high space overhead because it cannot reuse existing nodes. However, tracing-based garbage collection (GC) collects unused CCU nodes naturally and efficiently. To reduce space used by the remaining nodes, we present an efficient algorithm that piggybacks on GC to merge redundant nodes lazily.

We implement our CCU-based approach in a high-performance Java virtual machine and integrate it with memory leak and data race detectors so they can report context-sensitive sites that cause bugs. We show that despite allocating instead of reusing nodes, our CCU-based approach adds low overhead to these clients and keeps space overhead low by relying on GC and performing lazy merging. The CCU can thus provide low-overhead context sensitivity to a variety of dynamic analyses that report bug causes.

1. Introduction

To provide more functionality and to scale to hardware that provides more instead of faster cores, software is becoming increasingly complex and concurrent. These trends make it harder for developers to write correct programs and to reproduce, find, diagnose, and fix bugs in existing programs.

Dynamic program analysis can help developers make software more reliable by identifying errors and their *causes*. For example, data race detectors track the program locations that last accessed each variable [12, 20, 22, 36, 43]. When they detect a data race, they can thus report the two program locations involved in the data race: the program location that last accessed the variable, as well as the current program location. Other dynamic analyses such as memory leak detection [13, 16, 40], dynamic slicing [1, 54], and atomicity violation detection [23, 35] track program locations in order to report likely bug causes. To find and diagnose bugs that do not manifest during testing, dynamic analyses can run in *production systems*, where keeping overhead low is a key constraint.

To save time and space, almost all dynamic program analyses track *static* program locations, e.g., a method and line number. However, static locations are often not enough to understand *what the program was doing* at that point. Static locations are increasingly inadequate as software becomes more complex and concurrent. In complex, object-oriented software with many small, virtual methods, a static program location may be invoked from many unrelated contexts. Modern software often consists of integrated components written by many developers, which complicates this guessing game. In concurrent programs, the operation that causes a bug may execute on a different thread from the operation that detects the bug or triggers failure, again making it challenging to determine program behavior from a static location.

Context sensitivity. Developers need more information than a static program location to understand what the program was doing: they need to know the *dynamic execution context*. *Dynamic calling context* is the list of active call sites, similar to an exception stack trace. (Prior work on static analysis considers other forms of context sensitivity that includes object allocation sites and types [45].)

Developers are already used to getting a stack trace *upon failure or when a bug is detected*, to help them understand what the program was doing when it failed. Obtaining the calling context at these points is straightforward: the runtime system simply walks the current thread's call stack. Overhead is not a concern because walking the stack occurs only once.

To understand bugs, developers also need to know the calling context of program locations that are *potential bug causes*. For example, developers need to know the calling context of the *first* of two accesses involved in a data race, and they need to know the calling context of a program location that is leaking memory. Reporting these prior program locations' calling contexts is challenging. Dynamic analysis needs to record calling context extremely frequently, e.g., at every program read and write, since it is impossible to predict which accesses' calling contexts might need to be reported later.

Context sensitivity can also enhance dynamic analyses that infer bug causes from observing program behavior that is well correlated with failure [21, 27, 28, 34, 35]. Prior approaches are context insensitive: they use static program locations as features. By using context-sensitive locations, these approaches would become more precise and be able to detect bugs due to program locations called from both buggy and correct contexts.

Prior approaches. While most analyses record and report only static program locations, some prior work captures calling context

but either has serious limitations or overhead too high for production use. Walking the stack whenever context is needed is expensive unless it is rare [14, 24, 39, 44, 50]. Recent approaches reconstruct calling context when needed from limited information that is cheap to collect, but these techniques are often probabilistic, and none scale well with program complexity [11, 31, 37, 48].

Dynamic analysis can build and maintain each thread's current position in a *calling context tree* (CCT), in which each node represents a distinct calling context [4, 46, 55]. Each CCT node maintains a mapping from callee sites to callee contexts. This mapping can be implemented in various ways such as a hash table since the mapping is sparse, or as a list if the number of callee sites is relatively small. Each program call thus requires a nontrivial lookup to find or construct the corresponding callee context node, if any, slowing programs by two or more times.

The calling context uptree. This paper proposes a novel approach based on a new data structure called the *calling context uptree* (CCU). Each CCU node does *not* have pointers "down" to its children. Instead, each node points only "up" to its parent. This simple difference makes a significant impact. Instead of using an expensive lookup at each program call, a CCU-based approach simply allocates a new node at each program call, setting the node's parent to the caller context's node. Because CCU nodes do not reference their children, dynamic analysis at a program call cannot reuse existing callee context nodes. It adds space overhead by allocating, instead of mostly reusing, child nodes at every program call. However, tracing garbage collection (GC) collects transitively unreachable nodes efficiently, and we have developed an algorithm that piggybacks on GC to merge redundant CCU nodes lazily.

We implement our CCU-based approach in a high-performance research JVM and integrate it into two dynamic bug detection analyses—a staleness-based memory leak detector and a happens-before race detector—that track objects' "last access" sites to report likely bug causes. We show that the CCU provides dynamic context sensitivity to the leak detector by adding 33% or 93% overhead (depending on how leaf sites are stored), and to the race detector by adding 57% overhead. We show that this performance compares favorably with a CCT-based implementation. We also show that the overhead CCU adds to the sampling-based race detector scales naturally with the sampling rate. Finally, we evaluate two leaky sites and conclude that context sensitivity helps diagnose one leak and at least adds significant information to the other site. These results suggest that the CCU can provide low-overhead context sensitivity to dynamic bug detection analyses.

2. Background

2.1 Context-Sensitive Dynamic Analysis

Dynamic analyses such as memory leak and data race detectors keep track of program locations that allocated or last accessed program variables. This paper refers to such analyses as *client analyses*. Client analyses typically maintain references to program locations, which we call *client metadata*. Analyses often maintain per-variable client metadata by adding extra word(s) to object headers or adding shadow memory [38].

A client analysis that tracks *static* program locations is *context insensitive*. By using calling contexts to represent program locations, a client analysis becomes *context sensitive*. Being context sensitive enables an analysis to report context-sensitive program locations to programmers, or to detect bugs more precisely by differentiating program locations called from different contexts, or both.

In this paper, dynamic calling context is (1) the static program location, which we call the *client site*, plus (2) the set of active call sites. A (call or client) site consists of a method and a bytecode index (which uniquely identifies bytecode operations, unlike line numbers):

```
class Site {
  Method method;
  int bytecodeIndex;
}
```

In the following example calling context, the first line is the client site, and the other lines are call sites:[1]

```
org.apache.xerces.framework.XMLParser.parse:1111
org.apache.tomcat.util.digester.Digester.parse:1561
org.apache.catalina.startup.TldConfig.tldScanStream:514
org.apache.catalina.startup.TldConfig.tldScanJar:472
...
java.lang.Thread.run:595
```

```
1  class X { boolean flag; }
2  class A {
3    m(X x) {
4      if (x.flag) {
5        globalSet.add(x); // x escapes
6      }
7  } }
8  class B extends A {
9    m(X x) {
10     x.flag = true;
11     super.m(x);
12 } }
13 main() {
14   A a = new A(); B b = new B();
15   for(A tmp : {b, a, b}) {
16     tmp.m(new X());
17   }
18   ...
19 }
```

Figure 1: Example program written in Java pseudocode.

2.2 The Calling Context Tree

Ammons et al. introduce the calling context tree (CCT), in which each node represents a distinct calling context executed by the program [4, 46, 55]. Each node points to its existing child nodes:

```
class CCTNode {
  Site site;
  Map<Site,CCTNode> children;
}
```

Each node maintains a map from child sites to child nodes so that at each call, dynamic analysis (e.g., instrumentation added to the compiled program) can reuse the existing child context node, if any. Reusing existing nodes is important because programs execute many more dynamic than distinct contexts [14]. Direct mapping is impractical because a call site may have many statically possible callee call sites: a call site may call several virtual methods, and each of these methods may contain many call sites. Furthermore, the number of statically possible callee call sites grows over time due to dynamic class loading. Implementations must use a sparse mapping implementation such as a hash table, or a list if there are relatively few distinct, executed callee sites. Existing CCT-based dynamic analyses thus require a nontrivial lookup at essentially every call, slowing programs by two or more times, making them unsuitable for production systems [4, 46].

Figure 1 shows an example program written in Java-like pseudocode. In the example, main():16 has four statically possible

[1] We present calling contexts using line numbers instead of bytecode indices because line numbers are more helpful when examining source code.

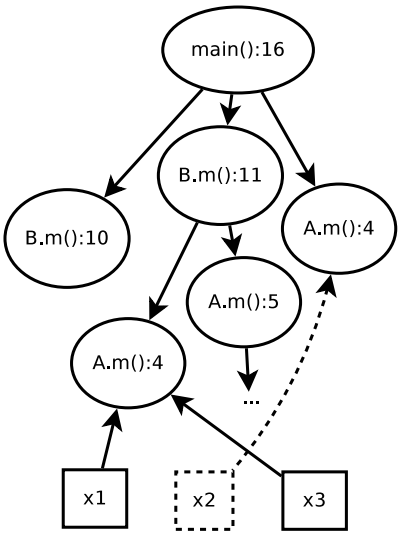


Figure 2: Calling context tree (CCT) corresponding to Figure 1.

callee sites since it has two possible callee methods (A.m() and B.m()) that each contain two sites (the client sites A.m():4 and B.m():10, and the call sites A.m():5 and B.m():11). Similarly, B.m():11 has two statically possible callee sites (A.m():4 and A.m():5). We suppose the program continues executing at line 18 (...), so long-term space overhead matters.

Figure 2 shows the CCT that a CCT-based dynamic analysis would allocate for the example program in Figure 1. Suppose a client analysis is recording the last access (read or write) to each object. Ovals represent CCT nodes, and squares represent heap objects, labeled x1–x3 by allocation order. Down edges point from CCT nodes to their children, and up edges point from per-object client metadata (e.g., object headers) to CCT nodes; the client metadata records the last access to each object. The edge from A.m():5 to "..." represents extra CCT nodes created by globalSet.add().

The program accesses x1 and x3 first at B.m():10 ← main():16, but later accesses each object at A.m():4 ← B.m():11 ← main():16. Nonetheless, the context B.m():10 ← main():16 remains in the CCT. Similarly, the x2 dies quickly (indicated with dashed lines) because A.m():5 does not add x2 to globalSet, but the last-access context A.m():4 ← main():16 survives because it is reachable. This paper refers to nodes that can no longer be used by client analyses as *irrelevant* nodes. Admittedly, the CCT could support garbage collecting irrelevant nodes using weak references [26], e.g., by using a weak hash map for child nodes and also maintaining parent pointers.

The more critical problem with the CCT is that nodes can have many statically possible callee contexts, which can grow over time, so finding a child context in the child map requires a relatively expensive indirect lookup at each program call. Our new approach addresses this problem, as well as how to collect irrelevant nodes efficiently.

Alternatives to the CCT. Instead of reusing existing context nodes, a *call tree* constructs a new node for every dynamic call. In this way, a call tree is related to our calling context uptree (presented next), except that each call tree node maintains pointers to its child nodes, making it expensive to construct nodes and difficult for GC to collect irrelevant nodes. Analyses can build and maintain a *dynamic call graph*, which maintains only one node per call site, losing the ability to reconstruct client sites' calling contexts.

3. The Calling Context Uptree

This section describes the *calling context uptree* (CCU), our new alternative to the CCT that provides efficient, always-available context sensitivity to dynamic analyses. After introducing the CCU, we describe how a CCU-based approach works: how dynamic analysis constructs CCU nodes, and how garbage collection (GC) and lazy merging of redundant nodes reduce space overhead.

3.1 The Basic Idea

The *calling context uptree* (CCU) is a new data structure in which each node points "up" to its parent instead of "down" to its children:

```
class CCUNode {
  Site site;
  CCUNode parent;
}
```

Because CCU nodes do not point to their child nodes, it is essentially impossible to reuse existing nodes. Our CCU-based approach thus *allocates a new node* at every program call and client site. This approach adds low time overhead at each site (especially when using optimizations described in Section 4.1), but space overhead is a serious concern because existing nodes are not reused, so many *redundant nodes* (nodes representing the same calling context) and *irrelevant nodes* (nodes no longer used by the client analysis) will be created. We show how GC naturally collects irrelevant nodes, and we also present an approach that lazily merges redundant CCU nodes to balance space and time.

3.2 Constructing the CCU

A CCU-based analysis must allocate, at a minimum, CCU nodes to represent the context of every client site (e.g., every read and write).[2] An analysis can construct CCU nodes eagerly or lazily. *Eager construction* allocates a node at every *call* site and passes the node as an extra, implicit call parameter:

```
caller(..., node) {
  ...
  // program call site
  callee(..., new CCUNode(callSite, node));
  ...
}
```

At every *client* site, eager construction allocates a new node from the client site and parent node, and uses it in an analysis-specific way such as storing the node in an accessed object's header:

```
o.metadata = new CCUNode(clientSite, node);
read o.f; // client site
```

In contrast, *lazy construction* allocates CCU nodes only at client sites:

```
o.metadata =
  new CCUNode(clientSite, getNode());
read o.f; // client site
```

The VM-specific function getNode() walks the stack and allocates CCU nodes recursively until it finds a stack frame for which the CCU node has already been constructed. Section 4.1 describes how our implementation stores pointers to CCU nodes on stack frames and uses method return addresses to represent sites efficiently.

We focus on lazy construction because it scales well with the client analysis, since it only constructs CCU nodes needed to represent the contexts of client sites.

Eager and lazy construction are not unique to a CCU-based approach. A *CCT*-based approach may also construct CCT nodes

[2] The CCU naturally handles recursive call sites because each CCU node represents one *dynamic* call site or client site.

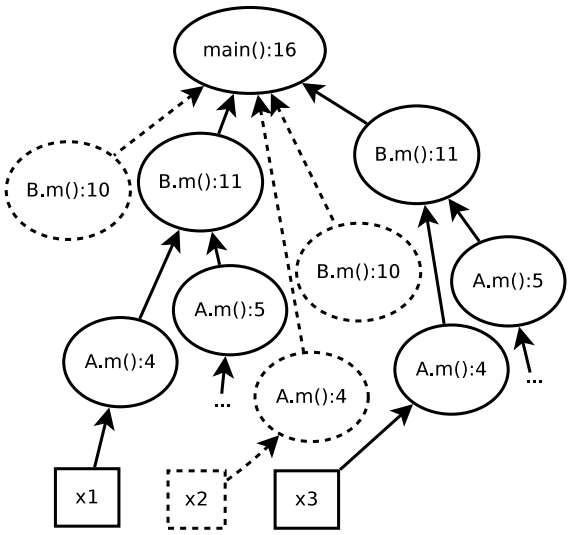


Figure 3: Calling context uptree (CCU) corresponding to Figure 1, without merging of redundant nodes. The dashed lines represent objects that become irrelevant (unreachable) by line 18 of the example program.

lazily or eagerly [50, 55]. A CCU-based approach and a CCT-based approach will each *look up* the same number of nodes; the key difference is that "looking up" a CCU node always means allocating a new node, whereas looking up a CCT node often means reusing an existing node.

3.3 Collecting Irrelevant Nodes

Because CCU nodes point only "up" to their parents, GC naturally collects irrelevant nodes (nodes that are transitively unreachable and thus are no longer used by the client analysis). In fact, CCU *relies* on GC to collect the many nodes that become irrelevant quickly. Tracing-based GC is well suited to collecting CCU nodes because tracing is proportional to the dead nodes, not the live nodes [32]. A CCU-based approach should work especially well with generational GC, which collects newly allocated objects more frequently than older objects, based on the weak generational hypothesis that many objects die young [29].

3.4 Example CCU

Figure 3 shows the CCU after executing the code in Figure 1. The contexts B.m():10 → main():16 and A.m():4 ← main():16 are irrelevant since x1 and x3's last access changed, and x2 died. We represent these nodes with dashed lines to show that GC collects them automatically since they are unreachable.

3.5 Merging Redundant Nodes

While GC naturally collects irrelevant CCU nodes, *redundant* nodes (nodes representing the same calling context) can still add significant space overhead in a CCU-based approach. In contrast, the *CCT* disallows redundant nodes by always reusing existing nodes. We contend that the CCT's approach essentially wastes time reusing nodes because many redundant nodes become irrelevant quickly.

To balance space and time, our CCU-based approach periodically merges redundant nodes so that each relevant context is represented by just one *unique* node. Our merging algorithm piggybacks on tracing GC, which already traverses all objects. We also believe our technique could piggyback on a *concurrent* tracing GC.

```
1  CCUNode traceAndMerge(CCUNode node) {
2    // first merge the parent
3    if (node.parent not yet processed) {
4      node.parent = traceAndMerge(node.parent);
5    }
6    uniqueNode =
7      node.parent.childMap.get(node.site);
8    if (uniqueNode == null) {
9      // node is the unique node
10     node = markLiveAndPossiblyCopy(node);
11     node.parent.childMap.put(node.site, node);
12     return node;
13   }
14   return uniqueNode;
15 }
```

Figure 4: GC tracing pseudocode for tracing and merging CCU nodes. Fine-grained synchronization (not shown) ensures atomicity of lines 6–11. Nodes without parents are not merged (not shown).

Our algorithm is not suitable for non-tracing GC, although efficient reference-counting algorithms still trace young objects [9], providing an opportunity for merging redundant nodes.

The goal of merging is to determine, for each node, its unique node—which may be the node itself or a different node—and redirect all incoming pointers to the unique node.

Eager tracing. GC typically traces (marks live and possibly moves) an object before tracing the object's children (referenced objects) [25]. Regular GC tracing is not well suited to merging, which needs to determine a node's unique node based on the node's unique parent. We modify GC tracing to be *eager* for nodes only; GC continues to trace other heap objects normally. Eager tracing traces a node's parent pointer recursively before tracing the node. Eager tracing is reasonable for CCU nodes because each node has only one outgoing pointer, and node chains are bounded by the program call depth, since nodes reflect program call chains.

Looking up unique nodes. To merge redundant nodes, our implementation needs to support looking up existing unique nodes based on node equality. Two nodes are *equal* if their sites are the same and their parent nodes are equal (or both null). One option is to maintain a global map from each site–parent node pair to the corresponding unique node. But a global, concurrent map would provide poor locality and require synchronization. Instead, we add to each unique node a map from its callee call sites to existing child nodes:

```
class UniqueCCUNode extends CCUNode {
  Map<Site,CCUNode> childMap;
}
```

Section 4.2 explains how our implementation uses different spaces for unmerged and merged (unique) nodes, in order to support using the childMap field only for unique nodes.

The *CCT* also maintains a child map—but for every node. In contrast, the CCU avoids the cost of updating the child map for the (many) irrelevant nodes that become unreachable between their allocation and the next GC.

Figure 4 presents pseudocode that extends GC tracing to merge redundant nodes by identifying a node's unique node and redirecting the node's incoming pointers to the unique node. GC calls traceAndMerge, instead of its regular tracing function, when tracing CCU nodes. traceAndMerge first recursively traces and merges the parent node, which returns the unique parent node. Then it checks for an existing unique node for node. If no unique node exists, node is the unique node, so traceAndMerge performs regular GC tracing on it (marking it live and copying it if applicable)

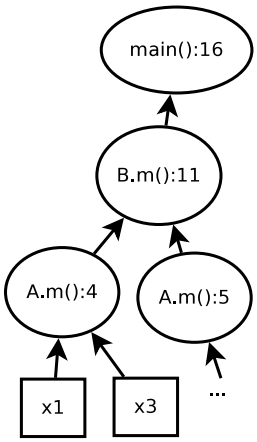


Figure 5: Calling context uptree (CCU) corresponding to Figure 1, after merging of redundant nodes. Irrelevant nodes have also been garbage collected.

and adds it to the parent's child map as the unique child. Section 4.2 describes how this algorithm applies to CCU nodes in both copying and non-moving spaces.

3.6 Example CCU After Merging

Figure 5 shows the CCU from Figure 3 after merging executes as part of GC, which also collects irrelevant nodes. Each relevant context is represented by one unique node.

4. Implementation Details

We implement our CCU-based approach in Jikes RVM 3.1.1, a high-performance research Java virtual machine [2, 3].[3] Jikes RVM provides performance within 15% of commercial JVMs.[4] This section describes implementation details of optimizations for constructing CCU nodes, merging redundant nodes, and integration with two bug detection clients.

4.1 Optimizing CCU Nodes

Using return addresses as sites. Each CCU node has two fields: its site and parent node. One option for representing the site is to assign each static site (method and bytecode index) a unique identifier. When compiling each call site and client site, the dynamic compiler would compute the identifer and insert instrumentation that constructs a CCU node using the identifer. However, this option would not work well with lazy node construction (Section 3.2), because *callee* methods are responsible for constructing nodes for caller call sites. For example, when call site B.m():11 in Figure 1 calls method A.m(), instrumentation in A.m() needs to construct the node for the call site B.m():11. Instrumentation at B.m():11 could potentially pass the site identifier to A.m(), but this would add overhead and lose much of the benefit of lazy construction.

Our implementation addresses this challenge by representing sites using the *return address* of the caller call site. The return address of the caller call site is already available on the current stack frame, in order to handle a return instruction. A return address maps to a specific caller call site, and the VM already provides methods to decode an instruction pointer to a caller call site, e.g., to support exception handling.

[3] `http://www.jikesrvm.org`

[4] `http://dacapo.anu.edu.au/regression/perf/9.12-bach.html`

Decoding return addresses is slow compared with node allocation. However, the implementation only needs to decode return addresses to call sites when reporting them to programmers, which is already expensive and infrequent. We modify the VM to prevent collection of unused compiled methods, so the VM can always decode a return address to its unique site.

The VM recompiles methods adaptively [5] and also compiles static sites multiple times by inlining methods and unrolling loops, so multiple return addresses may map to the same call site. Our implementation computes node equality using a node's return address, which may inhibit merging somewhat since two nodes with the same site and parent cannot be merged theirs sites are represented by different return addresses.

Whenever our implementation constructs a CCU node representing a method's caller context, it stores a pointer to the node on the method's stack frame. We add a slot to each stack frame for this purpose. This enables reusing CCU nodes constructed for existing stack frames and enables walking the stack until a stack frame is found that has already constructed its caller context node.

Inlined call sites. To reduce call overhead and increase optimization scope, the VM inlines small and hot methods. An inlined call site effectively represents multiple call sites. Our implementation constructs just one CCU node for each actual call site in an inlined method. Return addresses naturally represent inlined call sites, and the VM provides functionality to decode the call sites that make up a return address for an inlined call site.

Raw objects. Our implementation supports CCU nodes being pure Java objects. Java objects have a header for type information, locking, and garbage collection (GC); the header in Jikes RVM is two words by default. Since dynamic analyses allocate CCU nodes frequently, the cost of the header is significant in terms of cache footprint, space overhead, and header initialization time.

Our implementation thus also supports using "raw" memory for CCU nodes that do not have headers. We have created custom memory spaces that support raw CCU nodes: a copying space and a mark-sweep space. When GC traces nodes in these special spaces, it calls our custom tracing code. Our evaluation uses raw nodes since they offer better performance (Section 5).

4.2 Merging Redundant Nodes

Section 3.5 described an algorithm for merging redundant nodes. Our implementation supports two types of merging. *In-place merging* merges nodes that have been previously allocated or copied into a mark-sweep space. Because nodes are not copied, all nodes in the space must include the extra childMap field in case they become unique nodes.

The other type of merging, *copy-based merging*, merges nodes that are in a copy space. It only traces and copies nodes that are chosen as unique nodes. Copy-based merging copies unique nodes into a mark-sweep space, which necessarily contains *only* unique nodes. Copy-based merging has two advantages. First, nodes in the copy space, which are numerous, do not need the extra childMap field. Second, copy-based merging limits fragmentation better than in-place merging, since copy-based merging only copies unique nodes into the fragmentation-prone mark-sweep space.

Regardless of the type of merging, our implementation always uses a configuration with one copy node space and one mark-sweep node space. It allocates nodes into the copy space, which is fast because it uses bump-pointer allocation [32]. Copied nodes are copied to the mark-sweep space, which offers better space and time performance for long-lived nodes. The mark-sweep space is always a mature space, i.e., it is traced only during full-heap GCs. The copy space may be a mature space or nursery space, i.e., traced during nursery GCs.

We have found that if the copy node space is a nursery space, then our implementation adds high overhead due to *generational write barriers*, which track new pointers from mature to nursery objects in order to support high-performance generational GC [8]. For our leak and race detectors, any assignment of a CCU node into an object header requires a generational write barrier, and this barrier needs to record the pointer in a *remembered set* if the node is old [8]. Thus, all of our experiments use a mature copy space that is collected only during full-heap GCs.

Implementing the child map. Unique nodes have an extra field childMap that maps callee sites to child nodes. In our implementation, the child map is a hash-based map that uses an array of "buckets," where each bucket is simply a linked lists of nodes. To construct a lightweight linked list, each unique node has an extra field next that points to the next node in the list.

Our implementation piggybacks on parallel GC to perform merging when nodes are copied from the copy node space to the mark-sweep node space. Searching for a child node does not require synchronization (except for a load fence to ensure a happens-before edge from insertions). However, if a node is not found, it must be inserted in the appropriate bucket's list, which requires synchronization to ensure atomicity with respect to another thread adding the same or a different node to the same bucket. The implementation first uses atomic operations to "lock" the bucket to ensure exclusive access. Then it searches the bucket's list again to make sure the node is not already in the list. Finally it inserts the node and unlocks the bucket.

Child nodes should be allowed to die if they are not referenced transitively by client metadata via node parent pointers. Otherwise all merged nodes will be transitively reachable from child maps, so they will not be collected by GC. The implementation treats each child map reference like a *weak reference* [26] by tracing the map only at the end of regular tracing, and removing any nodes from the map that have not been marked live.

Implementing the CCT. For comparison purposes, we have also implemented support for CCT-based analysis. In the CCT, *every* node has a child map. To keep the comparison as close as possible, our CCT nodes are also "raw" nodes and represent their child maps in the same way as CCU nodes.

4.3 Integration with Client Analyses

Memory leak detector. State-of-the-art leak detectors track the sites that allocated and/or last accessed each memory location, in order to report the sites associated with leaked memory to programmers [13, 16, 52]. We have implemented a leak detector that detects leaks by inferring that *stale* (not recently used) objects are likely leaks [13, 16, 42, 52].

We have implemented a staleness-based leak detector that tracks staleness by instrumenting each load of an object reference to mark the referenced object as *not stale* [15]; it also updates the last-use site at each instrumentation site, making it a challenging CCU-based client. The leak detector supports both context-insensitive and context-sensitive modes. The context-insensitive detector adds a word to each object header to store the last-use client site. The context-sensitive detector supports two options: (1) one word for the client site and another word to point to a CCU node representing the caller context, or (2) one header word that points to a CCU node representing the entire context, i.e., including the client site.

Data race detector. Happens-before race detectors detect data races by identifying two conflicting accesses that are not ordered by synchronization [12, 17, 22, 36, 41, 49, 53]. A race detector typically tracks the sites that last read and wrote each field or array element. When the detector detects a data race, it reports both the the prior access(es) stored for the racy variable, and the current program location. To report the calling context of the current program location, the runtime system simply walks the call stack. Reporting the calling context of the prior access(es) requires recording the context of each access.

We have integrated CCU with *Pacer*, a sampling-based happens-before race detector [12] implemented in Jikes RVM and publicly available on the Jikes RVM Research Archive. Pacer maintains per-field metadata in each object's header. This metadata already stores the (context-insensitive) site that last wrote and site(s) that last read each field. These are the client sites. In addition, we modify the implementation to store the CCU node representing the caller context of each client site. Given the client site and its caller context, Pacer can report the full calling context.

At a 100% sampling rate, Pacer is functionally equivalent to *FastTrack* [22]. We primarily evaluate FastTrack (Pacer at 100%), which is a challenging client because it instruments essentially every read and write. Pacer and FastTrack are able to skip race detection analysis for accesses that occur within the same *epoch* (synchronization-free region) as the prior access, and we do not record the CCU node upon such "same epoch" cases. In addition to evaluating CCU-enabled FastTrack, we evaluate Pacer at a variety of sampling rates to evaluate CCU's ability to scale with the client analysis.

5. Evaluation

This section evaluates our CCU-based approach by evaluating the time and space it adds to two client analyses, data race and memory leak detection. While our primary goal is to show that a CCU-based approach is viable, we also compare to a CCT-based approach. The section concludes by qualitatively evaluating leaky calling contexts reported by our leak detector for two real memory leaks.

5.1 Methodology

Benchmarks. In our experiments, Jikes RVM executes the DaCapo Benchmarks [7] (version 2006-10-MR2) and a fixed-workload version of SPEC JBB2000 called pseudojbb [47]. We evaluate leak detection on all benchmarks, and race detection on the parallel benchmarks, except for multithreaded lusearch since we could not run it correctly with our changes, and single-threaded bloat because it has erratic performance even without our changes. For race detection, we execute the *medium* workload size of the DaCapo benchmarks because the *large* size runs out of memory, and we execute pseudojbb with four warehouses because eight warehouses runs out of memory.

Experimental setup. We build a high-performance configuration of Jikes RVM (FastAdaptive) that optimizes the VM and adaptively optimizes the application as it runs. We use Jikes RVM's high-performance generational Immix collector [10] (GenImmix). To account for run-to-run variability due to dynamic optimization guided by timer-based sampling, we execute 15 trials for each time measurement and take the median, and our bar graphs show 95% confidence intervals centered at the mean. We let the VM choose its own heap size adaptively because the client analyses, especially race detection, add high space overhead. We evaluate space overhead by measuring it explicitly in just one trial, since averaging plots of space versus time is not straightforward and may unrealistically hide peaks in space overhead.

Platform. Our experiments execute on a 4-core Intel i5 3.2-GHz system with 4 GB memory running Linux 2.6.32.

	CCU w/merging:			CCT
	None	In-place	Copy	
antlr	2,952	3,000	2,868	51,712
chart	46,000	41,112	2,536	4,672
eclipse	29,032	31,352	3,708	75,616
fop	4,456	4,564	2,696	4,192
hsqldb	147,140	98,108	3,684	59,456
jython	3,632	3,656	3,076	62,592
luindex	2,828	3,004	2,632	7,520
pmd	55,964	32,516	3,460	186,912
xalan	5,076	42,156	5,376	360,576
pseudojbb	71,216	57,776	3,048	4,384

Table 1: The memory consumed by CCU and CCT nodes, in KB, for context-sensitive leak detection with client sites stored in nodes. The CCU uses no merging or one of two merging algorithms.

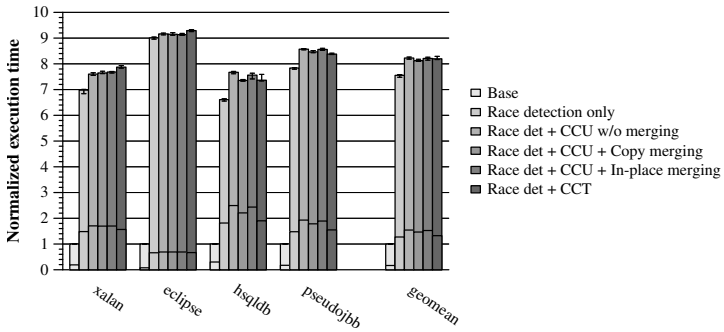


Figure 8: Time overhead of race detection with and without CCU- and CCT-based context sensitivity. Sub-bars bars are GC time. The (tiny) ranges are 95% confidence intervals.

5.2 Performance

This section evaluates the time and space overhead of CCU-based analyses. Because our implementation constructs CCU nodes lazily (Section 3.2), it adds no overhead without a client analysis.

5.2.1 Memory Leak Detection

This section evaluates the performance of our memory leak detector that tracks the last-use (i.e., last read) sites of all objects (Section 4.3).

Time overhead. Figures 6 and 7 show the normalized application time of various leak detection configurations. All bars are normalized to *Base*, which is unmodified Jikes RVM. The *Leak detection only* configuration records context-insensitive sites and adds about 11% on average to record last-use sites and track object staleness. The three *Leak det + CCU* configurations construct CCU nodes to represent the context of each last-use site and either do not merge redundant nodes, or use copy or in-place merging.

We experiment with two CCU configurations (Section 4.3). The first stores a pointer to a CCU node in each object's header that represents the last-use site's calling context, *including a node for its client site* (Figure 6). In this configuration, CCU-based context sensitivity adds 93–96% overhead, depending on the merging configuration. CCT-based context sensitivity adds substantially more overhead (196% on average) because the cost of looking up or constructing each node is substantially higher for the CCT than for the CCU. The second configuration stores a node representing the last-use site's *calling context*, and it uses a second header word for an identifier representing the client site (Figure 7). In this configuration, CCU-based context sensitivity adds only 33–35% overhead on average to leak detection. Using the CCT instead of the CCU adds on average 57% overhead.

In both configurations, across all benchmarks, the CCU (with merging) performs about the same as, or significantly better than, the CCT. The overhead of the first configuration (node for each client site) may be too high for some clients, in which case they can use the second configuration to avoid allocating a new node for each client site.

In both configurations, the CCU adds high overhead to hsqldb, with much of it due to GC (sub-bars are GC time). Unsurprisingly, this program benefits the most from merging of redundant nodes. Next we show that hsqldb allocates significantly more CCU nodes than the other programs.

Space overhead. Table 1 shows the maximum live memory added for CCU and CCT nodes by measuring the total memory consumed by the node space after each full-heap GC and reporting the maximum across each execution. For these results, client sites are stored in nodes (i.e., same as Figure 6). The results indicate that merging redundant nodes is sometimes critical: *In-place* merging often improves memory overhead significantly over *None* (no merging), and *Copy* merging improves memory overhead further. We have determined that both merging algorithms perform similarly in terms of the *number* of unique nodes, but *In-place* merging has higher space overhead because it fragments the heap significantly. In contrast, the *CCT* avoids redundant nodes but cannot collect irrelevant nodes, so its memory overhead is often very high, and always higher than for the merged CCU.

5.2.2 Data Race Detection

This section evaluates the overhead of the context-sensitive race detector that records context-sensitive sites at reads and writes (Section 4.3).

Time overhead. Figure 8 shows the performance overhead that race detection adds to programs. Context-insensitive race detection slows programs by about 7X on average, which matches prior results for the FastTrack algorithm and Pacer implementation [12, 22]. CCU-based context sensitivity with merging adds 57% average overhead (relative to *original program execution time*) over context-insensitive race detection. CCT-based context sensitivity adds 64% average overhead over context-insensitive race detection. In theory, the CCU should not add as much overhead to the race detector as to the leak detector because the race detector skips analysis (and thus CCU node lookup) at reads and writes that fall within the same synchronization epoch (Section 4.3). However, the CCU adds more overhead to the race detector (relative to original program execution time) because the race detector's heavyweight analysis happens inside an inserted method call, requiring more stack-walking to lookup and construct CCU nodes lazily, whereas the leak detector's analysis is inlined into the application method.

We also evaluate the overhead of CCU-based race detection at different sampling rates, in order to detect how well a CCU-based approach scales with the client analysis. Sampling-based race detection samples a fraction of reads and writes equal to the sampling rate, so a lower sampling rate should yield a less-demanding client analysis. Figure 9 shows how the amount of additional overhead added by the CCU-based approach scales approximately linearly with the sampling rate. This behavior is what we expect since our implementation constructs CCU nodes lazily at client sites (Section 3.2). We also measured (but do not show) time overhead across sampling rates for the CCT, which also scales well because it uses lazy construction. In contrast, we expect *space* to scale better with the sampling rate for CCU than for the CCT, but we have not measured space overhead across different sampling rates.

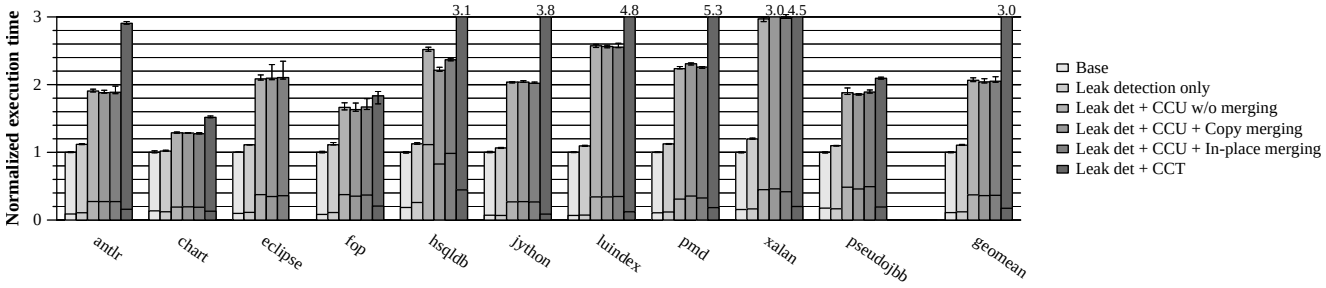


Figure 6: Normalized application time for leak detection with client site node, with and without CCU-based context sensitivity, and compared with CCT-based context sensitivity. Bars 3–5 in each group use the CCU without merging and with two merging algorithms. Sub-bars are GC time. The intervals are 95% confidence intervals.

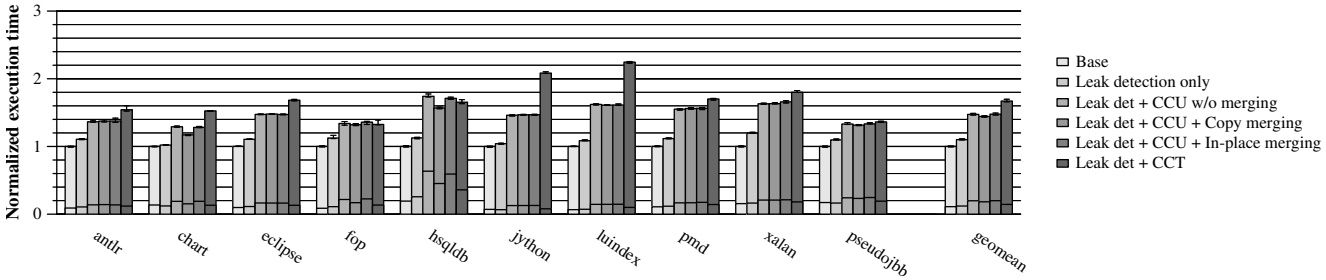
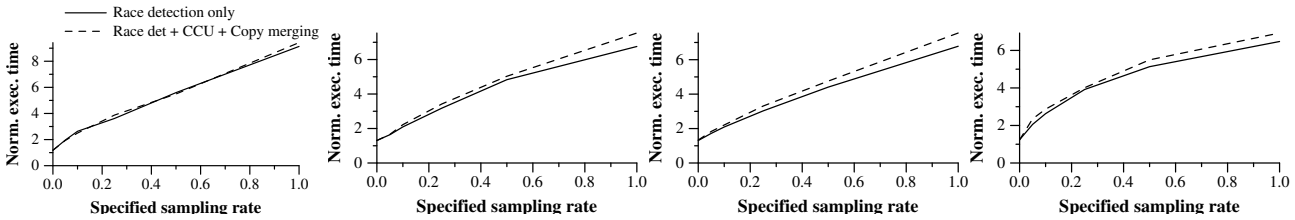
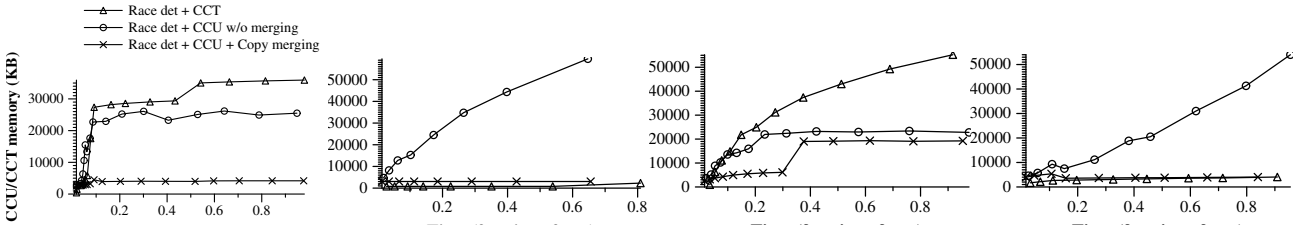


Figure 7: Normalized application time for leak detection with site in header, with and without CCU-based context sensitivity, and compared with CCT-based context sensitivity. Bars 3–5 in each group use the CCU without merging and with two merging algorithms. Sub-bars are GC time. The intervals are 95% confidence intervals.



(a) eclipse (b) hsqldb (c) xalan (d) pseudojbb

Figure 9: Normalized execution time of race detection, with and without CCU-based context sensitivity, at various sampling rates (0%, 5%, 10%, 25%, 50%, 100%).



(a) eclipse (b) hsqldb (c) xalan (d) pseudojbb

Figure 10: Space used by CCU and CCT nodes for context-sensitive race detection, with and without CCU node merging.

	Distinct		Dynamic
	CI	CS	
eclipse	41	59	667,230
hsqldb	9	22	336
xalan	11	14	138
pseudojbb	21	21	920,462

Table 2: Distinct, prior context-insensitive (CI) accesses and context-sensitive (CS) accesses. The last column is dynamic data races reported.

	6-10		16-20		26-30		36-40		46-50		
	1-5	11-15		21-25		31-35		41-45		51-55	
eclipse	17	4	2	9	2	3	5	7	4	3	3
hsqldb	10	4	8	0	0	0	0	0	0	0	0
xalan	4	4	0	0	2	4	0	0	0	0	0
pseudojbb	20	1	0	0	0	0	0	0	0	0	0

Table 3: Histogram showing the depth (in call sites) of context-sensitive sites reported by the race detector.

Space overhead. Figure 10 shows the memory used by CCU and CCT nodes across an execution. Time is normalized to the length of each execution. Each point represents the live CCU or CCT memory at the end of a full-heap GC. *Race det + CCU w/o merging*, which constructs CCU nodes for client sites but does not merge redundant nodes, clearly shows the need for merging. By merging redundant nodes, the space overhead added by CCU is reduced substantially. Furthermore, whereas CCU space overhead grows over time without merging (which is unsurprising since these programs' total live memory also grows over time), merging keeps CCU space overhead fairly constant over time. The CCT sometimes adds space overhead similar to CCU with merging, and sometimes adds more space than the CCU without merging. While the CCT avoids redundant nodes, it cannot collect irrelevant nodes.

Context-sensitive data races. We quantitatively (but not qualitatively) evaluate the calling contexts reported by the race detector. Table 2 shows the number of *prior racy accesses* reported by the race detector (reporting a race involves reporting a prior access and the current access). The first two columns compare the number of context-insensitive and context-sensitive prior accesses; for three programs context sensitivity yields more distinct prior accesses. Dynamic races (last column) varies greatly across the programs.

Table 3 is a histogram showing the depths, in terms of number of sites, of the distinct context-sensitive prior racy accesses. Many contexts, especially for eclipse, are dozens of sites long, suggesting the potential for calling contexts to provide significantly more information to developers than static program locations.

5.3 Qualitative Evaluation of Context-Sensitive Sites

This section evaluates whether context sensitivity provides useful information to bug reports. We evaluate two real leaks that were reproduced and evaluated by prior work [13, 33]: one in SPEC JBB2000 [47] and one in Eclipse.[5] Our reported leaky sites do not exactly match those from the prior staleness-based leak detector that evaluated the same leaks [13] because our detector updates an object's staleness and last-use site when a *reference to the object is loaded* [15], instead of when the object itself is modified.

SPEC JBB2000. The SPEC JBB2000 leak occurs because the program adds but does not correctly remove finished orders from an order list. Researchers from IBM and Intel discovered the leak fix, which involves replacing a call to spec.jbb.District.removeOldestOrder() with more complex logic to properly remove finished orders [13]. Our context-sensitive leak detector reports that the following context as a last-use site for a growing number of stale objects:

```
 <4> spec.jbb.infra.Factory.Container.deallocObject():352
<34> spec.jbb.infra.Factory.Factory.deleteEntity():659
 <1> spec.jbb.District.removeOldestOrder():285
 <1> spec.jbb.DeliveryTransaction.process():201
 <1> spec.jbb.DeliveryHandler.handleDelivery():103
 <2> spec.jbb.DeliveryTransaction.queue():363
 <1> spec.jbb.TransactionManager.go():449
 <1> spec.jbb.JBBmain.run():173
```

The numbers in brackets (e.g., <34>) indicate the number of static call sites that can potentially call each method (based on analyzing the source in the Eclipse IDE). This gives a sense of how "nontrivial" the context is, i.e., how hard it is for developers to guess the context from a context-*insensitive* site. In this case, developers need to find DeliveryTransaction.process():201, which will likely require a lot of work because Factory.deleteEntity() has 34 callers. However, in our experiments, the *client site* actually consists of the first *three* call sites due to inlining, from which it is relatively easy to find the buggy site. As in prior work, our implementation also reports a few other last-use sites (for both JBB and Eclipse) for a growing number of stale objects, but these sites are not directly related to the bug fix, so we do not show their contexts.

Eclipse. Eclipse bug #115789 leaks memory when a "recursive difference" is performed between two source trees.[6] Repeatedly comparing the source tree leads to a growing leak. Prior work shows that the leak occurs in a NavigationHistory component that enables navigating back to prior editor windows, but does not properly release old state. Our leak detector reports one last-use site in NavigationHistory (prior work reports this site and another in NavigationHistory; we believe the differences are due to the differing instrumentation strategies described earlier):

```
[* = org.eclipse]
*.ui.internal.NavigationHistory.createEntry():527
*.ui.internal.NavigationHistory.addEntry():307
*.ui.internal.NavigationHistory.access$9():291
*.ui.internal.NavigationHistory$2.run():160
... 13 call sites ...
*.compare.internal.CompareUIPlugin.compareResultOK():474
*.compare.internal.CompareUIPlugin.openCompareEditor():427
*.compare.CompareUI.openCompareEditorOnPage():138
*.compare.internal.CompareAction.run():36
*.compare.internal.BaseCompareAction.run():26
leakdiff.Harness$1.run():89
... 23 call sites ...
*.core.launcher.Main.main():948
```

This calling context illustrates how comparing source trees ultimately leads to stale last-use sites in NavigationHistory. The elided call sites are other Eclipse internal methods. Developers might find the exact connection between CompareUI.openCompareEditorOnPage() and NavigationHistory useful for understanding the leak. Admittedly, the leak fix is actually in NavigationHistory, so the context-*insensitive* site is useful by itself. In any case, this example shows that context-sensitive sites can provide significantly *more* information to developers for highly object-oriented programs.

[5] http://www.eclipse.org

[6] https://bugs.eclipse.org/bugs/show_bug.cgi?id=115789

6. Related Work

Section 2.2 discussed the most closely related work: the calling context tree (CCT). This section discusses other alternatives for providing dynamic context sensitivity, plus other related topics.

6.1 Walking the Stack

Client analyses can simply walk the entire call stack at each client site, which is expensive unless client sites execute infrequently [14, 24, 39, 44, 50, 55]. Stack-walking approaches typically store nodes in a CCT for space efficiency. Prior work walks the stack until it encounters a stack frame that has already looked up its CCT node [50, 55], which is equivalent to lazy construction of the CCT (Section 3.2).

6.2 Reconstructing Calling Context

Recent work introduces several approaches that represent calling contexts as integer values and reconstruct calling contexts from these values on demand. Ultimately, mapping contexts to values is challenging because the number of statically possible calling contexts easily exceeds 2^{64} for real, complex programs (not counting recursion, which leads to infinitely many statically possible contexts). Thus, none of these approaches scales well to complex programs, i.e., programs with many distinct calling contexts.

Perfect accuracy. Sumner et al. introduce *precise calling context encoding* (PCCE), which represents each calling context with a unique integer value [48]. Instrumentation at each call site incrementally computes the current calling context's value. PCCE computes these increments at compile time by applying Ball-Larus intraprocedural path profiling algorithm [6] to the call graph. This algorithm also enables efficient reconstruction of a calling context from its value. PCCE does not scale well with many distinct calling contexts: if the number of statically possible calling contexts exceeds the integer size (2^{32} or 2^{64}), PCCE uses multiple integers to represent each calling context, which slows execution and adds space overhead whenever the client analysis stores a calling context; CCU nodes amortize this cost by sharing parent nodes. Furthermore, PCCE does not handle dynamic class loading and virtual methods well: computing each context's value requires knowing the statically possible call targets in advance, and virtual method dispatch complicates adding instrumentation at these calls. While PCCE reports low time overhead, it cannot guarantee it has low time overhead for Java programs, which have higher call density than C/C++ programs. Wiedermann also applies Ball-Larus path profiling to the call graph but does not handle recursion nor the number of paths exceeding the integer size [51].

Imperfect accuracy. Unlike PCCE, *Breadcrumbs* handles dynamic class loading and virtual method dispatch, and it maps each calling context to a single integer word [11]. Breadcrumbs computes a probabilistically unique value for each calling context, by computing an incremental hash function at each call site [14]. Because the number of statically possible contexts greatly exceeds both 2^{64} and the number of dynamically executed contexts, Breadcrumbs also records some *dynamic* information—the context values observed at *cold* call sites—in order to help guide reconstruction of contexts. Nonetheless, reconstruction of contexts is complex, may take seconds, and may fail to reconstruct the correct context. Breadcrumbs thus provides a time–accuracy tradeoff, since collecting more dynamic information provides better reconstruction accuracy.

Mytkowicz et al. and Inoue and Nakatani propose to reconstruct contexts from existing runtime values such as program counters and stack depth [31, 37]. These approaches add virtually no overhead, but the values they use have significantly less entropy than Breadcrumbs' probabilistically unique values. To reduce value conflicts between stack depths, Mytkowicz et al. pad the call stack based on profiling, which helps somewhat but not enough to scale to complex programs with many distinct calling contexts.

6.3 Sampling and Mining

Prior work uses sampling and data mining to trade accuracy for lower overhead when collecting calling contexts [19, 30, 55]. This tradeoff is worthwhile for determining *hot* program behavior for performance optimization. However, *cold* program behavior is critical for bug detection [16, 36].

6.4 Uptrees

In tree-based data structures, each node typically points "down" to its children. In contrast, in an uptree each node points "up" to its parent. Uptrees are useful for implementing efficient disjoint-set data structures, where the uptrees enable near-constant amortized-time *find* and *union* operations [18]. The CCU's uptree property enables efficient construction and garbage collection of CCU nodes.

7. Conclusion

Growing complexity and concurrency mean that static program location is not enough to help programmers understand dynamic program behavior. This paper presents a new approach for providing context sensitivity to dynamic analyses, especially bug detectors that report bug causes. Calling context uptree (CCU) nodes cannot be reused but are fast to build—and tracing garbage collection and a lazy merging algorithm keep space overhead low. By using the CCU to add context sensitivity to leak and race detectors, we demonstrate the potential for the CCU to add low-overhead, always-available context sensitivity to dynamic bug detection analyses.

Acknowledgments

Thanks to Daniel Frampton, Sam Guyer, Kathryn McKinley, Feng Qin, and Nasko Rountev for valuable discussions, ideas, and support. Thanks to Todd Mytkowicz and Nasko Rountev for helpful feedback on the text.

References

[1] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *ACM Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

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

[3] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44:399–417, 2005.

[4] 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, 1997.

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

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

[7] 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.

[8] S. M. Blackburn and A. L. Hosking. Barriers: Friend or Foe? In *ACM International Symposium on Memory Management*, pages 143–151, 2004.

[9] S. M. Blackburn and K. S. McKinley. Ulterior Reference Counting: Fast Garbage Collection Without a Long Wait. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 344–358, 2003.

[10] S. M. Blackburn and K. S. McKinley. Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *ACM Conference on Programming Language Design and Implementation*, pages 22–32, 2008.

[11] M. D. Bond, G. Z. Baker, and S. Z. Guyer. Breadcrumbs: Efficient Context Sensitivity for Dynamic Bug Detection Analyses. In *ACM Conference on Programming Language Design and Implementation*, pages 13–24, 2010.

[12] 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*, pages 255–268, 2010.

[13] M. D. Bond and K. S. McKinley. Bell: Bit-Encoding Online Memory Leak Detection. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 61–72, 2006.

[14] 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.

[15] M. D. Bond and K. S. McKinley. Leak Pruning. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 277–288, 2009.

[16] 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.

[17] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *ACM Conference on Programming Language Design and Implementation*, pages 258–269, 2002.

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

[19] D. C. D'Elia, C. Demetrescu, and I. Finocchi. Mining Hot Calling Contexts in Small Space. In *ACM Conference on Programming Language Design and Implementation*, pages 516–527, 2011.

[20] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *ACM Conference on Programming Language Design and Implementation*, pages 245–255, 2007.

[21] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.

[22] 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.

[23] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *ACM Conference on Programming Language Design and Implementation*, pages 293–303, 2008.

[24] 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.

[25] R. Garner, S. M. Blackburn, and D. Frampton. Effective Prefetch for Mark-Sweep Garbage Collection. In *ACM International Symposium on Memory Management*, pages 43–54, 2007.

[26] B. Goetz. Plugging memory leaks with weak references, 2005. `http://www-128.ibm.com/developerworks/java/library/j-jtp11225/`.

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

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

[29] B. Hayes. Using Key Object Opportunism to Collect Old Objects. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 33–46, 1991.

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

[31] 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.

[32] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[33] M. Jump and K. S. McKinley. Cork: Dynamic Memory Leak Detection for Garbage-Collected Languages. In *ACM Symposium on Principles of Programming Languages*, pages 31–38, 2007.

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

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

[36] 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.

[37] 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.

[38] N. Nethercote and J. Seward. How to Shadow Every Byte of Memory Used by a Program. In *ACM/USENIX International Conference on Virtual Execution Environments*, pages 65–74, 2007.

[39] 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.

[40] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and Precisely Locating Memory Leaks and Bloat. In *ACM Conference on Programming Language Design and Implementation*, pages 397–407, 2009.

[41] E. Pozniansky and A. Schuster. MultiRace: Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. *Concurrency and Computation: Practice & Experience*, 19(3):327–340, 2007.

[42] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *International Symposium on High-Performance Computer Architecture*, pages 291–302, 2005.

[43] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *ACM Symposium on Operating Systems Principles*, pages 27–37, 1997.

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

[45] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick Your Contexts Well: Understanding Object-Sensitivity. In *ACM Symposium on Principles of Programming Languages*, pages 17–30, 2011.

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

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

[48] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise Calling Context Encoding. In *ACM International Conference on Software Engineering*, pages 525–534, 2010.

[49] C. von Praun and T. R. Gross. Object Race Detection. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–82, 2001.

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

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

[52] G. Xu and A. Rountev. Precise Memory Leak Detection for Java Software Using Container Profiling. In *ACM International Conference on Software Engineering*, pages 151–160, 2008.

[53] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *ACM Symposium on Operating Systems Principles*, pages 221–234, 2005.

[54] X. Zhang, N. Gupta, and R. Gupta. Pruning Dynamic Slices with Confidence. In *ACM Conference on Programming Language Design and Implementation*, pages 169–180, 2006.

[55] 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.