Efficient Context Sensitivity for Dynamic Analyses via Calling Context Uptrees and Customized Memory Management *

Jipeng Huang Michael D. Bond
Ohio State University
{huangjip,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, programmers need more than *static* program locations 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 (CCT) to track calling context—but in order to reuse existing nodes, CCT-based approaches require an expensive lookup.

This paper introduces a new approach for context sensitivity that avoids this expensive lookup. The approach uses a new data structure called the *calling context uptree* (CCU) that adds low overhead by avoiding the lookup and instead allocating a new node for each context. A key contribution is that the approach can mitigate the costs of allocating many nodes by extending tracing garbage collection (GC): GC collects unused CCU nodes naturally and efficiently, and we extend GC to merge duplicate nodes lazily.

We implement our CCU-based approach in a high-performance Java virtual machine and integrate it with a staleness-based memory leak detector and happens-before data race detector, so they can report context-sensitive program locations that cause bugs. We show that the CCU-based approach, in concert with an extended GC, provides a compelling alternative to CCT-based approaches for adding context sensitivity to dynamic analyses.

* This material is based upon work supported by the National Science Foundation under Grants CAREER-1253703 and CSR-1218695.

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-2374-1/13/10. . . $15.00.
http://dx.doi.org/10.1145/2509136.2509510

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Debuggers, Memory management, Run-time environments

Keywords calling context; context sensitivity; dynamic analysis; leak detection; race detection; garbage collection

1. Introduction

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

Dynamic program analysis helps developers make software more reliable by identifying errors and their likely causes. For example, data race detectors track the program locations that last accessed each variable [14, 21, 22, 33, 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 [15, 18, 38], dynamic slicing [1, 55], and atomicity violation detection [23, 32] track program locations in order to report likely bug causes. To find and diagnose bugs that do not manifest during testing, dynamic analyses must run in production settings, where minimizing overhead is the key constraint.

In order 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 is often invoked from many unrelated contexts. Modern software consists of integrated components written by many developers, which complicates this guessing game. In concurrent programs, a bug's cause and manifestation might execute on different threads, increasing the challenge of determining program behavior from a static location.

```
[* = 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 omitted ...
*.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 omitted ...
*.core.launcher.Main.main():948
```

Figure 1. A calling context from a leak in the Eclipse IDE.

Context sensitivity. By looking only at static sites, developers may not know *what* (or *how* or *why*) their programs are doing. They need more information than a static program location to understand what the program was doing. They need to know the dynamic calling context: the list of active call sites, similar to an exception stack trace.[1]

Developers are already accustomed 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 stack trace 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 not only need to know the calling context of program failures—they need to know the calling context of *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 site that is leaking memory. Reporting these prior program locations' calling contexts is challenging. Since it is impossible to predict which accesses' calling contexts might need to be reported later, dynamic analysis must record calling context frequently, e.g., at every program read and write.

The Java benchmark SPECjbb2000 has a site (static program location) that some leak detectors will pinpoint as a probable leak cause. However, this site can be invoked from 34 statically possible callers. Similarly, the Eclipse IDE has a memory leak that occurs in Eclipse's NavigationHistory class, but the calling context is needed in order to determine that the call stack includes a class CompareUIPlugin that indicates that the leak is called by comparing two code trees. Figure 1 shows (an abridged version of) this calling context; Section 6 has more details.

Prior approaches. While most analyses record and report only static program locations, some prior work captures calling context but either has serious limitations or adds high space and time overheads, limiting its applicability. Walking the stack whenever context is needed is expensive unless it is rare [16, 24, 37, 44, 50]. Recent approaches reconstruct calling context when needed from limited information that is cheap to collect, but these techniques are often probabilistic; their accuracy does not scale well with program complexity; they are unsuitable for bug detection analyses; and/or they cannot handle virtual method dispatch and dynamic class loading [13, 28, 35, 48]. Notably, *precise calling context encoding* (PCCE) [48] provides efficient calling context encoding and profiling but cannot handle bug detection analyses, virtual method dispatch, or dynamic class loading.

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 [3, 42, 46, 56]. Each CCT node maintains a mapping from child sites to child (i.e., 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 child sites is relatively small. Dynamic analysis thus frequently performs a nontrivial lookup to find the corresponding child context node, if any, which slows programs by two or more times [3, 42, 46].

Contributions. This paper proposes a novel approach for maintaining the calling context at run time and adding context sensitivity to dynamic analyses efficiently. Whereas a CCT-based approach looks up and reuses existing calling context nodes, our approach always allocates a new node. Making this approach efficient requires two main contributions. First, we introduce a new data structure called the *calling context uptree* (CCU) that supports allocating new nodes, instead of reusing existing nodes, efficiently. Second, we extend tracing garbage collection (GC) to support efficient collection of unused nodes and lazy merging of duplicate nodes (nodes representing the same context). This approach avoids relying on static analysis of the call graph, and it naturally supports dynamic class loading and virtual method dispatch.

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 our approach provides dynamic context sensitivity to the leak detector by adding 28 or 67% average overhead (relative to baseline program execution), depending on how leaf sites are stored, and to the race detector by adding 37% overhead. We show that our approach outperforms a comparable CCT-based implementation. These results suggest that the CCU-based approach offers a compelling new direction for adding efficient context sensitivity to dynamic analyses. While our CCU-based approach outperforms the CCT-based approach, we believe the larger takeaway is the counterintuitive result that the CCU-based approach can provide context sensitivity efficiently, demonstrating a promising direction for future work to improve performance further.

[1] Prior work on *static* analysis considers other forms of context sensitivity that includes object allocation sites and types [34, 45].

2. Background

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*. A client analysis instruments program locations that we call *client sites*. Which sites are client sites depends on the client analysis. For example, in a typical data race detector, all program loads and stores to potentially shared memory are client sites. A client analysis records client sites in per-variable *client metadata*. This metadata might be implemented by adding extra word(s) to object headers or by adding shadow memory [36]. A client analysis that records only client sites is *context insensitive*, while a client analysis that records client sites with their calling context is *context sensitive*.

In this paper, dynamic calling context is (1) a client site plus (2) the set of active call sites. A (call or client) site consists of a method and a bytecode index:[2]

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

In Figure 1, the first site, NavigationHistory.createEntry():527, is the client site, and the other sites are call sites.

It is challenging to add efficient context sensitivity to dynamic analyses. Dynamic analyses such as bug detectors typically execute client sites frequently (e.g., they record a site for every program memory access), so that if and when they detect a bug related to a memory location, they can report an associated site. A naïve analysis that frequently records calling context—which can be dozens of call sites long—will add high time and space overhead. Another challenge is handling two common language features: dynamic class loading, which grows the static call graph as the program executes, and virtual method dispatch, which complicates instrumenting calls.

2.1 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 [3, 42, 46, 56]. Each node points to its existing child nodes:

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

As mentioned earlier, the site can be a call site or a client site. Each node maintains a map childMap from child sites to child nodes; a child node represents a child (callee) calling context of the current node, and a child site is the site of a child node. The child map enables dynamic analysis to reuse an existing context node, if any, at a call or client site. Reusing existing nodes is important because programs execute many more dynamic than distinct contexts [16].

Inherent to this design is that each node has pointers pointing "down" to it child nodes, in order to reuse existing nodes. Direct mapping is impractical because a call site may have many statically possible child sites: a call site may statically call several virtual methods, and each of these methods may contain many call and client sites. Furthermore, the number of statically possible child sites grows over time due to dynamic class loading. Efficient implementations must use a sparse mapping implementation such as a hash table, or potentially a list if relatively few distinct child sites execute. Existing CCT-based dynamic analyses thus require a nontrivial lookup at essentially every call and/or client site, slowing programs by two or more times [3, 42, 46].

Figure 2 shows an example program written in Java-like pseudocode. Suppose the client analysis wants to record the last access (load or store) to each object. The call site main():16 has four statically possible child sites: A.m():4, A.m():5, B.m():10, and B.m():11. Note that the child sites are, by definition, the call sites and client sites inside the possible callees (A.m() and B.m()). Similarly, call site B.m():11 has two statically possible child sites, A.m():4 and A.m():5. We assume line 18 (...) performs additional work.

Figure 3 shows the CCT that a CCT-based dynamic analysis would allocate for the example program in Figure 2. Ovals represent CCT nodes; shaded nodes are client site nodes, while other nodes are call site nodes. Squares represent heap objects that are instances of class A, B, or X (x1–x3 are instances of X numbered 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 significance of this call is that x1 and x3 *escape* and are thus still alive at line 18.

The program accesses x1 and x3 first at B.m():10 ← main():16 and then at A.m():4 ← B.m():11 ← main():16. Nonetheless, the context B.m():10 ← main():16 remains in the CCT. Similarly, x2 dies quickly (indicated with dashed lines) because A.m():5 does *not* add x2 to globalSet; nonetheless, the last-access context A.m():4 ← main():16 survives because it is reachable from the tree root. This paper refers to nodes that are no longer used by client analyses as *irrelevant* nodes. Our new CCU-based approach relies on most context nodes becoming irrelevant (unreachable) fairly quickly so that tracing-based garbage collection (GC) can collect them. The CCT can also support GC of irrelevant

[2] Sites use bytecode indices because they uniquely identify bytecodes, unlike line numbers. Throughout the paper, we show line numbers for calling contexts because they are more intuitive when examining source code.

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

Figure 2. Example program written in Java pseudocode. Client sites are loads and stores. Each client or call site is annotated.

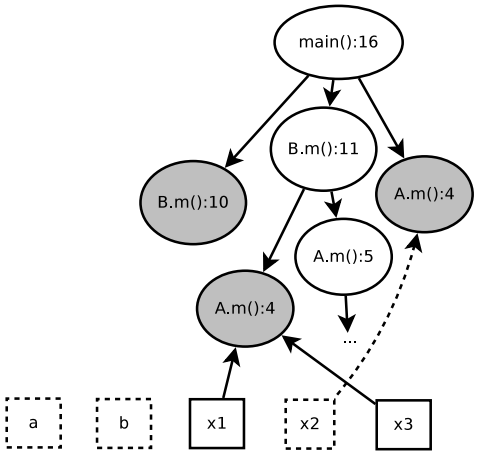


Figure 3. Calling context tree (CCT) corresponding to Figure 2. Ovals represent CCT nodes; shaded nodes are client site nodes, and others are call site nodes. Squares represent program objects.

nodes using weak references [26], e.g., by using a weak hash map for child nodes and also maintaining parent pointers.[3]

The more critical problem with the CCT is that nodes can have many statically possible child 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 approach addresses this problem, as well as how to delete irrelevant nodes.

[3] Client analyses that profile *all* contexts will not have irrelevant nodes. The CCT (without weak child references) is well suited for such analyses, while our CCU-based approach, which relies on irrelevant nodes, is not.

2.2 Alternatives to CCT-Based Approaches

Recent work encodes the calling context as a value or values. Some techniques trade accuracy for performance, but their accuracy does not scale well with program complexity [13, 28, 35]. In contrast, *precise calling context encoding* (PCCE) losslessly encodes each calling context as an integer value or values [48]. While PCCE provides efficient encoding and profiling of contexts, it is not well suited to bug detection analyses, which need to encode program locations as per-variable metadata, since PCCE represents contexts with a *variable* number of integers. Furthermore, PCCE relies on numbering a statically known call graph and instrumenting call edges with statically unique targets, so it inherently cannot handle dynamic class loading nor virtual method dispatch, limiting its applicability. Section 7 discusses these techniques in more detail.

3. A New Approach for Dynamic Context Sensitivity

This section introduces an approach based on a new data structure called the *calling context uptree* (CCU). 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 children, a node's callee context nodes are not accessible from it, making it essentially impossible to reuse existing nodes to represent reoccurring contexts. However, allocating new nodes is fast.

This section presents an approach for dynamic context sensitivity that uses the CCU and extends garbage collection (GC). We first describe how dynamic analysis constructs CCU nodes, and then how to extend GC to collect irrelevant nodes and merge duplicate nodes.

3.1 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). 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. In the example below, caller and callee are names of methods that each take an additional parameter:

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

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):

```
o.metadata = new CCUNode(clientSite, ccuNode);
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 internal, implementation-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.[4] Section 4.1 describes how our implementation stores pointers to CCU nodes on stack frames and uses method return addresses to represent sites efficiently.

Eager and lazy construction are not unique to a CCU-based approach. A CCT-based approach may also construct nodes lazily or eagerly [50, 56]. CCU- and CCT-based approaches will each *look up* the same number of nodes; the key difference is that "looking up" a CCU node means allocating a new node, whereas looking up a CCT node means searching for an existing node to reuse.

We focus on lazy construction because it scales well with the client analysis: it constructs nodes only for the calling contexts of client sites, not for all calling contexts.

3.2 Collecting Irrelevant Nodes

Because CCU nodes point only "up" to their parents, tracing GC naturally collects irrelevant nodes (nodes that are transitively unreachable and thus are no longer used by the client analysis). In fact, the CCU *relies* on GC to collect the plethora of nodes that become irrelevant quickly. Tracing GC is well suited to collecting CCU nodes because tracing is proportional to the live nodes, not the (more numerous) dead nodes [30].

Example CCU. Figure 5 shows the CCU after executing the (instrumented) code in Figure 2. 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 once they become unreachable.

3.3 Merging Duplicate Nodes

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

[4] We note that the CCU and CCT both naturally handle recursive call sites because each CCU node represents one *dynamic* call site or client site.

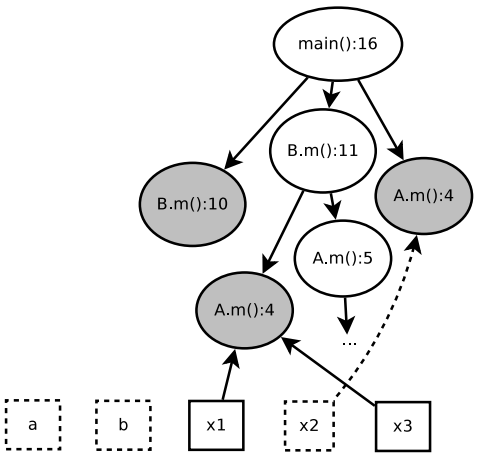


Figure 4. Calling context tree (CCT) corresponding to Figure 2. (Exact copy of Figure 3, for comparison purposes.)

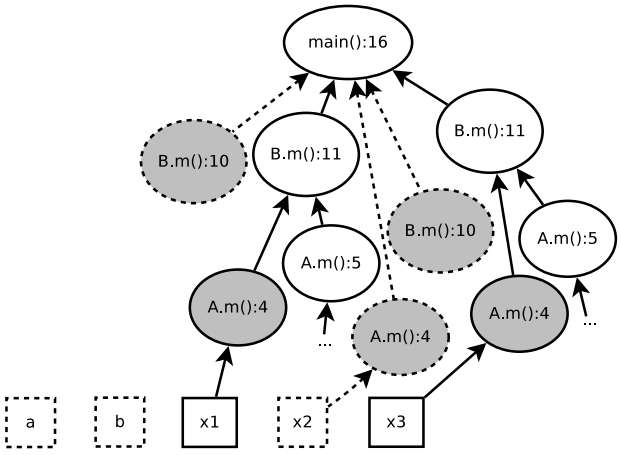


Figure 5. Calling context uptree (CCU) corresponding to Figure 2, without merging of duplicate nodes. The dashed lines represent objects that become irrelevant (unreachable) by line 18 of the example program. Other formatting is same as in Figure 3.

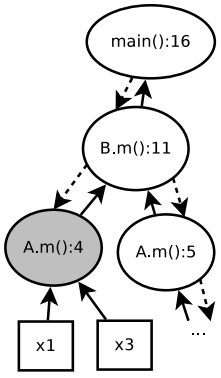


Figure 6. Calling context uptree (CCU) corresponding to Figure 2, after merging of duplicate nodes. Irrelevant nodes have also been garbage collected. Down arrows (dashed) represent child map pointers; after merging, all surviving nodes have child maps. Other formatting is same as in Figure 3.

To balance space and time, our CCU-based approach periodically merges duplicate nodes so that each relevant context is represented by just one *merged* node. The goal of merging is to determine, for each node, its merged node—which may be the node itself or a different node—and redirect all incoming pointers to the merged node.

Our merging algorithm piggybacks on tracing GC, which already traverses all objects. Merging is not suitable for *non-tracing* GC such as reference counting, although efficient reference-counting algorithms still trace young objects [11], providing an opportunity for merging duplicate nodes.

Looking up merged nodes. To merge duplicate nodes, our implementation needs to support looking up existing merged nodes based on node equality. Two nodes are *equal* if their sites are the same and their parent nodes are equal (or both null). We add to each merged node a map from its child sites to existing child nodes:

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

A merged node needs a child map to enable looking up the the unique (merged) node for each child node. Section 4.3 explains how our implementation uses different memory spaces for unmerged and merged nodes, in order to support adding the childMap field only for merged nodes.

Both the CCU and CCT maintain child maps, but the CCT maintains a child map for every node; a merged CCU node is essentially the same as a CCT node that has a parent pointer. 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, as supported by our results.

GC typically traces (i.e., marks live and possibly moves) an object o *before* tracing objects referenced by o [25]. This tracing order is not well suited to merging, which needs to determine a node's merged node based on the node's merged parent. We modify GC tracing to trace each CCU node's parent before tracing the node itself.

Figure 7 presents pseudocode that extends GC tracing to merge duplicate nodes by identifying a node's merged node and redirecting the node's incoming pointers to the merged node. GC calls traceAndMerge, instead of its regular tracing function traceObject, when tracing CCU nodes. traceAndMerge first recursively traces and merges the parent node, which returns the merged parent node. It then checks for an existing merged node for node. If no merged node exists, node is the merged node, so traceAndMerge performs regular GC tracing on it (marking it live and copying it if applicable) and adds it to the parent's child map as the merged child. Section 4.3 describes how this algorithm applies to CCU nodes in both copying and non-moving spaces.

Example CCU after merging. Figure 6 shows the CCU from Figure 5 after merging executes as part of GC, which also collects irrelevant nodes. Each relevant context is represented by one merged node.

```
1  // Transitive closure:
2  while (!workList.isEmpty()) { // initialized w/root objs
3    ObjectReference obj = workList.pop();
4    for (Address slot : getReferenceSlots(obj)) {
5      slot.store(traceObject(slot.loadObjectReference()));
6    }
7  }
8  ObjectReference traceObject(ObjectReference obj) {
9    if (isCCUNode(obj)) {
10     return traceAndMerge(obj);
11   }
12   if (!alreadyTraced(obj)) {
13     // first trace this node
14     obj = markLiveAndPossiblyCopy(obj);
15     // trace outgoing pointers later
16     workList.push(obj);
17   }
18   return obj;
19 }
20 CCUNode traceAndMerge(CCUNode node) {
21   // first trace and merge the parent
22   if (!alreadyTraced(node.parent)) {
23     node.parent = traceAndMerge(node.parent);
24   }
25   // then trace and merge this node
26   mergedNode = node.parent.childMap.get(node.site);
27   if (mergedNode == null) {
28     node = markLiveAndPossiblyCopy(node);
29     // now node is the merged node
30     node.parent.childMap.put(node.site, node);
31     return node;
32   }
33   return mergedNode;
34 }
```

Figure 7. Pseudocode showing how we modify GC tracing to trace and merge CCU nodes. The transitive closure traces each heap object. Normally, an object is traced *before* its referenced objects (traceObject). A CCU node is traced *after* tracing its parent (traceAndMerge). Fine-grained synchronization (not shown) ensures atomicity of lines 26–30. Nodes without parents are not merged (not shown).

4. Challenges for an Efficient Implementation

The main challenge in implementing a CCU-based approach for dynamic context sensitivity is that it allocates many nodes. Node allocation takes time, increases cache pressure, and increases GC frequency and workload. Walking the stack to support lazy construction also adds time overhead—for both the CCU and the CCT. This section describes how our implementation addresses these challenges: optimizations for constructing CCU nodes, merging duplicate nodes, and integrating with two bug detection clients.

We implement our CCU-based approach in Jikes RVM 3.1.1, a high-performance research Java virtual machine (JVM) [2] that provides performance competitive with commercial JVMs.[5] Our implementation is publicly available on the Jikes RVM Research Archive.[6]

The CCU-based approach and optimizations could be implemented in many existing managed language VMs that have the requisite features: tracing GC and support for introspection of both the stack and dynamically compiled method metadata. One could potentially even adapt the approach to a language such as C or C++ that uses explicit memory management, e.g., by providing specialized GC and merging of CCU nodes, and keeping track of pointers from ordinary heap objects to CCU nodes, since these pointers would be the roots of the transitive closure of CCU nodes.

4.1 Optimizing CCU Nodes

Although CCU nodes and sites can be represented as objects as described in Section 3, objects have headers that would bloat the code and add overhead to node construction. This section describes how we try to make CCU nodes and sites as lightweight as possible.

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 identifier and insert instrumentation to construct a CCU node using the identifier. However, this option would not work well with lazy node construction (Section 3.1) because *callee* methods are responsible for constructing nodes for caller call sites. For example, when call site B.m():11 in Figure 2 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, sacrificing 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 (for use by 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.

Decoding return addresses is slow compared to allocating nodes—but decoding occurs only when reporting call sites to programmers, which is already expensive and infrequent. To enable the VM to always decode a return address to its unique site, we modify the VM to prevent collection of unused compiled methods and their metadata.[7]

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

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

[7] We find that disabling collection of unused methods does not noticeably degrade performance (results not shown).

The VM recompiles methods adaptively [5] and compiles some 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 if their sites are represented by different return addresses.

Raw node 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 without headers. We have implemented custom memory spaces that support raw CCU nodes: a copying space and a mark-sweep space. When GC traces nodes in these spaces, it calls our custom tracing code. The evaluation uses raw nodes since they offer better performance.

4.2 Optimizing CCU Instrumentation

This section describes how the implementation avoids walking the stack and reuses nodes whenever possible.

Optimizing lazy construction. Whenever our implementation constructs a CCU node representing a method's caller context (i.e., the calling context not including the current call or client site), it stores a pointer to the node in a slot in the method's stack frame. This behavior 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.

We modify the optimizing compiler to introduce a new local variable in each method that holds the method's caller context node, to avoid looking on the current stack frame in the common case. This local variable starts as null; instrumentation initializes it at the method's first client site. The method getNode() constructs nodes lazily by walking the stack until it finds an already-constructed node.

Client sites in loops execute multiple times with the same calling context. We modify the compiler to perform a specialized form of loop-invariant code motion that moves the construction of client sites before the loop pre-header and lets each dynamic client site use the same node. The compiler only performs this optimization if the loop pre-header executes less frequently than the client site in the loop (based on existing edge profiling). Since this optimization applies only to client sites, not call sites, it benefits only the leak detection experiments that use client site nodes (Section 5.3).

The following pseudocode shows how the local variable and hoisting optimizations work:

```
foo() {
  CCUNode callerNode = null; // foo's caller context
  ...
  // Hoisted instrumentation:
  if (callerNode == null) {
    callerNode = getNode();
  }
  CCUNode csNode =
    new CCUNode(clientSite, callerNode);
  ...
  for (...) {
    o.metadata = csNode;
    read o.f; // client site
    ...
  }
}
```

Inlined call sites. To reduce call overhead and increase optimization scope, the VM inlines small methods and hot call sites. An inlined call site represents multiple call sites. Our implementation constructs just one CCU node for each non-inlined 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.

4.3 Merging Duplicate Nodes

Section 3.3 described an algorithm for merging duplicate nodes, a key optimization because our approach allocates many duplicate nodes, some of which remain reachable. Our implementation supports two types of merging. *In-place merging* merges nodes in a (non-moving) mark-sweep space. Because nodes cannot be moved, all nodes in the space must include the childMap field in case they become merged nodes.

Copy-based merging copies merged nodes into a mark-sweep space, which naturally contains *only* merged nodes. Copy-based merging has two advantages. First, nodes in the copy space, which are numerous and often duplicates, do not need the extra childMap field. Second, copy-based merging limits fragmentation better than in-place merging, since copy-based merging copies only merged 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 since it uses bump-pointer allocation [8]. GC copies surviving nodes 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 only 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 [10, 53]. 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 [10, 53]. Thus, all of our experiments use a mature copy space that is collected only during full-heap GCs.

Implementing the child map. Merged nodes have an extra field childMap that maps child sites to child nodes. In our implementation, the child map is a hash-based map that uses an array of "buckets"; each bucket is a linked list of nodes. To construct a lightweight linked list, each merged 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 supports treating 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. We also implement and evaluate an alternative that retains all merged nodes and thus avoids tracing merged nodes.

4.4 Integrating 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 [15, 18, 52]. We have implemented a leak detector that detects leaks by inferring that *stale* (not recently used) objects are likely leaks [15, 18, 41, 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* [17]. It also updates the target object's last-use site at each reference load, 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 header word for the client site and another word pointing

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 [14, 19, 22, 33, 39, 49, 54]. 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 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 publicly available sampling-based happens-before race detector implemented in Jikes RVM [14]. Pacer maintains per-field metadata in each object's header. This metadata already stores the (context-insensitive) sites that last wrote and read each field. These are the client sites. 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 a significant fraction of all reads and writes. 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. The implementation does not record the CCU node in such "same epoch" cases.

5. Quantitative Evaluation

This section primarily evaluates the time and space that CCU- and CCT-based approaches add to two client analyses: data race and memory leak detectors. It also evaluates a worst-case client and measures how much the CCU-based approach relies on GC.

5.1 Methodology

Implementing the CCT. For comparison purposes, we have also implemented support for CCT-based, context-sensitive 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, and they use the same efficient lazy construction as CCU nodes. One limitation of our evaluation is the possibility that the CCT underperforms its potential because of a suboptimal implementation. We note that child maps are sparse by design, so most lookups hit on the first attempt, so unimplemented optimizations, such as using inline caching and using a map from caller call site to per-callee arrays indexed by call sites, are unlikely to improve performance significantly.

Configuring the child map. Both the CCU- and CCT-based approaches use child maps: each CCT node has a child map, while only merged CCU nodes have child maps. GC can treat these child node references like weak references [26], reducing space but potentially slowing execution by tracing more nodes and allocating and/or copying more nodes; or GC can treat child node references like strong references, in which case our implementation safely elides tracing of child nodes.

Our evaluation explores this tradeoff by comparing four configurations:

- *CCU with weak child references*: By using weak child references, GC collects all irrelevant nodes, both unmerged and merged. For this configuration, we explore several merging strategies: no merging, in-place merging, and copy merging. We find that copy merging provides the best overall performance by eliminating duplicate nodes and limiting fragmentation.
- *CCU with strong child references*: In this configuration, GC collects irrelevant nodes that have not been merged, but it does not collect irrelevant merged nodes, allowing GC to avoid tracing merged nodes altogether since their child and parent references always point to other merged nodes. This configuration uses copy merging since it provides the best performance.
- *CCT with weak child references*: In this configuration, GC collects all irrelevant nodes since it treats child map references like weak references. To limit fragmentation, this configuration uses a multi-space strategy similar to copy merging—it allocates nodes into a copy space and copies surviving nodes into a mark-sweep space—except GC performs no merging, since CCT nodes are merged at allocation time.
- *CCT with strong child references*: In this configuration, strong child references keep all nodes live, and nodes get merged at allocation time, so GC cannot collect any nodes. This configuration thus allocates all nodes into an immortal space.

Benchmarks. In our experiments, Jikes RVM executes the DaCapo Benchmarks [9] (version 2006-10-MR2) and a fixed-workload version of SPECjbb2000 called pseudojbb [47]. We evaluate the leak and race detectors on all benchmarks except bloat because its run-to-run variability is unusually high (even without our modified JVM), making it difficult to produce high-confidence results. We execute the *large* workloads for all benchmarks, except we execute the *medium* workload for hsqldb with the race detector since the large workload runs out of memory, even without context sensitivity.

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 run

the race detector with Jikes RVM's default generational Immix collector [12] (GenImmix), and the leak detector with a generational mark-sweep collector (GenMS) since its object model leaves a few bits available for the leak detector to use to compute staleness.

To account for run-to-run variability due to dynamic optimization guided by timer-based sampling, we execute 15 trials for each measurement and take the *median*, which minimizes effects of machine noise. We also show show 95% confidence intervals centered at the *mean*, which typically differs little from the median. We let the VM choose its own heap size adaptively because the client analyses, especially race detection, add high space overhead. For plots of space overhead versus time, we show just one trial since averaging such plots is not straightforward and might unrealistically hide overhead peaks.

Platform. Our experiments execute on a 4-core Intel i5 3.3-GHz system with 4 GB memory running Linux 2.6.32.

5.2 Key Questions

Our evaluation aims to address the following questions:

- *Can the CCU-based approach avoid the vast majority of the expensive lookups performed by a CCT-based approach?* Our results show the vast majority of child map lookups are eliminated and replaced by allocations of CCU nodes (e.g., Table 1).
- *Is it cheaper to allocate a CCU node than to look up a CCT node?* Our results show that each CCT lookup is replaced with roughly one CCU allocation (Table 1), and the CCU-based approach on average provides significantly better time performance than the CCT-based approach (Figures 8, 9, 12).
- *Can GC and lazy merging of duplicate nodes provide competitive space overhead for the CCU-based approach?* These features allow duplicate CCU nodes to be merged and unreachable nodes to be collected, so the CCU adds space overhead comparable to the CCT (Table 2 and Figure 11).
- *Can the overall performance of a CCU-based approach be competitive with or better than a CCT-based approach?* On average and across nearly all experiments, the CCU-based approach provides significantly better time performance and similar space performance to the CCT-based approach (Figures 8, 9, 11, 12 and Table 2).
- *How do the context-sensitive sites reported by leak and race detectors compare to context-insensitive sites?* The context-sensitive sites provide significantly more information about dynamic behavior (Section 6).

Next we evaluate the performance of CCU-based approaches and compare to CCT-based approaches. We always evaluate using a client analysis because the approaches add zero overhead without a client, due to lazy construction.

	Allocations		Hash lookups		DCS	LVA	LVAF
	CCU	CCT	CCU	CCT			
antlr	329.8	3.4	0.2	329.0	86%	96%	14%
chart	387.3	0.4	5.5	380.0	71%	98%	29%
eclipse	4561.0	68.5	8.3	4546.0	88%	98%	11%
fop	42.4	0.2	0.3	41.1	85%	80%	8%
hsqldb	490.8	4.5	17.3	546.0	100%	98%	9%
jython	3551.0	4.1	0.2	3535.6	79%	99%	20%
luindex	1068.3	0.4	0.4	1066.4	84%	99%	19%
lusearch	1331.8	0.8	5.5	1305.0	85%	90%	14%
pmd	1645.1	12.8	6.3	1635.4	90%	99%	9%
xalan	6242.2	24.2	8.6	6198.6	92%	96%	8%
pseudojbb	995.8	0.2	15.7	996.3	92%	99%	8%

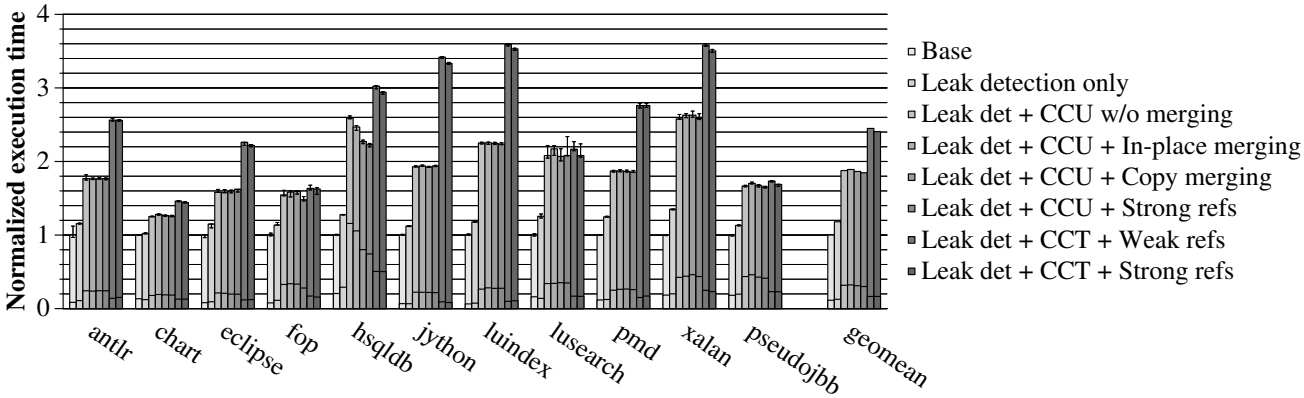
Table 1. Nodes allocated and hash lookups performed for the CCU- and CCT-based approaches (all in millions). The last three columns report how often client sites benefit from optimizations.

5.3 Memory Leak Detection

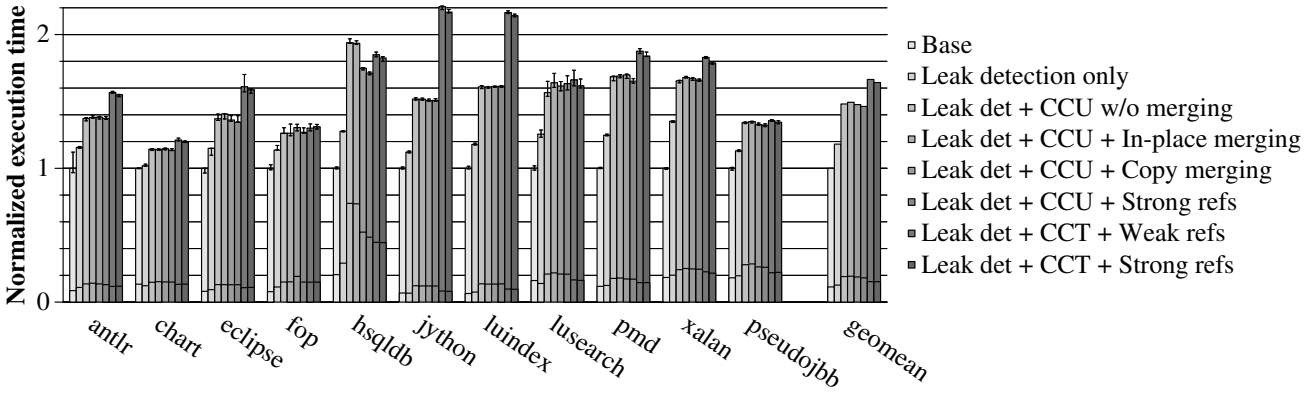
This section evaluates the performance of our memory leak detector that tracks the last-use (i.e., last read) sites of all objects (Section 4.4). We experiment with two CCU/CCT configurations. The first stores a pointer to a CCU or CCT node in each object's header that represents the last-use site's calling context, *including a node for its client site*. The second configuration stores a node representing the last-use site's *calling context not including the client site*, and it uses a second header word for an identifier representing the client site. We say the first configuration uses *client site nodes*, and the second configuration does not. Client site nodes can potentially reduce space overhead since (1) program location can be recorded in just one word of client metadata (instead of two) and (2) multiple client metadata can point to the same client site node after merging. Client site nodes stress CCU and CCT performance more, since more nodes are constructed or looked up.

Run-time characteristics. Table 1 shows run-time statistics for the configuration that uses client site nodes; all numbers are in millions. These results use the CCU with weak child references and CCT with strong child references; we expect weak versus strong references to have a modest impact on these results. *Allocations* is the number of nodes allocated. The CCU-based approach allocates roughly 2–3 orders of magnitude more nodes than the CCT-based approach. However, the CCU-based approach performs about the same factor fewer *Hash lookups* than the CCT-based approach, indicating that the vast majority of CCU nodes become unreachable before they can be merged.

The last three columns are for the CCU-based approach (results for the CCT-based approach are similar). *Dynamic client sites (DCS)* is the fraction of nodes allocated for client sites (rather than call sites). *Local variable attempts (LVA)* is the fraction of *DCS* that use the local variable optimization from Section 4.2. Values are close to 100% because all optimized code uses this optimization; some cold methods are not compiled by the optimizing compiler. *Local variable attempt failure (LVAF)* is the fraction of *LVA* that the local



(a) CCU and CCT configurations allocate client site nodes.



(b) CCU and CCT configurations store client sites in object headers.

Figure 8. Normalized execution time for leak detection (a) using client site nodes and (b) with client sites in object headers. The graphs compare context-insensitive leak detection and CCU- and CCT-based context sensitivity. Bars 3–5 in each group use the CCU (with weak child references) without merging and with two merging algorithms. Sub-bars are GC time.

node variable is null—often only about 10% of the time, suggesting that using the local node variable is a worthwhile optimization.

Time overhead. Figure 8 shows the normalized application time of various leak detection configurations. All bars are normalized to unmodified Jikes RVM (*Base*). The *Leak detection only* configuration records context-insensitive sites and adds 18% overhead on average to record last-use sites and track object staleness. The four *Leak det + CCU* configurations construct CCU nodes to represent the context of each last-use site. The first three CCU configurations use weak strong child references and either do not merge duplicate nodes, or use copy or in-place merging; the fourth CCU configuration uses copy merging with strong child references.

Figure 8(a) corresponds to the configuration that uses client site nodes. CCU-based context sensitivity adds 67–72% overhead over context-insensitive leak detection, depending on the merging configuration and the child reference type. CCT-based context sensitivity adds substantially more overhead—123 or 127% depending on the child reference type—because the cost of looking up or constructing each node is substantially higher for the CCT than for the CCU. Figure 8(b) shows overhead for storing the client site in object headers. CCU-based context sensitivity adds only 28–31% overhead on average over context-insensitive leak detection, depending on the merging configuration and the child reference type. Using the CCT instead of the CCU adds on average 46 or 48% overhead over context-insensitve leak detection, depending on the child reference type.

For the CCU-based approach, using strong child references has no significant impact on performance, with similar results across all benchmarks (1% less overhead on average than using weak reference with copy merging). This result is not surprising since only longer-lived nodes have child references, so most child references remain live in any case. The CCT-based approach benefits more (2–4%, relative to baseline execution), since the cost of creating new nodes is greater: allocating a node that is not found on lookup requires repeating the lookup in a small critical section.

In both configurations, across all benchmarks, the CCU-based approach (with copy merging) performs about the same as, or significantly better than, the CCT-based approach. For some programs, such as jython and luindex, the CCU-based approach drastically outperforms the CCT-based approach.

Merging has little effect on time overhead, although we show next that it reduces space overhead significantly. Merging's time benefit is modest since (1) merging adds its own GC overhead and (2) our experiments let the JVM grow the heap automatically, so the extra memory pressure without merging does not necessarily lead to more frequent GC.

In both configurations, the CCU-based approach adds high overhead to hsqldb, with much of it due to GC (subbars are GC time). Unsurprisingly, hsqldb, which we show allocates significantly more CCU nodes than the other programs, benefits the most from merging of duplicate nodes.

Figure 8(a)'s results include hoisting of client site node allocation out of hot loops (Section 4.2). Without this optimization, the CCU- and CCT-based approaches' performance degrades by 15% and 66%, respectively (results not shown). The CCT-based approach is helped more by this optimization because its hash lookup at each client site is more expensive than the CCU-based approach's node allocation.

The CCU-based approach adds extra GC overhead due to both (1) increasing the allocation rate, which triggers GC more frequently, and (2) increasing the GC workload from transitively reachable CCU nodes. We estimate the contribution of each cause by executing a configuration of CCU-based leak detection (with client site nodes) *without storing CCU pointers into per-object metadata*, which measures the allocation rate increase but not the workload increase. We find that the GC overhead of this configuration is 15%, whereas Figure 8(a) shows the full CCU configuration incurs 20% GC overhead (both relative to context-insensitive leak detection). We conclude that the majority (about three-quarters) of GC overhead comes from the allocation rate increase, and the rest comes from the GC workload increase.

Space overhead. Table 2 shows the average 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 average across the execution. Table 2(a) uses client site nodes, and Table 2(b) does not (i.e., same as Figures 8(a) and 8(b), respectively). The results indicate that merging duplicate nodes is sometimes critical to avoid high space overhead. *Copy* merging in particular improves memory overhead significantly. We have determined that both merging algorithms perform similarly in terms of the *number* of merged nodes, but *In-place* merging has higher space overhead because it fragments the heap significantly and adds two words for every node (one for the childMap and one for the next pointer; Section 4.3), not just merged nodes.

	CCU Weak w/merging:			CCU	CCT	
	None	In-place	Copy	Strong	Weak	Strong
antlr	1,675	1,318	1,241	1,286	1,239	24,347
chart	33,299	16,088	1,222	1,259	1,187	1,594
eclipse	56,324	30,708	3,481	8,441	3,315	283,313
fop	2,048	1,955	1,197	1,210	1,185	1,214
hsqldb	59,796	32,840	1,283	1,287	1,402	10,304
jython	3,499	3,466	2,932	3,090	3,103	48,878
luindex	2,822	1,708	1,602	1,676	1,605	10,972
lusearch	44,644	33,143	1,994	2,044	1,656	6,598
pmd	40,348	12,386	1,662	2,246	1,532	62,239
xalan	62,382	36,924	4,309	26,880	3,803	146,239
pseudojbb	41,091	26,264	2,754	2,775	2,780	980

(a) CCU and CCT configurations allocate client site nodes.

	CCU Weak w/merging:			CCU	CCT	
	None	In-place	Copy	Strong	Weak	Strong
antlr	1,355	1,256	1,209	1,198	1,197	4,791
chart	17,520	8,186	1,187	1,191	1,174	665
eclipse	26,336	21,404	2,993	3,245	2,988	59,640
fop	1,637	1,785	1,172	1,172	1,166	288
hsqldb	43,705	25,763	1,248	1,246	1,267	2,055
jython	2,941	3,032	2,723	2,735	2,834	17,142
luindex	1,902	1,653	1,594	1,608	1,604	1,880
lusearch	21,246	16,418	1,505	1,531	1,417	2,534
pmd	15,243	7,815	1,469	1,727	1,425	10,130
xalan	22,341	7,115	3,293	3,740	3,194	32,044
pseudojbb	18,058	11,432	2,644	2,657	2,723	312

(b) CCU and CCT configurations store client sites in object headers.

Table 2. Memory consumed by CCU and CCT nodes, in KB, for context-sensitive leak detection with (a) client sites stored in nodes and (b) with client sites stored in object headers. The first three CCU configurations use weak child references. *CCU Strong* uses copy merging but avoids tracing merged nodes. *CCT Weak* mimics copy merging by allocating into a copy space and copying to a mark-sweep space. *CCT Strong* uses an immortal space.

For the CCU, *Strong* child references do not add much space overhead over weak references using copy merging, since only longer-lived nodes get merged. Weak references do reduce space overhead in a few cases, e.g., xalan with client site nodes. Weak references are more important for the CCT; using strong references leads to high space overhead in several cases because irrelevant nodes are never collected. Weak references add more space overhead than strong references in a few cases (pseudojbb in both tables; chart and fop in Table 2(b)) because weak references lead to fragmentation in the mark-sweep node space.

5.4 Data Race Detection

This section evaluates the overhead of the context-sensitive race detector that records context-sensitive sites at reads and writes. The race detector does not allocate client site nodes and instead uses a separate word per field and array element for the context-insensitive client site (Section 4.4).

Time overhead. Figure 9 shows the run-time overhead that race detection adds to programs. Context-insensitive race detection slows programs by about 7.8X on average, which

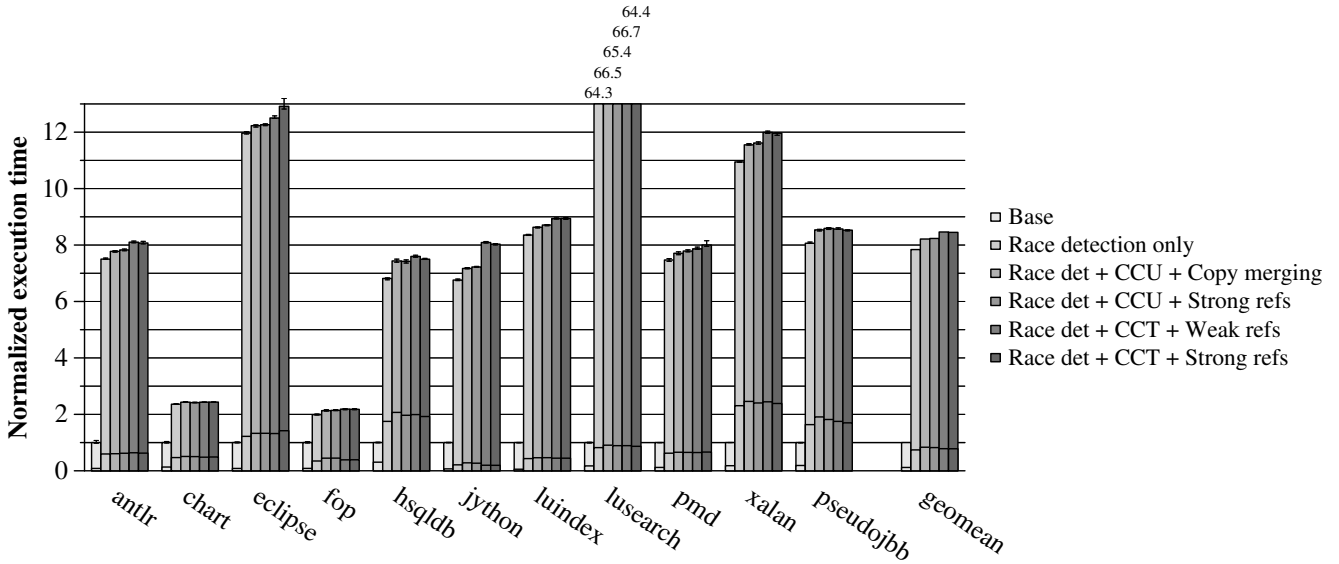


Figure 9. Time overhead of race detection with and without CCU- and CCT-based context sensitivity. Sub-bars are GC time.

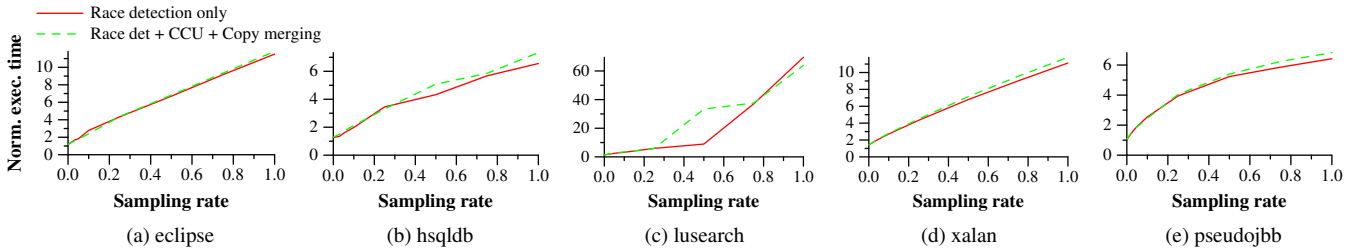


Figure 10. Normalized execution time of race detection, with and without CCU-based context sensitivity, at various sampling rates (0%, 5%, 10%, 25%, 50%, 100%).

matches prior results for the FastTrack and Pacer implementations [14, 22]. CCU-based context sensitivity with merging (with weak child references) adds 37% average overhead, relative to *original program execution time*, over context-insensitive race detection, while CCU-based context sensitivity with strong child references adds 40% average overhead. CCT-based context sensitivity with strong child references adds 61% average overhead while CCT-based context sensitivity with weak child references adds 62% average overhead over context-insensitive race detection. If we consider only the multithreaded benchmarks (eclipse, hsqldb, lusearch, xalan, and pseudojbb), the CCU-based approach adds 61% and 60% (weak and strong child references, respectively), while the CCT adds 96% and 79% (weak and strong child references, respectively). .

We further break down the overhead for the multithreaded benchmarks into three parts (results not shown): stack walking adds about 20% overhead (relative to original program execution); storing CCU nodes into metadata adds about 10%; and allocation and GC of nodes add about 20%.

We evaluate the overhead of the CCU-based race detector at different sampling rates, in order to detect how well a CCU-based approach's overhead 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 10 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 3.1). We also measured (but do not show) time overhead across sampling rates for the CCT, which also scales well because it uses lazy construction.

Space overhead. Figure 11 shows the memory used by CCU and CCT nodes across an execution. We omit CCU with strong child references since its space overhead is similar to CCU with weak child references (e.g., Table 2). 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 but does not merge duplicate nodes, clearly shows the need for merging. By merging duplicate nodes, the space overhead added by the CCU is reduced substantially. Furthermore, whereas CCU space overhead grows

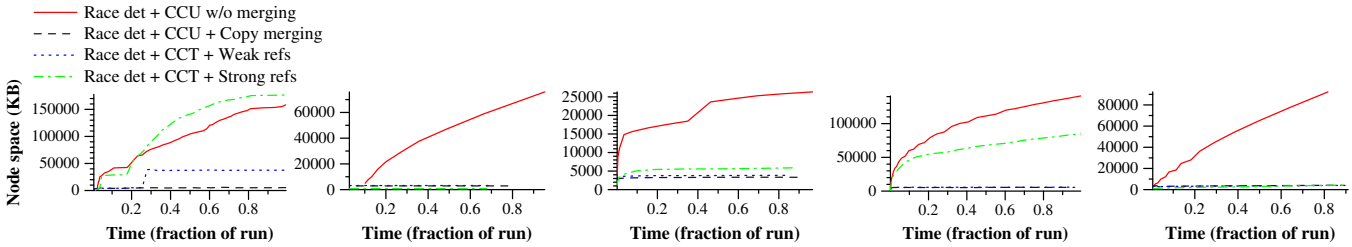


Figure 11. Space used by CCU and CCT nodes for context-sensitive race detection, with and without CCU node merging.

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 CCU with copy merging provides space overhead similar to, or significantly lower than, the CCT with both strong and weak child references.

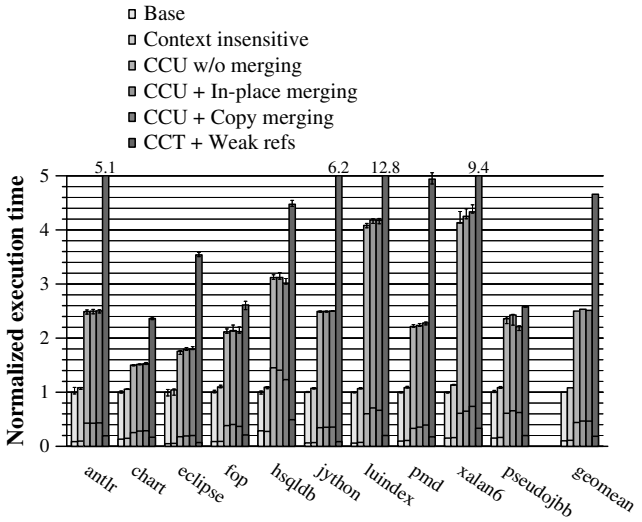
5.5 Evaluating a Worst-Case Client

Since our leak detector instruments only reference reads and our race detector avoids instrumenting reads and writes for "same epoch" cases (Section 4.4), we evaluate a simple, plausible worst case for CCU- and CCT-based analyses: instrumenting every read and write to record the last-access site in per-object metadata. To keep the context-insensitive client simple and its overhead low, this client's instrumentation does not perform any synchronization.

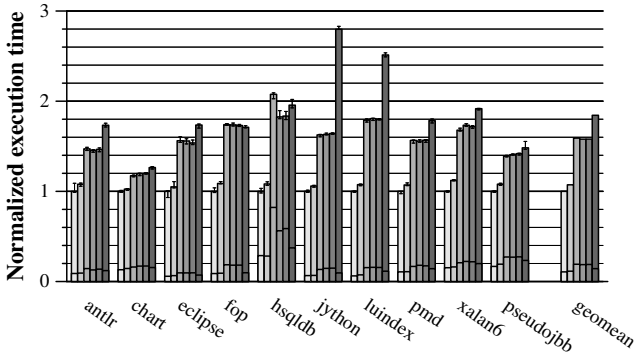
Our evaluation excludes lusearch because it crashes for unknown reasons with context-*insensitive* instrumentation. We execute the medium workload for eclipse because the CCT-based configuration runs out of memory on the large workload. We experiment with configurations with and without client site nodes, as for the leak detector (Section 5.3).

Figures 12(a) and 12(b) show the normalized application time with and without client site nodes, respectively. The *Context insensitive* configuration records context-insensitive sites and adds about 8% overhead on average. The *CCU* and *CCT* configurations are the same as the weak reference configurations from Figure 8.

When the CCU and CCT construct client site nodes (Figure 12(a)), CCU-based context sensitivity adds 140–145% overhead, depending on the merging configuration and weak versus strong child references. CCT-based context sensitivity adds substantially more overhead—356 and 358% on average for strong child reference and weak child reference, respectively—because the cost of looking up or constructing each node is substantially higher for the CCT than for the CCU. For configurations that store client sites in object headers (Figure 12(b)), CCU-based context sensitivity adds 49–52% overhead on average to leak detection, depending on the merging configuration and child reference type. The CCT-based approach adds 77% overhead on average.



(a) CCU and CCT configurations allocate client site nodes.



(b) CCU and CCT configurations store client sites in object headers.

Figure 12. Normalized execution time for instrumenting all reads and writes so they store last-access sites. The context-insensitive, CCU, and CCT configurations correspond to Figures 8(a) and 8(b).

	CCU		**CCU**	CCT	
	Weak	Strong	**Immortal**	Weak	Strong
antlr	1,203	1,220	145,575	1,194	4,591
chart	1,176	1,189	258,242	1,174	647
eclipse	2,679	2,690	28,390	2,664	1,305
fop	1,171	1,171	18,484	1,168	272
hsqldb	1,251	1,245	180,469	1,269	3,289
jython	2,670	2,675	293,213	2,726	12,306
luindex	1,595	1,615	527,197	1,604	1,821
lusearch	1,434	1,532	477,792	1,418	2,199
pmd	1,476	1,738	403,116	1,429	9,558
xalan	2,982	3,073	266,059	2,944	10,342
pseudojbb	2,644	2,646	216,943	2,732	327

Table 3. Memory consumed by CCU and CCT nodes, in KB, for context-sensitive leak detection without client site nodes. *CCU Immortal* is the immortal CCU-based approach, and the other columns are the same as Table 2(b); *CCU Weak* uses copy merging.

5.6 Reliance on Tracing GC

The CCU-based approach avoids the cost of looking up existing calling context nodes, but it allocates substantially more nodes than a CCT-based approach that reuses nodes. This section evaluates how much the CCU-based approach relies on tracing GC to collect irrelevant nodes—by disabling GC of irrelevant nodes.[8]

Implementation and methodology. We have implemented an alternative "immortal" CCU-based approach that allocates CCU nodes into an immortal space that GC does not trace, saving GC time but using extra heap memory and triggering GC more frequently. This approach is thus related to our CCT-based approach with strong child references, which also allocates nodes into an immortal space that GC does not trace.

The immortal CCU-based approach runs out of memory for several benchmarks when used with the leak detection client, especially when using client site nodes, so we use a configuration without client site nodes. Some programs still run out of memory with the large workload size; we use the small size for eclipse and medium size for jython and xalan.

Space overhead. Table 3 shows the live memory used by CCU and CCT nodes, by measuring the memory consumed by the node space(s) at each full-heap GC and reporting the average across the execution. The results are similar to Table 2(b), except that Table 3 adds the immortal CCU-based approach. The immortal CCU-based approach uses 1–2 orders of magnitude more memory than the other CCU- and CCT-based approaches that use GC, including even the CCT with strong child references.

Time overhead. We find that the immortal CCU-based approach actually improves the time performance slightly over the tracing-baed CCU-based approaches: by 2% on average, relative to baseline execution time (results not shown). This improvement comes from reducing GC time, since immortal CCU nodes are not traced or merged.

The results indicate that the CCU-based approach relies on tracing GC to collect irrelevant nodes—which increase space overhead drastically if not collected or merged. Although irrelevant nodes increase memory pressure, GC time does not increase significantly in our experiments since they allow the heap size to grow automatically.

5.7 Summary of Quantitative Evaluation

In summary, the CCU-based approach almost always outperforms the CCT-based approach in terms of time and performs comparably in terms of space. More significantly, this result demonstrates the potential of the CCU-based approach—which might seem counterintuitive at first—to provide dynamic context sensitivity efficiently.

6. Evaluating Context Sensitivity

While this paper mainly evaluates performance, this section evaluates whether context sensitivity provides useful information to bug reports.

Qualitatively Evaluating Context-Sensitive Leaky Sites

We first qualitatively evaluate two real leaks that were reproduced and evaluated by prior work [15, 31]: one in SPECjbb2000 [47] and one in Eclipse.[9] Our reported leaky sites do not exactly match those from the prior staleness-based leak detector that evaluated the same leaks [15] because our detector updates an object's staleness and last-use site when a *reference to the object is loaded* [17], instead of when the object itself is modified.

SPECjbb2000 leak. The SPECjbb2000 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 logic to properly remove finished orders [15]. Our context-sensitive leak detector reports 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 with the Eclipse IDE). This

[8] We considered implementing a variant that allocates CCU nodes and does not collect irrelevant nodes, but merges duplicate nodes. However, this variant and the standard CCT-based approach should perform similarly since they each merge every allocated node.

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

gives a sense of how "nontrivial" the context is, i.e., how hard it might be 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 method inlining by the optimizing compiler, from which it is relatively easy to find the buggy site.

Eclipse leak. Eclipse bug #115789 leaks memory when a "recursive difference" is performed between two source trees.[10] 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 [15]. Our leak detector reports one last-use site in NavigationHistory, shown in Figure 1.[11] This calling context illustrates how comparing source trees in Eclipse ultimately leads to stale last-use sites in NavigationHistory. The elided call sites are other Eclipse internal methods. We note that the leak *fix* involves changing NavigationHistory, so the context-*insensitive* site is useful by itself. On the other hand, code tree comparisons cause the leak, so developers might find the connection between CompareUI.openCompareEditorOnPage() and NavigationHistory useful for understanding the leak. In any case, this example shows that context-sensitive sites can provide significantly *more* information to developers for highly object-oriented programs.

As in prior work, our implementation reports a few other last-use sites (for both leaks) for a growing number of stale objects, but these sites are not directly related to the bug fix, so we do not evaluate their contexts.

Quantitatively Evaluating Context-Sensitive Racy Sites

To avoid spending substantial time understanding reported data races, we have only quantitatively (not qualitatively) evaluated the calling contexts reported by the race detector. Table 4 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) vary greatly across the programs.

We have evaluated the possibility that adding context sensitivity with the CCU or CCT could impact the number of races reported (results not shown). However, the number of distinct and dynamic races reported do not vary significantly across configurations: the 95% confidence intervals always overlap—except in one case, which is not significant given the number of independent comparisons involved.

	Distinct		Dynamic
	Context insensitive	Context sensitive	
eclipse	41	59	667,230
hsqldb	9	22	336
lusearch	81	81	21,035,554
xalan	11	14	138
pseudojbb	21	21	920,462

Table 4. Distinct, prior context-insensitive accesses and context-sensitive 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
lusearch	35	15	31	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 5. Histogram showing the depth (in call sites) of context-sensitive sites reported by the race detector.

Table 5 is a histogram showing the depths, in terms of number of sites, of 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.

7. Related Work

Section 2.1 compared our CCU-based approach with a CCT-based approach. This section discusses other alternatives for providing dynamic context sensitivity, plus other related topics.

Walking the stack. Client analyses can simply walk the entire call stack at each client site [16, 24, 37, 44, 50, 56], which is expensive when client sites execute frequently. 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, 56], which is equivalent to lazy construction of the CCT (Section 3.1).

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 including recursion, which leads to infinitely many statically possible contexts.

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

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

[11] Prior work reports this site and another in NavigationHistory [15]. This difference is due to the differing instrumentation strategies described earlier.

computes these increments at compile time by applying the Ball–Larus intraprocedural path profiling algorithm [7] to the call graph.[12] This algorithm also enables efficient reconstruction of a calling context from its value.

While PCCE enables efficient encoding and profiling of calling contexts, it is fundamentally solving different problems than our CCU-based approach. First, PCCE is not well suited to providing context sensitivity for bug detection analyses. To handle the challenge of many statically possible paths, PCCE represents a single context with a variable number of integers. We believe that supporting variable-sized, per-object metadata would add significant overhead to a client, but the PCCE paper does not evaluate any clients [48].

Second, PCCE is inherently unable to handle dynamic class loading and virtual method dispatch, limiting its applicability. PCCE relies on knowing the static call graph at compile time in order to number the call graph using the Ball–Larus algorithm, which is not possible under dynamic class loading. PCCE relies on instrumenting call edges efficiently, but instrumenting virtual method calls incurs extra costs [40]. PCCE handles indirect calls (e.g., indirect calls via method pointers in C) by adding an integer to the stack of integers that represent context, which is reasonable if indirect calls are rare, but would add high time and space overhead if used to handle the uncertainty introduced by dynamic class loading and virtual method dispatch.

Breadcrumbs handles dynamic class loading and virtual method dispatch, and it maps each calling context to a single integer word [13]. It computes a probabilistically unique value for each calling context by computing an incremental hash function at each call site [16]. 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 run-time values such as program counters and stack depth [28, 35]. 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 accuracy somewhat but not enough to scale to complex programs with many distinct calling contexts.

Other approaches. A *call tree* constructs a new node for every dynamic call [3], but each call tree node maintains pointers to its child nodes, making it expensive to construct nodes and difficult for GC to collect irrelevant nodes (since all nodes remain reachable). Analyses can build and maintain a *dynamic call graph*, which maintains only one node per static call site, losing the ability to reconstruct client sites' calling contexts [40].

Recent work profiles all contexts using a CCT-based, VM-independent approach, but the technique focuses on portability and minimizing space overhead and adds higher overhead than high-performance CCT profilers [42].

Recent work proposes an alternative to the CCT in which contexts are bounded to depth k to save space and increase utility compared to an infinite-depth CCT [6]. This approach still requires an expensive hash lookup like the CCT.

Prior work uses sampling and data mining to trade accuracy for lower overhead when collecting calling contexts [20, 27, 56]. This tradeoff is worthwhile for determining *hot* program behavior for performance optimization. However, *cold* program behavior is critical for bug detection [18, 33].

Hash consing merges equivalent objects [4], and prior work uses hash consing to compact dynamic call trees [29]. The CCU- and CCT-based approaches are essentially performing hash consing of nodes: the CCT performs hash consing eagerly, and the CCU performs hash consing lazily.

8. 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 construct. Tracing garbage collection and a lazy merging algorithm are key components that keep space overhead low. We demonstrate a CCU-based approach's potential to outperform a CCT-based approach when adding context sensitivity to bug detection analyses, offering an appealing direction for future work on context-sensitive dynamic analysis.

Acknowledgments

We thank Daniel Frampton, Sam Guyer, Kathryn McKinley, Feng Qin, and Nasko Rountev for valuable discussions, ideas, and support. Thanks to Swarnendu Biswas, Todd Mytkowicz, Nasko Rountev, Aritra Sengupta, Xiangyu Zhang, and the anonymous reviewers for helpful feedback on the text; and to Man Cao, Aritra Sengupta, and Minjia Zhang for help with the artifact evaluation submission.

[12] 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].

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, 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.

[3] 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.

[4] A. W. Appel and M. J. R. Goncalves. Hash-Consing Garbage Collection. Technical Report TR-412-93, Princeton University, 1993.

[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] G. Ausiello, C. Demetrescu, I. Finocchi, and D. Firmani. k-Calling Context Profiling. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 867–878, 2012.

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

[8] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and Realities: The Performance Impact of Garbage Collection. In *ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 25–36, 2004.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[28] 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.

[29] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing Interactions in Program Executions. In *ACM International Conference on Software Engineering*, pages 360–370, 1997.

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

[31] 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.

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

[33] D. 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.

[34] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-to and Side-Effect Analyses for Java. In *ACM International Symposium on Software Testing and Analysis*, pages 1–11, 2002.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, M. Schoeberl, and M. Mezini. Portable and Accurate Collection of Calling-Context-Sensitive Bytecode Metrics for the Java Virtual Machine. In *ACM International Conference on Principles and Practice of Programming in Java*, pages 11–20, 2011.

[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] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking. Barriers Reconsidered, Friendlier Still! In *ACM International Symposium on Memory Management*, pages 37–48, 2012.

[54] 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.

[55] 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.

[56] 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.