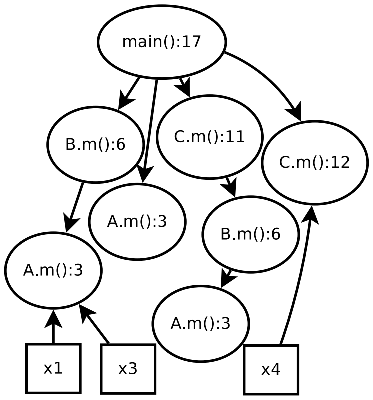
Efficient, Context-Sensitive Dynamic Analysis via Calling Context *Up*trees

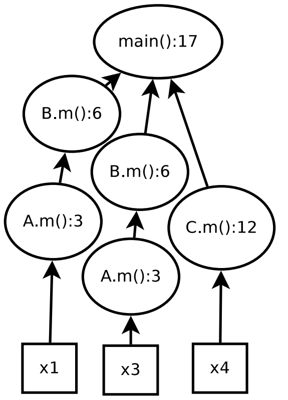
Jipeng Huang & Michael D. Bond (Ohio State University)

```
 1 class X { int f; }
 2 class A {
 3   m(x) { read x.f; }}
 4 class B extends A {
 5   m(x) {
 6     super.m(x);
 7     globalSet.add(x);
 8   }}
 9 class C extends B {
10   m(x) {
11     super.m(x);
12     read x.f;
13   }}
14 main() {
15   A a; B b; C c; ...
16   for(A tmp:{b,a,b,c}) {
17     tmp.m(new X());
18   }}
```

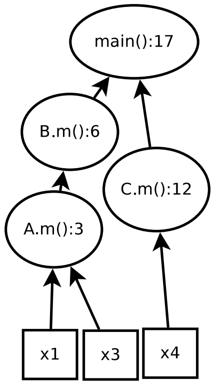
Example program



CCT



CCU (before merging)



CCU (after merging)

Problem. To diagnose and fix bugs such as data races and memory leaks, programmers need to know which program locations cause these bugs. Dynamic program analysis can report buggy program locations by recording, for example, the last program location to read and write each variable, and then reporting these locations for variables that the analysis later determines are involved in data races or memory leaks. *Static* program locations are often not enough to understand what the program was doing—especially as software becomes more complex and concurrent—and programmers really want to know a program location's *calling context* (in this work, the active call sites).

A dynamic analysis can record the calling context of program statements by constructing and maintaining each thread's position in a *calling context tree* (CCT) [1]. Each CCT node represents a distinct calling context and consists of a call site and a mapping from callee call sites to child nodes. In the example CCT above, we suppose that a client is recording the context of the last read of each object (read x.f). The objects x1–x4 are the instances of new X(), and each points to the context that last read it. (There is no "x2" because the program does not store it into globalSet, so garbage collection (GC) collects it quickly.)

Constructing and maintaining each thread's position in a CCT slows programs by 2–3X or more because it is expensive to find and reuse an existing child node at every program call. Each call site has many statically possible callee call sites because each call site may call multiple virtual methods, and each of these methods may contain multiple call sites. The number can grow over time due to dynamic class loading. At run time, a given context executes relatively few of its statically possible callee call sites, so efficient implementations use an indirect lookup such as a hash table to find the existing child node (if any) for a callee call site. Thus, each program call requires an indirect lookup to keep track of the current CCT node. The CCT also adds high space overhead due to millions of distinct contexts, many of which are irrelevant to the client analysis.

Solution. We propose the *calling context uptree* (CCU), which does *not* maintain pointers to child nodes. A CCU node consists of its call site and a pointer to its parent node. At each program call, a CCU-based approach allocates a new node and sets its parent pointer to the parent calling context node. The CCU thus **trades space for time**: it avoids an indirect lookup at each program call but creates a new node instead.

However, the **extra space may not be a real problem.** Many nodes do not live (stay reachable) for long. For example, a race detector stores the context of only the last read and write to a variable, so prior accesses' contexts will die (become unreachable); also, the variables themselves may die. Tracing-based GC, which is proportional to the live (not dead) objects, naturally and efficiently collects the many dead nodes. In the example CCU (before merging), GC has collected two unreachable contexts that were present in the CCT, main():17 → A.m():3 and main():17 → C.m():11 → B.m():6 → A.m():3.

Still, the CCU contains redundant nodes for context main():17 ← B.m():6 ← A.m():3. We propose *lazily merging* redundant nodes, to achieve the example CCU (after merging). In spirit, merging is similar to a CCT reusing existing child nodes. However, by merging only long-lived nodes, a CCU can avoid a lot of work because many nodes die young and are never merged.

To avoid unnecessary work, a client analysis can build CCU nodes (or CCT nodes, in fact) *on demand*. Each stack frame stores a pointer to the caller call site's node, if it has been constructed. At a program read or write (if the client is race detection), the on-demand algorithm walks the stack and constructs nodes until it finds an existing node on the stack or reaches the top of the stack.

Related work. Dynamic analysis can walk the stack whenever context is needed [4]; stack-walking is cheap only if it is rare. Recent probabilistic approaches trade accuracy for performance but do not scale well to many distinct contexts [2, 3]. Uniquely numbering each context, for example via path profiling [5], does not scale well because even without considering recursion, the number of statically possible contexts is much larger than 2^{64} in real programs.

Conclusion. The CCU trades time for space in order to provide efficient context sensitivity to dynamic bug detection analyses. We have implemented the CCU (but not yet merging, nor a few other optimizations), as well as race and leak detectors that use the CCU, in a Java Virtual Machine. Preliminary results suggest that CCU-based bug detection adds low enough overhead for all-the-time use in production systems, where it can help programmers better understand the causes of software bugs.

Thanks to Todd Mytkowicz for helpful feedback on the text.

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97*.

[2] M. D. Bond, G. Z. Baker, and S. Z. Guyer. Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses. In *PLDI '10*.

[3] T. Mytkowicz, D. Coughlin, and A. Diwan. Inferred call path profiling. In *OOPSLA '09*.

[4] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07*.

[5] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. In *ICSE '10*.