Efficient, Context-Sensitive Dynamic Analysis via Calling Context Uptrees
Jipeng Huang & Michael D. Bond (Ohio State University)

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 from callee call sites to child nodes. In the example CCT above, the client analysis.

Constructing and maintaining each thread’s position in a CCT slows programs by 2–3X or more because it is expensive to find an existing node on the stack or reach the top of the stack. 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 $2^{30}$ in real programs.

Conclusion. The CCT trades time for space in order to provide efficient context sensitivity for dynamic bug detection analyses. We have implemented the CCT (but not yet merging), and a few other optimizations), as well as race and leak detectors that use the CCT, in a Java Virtual Machine. Preliminary results suggest that CCCU-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.