Review of Software and the Concurrency Revolution by Herb Sutter and James Larus

Riley Wood

March 30, 2016

Key Ideas

The main goal of this paper is to describe the necessary trajectory for programming languages and compilers given the recent trend toward multiprocessor systems. Sutter and Larus highlight several key shortcomings in how concurrenct is exposed to programmers today; they also summarize existing approaches to these problems as well as suggest what they believe to be the right solutions. They describe how the global nature of synchronization locks defies modularity and makes it hard to blackbox functionality, because a function that secretly uses a lock could cause deadlock to occur. Locks also have no easy way of being coupled with the data they protect - it is simply up to the programmer to remember how locks and data are correlated. At the time of writing, programming languages were also not capable of catching the new programming errors that concurrency introduces, such as deadlock, livelock, data races, and others. It also becomes more dificult to debug programs because concurrency grows the program's state-space. In short, software researchers will need to develop deterministics ways of debugging and proving the reliability of potentially-nondeterministics concurrent programs.

Review

Reading through the article, I see that some of the problems mentioned have been addressed since the article was published. Other problems have yet to be solved. After reading over the different categories of approaches to concurrency in programming languages, I wonder how LabView would be categorized in terms of its concurrency. LabView is essentially parallelized by default. Operations are executed as soon as their inputs are available, and you can have many concurrent data paths all execute at once. A visual programming approach makes it easier to think about concurrency, since you can see all of the datapaths at once. Also, on the subject of catching concurrency errors, languages like Rust can catch such errors at compile time. Rust allows data to be bound to only one identifier (aka no making copies of pointers) as a way of

avoiding data races. This is called its ownership system; ownership of data can belong to only one binding at a time. Ownership is also transferred when passing data to a function; alternatively, data can be borrowed in either a mutable or immutable state by functions. This actually reminds me of the MESI cache coherence protocol in which data can be shared among cores with different sets of permissions. Ultimately, these constructs should help programmers feel more comfortable embracing concurrency knowing that the language itself will keep them safe from difficult-to-find concurrency errors. On the subject of coupling locks with the data they protect, I haven't heard of any frameworks that do such a thing, but I expect they must exist - it can't be hard. In fact, it seems like Rust also takes care of this problem; its Mutexes return the data they protect when the lock is acquired. Rust enforces "locking data, not code". On the subject of debugging, right now Rust is debugged with gdb which, as far as I know, doesn't have mechanisms for evaluating all corner cases of concurrent programs. As recently as 2013, research on debugging concurrent software was still ongoing [1], so there is not yet a widely-used concurrent debugging solution.

Conclusions

Sutter and Larus raise valid concerns about the state of software in 2005 with regards to concurrency. They were also right about the increasing importance of concurrency in software - since 2005 multicore has dominated the market and been the primary source of performance gains over clock frequency, which has not changed much in the past ten years on average. Since the article was written, most of the problems they acknowledge have been addressed with new programming languages like Rust. Rust helps keep the programmer "safe" when writing concurrent programs with new language constructs and a different take on scope. Sutter and Larus point out that new debugging techniques will also need to emerge to convincingly validate concurrent programs, yet few concurrent debuggers have been released. This should be the next focus of software research.

References

[1] Shaoming Huang. *Effective Methods for Debugging Concurrent Software*. PhD thesis, The Hong Kong University of Science and Technology, 2013.