Question being answered

What is the most inventive or innovative thing you've done? It doesn't have to be something that's patented. It could be a process change, product idea, a new metric or customer facing interface – something that was your idea. It cannot be anything your current or previous employer would deem confidential information. Please provide us with context to understand the invention/innovation. What problem were you seeking to solve? Why was it important? What was the result? Why or how did it make a difference and change things?

Background

A team of engineers created a family of concurrent algorithms (Read-Copy Update, RCU) that achieve near optimal multiprocessor scalability by allowing concurrent accessors and a mutator to access possibly different versions of a data object. Unfortunately, it is not known why these algorithms work because the engineers who created them are unable to formalize the mechanism of these algorithms. The lack of a proof or verification means that the algorithms cannot be used in certain "critical" applications that are regulated by the FCC, FDA, DoD, etc. Additionally, vendors with specialized hardware devices won't use them because of the penalties involved when SLA's are not met (e.g., how do I fix a problem when I don't understand how it works?).

A formal demonstration (I am going to elide over the difference between a proof and a verification) of correctness of an algorithm is arbitrarily hard; I often don't know if the reason such a demonstration eludes me is because no such demonstration exists (the unprovability of provability) or that I am not clever enough to find it. In the case of RCU, I showed that RCU resisted rigorous analysis for (at least) 3 reasons: 1) Traditional semantic based approaches under approximated modern processor-memory systems; 2) RCU relied on arcane operating system interactions that aren't reconciled with language semantics; and 3) RCU lacked behavioral specifications (i.e., how would I know a correct implementation from an incorrect one?).

In a previous effort, I showed an isomorphism between RCU and a garbage collection system and proposed that given this isomorphism the practical behavioral properties of RCU were that it didn't "collect" live objects nor did it access memory that had been collected; I called these properties live object safety and memory safety, respectively. I worked with the engineers who created RCU and gained consensus that these were the salient properties impeding the use of RCU in certain proprietary OS's and applications.

The immediate problem was to show that RCU had these properties; but, there lurked a larger problem: How do you reason about the behavior of a program whose correctness relies on the behavior of the system on which it executes and which isn't captured in the semantics of the language in which the program is written? Equally important: What if the programmer takes advantage of some "internal" behavior exhibited by a class of processors? That program won't necessarily work when executed on a different class of processors even though the program hasn't changed.

Solution Overview

I created a minimal language capable of expressing the RCU algorithm. I incorporated context switch behavior in the language. The language semantics were defined using small-step semantics that mapped language constructs to configuration transformations. Traditionally, context switches have the same semantics as a "NoOP" (an instruction that doesn't do anything) but that's not quite correct. For example, context switches in X86 Linux typically cause all issued writes to complete.

I used linear temporal logic to express the live object and memory safety properties. The verification uses the set of traces comprised of sequences of configurations that are created by simulated

execution of the RCU program via the operational semantics as models to determine the validity of these properties.

To make the verification "easy," I instrumented the semantics — for example, I added semantics that caused the program to "crash" if reclaimed memory was accessed and I instrumented the code by adding statements that made it very easy to know when an object was safe to reclaim. The result of this effort was a verification that a program P' (which looked a lot like RCU) executing semantics S' (which looked a lot like the semantics of the programming language in which RCU was written) had the desired properties. The problem, of course, is that P' wasn't RCU and S' wasn't the actual semantics.

In what follows, I continue to use P' to mean the instrumented code and use P to mean the actual RCU code. Similarly S' means the instrumented semantics and S means the actual semantics. The verification I performed is written as (S', P') |= V and means that all traces (remember that concurrent programs have interleaving semantics) of P' executing semantics S' satisfy property V (live object and memory safety).

Let us say we find restrictions R such that the instrumented code in P' is dead and suppose that the same restrictions make the instrumented semantics in S' dead as well. Then, we have

(R => (S', P') = (S, P)), ((S', P') |= V) => (S, P) |=V

As a verification strategy, this reads — I find a suitable instrumentation to create an instrumented program and semantics that enable me to demonstrate (prove/verify) that a program meets specification V. I then introduce restrictions on the program that eliminate the instrumentation giving me the "original" program and semantics. Under these restrictions, the original program meets specification V (but now, having lost the instrumentation, I can't technically prove/verify it). The restrictions R are useful to check implementations of RCU — if the restrictions don't hold, the implementation isn't correct.

Did This Change The World?

This work and insights from this and related work by me have enabled RCU to be used in proprietary systems resulting in improved performance (happy customers!) and improved revenue (happy management!). Insights from this work have also worked their way into a few Ph.D. theses.

This work is not without controversy. Over the past many (more than 50 at this point) years, language semantics have focused on presenting a model where each statement is executed and effects made visible before the next statement is executed. Dijkstra admonished that without this execution model; reasoning about programs would be near impossible. My work shows that it is feasible to verify algorithms that rely on complex multiprocessor-memory behavior and abstruse operating system interactions that can't be cast into the step-by-step semantics Dijkstra sought.

On a personal level, this work helped me understand the simplicity that comes from a deep analysis as opposed to a simple-minded approach that leads to hopeless complexities. And that understanding has helped me build effective products, top teams and mentor engineers and managers.