Technical perspective: Unexpected connections

Marc Shapiro

Sorbonne-Universités-UPMC-LIP6 & Inria Paris
http://lip6.fr/Marc.Shapiro/

Revision 7035 (2017-01-09)

Scalability is the capability of a parallel program to speed up its execution as we provide it with more CPUs. Back in 1967, Gene Amdahl noticed that the sequential part of a parallel program has a disproportionate influence on scalability [1]. Suppose that some program takes 100 s to run on a sequential processor. Now let's run it on a parallel computer. If we are able to parallelise, say, 80% of the code, with enough CPUs, that 80% would take essentially zero time, but the remaining sequential portion will not run any faster. This means that the parallel program will always take at least 20 s to run, a maximum speed-up of only 5×. If we are able to parallelise 95% of the code, speed-up is still limited to 20×, even with an infinite number of CPUs! This back-of-the-envelope calculation, known as Amdahl's Law, does not take into account other factors, such as increased memory size, but remains an important guideline.

In 1967, parallelism was a niche topic, but not any more. To improve program performance on today's multicore computers requires the developer to pay serious attention to scalability. Even more so for basic platform software such as operating systems, the focus of the our Research Highlight, "The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors." The paper makes the observation that, when a thread updates some shared data, and another thread wants to read or write the most recent version of that data (they *conflict*), this ends up constituting a sequential bottleneck. This is a general result that does not depend on any particular implementation; it remains true even with efficient hardware support, for instance the MESI protocol of cache-coherent processors, as detailed in the paper. Now, shared state is the bread-and-butter of operating systems. An OS contains a mass of shared state, such as process state, file system state, memory management state, resource allocation state, and so on; just about every system call reads or writes some of that data.

Here comes the paper's main insight. If two concurrent procedure calls commute (i.e., executing them in either order is equivalent), this means that neither one depends on the result of the other. Therefore, *there is no inherent reason why these calls should conflict*; and *it is possible to implement them in a way that scales well*, even if it won't

be the most obvious, or the simplest, or the most efficient implementation. Reasoning about commutativity enables us to reason about scalability independently of a particular implementation, benchmark or workload. We can *design interfaces* for scalability by ensuring that calls commute.

The advantages of commutativity in software have been known for a long time; see the paper for relevant references. It is only recently however that focus has shifted from leveraging commutativity to designing for commutativity (shameless plug for my own work [2–4]). The current paper goes well beyond previous work. First, instead of simple abstract data types, it considers the much more complex case of an OS with its intricate interface and massive amount of state. Second, it goes beyond a black-and-white "commutes/doesn't-commute" characterisation, and considers calls that may commute in some cases and not in others. This is especially important when commuting is the common case, as in many OS calls. Finally, it leverages static program verification techniques, providing a tool that automates reasoning about commutativity.

****

simple and powerful idea revisits common assumptions use of static verification techniques

Fast as long as no shared state (embarrassingly parallel). Shared state is inherent in many algos (e.g., OS). Non-scalability: coherence protocol (or its software equivalent). This is a principle, independent of implementation (MESI or other).

commutativity

shared-memory multiprocessor scalability

relate to distributed systems + CRDTs

beyond simple data types: commutative wrt current state. It's a win if commutativity is the common case. Many interfaces non-commute only in rare situations (resource exhausted).

principled, reason about scalability independently of implementation

connection between math theory and OS practice

high-performance data structures don't scale

use of static verification techniques

built a tool, checks a model of the APIs

designed an OS, most interfaces commute, most commuting interfaces are scalable (sometimes it's not worthwhile doing the work).

1 Conclusion

References

[1] Gene Myron Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proc.*, volume 30 of *AFIPS Conference Proc.*, pages 483–485, Atlantic City, NJ, USA, April 1967. AFIPS, AFIPS Press.

[2] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de Recherche 7506, Institut National de la Recherche en Informatique et Automatique (Inria), Rocquencourt, France, January 2011.

[3] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and V. Villain, editors, *Int. Symp. on Stabilization, Safety, and Security of Dist. Sys. (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pages 386–400, Grenoble, France, October 2011. Springer-Verlag. doi: 10.1007/978-3-642-24550-3_29. URL http://www.springerlink.com/content/3rg39l2287330370/.

[4] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, (104):67–88, June 2011. URL http://www.eatcs.org/images/bulletin/beatcs104.pdf.