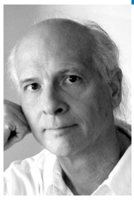
RIK FARROW

musings



Rik is the Editor of *;login:*.

rik@usenix.org

WHERE WAS I? OH YEAH, IN SAN DIEGO again, but this time for OSDI. OSDI and SOSP are the two big operating system conferences in the US, and I've particularly enjoyed being able to attend OSDI. I sat in the second row during all sessions, listening intently. In this issue, I've worked with the authors for two OSDI papers to share some of what I learned, and I asked researchers at the University of Rochester to write a survey article about transactional memory (TM).

You may not have heard of TM (unless you are old enough to remember Transcendental Meditation), but TM may become very important in both hardware and programming languages in the near future. As Dave Patterson said during his keynote at USENIX Annual Tech in 2008, multicore is here. Multicore processors have not just arrived, they are the clear path forward in increasing processor performance, and TM looks like a good way to make parallel programming techniques accessible to people besides database and systems programmers.

Locks

Mutual exclusion (mutex) locks have been the technique of choice for protecting critical sections of code. You may recall past *;login:* articles about removing the "big lock" from Linux or FreeBSD kernels. The big lock refers to having one lock that insures that only one thread can be running within the locked code at a time. Having one big lock is inefficient, as it blocks parallel execution of key kernel code.

Over time, system programmers refined locking by replacing one big lock with fine-grained locks—locks that protect much smaller code segments. Programmers have to work carefully, because having multiple locks can result in code that deadlocks when one locked section requires access to a mutex for another already locked resource, which in turn requires access to the first locked section.

TM replaces locking with transactions, where the results of an operation are either committed atomically (all at once) or aborted. Within a processor's ABI there are precious few atomic operations, as these play havoc with instruction pipelines. These operations are useful for implementing mutexes, but not for handling transactions that will span the much larger blocks of code found in critical sections of locked code.

In Shriraman et al. you will learn of the various techniques in hardware, in software, and in mixed approaches to support TM. Hardware approaches are faster but inflexible and limited in scope. Software approaches are painfully slow, as they must emulate hardware features, and that also adds considerably to the amount of memory involved in a transaction.

As I was reading this article, I found myself wanting to reread Hennesey and Patterson [1] about caches and cache coherence. If you don't have access to this book, Wikipedia has a very decent entry on caches [2]. Many TM approaches rely on tags added to caches for their operation, and the tags themselves are related to cache coherency.

Recall that only registers can access data within a single processor clock cycle. Level 1 (L1) caches provide more memory than registers, but accessing the data requires multiple clock cycles. As the caches become larger (L2 and L3 caches), the number of clock cycles increases because of the hardware involved in determining whether a particular cache contains valid data for the desired memory address.

In single-threaded programs and programs that do not explicitly share memory, coherency issues do not arise. Only one thread has access to each memory location. But in multi-threaded programs and programs that share memory, the caches associated with a core, such as L1 cache, will contain data that needs to be consistent with the data found in the rest of the memory system. Cache coherency systems handle this by tagging each cache block with status, such as *exclusive*, *shared*, *modified*, and *invalid*. Processor-level hardware then manages coherency between the different caches by updating an invalided cache block when it is accessed, for example.

Shriraman et al.'s favored solution involves extending coherency mechanisms to support flexible TM. I like this article, as it is a thorough survey of the approaches to TM, as well as a clear statement of the issues, such as how TM can be an easier mechanism for programmers to use that avoids deadlock and approaches the performance of fine-grained locks.

Although parallel programming is largely the domain of systems, databases, and some gaming programmers, the wider use of multicore processors suggests that more programmers who require high performance will be looking to add parallelism to their code. TM appears to be a workable approach to writing efficient and easy-to-debug parallel code.

Memory and Syscalls

The next two articles don't go as deeply into the use of processor features. Gupta et al. consider the use of sharing portions of pages of memory as well as compressing memory. VMware ESX can share identical pages of memory, something that occurs as much as 40% of the time when homogeneous guests are running within VMs. By extending sharing to partial pages and by compressing rarely used pages, Difference Engine can save much more memory, allowing more VMs to run on the same system with an increase in throughput.

Xax, described in "Leveraging Legacy Code for Web Browsers," relies on the system call interface for isolating a process. The system call interface is the gateway linking kernel services such as file system access, allocating memory, and communications with the network. Only the operating system has access to these hardware-mediated services, so it is possible to isolate a process effectively by interposing on system calls. During the paper presentation, I found myself wondering about this, but further reflection and a few words with the paper's presenter, Jon Howell, helped me recall that the system call mechanism is inviolable because of hardware features—not just the trap instruction, but also memory management.

Dave Beazley gets into the nitty-gritty of Python 3. I had heard Guido van Rossum talk about this new version of Python back in the summer of 2007, and even then he was talking about how older Python programs will not run unmodified under the new version. Dave explains, with examples, some of the reasons for the departure from backward compatibility while also showing exactly what pitfalls await those who venture unprepared into the new version.

Rudi van Drunen begins a series of articles on hardware, starting with the basics of electricity. If you find yourself wondering just how much voltage will drop along a run of 12-gauge wire or what exactly is meant by three-phase power, you will want to read this article. A must-read for anyone designing or overseeing machine rooms or even just racks of systems.

To go back to the beginning of this issue, immediately following these Musings, Mark Burgess expresses his misgivings (putting it mildly) about the "new" rage, cloud computing. Perhaps I should be writing "a new buzzword," as the cloud really isn't all the new, nor particularly shiny white, either.

We also have the usual array of columns, and I am not going to attempt to introduce them this issue. There are many more book reviews than usual this time around, including reviews of several programming books.

Finally, we have the reports on OSDI and some co-located workshops.

When it is time to write this column, I often go back and read past columns to get myself into the "write" mood (pun intended). I noticed how often I have written about operating systems and security (or lack thereof), wondering what a secure yet usable operating system might look like. As you will have noticed, we still don't have secure systems, and that goal appears as elusive as ever. But we do have steps that may lead us to more flexible systems that will include some steps toward better security, such as the isolation mechanism seen in Xax, as well as clever hacks, such as Difference Engine and forward-thinking designs, such as FlexTM. I find that I like computer systems research as much as ever, and I am proud to be part of the community that does this work.

REFERENCES

[1] J.L. Hennesey and D.A. Patterson, *Computer Architecture*, Fourth Edition (San Francisco: Morgan Kaufman, 2006), Section 4.2.

[2] http://en.wikipedia.org/wiki/CPU_cache.