Reed Electronics Group

MICROPROCESSOR REPORT

www.MPRonline.com

THE INSIDER'S GUIDE TO MICROPROCESSOR HARDWARE

THE EDITORIAL VIEW

The Dread of Threads

By Tom R. Halfhill {4/30/07-02}

Recently I discovered a bug in a Java applet I wrote more than 10 years ago. Luckily, the bug was cosmetic and didn't affect the applet's calculations. But I was taken aback that an embarrassing error could lurk for so long in a program that's the most popular attraction on my personal website. Since the mid-1990s, thousands of people have used the applet without reporting the problem, which escaped my attention despite yearly updates.

All programmers know this fear: the lurking bug that evades testing, only to surface and bare its teeth much later. At least I'm in good company. Around the same time I found and squashed my little bug, a much worse programming error disabled an entire flight of brand-new F-22 Raptors of the U.S. Air Force. On the way to Japan for their first overseas deployment, the $120 million jet fighters suffered major failures of their navigation and communications systems after crossing the international date line. When the high-tech avionics refused to reboot, the crippled planes had to follow their air tankers back to Hawaii, humbly observing visual flight rules the whole way. Later, engineers determined that the F-22's software couldn't handle the sudden change in compass heading from 180 degrees west to 180 degrees east. Whoops!

Any programmer can tell similar tales. Bugs happen. However, the trend toward multicore processors is leading the computer industry into uncharted territory. There might be entire minefields of hidden bugs we haven't considered before. Two papers I've read on this subject are disturbing, especially because they warn that we have few alternatives.

Hidden Holes in Threadbare Software

One paper is "The Problem With Threads," by Dr. Edward A. Lee, chairman of electrical engineering at the University of California at Berkeley. (For a web link, see our "For More Information" box.) Published last year, this detailed paper examines new problems that can appear when multithreaded programs run on systems with multiple processors or multicore processors. Programs thought to be rock-solid after rigorous testing and everyday use can suddenly fail. They behave just fine on single-processor or single-core systems, but suffer mysterious thread-contention deadlocks (sometimes called "threadlocks") when running on systems with multiple processors or cores. These failures can happen even if the programmers tested and debugged the software with thread-analysis tools.

Lee cites the example of the Ptolemy Project at UC-Berkeley. In early 2000, a team of programmers began writing the kernel of an interactive concurrency modeling environment. The team thoroughly tested and debugged this multithreaded Java program, using multiple levels of design reviews, code reviews, daily rebuilds, and 100%-coverage regression tests. The finished program ran great for four years. Until one day, when running on a dual-processor system, it deadlocked.

Lee concludes: "It is certainly true that our relatively rigorous software-engineering practice identified and fixed many concurrency bugs. But the fact that a problem as serious as a deadlock that locked up the system could go undetected for four years despite this practice is alarming. How many more such problems remain? How long do we need test before we can be sure to have discovered all such problems? Regrettably, I have to conclude that testing may never reveal all the problems in nontrivial multithreaded code."

That's a frightening conclusion, especially coming from a respected professor at a leading engineering school. Lee's paper goes on to discuss the shortcomings of today's programming languages and software-development tools and suggests some solutions. Nevertheless, his experience with the Ptolemy II kernel and his other research is disquieting. What does it foretell for millions of ordinary PCs and servers? Dual-core processors are now widespread, even in low-end systems, and quad-core chips are rapidly penetrating the market. Some of these multicore processors compound the potential problem by using hardware multithreading, too.

One hope is that concurrency bugs are less likely to survive for years in code written for PCs, which must endure the teeming hothouse environment of the world at large. Software companies commonly release new PC operating systems and application programs for public beta testing. With millions of people hammering on the code, surely the bugs will surface sooner rather than later, right? Maybe not. Identifying bugs is difficult, even with the automatic bug-reporting features built into Windows and other software. Failures may be caused by interactions among two or more multithreaded programs—a situation difficult to reproduce in the lab.

Are embedded systems in greater danger? Most aren't as easily patched as PCs. They often lack network connectivity or any other means of receiving bug fixes, and their firmware may be locked in ROM. Such is the case with millions of DVD players, VCRs, and other products whose real-time clocks are no longer in sync with the daylight-savings time changes in the U.S. (That problem was caused by Congress, not by multithreading, but it illustrates the difficulty of patching embedded systems in the field.) On the other hand, embedded developers were designing multicore processors and multiprocessor systems for years before PCs joined the bandwagon. Perhaps greater experience will afford some protection.

No Alternatives on the Horizon

In any case, there are no ready alternatives to parallel processing, according to another UC-Berkeley paper written by 11 experts on microprocessor architecture. (For a web link, see the "For More Information" box.) In "The Landscape of Parallel Computing Research: A View From Berkeley," they write that the only path toward significantly faster CPUs is chip multiprocessing. The hardships of controlling power consumption and heat dissipation rule out all other approaches for the foreseeable future. Programmers will simply have to adapt by writing concurrent code, regardless of any consequential problems with threads.

HPCwire, a publication for the high-performance computing community, interviewed two of the Berkeley paper's coauthors: David Patterson, of *Computer Architecture: A Quantitative Approach* fame, and John Shalf, a computer scientist at the National Energy Research Scientific Computing Center. It's a fascinating interview. (See the web link in our "For More Information" box.)

Patterson says programmers are only beginning to grasp that they must rewrite much of their code to get any benefit from the rising number of processor cores per chip. "The industry is already betting on multicore for future improvements in computing performance," Patterson told *HPCwire*. "To use a football analogy, the computing industry has already thrown a Hail Mary pass with the first round of multicore designs. The ball is in the air, but nobody is running yet [to catch it]. That's where things stand today."

The "View From Berkeley" paper distinguishes between today's modest dual- or quad-multicore chips and future "manycore" designs, which will be far more aggressive. They will integrate dozens, hundreds, or even thousands, of cores. Indeed, *Microprocessor Report* has been writing about such designs for years. They appeared first in the embedded market, where specialized workloads encourage the development of specialized processors. In recent years, *MPR* has described massively parallel or manycore designs from Ambric, ClearSpeed, Connex Technology, Elixent, Eutecus, Intel, MathStar, PicoChip, and Xelerated. Some devices from these companies cram more than 4,000 processor cores on a single chip. In addition, other companies have developed their own manycore ASICs and SoCs for internal use. A prominent example is Cisco Systems' Silicon Packet Processor, which integrates 188 Tensilica Xtensa processor cores.

Naturally, specialized workloads, such as packet routing, digital-video encoding, and scientific computing, lend themselves to parallel processing. In the terminology of the computer science community, such applications are "embarrassingly parallel." Patterson says there's no need to apologize for parallelism. He suggests describing such applications as "successfully parallel" or "brilliantly parallel."

Yet Another New Language?

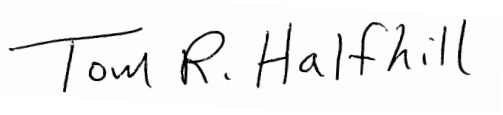
Putting terminology aside, the greater challenge is finding parallelism in commonplace software—the Holy Grail of computer science for decades. Patterson worries that some software developers are giving up without trying, making the excuse that they don't need parallelism. "We get questions along the lines of, 'What could you possibly run that needs 128 cores on a laptop?,'" Patterson told *HPCwire*. "This reminds me of the story of the patent examiner in 1870 who decided that everything of importance had been invented, so he quit his job to look for something permanent. Or that 640KB ought to be enough memory for PCs. We think the most exciting software has yet to be written, and it's going to be highly parallel."

If programmers cannot find any parallelism in a particular program, then that program (and others like it) will soon reach a performance plateau. Sure, they will run a little faster, as long as clock speeds keep inching upward. But the big performance gains taken for granted in past years are history. Only by exploiting parallelism—on a scale as massive as

the manycore designs themselves—can software continue delivering significantly higher performance.

As this reality sinks in, the software community has little choice. The industry must create new development tools and perhaps new programming languages as well. Of course, there are already several concurrent programming languages, and more are coming. But, to date, there's no widely accepted standard in the same way that ANSI C, C++, or Java is considered standard for writing sequential code or moderately parallel code. All the manycore-processor companies we mentioned have their own proprietary software tools, often with specially modified versions of sequential languages. The Defense Advanced Research Projects Agency (DARPA) is funding projects at Cray, Sun Microsystems, and other companies that could lead to a new concurrent language. IBM, Intel, and Microsoft are working in the same direction, as are numerous smaller companies and startups.

Perhaps, in time, we'll see an industry-standard programming language that makes it easier to express massive parallelism while avoiding nasty threadlocks. To become a true standard, such a language must work on manycore processors from numerous vendors. Because different manycore processors have radically different architectures, the language will almost certainly need a Java-like abstraction layer that insulates programmers from details of the hardware. If the Berkeley experts are right, this approach may be the only solution to problems with threads and parallelism. ◇



For More Information

"The Problem With Threads":
- *www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf*

"The Landscape of Parallel Computing Research: A View From Berkeley":
- *www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf*

"Confronting Parallelism: The View From Berkeley" (HPCwire):
- *www.hpcwire.com/hpc/1288079.html*