David A. Patterson

Computer Science Education in the 21st Century

To draw students to CS, we must first look to create a curriculum that reflects the exciting opportunities and challenges of IT today versus the 1970s. Future students and faculty would greatly benefit from a reinvigorated CS curriculum.

This letter is based on my position statement for a workshop on the preparation of IT graduates for 2010 and beyond [4].

Whereas in the past we created obstacles to reduce the number of CS majors, today we must recruit students to have the work force needed to meet the challenges and opportunities of information technology in this century. We should take advantage of the reduced pressures from the dip in enrollments to revamp our curriculum.

First, let's start with the state of the world in 2006 rather than 1976. Second, let's create courses that we would love to take if we were students, and that we would love to teach if given the chance. Such enthusiasm would be attractive and contagious.

Rather than wax on philosophically, I'll confine my remarks to four concrete suggestions: two technological upgrades and two examples of courses I would love to take and teach. All these suggestions leverage technology not available when the CS curriculum was first created.

Technological Upgrade #1 for 21st Century CS: *Use Tools and Libraries.*

There is a huge disconnect between the experience of most professors, who have never worked as professional programmers and often write software for a 30-year-old environment, and the way in which cutting-edge software is written today.

For example, although many professors use a more recent programming language like Java, surprisingly few have embraced modern programming environments like the Eclipse development platform and the JUnit testing framework. Moreover, a great many exciting programming projects today build on existing components such as .NET or NETBeans.

For many CS courses, a dramatic change would simply be if students first wrote a clear specification and then built software using modern tools and software components.

Technological Upgrade #2 for 21st Century CS: *Parallelism.*

In case you weren't paying attention, the era of doubling performance every 18 months ended in 2002, as the figure here documents [3]. Three factors have combined to grind uniprocessor performance improvement almost to a halt.

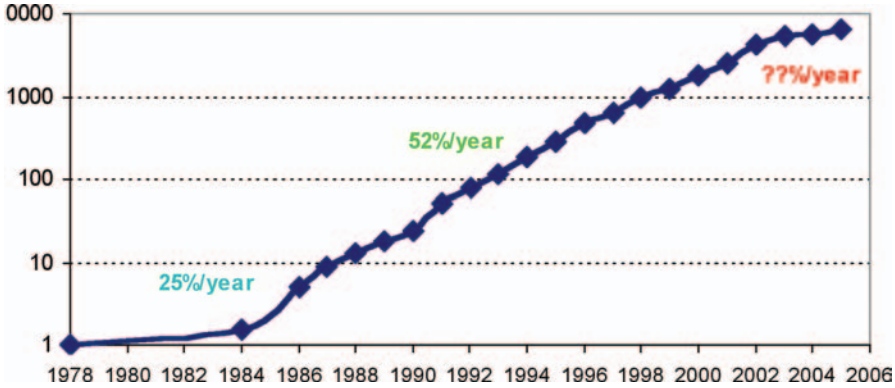
- The lack of additional power for a chip to dissipate,
- The lack of additional instruction-level parallelism to exploit, and
- The lack of improvement in memory latency.

This sea change is demonstrated by the change in direction of microprocessor companies away from increasing clock rates and uniprocessor performance. AMD, Intel, IBM, Sun Microsystems, among others

are either already shipping multiprocessors on a chip as their mainline product or will in near future. Apple just announced the next generation of Macintoshes, and all will include two processors—even laptops!

The table on the next page shows the number of processors ("cores") on a chip and number of hardware-supported threads per processor on current and upcoming microprocessors.

Hardware companies will market the number of processors per chip rather than clock rate from now on, and the number of processors is predicted to double every two years. If the prediction holds, by the time an entering freshman class graduates, standard desktop PCs will include at least eight processors.



Growth in processor performance since 1978 relative to the VAX 11/ 780 as measured by the SPECint benchmarks (from SPECint89 to SPECint2000). Prior to the mid-1980s, processor performance growth was largely technology driven, and averaged about 25% per year. The increase in growth from 1986 to 2002 to about 52% was due to more advanced architectural and organizational ideas exploiting Moore's Law. It's less than 20% per year since 2002 [3].

In our past, some have prematurely claimed that parallel programming was the only option for the immediate future. The steep part of the graph belies that claim. The difference today is that no hardware company is planning to sell much faster sequential processors. If you can't buy them, parallel procrastination will be penalized instead of rewarded.

Given this sea change, how much of the curriculum—and what fraction of the CS faculty—is oblivious to concurrency? How many algorithms, data structures, languages, compilers, debuggers, operating systems, books, and lectures must change to match the transformation in underlying technology if they are to remain relevant in the 21st century?

Educating faculty so they can train students properly is not a trivial bootstrapping problem. At research universities, if we can ensure that all future research assumes concurrent computing, the skills gained there hopefully can trickle down quickly to the classroom.

Course I Would Love to Take #1: *Join the Open Source Movement.*

Most schools teach "write programs from blank sheet of paper" programming, of which there is very little real-world bearing. A different approach is to leverage high-quality examples of the open source movement. For example, a database class might be based on PostgreSQL and an operating system class might use BSD Unix. (Well-documented and well-crafted code trumps popular for pedagogic reasons.) The high-level idea is taking advantage of a technology that has been created for other reasons that can make the classroom a much more exciting and realistic place.

If this works, it would be inspiring for students working on real production software. Even companies that don't use open source software may benefit from students who can do more than just write programs from scratch. In addition, it could be a differentiator on a college campus. Do civil engineering students get to contribute to the construction of a real bridge in the classroom? Do history courses allow students to help write a book? The recruiting pitch is to join CS and learn in part by contributing immediately to the real world.

To help learn a large system, writing documentation for portions of open source code could be an assignment. Documentation is important yet rare in the classroom and the open source movement. It's likely that open source developers would welcome good documentation, given its absence. Students would be inspired that their coursework could be

Let's create courses that we would love to take if we were students, and that we would love to teach if given the chance. Such enthusiasm would be attractive and contagious.

used in real systems, as well as bear the responsibility of that opportunity.

Another assignment is to have teams of students help debug this large system, in part by using the documentation that other students have written. While they might start with a local version of the code that includes bugs inserted by a teaching assistant, the exciting challenge would try to take on real open bugs. They could post the source of the problem to the open source group in addition to their proposed fixes. Since you won't run out of new bugs to assign, instructors don't have to worry about students copying their solutions from the last term.

Manufacturer/Year	AMD/2005	Intel/2006	IBM/2004	Sun/2005
Processors/chip	2	2	2	8
Threads/Processor	1	2	2	4
Threads/Chip	2	4	4	32

The number of processors on a chip and number of hardware-supported threads per processor on current and future microprocessors.

A third assignment might be to propose a new feature for open source software. Open source projects have a list of improvements they are considering; why not have teams of students propose how one might be done? If time permits, they could code the upgrade and benchmark the results. This assignment has been in database courses for a while at Berkeley and CMU [1].

Presumably lectures and reading assignments would supplement the projects covering aspects of the programming: good coding styles, good approaches to testing, the software life cycle, and so on. Now students could look at real code to form their opinion of these ideas based on hands-on experience. If the approach works, I'm sure new textbooks would document the good open source examples, which would increase the plausibility of this approach.

The challenge is preventing the large code base from overwhelming the students, interfering with learning the conceptual material of the course. If we can meet this challenge, this is a course I'd love to take.

Course I Would Love to Take #2: *Build your own Supercomputer.*

Like open source software, Field Programmable Gate Arrays (FPGAs) is another technology that makes the classroom a much more exciting and realistic place. FPGAs were created to collect the miscellaneous pieces of logic on a board into a single chip and to give the hardware engineer flexibility in making changes late in the design. They are programmable hardware, as the logic blocks are just memory that can be programmed to perform any logical function, and the rest of the chip is interconnected to join these programmable logic blocks in interesting ways. Since they are so simple to manufacture, they still are on Moore's Law, doubling hardware resources every 18 months. FPGAs currently represent a billion-dollar-a-year industry.

The computer-aided design tools for FPGAs are much like those of a real chip: logic synthesis, placement, routing, and so on. Hence, many universities already use FPGAs in their logic design courses because FPGAs have many of the aspects of real hardware design without the time delays and cost of fabricating chips. Moreover, FPGAs have credibility; if a design works well in an FPGA, it will likely work well in a real chip.

For a few hundred dollars, students can attach a board to their PCs and get the CAD software to build their own computer. Although the processors might only run at 200MHz, that is still plenty fast enough to run operating systems and real programs. Since hardware generally doesn't work until most bugs are fixed, there are few more exciting events

President's Letter

A group of us at a half-dozen universities [have joined] forces around a common hardware design that connects 40 FPGAs together. We plan to develop a 1,000-way multiprocessor based on standard instruction sets that can run standard software stacks. We would then share that "gateware" and software with the research community.

than to see your computer run real programs successfully for the first time.

In addition to FPGAs, we would leverage the small open source movement for hardware (see opencores.org). They offer big hardware blocks that work, like processors, Ethernet controllers, and so on that can be used for either real chips or FPGAs. Thus, the processor is the lowest-level building block, taking over from the transistor or nand gate.

The growth of FPGA resources plus the open source hardware movement creates the new opportunity. We can place 25 pipelined-processors inside a single large FPGA today, and the number of processors should double every 18 months. By starting with working processors as the building blocks, students can tackle interesting issues facing the field today, such as how to design computer systems to make it easier to write parallel programs or memory architectures for high-performance garbage collection. I'd love to design a supercomputer in such a course.

This technological opportunity has inspired a group of us at a half-dozen universities to join forces around a common hardware design that connects 40 FPGAs together. We plan to develop a 1,000-way multiprocessor based on standard instruction sets that can run standard software stacks. We would then share that "gateware" and software with the research community. We believe it will be an attractive platform for ramping up hardware and software research in multiple processors, and that it can be economically replicated so that many departments could have one. This vision gave the project its name: Research Accelerator for Multiple Processors, or RAMP [2]. When completed, it will enable students to explore and share even grander designs than a single FPGA. (To learn more, see ramp.eecs.berkeley.edu/.)

Conclusion

Our computer science curriculum was developed at a time when it was the first introduction students had to computing and the software we use everyday had not yet been written. There have been extraordinary developments since then, and most students arrive on campus today with years of computer experience. While one challenge to our curriculum is to catch up to the technology of the 21st century, another is we haven't leveraged the better background of students. A third challenge is for CS faculty to learn new ways.

Let's learn and try creating CS courses that we'd love to take and teach, and that will capture the exciting opportunities and challenges of our field and of our students. C

References

1. Anastassia A. and Hellerstein, J.M. Exposing undergraduate students to database system internals. *ACM SIGMOD Record 32,* 3 (Sept. 2003), 18–20.
2. Arvind, K.A., Chiou, D., Hoe, J.C. Kozyrakis, C., Lu, S-L, Oskin, M., Patterson, D., Rabaey, J., and Wawrzynek, J. *RAMP: Research Accelerator for Multiple Processors—A Community Vision for a Shared Experimental Parallel HW/SW Platform.* Technical Report UCB//CSD-05-1412, University of California, Berkeley, Sept. 2005.
3. Hennessy, J. and Patterson, D. *Computer Architecture: A Quantitative Approach, 4th Edition* (to appear Oct. 2006).
4. NSF Workshop. Interative Computing Education and Research—Preparing Graduates for 2010 and Beyond. (Jan. 27–28, Palo Alto, CA); www.evergreen.edu/icer.

These ideas came from conversations with many friends and colleagues, especially Nina Bhatti of Hewlett-Packard Labs, James Gosling of Sun Microsystems, Joe Hellerstein of UC Berkeley, John Hennessy of Stanford, Jim Larus of Microsoft Research, and Kathy Yelick of UC Berkeley.

David A. Patterson (pattrsn@cs.berkeley.edu) is president of ACM and the Pardee Professor of Computer Science at the University of California at Berkeley.