John Hennessy
Stanford University

The Future of Systems Research

After 20 years in academia and the Silicon Valley, the new Provost of Stanford University calls for a shift in focus for systems research. Performance—long the centerpiece—needs to share the spotlight with availability, maintainability, and other qualities.

It's time to be more forceful in thinking about some of the problems that we ought to be working on in the computer systems research communities. The performance increases over the past 15 years have been truly amazing, but it will be hard to continue these trends by sticking to the basically evolutionary path we are on.

We need to rethink the types of problems we should be addressing in terms of how we exploit parallelism within processors and how we build memory hierarchies. We need to move toward a more integrated approach that doesn't treat hardware and software as separate entities. And we need to look for new focuses for research.

WHERE WE'VE COME FROM

To look at performance trends, I plotted processor performance for the past 10 years based on the SPEC benchmarks. Although the performance increases are generally amazing, what's really impressive is the integer performance. We might have expected huge performance gains on the floating-point side—there is lots of parallelism in floating-point calculations—but the fact that we've managed to grow performance this dramatically on the integer side is pretty astounding.

Computer architects are largely indebted to integrated circuit technology for these gains because it provided the fuel for this revolution by shrinking densities and giving us more transistors. CMOS technology has followed Moore's law for more than 20 years.

The real magic that computer architects have managed is not only in using those faster transistors—which of course gives you better clock rate—but also in making good use of the exploding number of transistors. Each linear shrink gives you both a *linear* improvement in clock rate and a *quadratic* increase in the number of transistors.

So the architecture community's job has been to turn this exploding transistor count into performance.

Premiere Computer Architecture Conference

These ideas were first presented at the 26th Annual International Symposium on Computer Architecture. This joint IEEE Computer Society/ACM conference is the leading research conference in computer architecture, often offering the first glimpses of research that will influence commercial computer designs in subsequent years.

For example, this year's conference presented a session on value prediction, a technique under active consideration by several microprocessor design teams.

In 2000, ISCA will meet in Vancouver, BC; see http://www.cs.rochester.edu/~ISCA2k for details.

All of the ideas developed over the past 15 to 30 years—pipelining, multiple-instruction issue, and various caching methods—have been about using those extra transistors to produce performance.

When you think about what today's machines do—they look at the instruction stream dynamically, find the parallelism on the fly, execute instructions out of order, and speculate on branch outcomes—it's amazing that they work.

Similarly, it's amazing that caches and memory hierarchies are hiding the latency between processors and DRAM(dynamic random access memory). Today's processors run 30 to 50 times faster than DRAM.

But in another way, the ideas built into these machines are less astounding: They were all basically around (conceptually) 15, 20, even 25 years ago. In some sense, we've been following an evolutionary path in computer architecture for the past 15 to 20 years. It has taken a lot of research and innovation to flesh out these concepts, but thoughtful research had laid the groundwork much earlier.

Although this description is from a hardware perspective, the key to delivering increased performance has been accompanying innovations on the compiler side. So the interplay between hardware and software is also important.

WHERE WE ARE TODAY

Today the wide variety of approaches to processor architecture falls into two broad categories:

- software techniques that focus on using compiler technology to find instruction-level parallelism and speculation opportunities and perform alias pointer analysis; and
- techniques that focus on performing the same functions in hardware.

The software techniques are less complex and can look further down an instruction stream. Software techniques, however, are at a disadvantage because they don't usually have information about the executing program that is as accurate as that available to a hardware technique. That is, hardware makes decisions based on the program as it actually runs; software can only guess about what will happen at runtime.

The primary advantage of hardware techniques is that they yield more stable performance across a range of programs and don't require as complex a compilation process. But they do so at the cost of added complexity.

What puzzles me is that it's not clear that either the software or hardware techniques are obviously best. The answer may be a blend, but, as of now, we don't understand how to blend the ideas from these approaches together.

All these approaches seek to improve execution time through exploitation of parallelism. One way to increase the amount of parallelism we exploit is to use more explicit approaches to exploiting parallel processing. Unfortunately, making programs run well on today's parallel processors involves a tremendous amount of effort—way too much. We need a more evolutionary approach to these problems, the approach needs to be more integrated, and we need to think about hardware and software as a continuum, not as separate parts.

TOWARD AN INTEGRATED APPROACH

To support an integrated approach, we probably

New Era, New Expectations

It's really been amazing how much progress we've made in terms of the computer systems we build. What I love about working in computing is the dynamism between academic research and the industry over the past 20 years. We researchers have the best of both worlds: We get to work in this great field and have this tremendous impact on a giant industry. Well, sometimes it's a small impact, but the industry is now big enough so that it magnifies our contributions enormously.

We're on the threshold of a new era in which computing is ubiquitous, where everyone will use information services and everyday utilities. When everyone starts using these systems, guess what? They expect them to work and to be easy to use. They've been sold on this promise, and, believe me, they expect us to deliver. The general public thinks the Web and the Internet are their future.

Given these circumstances, the opportunity for disappointing them is giant. Computer people are pretty tough characters—we're used to networks that aren't up and computers that crash. But computing will move into the everyday lives of people who turn on the TV and expect it to work. Computing will become like an automobile, which people expect to work most of the time and break down very gradually. Consumers are used to products that work and to having that commitment from the products they buy.

Computer people are, I believe, obligated to think about this mismatch of expectations and reality, which will only get worse. By the time parts of industry wake up to these expectations, it may be too late. So industry should articulate the future challenges and technology gaps for the research community. In this way, when industry realizes how important these commitments to reliability are becoming, they will find that the research community has explored the technology. Such was the case when industry came to the research communities to find the technology to building faster processors and better memory hierarchies.

need to think about different forms of parallelism and changes in the ways we use memory hierarchies.

Parallelism

Programs need to be more explicit about parallelism, but the transition from the existing methods for writing programs to the next generation of methods has to be a gentle slope. The new methods will need to build on existing methods, such as instruction-, loop-, and task-level parallelism, but it must treat them as concepts on a continuum; today we treat them as completely different entities.

To accomplish this integration, we need innovative software and more research into algorithms. We need research into what happens to algorithms when you have caches on machines. We need algorithms that reflect the fact that caches are the only efficient way to access memory and that modern DRAMs do not really provide random access in the sense that all locations are equally costly to access.

Modern memory hierarchies are extremely complicated, so we will need abstractions that encourage programmers to focus on locality without requiring that they become experts in computer architecture.

Memory hierarchies

It's even time to question whether explicit memory hierarchies have a role. We're not going to completely abandon the idea of caches, but there could be something in addition to caches that would help.

Today we see that caches often consume 80 percent of the total transistors and 50 percent of the area. That's a good-news/bad-news situation. The good news is that caches are easy to design; designers can use up lots of transistors in a relatively straightforward way. For some programs, larger caches provide enhanced performance.

But we're getting to the point where we probably can't rely strictly on larger caches to continue to hide the CPU-DRAM gap. So this problem represents a complex set of trade-offs, the understanding of which could benefit from further research.

Along these lines, one topic we need to investigate is exploiting the interaction between parallelism and memory systems. There really are opportunities here; multithreading is one of them.

POST PC

These integrated approaches, like most computers systems research, focus on performance, and the computer industry has been supremely successful at delivering better performance. Although I don't advocate abandoning performance, it should share the spotlight with some other qualities, which are increasing in importance with emerging, new applications of computing.

The new applications come from the new era we are moving into; *Post PC* is the name David Patterson gives it. Maybe post-desktop is better, because desktop systems have been the focus for so many years, and we're changing our focus.

Information appliances are absolutely a big part of the future, and Mark Weiser's views on ubiquitous computing were right all along. I'm just sorry he passed away earlier this year and won't see the real explosion in information appliances, which he so insightfully predicted. These devices offer limited functionality, so they're cheap and easy to use—which is much more than I can say for most of our current computers—and they come in an explosive variety.

We're moving into a world where there are many more than just one or two computers per person. It's an Internet- and Web-centric environment in which services are the killer applications. In such a world, your net connection becomes more important than any one computing device.

I learned the importance of a net connection when the power connector on my laptop broke after I arrived at the Federated Computing Research Conference. My battery ran down, so the night before my plenary speech, I couldn't access my presentation! *But* because I had a net connection, I contacted my son, who knows something about computers. He e-mailed my presentation to Mark Horowitz, who put it on a flash card in his portable and then transferred it onto Dave O'Felt's machine—and that's how I had a presentation that day.

This was a lesson about the value of redundancy and the importance of communications. The first quality that's critical in information services is that they're reliable—we expect them to always be up, and they usually are. Thank God both my kid and the machine at Stanford were up!

The second critical quality is that information services scale with demand. For example, if you try to get to E*Trade on a day when the stock market's going crazy, you'll have a terrible time. If you try to get to the Olympics' Web site during the Olympics, you'll have a terrible time. And yet, those are the most important times to be able to get the service.

This shift to information appliances is also chang-

Same Old Rock

Here's my big picture of the progress and challenges in computer architecture: There's been a lot of discussion in the architecture community about limits to instruction-level parallelism, memory walls, and so on, but I'm not quite so pessimistic. Instead, I simply see these problems as hills we are trying to climb while pushing a big rock. And we've been pushing rocks up these hills for something like 15 to 20 years.

There isn't an immediate wall or a cliff that will set an absolute bound on performance, but the slope of the hill is certainly getting a lot steeper, and you need a lot more people pushing that big rock. Occasionally, we have processors that get so big they roll backwards down the hill, rolling over the design team. So we can keep on pushing our ever larger rocks up these hills, but it's going to get harder and harder, and it may not be the smartest use of all our design talent.

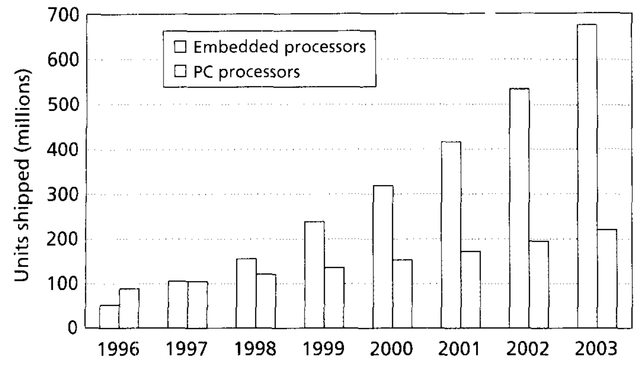

Figure 1. Marketing projections show shipments of embedded processors outpacing those of PC processors.

ing the microprocessor world. Sales of embedded microprocessors are outpacing those for microprocessors for desktop systems. Now, you might think all these embedded processors are going into automobiles, microwave ovens, or similar low-end control applications; but that's not true. Figure 1 shows only 32- and 64-bit microprocessors—they're not the ones in cars and can openers. Last year, the number of shipped embedded processors exceeded the number of those destined for desktop systems. In the next five years, these embedded processors are expected to outnumber PC processors by about three to one. The world is changing.

NEW FOCUS

Given this dramatic change, what should the computer industry and computer research focus on? I think the focus will be on different sorts of problems, and although still important, performance should be less of an emphasis. Instead, other qualities will become crucial:

- *Availability* of both of appliances and services is key. For example, during my laptop problems, there should have been another way to plug in the power adapter.
- *Maintainability* enhances availability by preventing failure and making recovery easy.
- *Scalability* of services also becomes an issue as user demand grows.

In all these new areas, *metrics* will be important. *Cost* will become more crucial, both for the appliance and per transaction at a server. And *performance* will remain important, but the benchmarks will be different: Communication performance will be key, and performance will be limited by available power since many appliances will not have wall plugs.

Availability

Webster's Revised Unabridged Dictionary defines availability as "the quality of having sufficient power, capability, or efficacy for the project." What I like about this definition is that it includes both a notion that the service is there and that it can get the job done. Availability has to speak to achieving a performance level, which allows you to operate, as well as just being up.

Our challenge is quite simple. Both software and hardware components are inherently unreliable, and the key question is, how do you build reliable systems from unreliable components? It's a problem at the level of individual machines, as well as servers, because individual machines simply crash too often. Imagine what would happen if your car broke down and crashed as often as your computer; automobile manufacturers would be out of business.

We really need to pay more attention to making the machines more reliable both on the hardware and software sides. For servers—if access to services on servers is the killer app—availability is *the* key metric. And we can't depend solely on the reliability of individual components to support availability. The interaction between software and hardware in large, integrated systems inevitably seems to produce failures. Research needs to address the difficulty in putting those components together and their unexpected interactions.

When I say this, fault-tolerance research immediately comes to mind. Fault tolerance is a hard problem and has been here forever; some say there won't be any progress in this area. I don't think we should give up on fault tolerance yet; perhaps there are other ways to approach it.

For example, a key insight is that systems often fail gradually. In real hardware, catastrophic failures happen, but they usually happen when you take the power adapter out of your machine, break it, and can't get power anymore—as I did with my laptop.

Many other failures are gentle. For example, a file system initially often displays small errors indicating problems with the file system data structure or underlying problems with the disk hardware. These small problems often occur many times before a catastrophic failure that leads to the entire disk becoming inaccessible. Many systems simply hide these small transient failures by error correction techniques or retries, thus maintaining availability but sometimes ignoring the evidence. We need research that defines more techniques that help ensure this gradual failure.

Encapsulation or confinement of faults is one key technique that can minimize the impact of failures. An example of the use of this technique comes from the Stanford project developing the Hive operating system, in which Mendel Rosenblum and his colleagues are using a large, shared-memory machine. A disadvantage of having a shared-memory infrastructure is that when you get a bug, your data can be scattered across the machine. So can we do anything to confine faults and aid recovery?

According to this research, the answer is "yes," and—this is another key insight—confinement and recovery can work much better with the right hardware support. Hardware support can cost almost nothing and play a vital role in bringing a processor back after a failure.

Another example of how reliability can be improved comes from the research on RAID (redundant arrays of inexpensive disks) systems. RAID is a particularly interesting concept, since it demonstrates how redundancy can improve availability as well as performance.

Maintainability

ENIAC had 19,000 tubes, and its initial uptime was terrible because tubes would burn out all the time. Its operators realized that the biggest cause of failures were filament failures that were exacerbated by turning the machine on and off. So they kept the machine powered up 24 hours a day. This action was step one in improved ENIAC reliability.

The next problem was the "infant mortality" of tubes, so the operators started doing burn-in on the tubes—checking them before installation—early (step two).

Step three was establishing a maintenance schedule that replaced tubes whether or not they were completely dead. Having the machine down while you search through 19,000 tubes is not particularly efficient. So you enhance availability by maintaining the machine—you replace stuff before it breaks.

As an example, Figure 2 shows some data from the UC Berkeley IStore project. It shows that you start to get errors on disks before they go solidly bad, so the disk is trying to tell you something is becoming a problem. This is a good time to swap the disk out before it fails completely, an action that enhances availability.

The other key part to maintainability is ease of upgrading for both hardware and software. I still upgrade my own PC because I want to understand the pain we put users through when they upgrade. Major software upgrades—such as replacing the operating system to support upgraded hardware—are a major pain in the neck. This is a problem we need to pay more attention to.

As another example, think about the servers at Yahoo. The system administrators there absolutely have to upgrade, add disks, change disks, and add hardware. And they have to be live *all the time* because an around-the-world user community is using the service 24 hours a day. Any maintenance they do has to be done online.

Some people assume information appliances will be so cheap that they don't need maintenance—we'll just throw them away. This is a wasteful notion and isn't environmentally conscious either. Such appliances ought to be both hardware and software upgradable. Many designers of such appliances are opting to have some mechanism to download software because they believe that at least software will be upgraded.

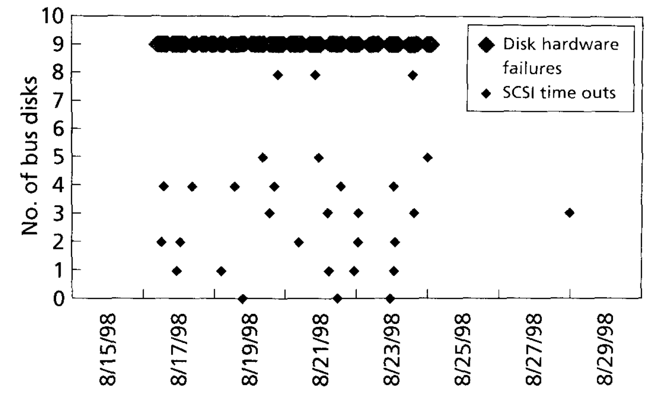

Figure 2. Data from the UC Berkeley IStore project shows several disk errors before disks go solidly bad.

Scalability

According to Greg Papadopoulos, chief technology officer at Sun, the amount of online storage on servers is currently scaling faster than Moore's law. This fact gives you some idea of the urgency with which the research community should be considering scalability issues.

System issues. How do we design a system that must scale to large sizes while providing full functionality and almost perfect availability? For such large systems, performance remains a problem. We don't know how to design really big systems to work as well as we think they should. In my mind, there are two underlying issues:

- *modularity* concerns how we construct the subsystems that will make up a large, scalable system, and
- *composability* concerns how we compose components in a way that maintains and makes the system's overall functionality reliable.

Scalability is critical for systems that are inherently large—systems like the Internet, Web servers, and modern microprocessors. We need to build them as subsystems and compose them. With the size and complexity of today's hardware and software systems, modular, composable design is the only approach that works. Yet, we understand very little about how to compose subsystems while maintaining correct functionality. Research into this area would be a well-spent investment.

Microprocessor design. As an example of the need to use more modular design, Table 1 shows three gen-

Table 1. Scalability of MIPS microprocessor design.

Processor	Year shipped	Transistor count (millions)	Design team (no. of people)	Time to design release (months)	Verification (percentage of total effort)
R2000	1985	0.10	20	15	15
R4000	1991	1.40	55	24	20
R10000	1996	6.80	> 100	36	> 35

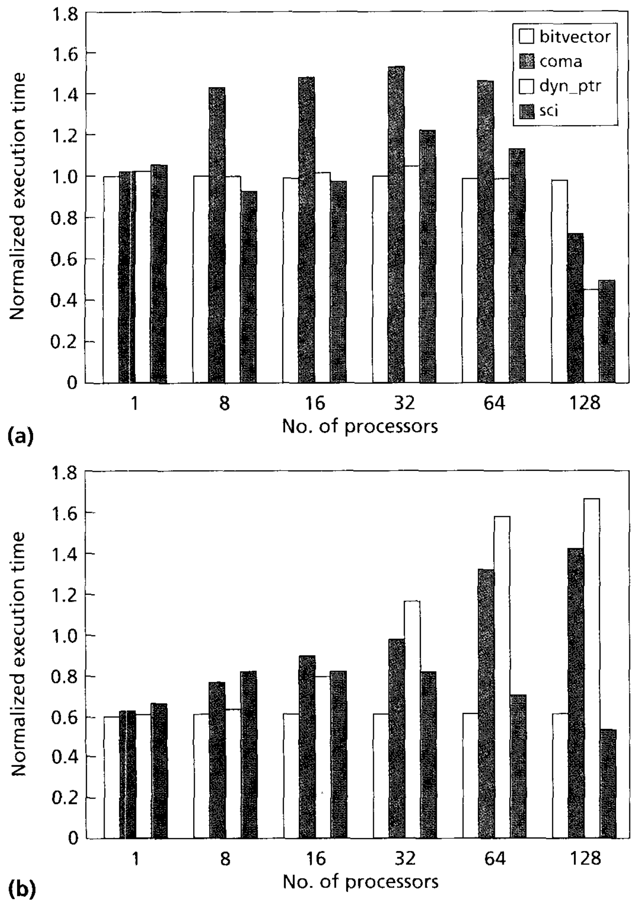

Figure 3. Execution time for each processor is normalized to the time for bitvector for the given number of processors. Shorter execution time is better. Under one 128-processor configuration (a) the algorithms dyn_ptr and sci have the best performance. Under another configuration (b) dyn_ptr has the worst performance.

erations of MIPS microprocessors in which transistor counts go from 0.1 million to 6.8 million, team sizes grow from 20 to more than 100 people, and design time increases from 15 months to three years. But the most important change is the fraction of the team that works on verification: from 15 to more than 35 percent.

The good news about multimillion-transistor chips is that they work. The bad news is that it takes more people to design them, and the number continues to climb. The Pentium design team is probably two to three times larger than the largest project in this table, and Pentium verification supposedly employed close to half the entire design effort.

Large-scale multiprocessors. As another example, consider our ability to understand performance in large-scale multiprocessors. The data from experiments on the Flash machine at Stanford, shown in Figure 3, quantifies how well coherence algorithms scale as processor count, problem size, and cache size change. To assess the four algorithms studied, researchers used two different versions of basically the same application, running them with different cache sizes and slightly different levels of compiler tuning. What's interesting is that the best coherence algorithm is radically different in these two situations. The configuration in Figure 3a shows that the coherence algorithms dyn_ptr and sci have the best performance at 128 processors, while in Figure 3b (data for a configuration using smaller caches and less optimized code), dyn_ptr has the worst performance!

This data says that we really don't understand these performance algorithms; we can't predict their behavior. The only way you can understand them is by doing a detailed simulation that takes a long time to run and includes all the underlying hardware details. And worst of all, we don't have a coherency scheme that does well under all these situations: from small to large processor counts, different levels of optimization, and differing cache sizes. It seems clear that we don't really understand performance scalability in these large systems.

Metrics and other key issues

A key issue in any sort of scientific research is the whole question of metrics. The difficulty, of course, is that scalability, availability, and maintainability are tougher to evaluate than performance.

Even performance evaluation was not such a smooth road as you might think. In the early days, we were using MIPS or megaflops and running benchmarks like Dhrystone and Whetstone before SPEC. So performance evaluation—this relatively simple task—was actually a mess, and we didn't understand the decomposition of performance into its contributing factors or analysis based on cycles per instruction, which are now second nature for us to use.

This lack of metrics was one reason I and the other developers of RISC had a hard time selling those ideas: We really couldn't explain in a scientific way what was going on. So performance wasn't so clear cut early on.

Evaluating cost is still difficult because volume has such a big effect on it, due to manufacturing costs as well as research and development amortization. In place of cost, we use some more basic measures like silicon area or power.

Power and silicon area have become more important constraints on designers in the Post PC era. In earlier times, microprocessor designers put little emphasis on cost and often designed chips with as large a silicon area as possible. With the explosion of low-cost devices, cost-constrained processors with more modest die area have become much more important.

Similarly, battery-based information appliances and systems without fans dramatically increase the importance of power, which in earlier times was given a lower priority. Power limitations and cost sensitivity are transforming processor design for information appliances, and cost and power will need to become important parts of our metrics.

Any metrics that we develop also need to continue paying attention to performance because there's clearly a trade-off between availability and performance, and maintainability and performance.

Twenty years have elapsed since the first Xerox Altos were delivered to Stanford in 1979. As a measure of our industry's progress since then, my current desktop machine indicates mixed improvement. My desktop is slightly more reliable than the Alto. It is not, though, much easier to use. Given that the processor is 1,000 times faster, you might expect applications to be significantly better. But I have the experience to tell you that Word, although certainly better, is not 1,000 times better than Bravo (the Alto's text processor) by a long shot.

This simple comparison says there are a lot of challenging problems left in systems research. We need to think long and hard about where to direct our efforts. ❖

John Hennessy became provost of Stanford University on 1 July. He is a pioneer in RISC architecture, having initiated the MIPS project, and also led the Stanford DASH (distributed architecture for shared memory) project. Hennessy has a PhD in computer science from the State University of New York, Stony Brook. Together with David Patterson, he is the coauthor of Computer Architecture: A Quantitative Approach *and* Computer Organization & Design: The Hardware/Software Interface *(Morgan Kaufmann, San Francisco, 1998). He is a member of the IEEE, the ACM, the National Academy of Engineering, and the American Academy of Arts and Sciences.*

Contact the author at Stanford Univ., Office of the Provost, Bldg. 10, Stanford, CA 940305-2061; jlh@vsop.stanford.edu.