The Case for Broader Computer Architecture Education

William J. Dally
Computer Systems Laboratory
Stanford University

Keynote address

Most introductory computer architecture courses – at both the graduate and undergraduate level – are primarily courses on CPU architecture. They tend to cover instruction set architecture, processor microarchitecture, and perhaps caches. While this is all useful content, our students would be better served by an introductory course that paints a more complete picture of computer architecture and one that better places computer architecture in context with the related fields of digital design and compilers.

Most computers today are not the desktop, laptop, or server boxes that we have historically associated with computing, but rather are embedded computing devices that control the engines in our cars, perform the modem functions in our cell phones, process the images in our cameras, TVs, and printers, or process packets traversing networks. The computing in these devices is typically performed by a combination of CPUs and special purpose hardware. The hard problems solved by architects in these systems do not involve the CPU, but rather the system-level organization of the device – the division of the problem over computing resources and the interconnect, memory organization, and I/O organization of the system. The CPU(s) is (are) typically not a major contributor to either cost (a RISC CPU with I and D caches is less than $1mm^2$ today – and most of that is cache) or performance (most of the heavy lifting is done by special-purpose devices).

Even for PCs and servers, where CPUs do have a large impact on cost and performance, the CPU is not where the architect spends the bulk of their time. System-level interconnect, memory bandwidth, and I/O bandwidth tend to dominate.

To better reflect the challenges faced by actual architects, our introductory courses should broaden their coverage of architecture by treating system level issues and considering the architecture of embedded computing devices – not just traditional "computers." This will better serve both the students who plan to specialize in architecture and those for whom the introductory course will be their only exposure. For the specialists, a system-oriented course will expose them to the type of architecture they are more likely to be practicing. Very few people architect CPUs. Many people architect systems using CPUs – and other computing devices. For the non-specialists a system-oriented course will give them a better overview of computer architecture as a field than a narrow treatment of CPUs.

To make room for the systems content in an introductory course, much of the detailed treatment of CPUs must be dropped from such a course. Such material rightly belongs in an advanced course on CPU architecture – much as detailed treatment of interconnection networks is deferred for an advanced course.

Many of the problems faced by architects cannot be solved entirely within the domain of architecture. Digital design and compiler technology are critical to solving many architectural problems. Yet many architects are not proficient in these areas.

Choosing between alternative organizations typically requires estimating the delay, power, and area of memories, interconnect, and logic. Performing such estimates is remarkably easy. However, many computer architects do not have this skill. Instead they rely on a separate "design group" to give them estimates, or use "canned" programs that perform estimates for a particular structure (e.g. Cacti for caches). Neither of these solutions really works because the architect does not develop an intuition about the alternatives that comes from understanding how they work at the next level down – and hence cannot use this intuition to arrive at the "right" alternative – which is almost never one of the initial alternatives. For example, I am constantly astounded by the large number of practicing architects that do not have a good feel for the speed/power/area tradeoffs of memories and hence believe that DRAMs are inherently slower than SRAMs. Also, even though power is a critical issue, few architects know the energy required by a particular operation (add, read an 8K RAM, lookup in a 32-entry CAM, clock a word into a pipeline register, transfer a word 10mm across a chip). This makes it nearly impossible for them to accurately estimate the power of proposed architectures.

A detailed understanding of digital design takes years of experience; however, simple models of area, power, and delay can be taught in a week or two. The use of such models drastically changes the nature of microarchitecture exploration. No longer is the task to develop the system that increases performance at any expense (and without regard for impact on clock rate) but the task becomes one of achieving specified performance – including the impact on clock rate - while staying within area and power constraints. Such simple models of delay, area, and power should ideally be included in an introductory course.

Many problems faced by an architect are better solved at compile time than run time – or even by JIT compilers that are invoked at run-time. Statically scheduling a sequence of instructions is far more efficient (and results in a better schedule – if all the information is available) than scheduling them dynamically. Similarly, specializing a piece of code given the data type or value of a variable using partial evaluation is far more efficient than "prediction" of various types. In a system with many computing "elements" (some CPUs, others specialized), a compiler plays a key role in "mapping" the problem to the elements. The best solution to most architecture problems is usually a combination of compile-time software and run-time hardware.

Many (not all) architects, unfortunately, view the compiler as a given. They see the architecture problem as one of running existing binaries, or compilers as someone else's problem. This may be appropriate for a CPU architect tasked with developing the next generation x86, where they really do have to run old binaries. However, it is not an attitude that we should cultivate in our students. Such an attitude is extremely limiting, ruling out entire classes of solutions to problems.

By including a small amount of back-end compiler technology in an introductory architecture course – preferably with an exercise that illustrates the advantages of solving a problem with a combination of hardware and software – we enable these students to view compiler technology as another tool in their arsenal and open up a range of solutions not accessible to those who view a compiler as a black box.

Our architecture students would benefit greatly from a broader introduction to computer architecture – one that focuses on system (rather than CPU) architecture and considers a broad class of computing systems (not just traditional "computers"). At the same time, we need to enable our students by giving them a broad range of tools to apply to architecture problems. Two key tools are back-end compiler technology and simple models of delay, area, and power.