1.6 The Sea Change: The Switch from Uniprocessors to Multiprocessors

"Up to now, most software has been like music written for a solo performer; with the current generation of chips we're getting a little experience with duets and quartets and other small ensembles; but scoring a work for large orchestra and chorus is a different kind of challenge."

Brian Hayes, *Computing in a Parallel Universe*, 2007.

The power limit has forced a dramatic change in the design of microprocessors. Figure 1.16 shows the improvement in response time of programs for desktop microprocessors over time. Since 2002, the rate has slowed from a factor of 1.5 per year to less than a factor of 1.2 per year.

Rather than continuing to decrease the response time of a single program running on the single processor, as of 2006 all desktop and server companies are shipping microprocessors with multiple processors per chip, where the benefit is often more on throughput than on response time. To reduce confusion between the words processor and microprocessor, companies refer to processors as "cores," and such microprocessors are generically called multicore microprocessors. Hence, a "quadcore" microprocessor is a chip that contains four processors or four cores.

Figure 1.17 shows the number of processors (cores), power, and clock rates of recent microprocessors. The official plan of record for many companies is to double the number of cores per microprocessor per semiconductor technology generation, which is about every two years (see Chapter 7).

In the past, programmers could rely on innovations in hardware, architecture, and compilers to double performance of their programs every 18 months without having to change a line of code. Today, for programmers to get significant improvement in response time, they need to rewrite their programs to take advantage of multiple processors. Moreover, to get the historic benefit of running faster on new microprocessors, programmers will have to continue to improve performance of their code as the number of cores doubles.

To reinforce how the software and hardware systems work hand in hand, we use a special section, *Hardware/Software Interface*, throughout the book, with the first one appearing below. These elements summarize important insights at this critical interface.

Hardware/Software Interface

Parallelism has always been critical to performance in computing, but it was often hidden. Chapter 4 will explain pipelining, an elegant technique that runs programs faster by overlapping the execution of instructions. This is one example of *instruction-level parallelism*, where the parallel nature of the hardware is abstracted away so the programmer and compiler can think of the hardware as executing instructions sequentially.

Forcing programmers to be aware of the parallel hardware and to explicitly rewrite their programs to be parallel had been the "third rail" of computer architecture, for companies in the past that depended on such a change in behavior failed (see **Section 7.14** on the CD). From this historical perspective, it's startling that the whole IT industry has bet its future that programmers will finally successfully switch to explicitly parallel programming.

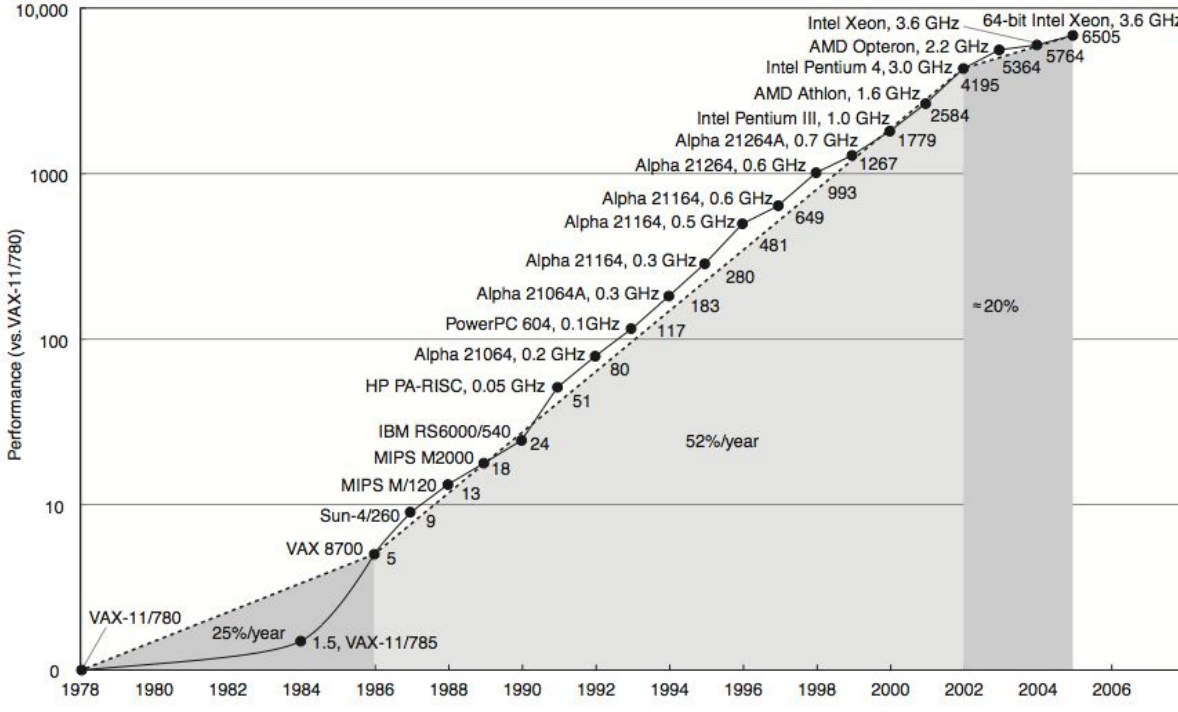

FIGURE 1.16 Growth in processor performance since the mid-1980s. This chart plots performance relative to the VAX 11/780 as measured by the SPECint benchmarks (see Section 1.8). Prior to the mid-1980s, processor performance growth was largely technology-driven and averaged about 25% per year. The increase in growth to about 52% since then is attributable to more advanced architectural and organizational ideas. By 2002, this growth led to a difference in performance of about a factor of seven. Performance for floating-point-oriented calculations has increased even faster. Since 2002, the limits of power, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance recently, to about 20% per year.

Product	AMD Opteron X4 (Barcelona)	Intel Nehalem	IBM Power 6	Sun Ultra SPARC T2 (Niagara 2)
Cores per chip	4	4	2	8
Clock rate	2.5 GHz	~ 2.5 GHz ?	4.7 GHz	1.4 GHz
Microprocessor power	120 W	~ 100 W ?	~ 100 W ?	94 W

FIGURE 1.17 Number of cores per chip, clock rate, and power for 2008 multicore microprocessors.

Why has it been so hard for programmers to write explicitly parallel programs? The first reason is that parallel programming is by definition performance programming, which increases the difficulty of programming. Not only does the program need to be correct, solve an important problem, and provide a useful interface to the people or other programs that invoke it, the program must also be fast. Otherwise, if you don't need performance, just write a sequential program.

The second reason is that fast for parallel hardware means that the programmer must divide an application so that each processor has roughly the same amount to

do at the same time, and that the overhead of scheduling and coordination doesn't fritter away the potential performance benefits of parallelism.

As an analogy, suppose the task was to write a newspaper story. Eight reporters working on the same story could potentially write a story eight times faster. To achieve this increased speed, one would need to break up the task so that each reporter had something to do at the same time. Thus, we must *schedule* the subtasks. If anything went wrong and just one reporter took longer than the seven others did, then the benefits of having eight writers would be diminished. Thus, we must *balance the load* evenly to get the desired speedup. Another danger would be if reporters had to spend a lot of time talking to each other to write their sections. You would also fall short if one part of the story, such as the conclusion, couldn't be written until all of the other parts were completed. Thus, care must be taken to *reduce communication and synchronization overhead*. For both this analogy and parallel programming, the challenges include scheduling, load balancing, time for synchronization, and overhead for communication between the parties. As you might guess, the challenge is stiffer with more reporters for a newspaper story and more processors for parallel programming.

To reflect this sea change in the industry, the next five chapters in this edition of the book each have a section on the implications of the parallel revolution to that chapter:

- *Chapter 2, Section 2.11: Parallelism and Instructions: Synchronization*. Usually independent parallel tasks need to coordinate at times, such as to say when they have completed their work. This chapter explains the instructions used by multicore processors to synchronize tasks.
- *Chapter 3, Section 3.6: Parallelism and Computer Arithmetic: Associativity*. Often parallel programmers start from a working sequential program. A natural question to learn if their parallel version works is, "does it get the same answer?" If not, a logical conclusion is that there are bugs in the new version. This logic assumes that computer arithmetic is associative: you get the same sum when adding a million numbers, no matter what the order. This chapter explains that while this logic holds for integers, it doesn't hold for floating-point numbers.
- *Chapter 4, Section 4.10: Parallelism and Advanced Instruction-Level Parallelism*. Given the difficulty of explicitly parallel programming, tremendous effort was invested in the 1990s in having the hardware and the compiler uncover implicit parallelism. This chapter describes some of these aggressive techniques, including fetching and executing multiple instructions simultaneously and guessing on the outcomes of decisions, and executing instructions speculatively.

- *Chapter 5, Section 5.8: Parallelism and Memory Hierarchies: Cache Coherence.* One way to lower the cost of communication is to have all processors use the same address space, so that any processor can read or write any data. Given that all processors today use caches to keep a temporary copy of the data in faster memory near the processor, it's easy to imagine that parallel programming would be even more difficult if the caches associated with each processor had inconsistent values of the shared data. This chapter describes the mechanisms that keep the data in all caches consistent.
- *Chapter 6, Section 6.9: Parallelism and I/O: Redundant Arrays of Inexpensive Disks.* If you ignore input and output in this parallel revolution, the unintended consequence of parallel programming may be to make your parallel program spend most of its time waiting for I/O. This chapter describes RAID, a technique to accelerate the performance of storage accesses. RAID points out another potential benefit of parallelism: by having many copies of resources, the system can continue to provide service despite a failure of one resource. Hence, RAID can improve both I/O performance and availability.

In addition to these sections, there is a full chapter on parallel processing. Chapter 7 goes into more detail on the challenges of parallel programming; presents the two contrasting approaches to communication of shared addressing and explicit message passing; describes a restricted model of parallelism that is easier to program; discusses the difficulty of benchmarking parallel processors; introduces a new simple performance model for multicore microprocessors and finally describes and evaluates four examples of multicore microprocessors using this model.

Starting with this edition of the book, Appendix A describes an increasingly popular hardware component that is included with desktop computers, the graphics processing unit (GPU). Invented to accelerate graphics, GPUs are becoming programming platforms in their own right. As you might expect, given these times, GPUs are highly parallel. Appendix A describes the NVIDIA GPU and highlights parts of its parallel programming environment.

I thought [computers] would be a universally applicable idea, like a book is. But I didn't think it would develop as fast as it did, because I didn't envision we'd be able to get as many parts on a chip as we finally got. The transistor came along unexpectedly. It all happened much faster than we expected.

J. Presper Eckert, coinventor of ENIAC, speaking in 1991

1.7 Real Stuff: Manufacturing and Benchmarking the AMD Opteron X4

Each chapter has a section entitled "Real Stuff" that ties the concepts in the book with a computer you may use every day. These sections cover the technology underlying modern computers. For this first "Real Stuff" section, we look at how integrated circuits are manufactured and how performance and power are measured, with the AMD Opteron X4 as the example.