CHAPTER 1
The Case for CMPs

The performance of microprocessors that power modern computers has continued to increase exponentially over the years, as is depicted in Fig. 1.1 for Intel processors, for two main reasons. First, the transistors that are the heart of the circuits in all processors and memory chips have simply become faster over time on a course described by Moore's law [1], and this directly affects the performance of processors built with those transistors. Moreover, actual processor performance has increased faster than Moore's law would predict [2], because processor designers have been able to harness the increasing number of transistors available on modern chips to extract more parallelism from software.

An interesting aspect of this continual quest for more parallelism is that it has been pursued in a way that has been virtually invisible to software programmers. Since they were first

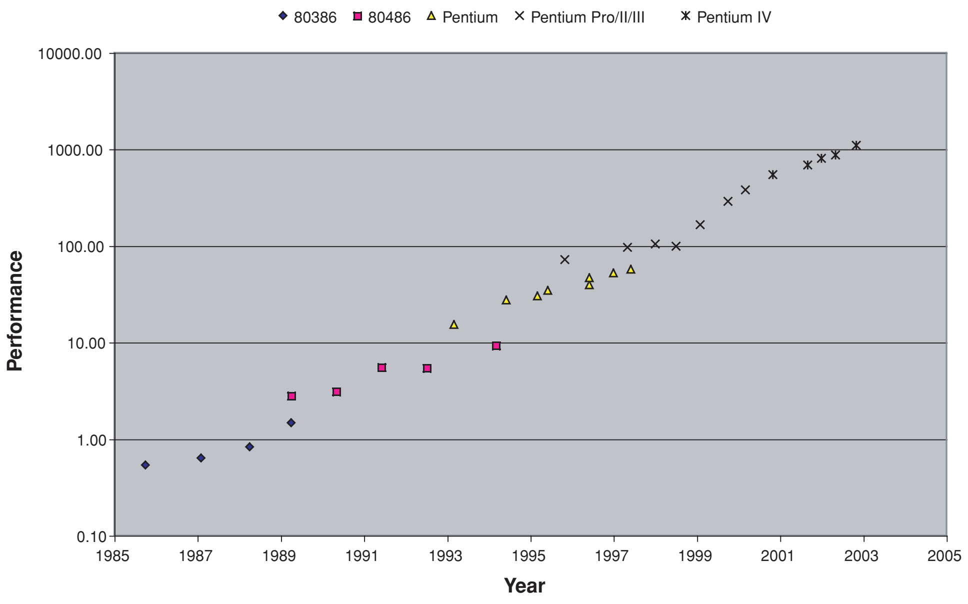

FIGURE 1.1: Intel processor performance over time, tracked by compiling published SPEC CPU figures from Intel and normalizing across varying suites (89, 92, 95, 2000).

invented in the 1970s, microprocessors have continued to implement the conventional Von Neumann computational model, with very few exceptions or modifications. To a programmer, each computer consists of a single processor executing a stream of sequential instructions and connected to a monolithic "memory" that holds all of the program's data. Because of the economic benefits of backward compatibility with earlier generations of processors are so strong, hardware designers have essentially been limited to enhancements that have maintained this abstraction for decades. On the memory side, this has resulted in processors with larger cache memories, to keep frequently accessed portions of the conceptual "memory" in small, fast memories that are physically closer to the processor, and large register files to hold more active data values in an extremely small, fast, and compiler-managed region of "memory." Within processors, this has resulted in a variety of modifications that are designed to perform one of two goals: increasing the number of instructions from the processor's instruction sequence that can be issued on every cycle, or increasing the clock frequency of the processor faster than Moore's law would normally allow. Pipelining of individual instruction execution into a sequence of stages has allowed designers to increase clock rates as instructions have been sliced into a larger number of increasingly small steps, which are designed to reduce the amount of logic that needs to switch during every clock cycle. Instructions that once took a few cycles to execute in the 1980s now often take 20 or more in today's leading-edge processors, allowing a nearly proportional increase in the possible clock rate. Meanwhile, superscalar processors were developed to execute multiple instructions from a single, conventional instruction stream on each cycle. These function by dynamically examining sets of instructions from the instruction stream to find ones capable of parallel execution on each cycle, and then executing them, often out-of-order with respect to the original sequence. This takes advantage of any parallelism that may exist among the numerous instructions that a processor executes, a concept known as *instruction-level parallelism* (ILP). Both pipelining and superscalar instruction issues have flourished because they allow instructions to execute more quickly while maintaining the key illusion for programmers that all instructions are actually being executed sequentially and in-order, instead of overlapped and out-of-order.

Unfortunately, it is becoming increasingly difficult for processor designers to continue using these techniques to enhance the speed of modern processors. Typical instruction streams have only a limited amount of usable parallelism among instructions [3], so superscalar processors that can issue more than about four instructions per cycle achieve very little additional benefit on most applications. Figure 1.2 shows how effective real Intel processors have been at extracting instruction parallelism over time. There is a flat region before instruction-level parallelism was pursued intensely, then a steep rise as parallelism was utilized usefully, and followed by a tapering off in recent years as the available parallelism has become fully exploited. Complicating matters further, building superscalar processor cores that can exploit more than a

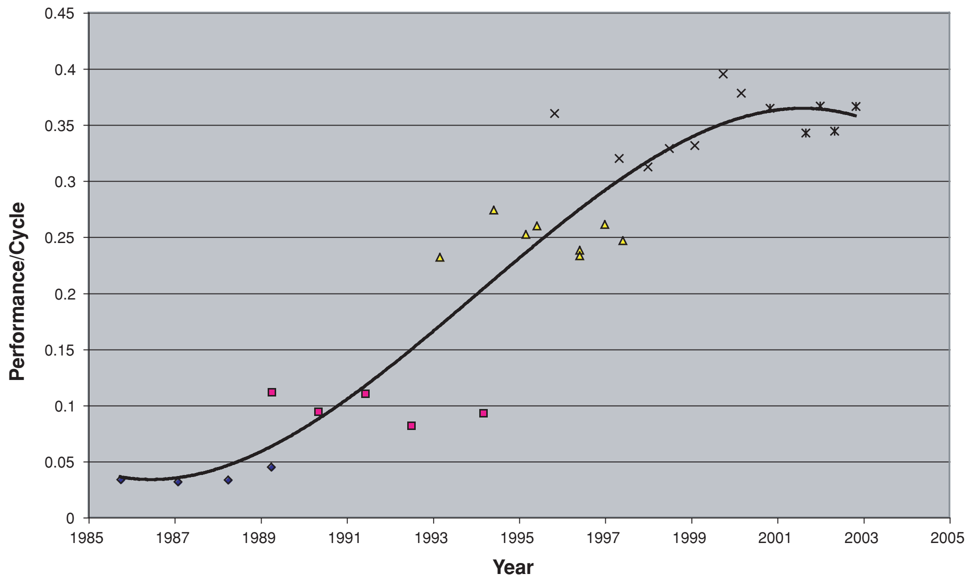

FIGURE 1.2: Intel processor normalized performance per cycle over time, calculated by combining intel's published (normalized) SPEC CPU figures and clock frequencies.

few instructions per cycle becomes very expensive, because the complexity of all the additional logic required to find parallel instructions dynamically is approximately proportional to the square of the number of instructions that can be issued simultaneously. Similarly, pipelining past about 10–20 stages is difficult because each pipeline stage becomes too short to perform even a basic logic operation, such as adding two integers together, and subdividing circuitry beyond this point is very complex. In addition, the circuitry overhead from adding additional pipeline registers and bypass path multiplexers to the existing logic combines with performance losses from events that cause pipeline state to be flushed, primarily branches, to overwhelm any potential performance gain from deeper pipelining after about 30 stages or so. Further advances in both superscalar issue and pipelining are also limited by the fact that they require ever-larger number of transistors to be integrated into the high-speed central logic within each processor core—so many, in fact, that few companies can afford to hire enough engineers to design and verify these processor cores in reasonable amounts of time. These trends slowed the advance in processor performance, but mostly forced smaller vendors to forsake the high-end processor business, as they could no longer afford to compete effectively.

Today, however, progress in processor core development has slowed dramatically because of a simple physical limit: *power*. As processors were pipelined and made increasingly superscalar over the course of the past two decades, typical high-end microprocessor power went from less than a watt to over 100 W. Even though each silicon process generation promised a reduction

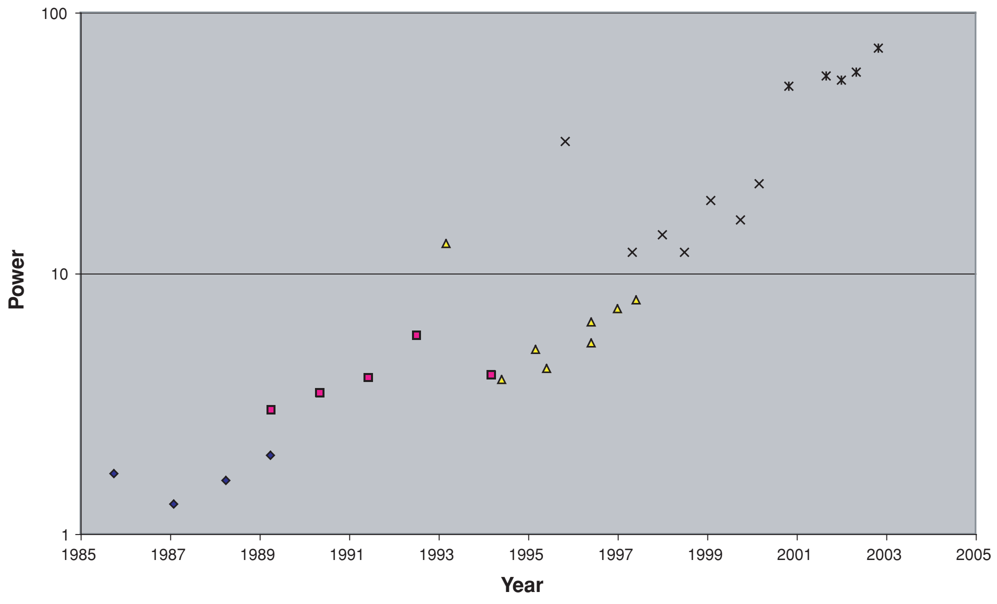

FIGURE 1.3: Intel processor power over time, calculated by combining published (normalized) SPEC and power numbers from Intel.

in power, as the ever-smaller transistors required less power to switch, this was only true in practice when existing designs were simply "shrunk" to use the new process technology. However, processor designers kept using more transistors in their cores to add pipelining and superscalar issue, and switching them at higher and higher frequencies, so the overall effect was that *exponentially more* power was required by each subsequent processor generation (as illustrated in Fig. 1.3). Unfortunately, cooling technology does not scale exponentially nearly as easily. As a result, processors went from needing no heat sinks in the 1980s, to moderate-sized heat sinks in the 1990s, to today's monstrous heat sinks, often with one or more dedicated fans to increase airflow over the processor. If these trends were to continue, the next generation of microprocessors would require very exotic cooling solutions, such as dedicated water cooling, that are economically impractical in all but the most expensive systems.

The combination of finite instruction parallelism suitable for superscalar issue, practical limits to pipelining, and a "power ceiling" set by practical cooling limitations limits future speed increases within conventional processor cores to the basic Moore's law improvement rate of the underlying transistors. While larger cache memories will continue to improve performance somewhat, by speeding access to the single "memory" in the conventional model, the simple fact is that without more radical changes in processor design, one can expect that microprocessor performance increases will slow dramatically in the future, unless processor designers find new ways to *effectively* utilize the increasing transistor budgets in high-end silicon chips to improve performance in ways that minimize both additional power usage and design complexity.

1.1 A NEW APPROACH: THE CHIP MULTIPROCESSOR (CMP)

These limits have combined to create a situation where ever-larger and faster uniprocessors are simply impossible to build. In response, processor manufacturers are now switching to a new microprocessor design paradigm: the chip multiprocessor, or CMP. While we generally use this term in this book, it is also synonymous with "multicore microprocessor," which is more commonly used by industry. (Some also use the more specific term "manycore microprocessor" to describe a CMP made up of a fairly large number of very simple cores, such as the CMPs that we discuss in more detail in Chapter 2, but this is less prevalent.) As the name implies, a chip multiprocessor is simply a group of uniprocessors integrated onto the same processor chip so that they may act as a team, filling the area that would have originally been filled with a single, large processor with several smaller "cores," instead.

CMPs require a more modest engineering effort for each generation of processors, since each member of a family of processors just requires stamping down a number of copies of the core processor and then making some modifications to relatively slow logic connecting the processors to tailor the bandwidth and latency of the interconnect with the demands of the processors—but does not necessarily require a complete redesign of the high-speed processor pipeline logic. Moreover, unlike with conventional multiprocessors with one processor core per chip package, the system board design typically only needs minor tweaks from generation to generation, since externally a CMP looks essentially the same from generation to generation, even as the number of cores within it increases. The only real difference is that the board will need to deal with higher memory and I/O bandwidth requirements as the CMPs scale, and slowly change to accommodate new I/O standards as they appear. Over several silicon process generations, the savings in engineering costs can be very significant, because it is relatively easy to simply stamp down a few more cores each time. Also, the same engineering effort can be amortized across a large family of related processors. Simply varying the numbers and clock frequencies of processors can allow essentially the same hardware to function at many different price and performance points.

Of course, since the separate processors on a CMP are visible to programmers as separate entities, we have replaced the conventional Von Neumann computational model for programmers with a new *parallel programming model*. With this kind of model, programmers must divide up their applications into semi-independent parts, or *threads*, that can operate simultaneously across the processors within a system, or their programs will not be able to take advantage of the processing power inherent in the CMP's design. Once threading has been performed, programs can take advantage of *thread-level parallelism* (TLP) by running the separate threads in parallel, in addition to exploiting ILP among individual instructions within each thread. Unfortunately, different types of applications written to target "conventional" Von Neumann uniprocessors respond to these efforts with varying degrees of success.

1.2 THE APPLICATION PARALLELISM LANDSCAPE

To better understand the potential of CMPs, we survey the parallelism in applications. Figure 1.4 shows a graph of the landscape of parallelism that exists in some typical applications. The *X*-axis shows the various conceptual levels of program parallelism, while the *Y*-axis shows the *granularity* of parallelism, which is the average size of each parallel block of machine instructions between communication and/or synchronization points. The graph shows that as the conceptual level of parallelism rises, the granularity of parallelism also tends to increase although there is a significant overlap in granularity between the different levels.

- *Instruction*: All applications possess some parallelism among individual instructions in the application. This level is not illustrated in the figure, since its granularity is simply single instructions. As was discussed previously, superscalar architectures can take advantage of this type of parallelism.
- *Basic Block*: Small groups of instructions terminated by a branch are known as basic blocks. Traditional architectures have not been able to exploit these usefully to extract any parallelism other than by using ILP extraction among instructions within these small blocks. Effective branch prediction has allowed ILP extraction to be applied

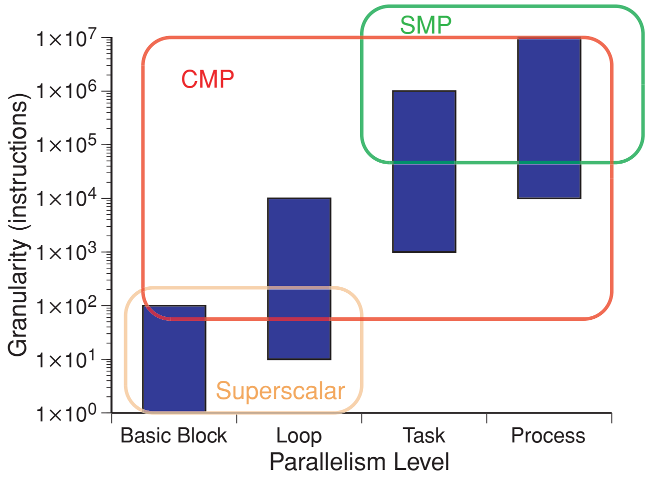

FIGURE 1.4: A summary of the various "ranges" of parallelism that different processor architectures may attempt to exploit.

across a few basic blocks at once, however, greatly increasing the potential for superscalar architectures to find potentially parallel instructions from several basic blocks simultaneously.

- *Loop Iterations*: Each iteration of a typical loop often works with independent data elements, and is therefore an independent chunk of parallel work. (This obviously does not apply to loops with highly dependent loop iterations, such as ones doing pointer-chasing.) On conventional systems, the only way to take advantage of this kind of parallelism is to have a superscalar processor with an instruction window large enough to find parallelism among the individual instructions in multiple loop iterations simultaneously, or a compiler smart enough to interleave instructions from different loop iterations together through an optimization known as *software pipelining*, since hardware cannot parallelize loops directly. Using software tools such as OpenMP, programmers have only had limited success extracting TLP at this level because the loops must be extremely parallel to be divisible into sufficiently large chunks of independent code.
- *Tasks*: Large, independent functions extracted from a single application are known as *tasks*. For example, word processors today often have background tasks to perform spell checking as you type, and web servers typically allocate each page request coming in from the network to its own task. Unlike the previous types of parallelism, only large-scale symmetric multiprocessor (SMP) architectures composed of multiple microprocessor chips have really been able to exploit this level of parallelism, by having programmers manually divide their code into threads that can explicitly exploit TLP using software mechanisms such as POSIX threads (pthreads), since the parallelism is at far too large a scale for superscalar processors to exploit at the ILP level.
- *Processes*: Beyond tasks are completely independent OS processes, all from different applications and each with their own separate virtual address space. Exploiting parallelism at this level is much like exploiting parallelism among tasks, except that the granularity is even larger.

The measure of application performance at the basic block and loop level is usually defined in terms of the *latency* of each task, while at the higher task or process levels performance is usually measured using the *throughput* across multiple tasks or applications, since usually programmers are more interested in the number of tasks completed per unit time than the amount of time allotted to each task.

The advent of CMPs changes the application parallelism landscape. Unlike conventional uniprocessors, multicore chips can use TLP, and can therefore also take advantage of threads to utilize parallelism from the traditional large-grain task and process level parallelism province of SMPs. In addition, due to the much lower communication latencies between processor cores and their ability to incorporate new features that take advantage of these short latencies, such as speculative thread mechanisms, CMPs can attack fine-grained parallelism of loops, tasks and even basic blocks.

1.3 A SIMPLE EXAMPLE: SUPERSCALAR VS. CMP

A good way to illustrate the inherent advantages and disadvantages of a CMP is by comparing performance results obtained for a superscalar uniprocessor with results from a roughly equivalent CMP. Of course, choosing a pair of "roughly equivalent" chips when the underlying architectures are so different can involve some subjective judgment. One way to define "equivalence" is to design the two different chips so that they are about the same size (in terms of silicon area occupied) in an identical process technology, have access to the same off-chip I/O resources, and run at the same clock speed. With models of these two architectures, we can then simulate code execution to see how the performance results differ across a variety of application classes.

To give one example, we can build a CMP made up of four small, simpler processors scaled so that they should occupy about the same amount of area as a single large, superscalar processor. Table 1.1 shows the key characteristics that these two "equivalent" architectures would have if they were actually built. The large superscalar microarchitecture (SS) is essentially a 6-way superscalar version of the MIPS R10,000 processor [4], a prominent processor design from the 1990s that contained features seen in virtually all other leading-edge, out-of-order issue processors designed since. This is fairly comparable to one of today's leading-edge, out-of-order superscalar processors. The chip multiprocessor microarchitecture (CMP), by contrast, is a 4-way CMP composed of four identical 2-way superscalar processors similar to early, in-order superscalar processors from around 1992, such as the original Intel Pentium and DEC Alpha. In both architectures, we model the integer and floating point functional unit result and repeat latencies to be the same as those in the real R10,000.

Since the late 1990s, leading-edge superscalar processors have generally been able to issue about 4–6 instructions per cycle, so our "generic" 6-way issue superscalar architecture is very representative of, or perhaps even slightly more aggressive than, today's most important desktop and server processors such as the Intel Core and AMD Opteron series processors. As the floorplan in Fig. 1.5 indicates, the logic necessary for out-of-order instruction issue and scheduling would dominate the area of the chip, due to the quadratic area impact of supporting very wide instruction issue. Altogether, the logic necessary to handle out-of-order instruction issue occupies about 30% of the die for a processor at this level of complexity. The on-chip memory hierarchy is similar to that used by almost all uniprocessor designs—a small,

TABLE 1.1: Key characteristics of approximately equal-area 6-way superscalar and 4 × 2-way CMP processor chips.

	6-WAY SS	4 x 2-WAY MP
♯ of CPUs	1	4
Degree superscalar	6	4 × 2
♯ of architectural registers	32int/32fp	4 × 32int/32fp
♯ of physical registers	160int/160fp	4 × 40int/40fp
♯ of integer functional units	3	4 × 1
♯ of floating pt. functional units	3	4 × 1
♯ of load/store ports	8 (one per bank)	4 × 1
BTB size	2048 entries	4 × 512 entries
Return stack size	32 entries	4 × 8 entries
Instruction issue queue size	128 entries	4 × 8 entries
I cache	32 KB, 2-way S. A.	4 × 8 KB, 2-way S. A.
D cache	32 KB, 2-way S. A.	4 × 8 KB, 2-way S. A.
L1 hit time	2 cycles	1 cycle
L1 cache interleaving	8 banks	N/A
Unified L2 cache	256 KB, 2-way S. A.	256 KB, 2-way S. A.
L2 hit time/L1 penalty	4 cycles	5 cycles
Memory latency/L2 penalty	50 cycles	50 cycles

fast level one (L1) cache backed up by a large on-chip level two (L2) cache. The wide issue width requires the L1 cache to support wide instruction fetches from the instruction cache and multiple loads from the data cache during each cycle (using eight independent banks, in this case). The additional overhead of the bank control logic and a crossbar required to arbitrate between the multiple requests sharing the 8 data cache banks adds a cycle to the latency of the L1 cache and increases its area/bit cost. In this example, backing up the 32 KB L1 caches is a

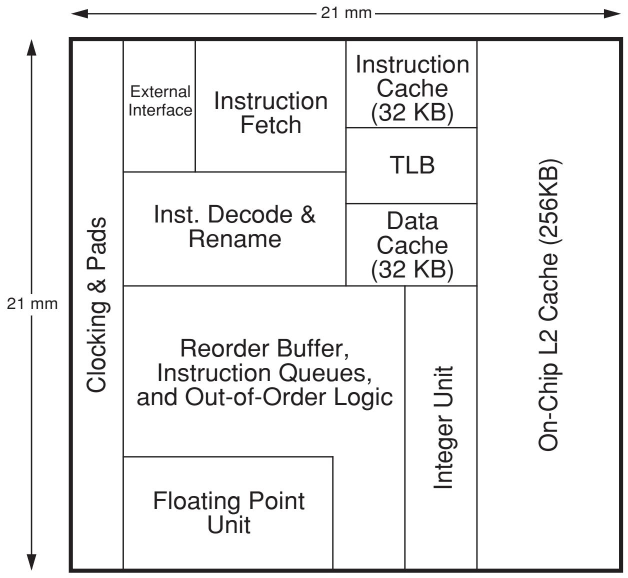

FIGURE 1.5: Floorplan for the 6-issue dynamic superscalar microprocessor.

unified 256 KB L2 cache that takes four cycles to access, which is smaller but faster than most L2 caches in typical processors from industry today.

In contrast, the CMP architecture is made up of four 2-way superscalar processors interconnected by a crossbar that allows the processors to share the L2 cache. On the die, the four processors are arranged in a grid with the L2 cache at one end, as shown in Fig. 1.6. The number of execution units actually increases in the CMP scenario because the 6-way processor included three units of each type, while the 4-way CMP must have four—one for each core. On the other hand, the issue logic becomes *dramatically* smaller, due to the decrease in the number of instruction buffer ports and the smaller number of entries in each instruction buffer. The scaling factors of these two units balance each other out, leaving the entire 4 × 2-way CMP very close to one-fourth of the size of the 6-way processor. More critically, the on-chip cache hierarchy of the multiprocessor is significantly different from the cache hierarchy of the 6-way superscalar processor. Each of the four cores has its own single-banked and single-ported instruction and data caches, and each cache is scaled down by a factor of 4, to 8 KB. Since each cache can only be accessed by a single processor's single load/store unit, no additional overhead is incurred to handle arbitration among independent memory access units. However, since the four processors now share a single L2 cache, that cache requires extra latency to allow time for interprocessor arbitration and crossbar delay.

These two microarchitectures were compared using nine small benchmarks that cover a wide range of possible application classes. Table 1.2 shows the applications: two SPEC95

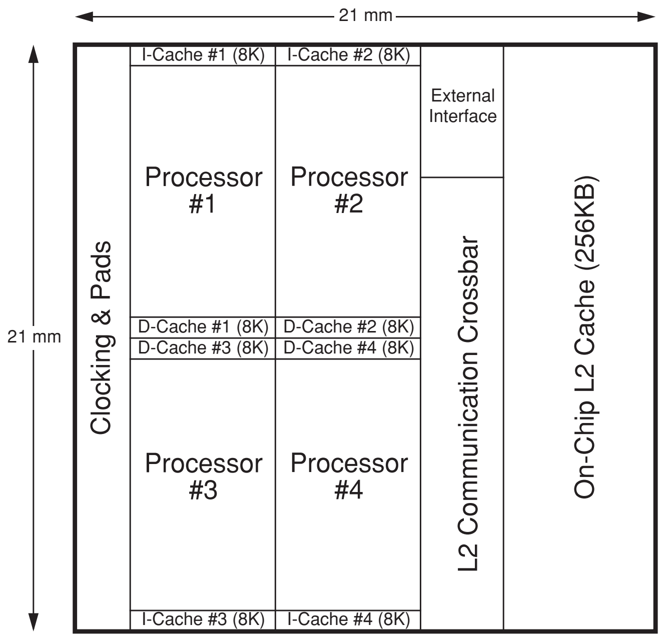

FIGURE 1.6: Floorplan for the 4-way CMP.

TABLE 1.2: Benchmarks used to compare the two equal-area architectures

INTEGER APPLICATIONS	
compress	compresses and uncompresses file in memory
eqntott	translates logic equations into truth tables
m88ksim	Motorola 88000 CPU simulator
MPsim	VCS compiled Verilog simulation of a multiprocessor
FLOATING POINT APPLICATIONS	
applu	solver for parabolic/elliptic partial differential equations
apsi	solves problems of temperature, wind, velocity, and distribution of pollutants
swim	shallow water model with 1K × 1K grid
tomcatv	mesh-generation with Thompson solver
MULTIPROGRAMMING APPLICATION	
pmake	parallel make of gnuchess using C compiler

integer benchmarks (*compress*, *m88ksim*), one SPEC92 integer benchmark (*eqntott*), one other integer application (*MPsim*), four SPEC95 floating point benchmarks (*applu*, *apsi*, *swim*, *tomcatv*), and a multiprogramming application (*pmake*). The applications were parallelized in different ways to run on the CMP microarchitecture. *Compress* was run unmodified on both the SS and CMP microarchitectures, using only one processor of the CMP architecture. *Eqntott* was parallelized manually by modifying a single bit vector comparison routine that is responsible for 90% of the execution time of the application [5]. The CPU simulator (*m88ksim*) was also parallelized manually into three "pipeline stage" threads. Each of the three threads executes a different phase of simulating a different instruction at the same time. This style of parallelization is very similar to the overlap of instruction execution that occurs in hardware pipelining. The *MPsim* application was a Verilog model of a bus-based multiprocessor running under a multithreaded compiled code simulator (*Chronologic VCS-MT*). The multiple threads were specified manually by assigning parts of the model hierarchy to different threads. The *MP-sim* application used four closely coupled threads, one for each of the processors in the model. The parallel versions of the SPEC95 floating point benchmarks were automatically generated by the SUIF compiler system [6]. The *pmake* application was a program development work-load that consisted of the compile phase of the Modified Andrew Benchmark [7]. The same *pmake* application was executed on both microarchitectures; however, the OS was able to take advantage of the extra processors in the MP microarchitecture to run multiple compilations in parallel.

In order to simplify and speed simulation on this simple demonstration, these are older benchmarks with relatively small data sets compared with those that would be used on real machines today, such as the SPEC CPU 2006 suite [8]. To compensate, the caches and rest of the memory hierarchy were scaled accordingly. For example, while a processor today would tend to have an L2 cache of a few megabytes in size, our simulated systems have L2 caches of only 256 KB. Given this memory system scaling and the fact that we are primarily interested in the performance effects of the processor core architecture differences, the results from this scaled-down simulation example should give a reasonable approximation of what would happen on real systems with larger applications and better memory systems.

1.3.1 Simulation Results

Table 1.3 shows the IPC, branch prediction rates and cache miss rates for one processor of the CMP, while Table 1.4 shows the instructions per cycle (IPC), branch prediction rates, and cache miss rates for the SS microarchitecture. Averages are also presented (geometric for IPC, arithmetic for the others). The cache miss rates are presented in the tables in terms of *misses per completed instruction* (MPCI), including instructions that complete in kernel and user mode. A key observation to note is that when the issue width is increased from 2 to 6 the

TABLE 1.3: Performance of a single 2-issue superscalar processor, with performance similar to an Intel Pentium from the early to mid-1990's

PROGRAM	IPC	BP RATE %	I-CACHE % MPCI	D-CACHE % MPCI	L2 CACHE % MPCI
compress	0.9	85.9	0.0	3.5	1.0
eqntott	1.3	79.8	0.0	0.8	0.7
m88ksim	1.4	91.7	2.2	0.4	0.0
MPsim	0.8	78.7	5.1	2.3	2.3
applu	0.9	79.2	0.0	2.0	1.7
apsi	0.6	95.1	1.0	4.1	2.1
swim	0.9	99.7	0.0	1.2	1.2
tomcatv	0.8	99.6	0.0	7.7	2.2
pmake	1.0	86.2	2.3	2.1	0.4
Average	0.9	88.4	1.2	2.7	1.3

actual IPC increases *by less than a factor of 1.6* for all of the integer and multiprogramming applications. For the floating point applications, more ILP is generally available, and so the IPC varies from a factor of 1.6 for *tomcatv* to 2.4 for *swim*. You should note that, given our assumption of equal—but not necessarily specified—clock cycle times, IPC is directly proportional to the overall performance of the system and is therefore the performance metric of choice.

One of the major causes of processor stalls in a superscalar processor is cache misses. However, cache misses in a dynamically scheduled superscalar processor with speculative execution and nonblocking caches are not straightforward to characterize. The cache misses that occur in a single-issue in-order processor are not necessarily the same as the misses that will occur in the speculative out-of-order processor. In speculative processors there are misses that are caused by speculative instructions that never complete. With nonblocking caches, misses may also occur to lines which already have outstanding misses. Both types of misses tend to inflate the cache miss rate of a speculative out-of-order processor. The second type of miss is mainly responsible for the higher L2 cache miss rates of the 6-issue processor compared to the 2-issue processor, even though the cache sizes are equal.

TABLE 1.4: Performance of the 6-way superscalar processor, which achieves per-cycle performance similar to today's Intel Core or Core 2 out-of-order microprocessors, close to the limit of available and exploitable ILP in many of these programs

PROGRAM	IPC	BP RATE %	I-CACHE % MPCI	D-CACHE % MPCI	L2 CACHE % MPCI
compress	1.2	86.4	0.0	3.9	1.1
eqntott	1.8	80.0	0.0	1.1	1.1
m88ksim	2.3	92.6	0.1	0.0	0.0
MPsim	1.2	81.6	3.4	1.7	2.3
applu	1.7	79.7	0.0	2.8	2.8
apsi	1.2	95.6	0.2	3.1	2.6
swim	2.2	99.8	0.0	2.3	2.5
tomcatv	1.3	99.7	0.0	4.2	4.3
pmake	1.4	82.7	0.7	1.0	0.6
Average	1.5	88.7	0.5	2.2	1.9

Figure 1.7 shows the IPC breakdown for one processor of the CMP with an ideal IPC of two. In addition to the actual IPC achieved, losses in IPC due to data and instruction cache stalls and pipeline stalls are shown. A large percentage of the IPC loss is due to data cache stall time. This is caused by the small size of the primary data cache. *M88ksim*, *MPsim*, and *pmake* have significant instruction cache stall time, which is due to the large instruction working set size of these applications in relation to these small L1 caches. *Pmake* also has multiple processes and significant kernel execution time, which further increases the instruction cache miss rate.

Figure 1.8 shows the IPC breakdown for the SS microarchitecture. A significant amount of IPC is lost due to pipeline stalls. The increase in pipeline stalls relative to the 2-issue processor is due to limited ILP in the applications and the longer L1 data cache hit time. The larger instruction cache in the SS microarchitecture eliminates most of the stalls due to instruction misses for all of these scaled-down applications except *MPsim* and *pmake*. Although the SPEC95 floating point applications have a significant amount of ILP, their performance is limited on the SS microarchitecture due to data cache stalls, which consume over one-half of the available IPC.

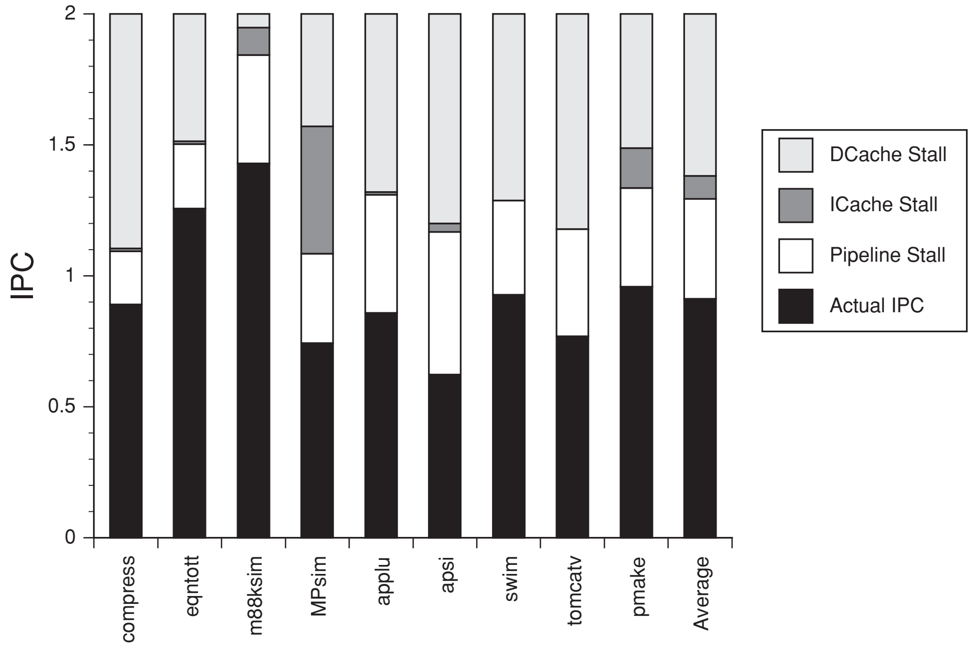

FIGURE 1.7: IPC breakdown for a single 2-issue processor.

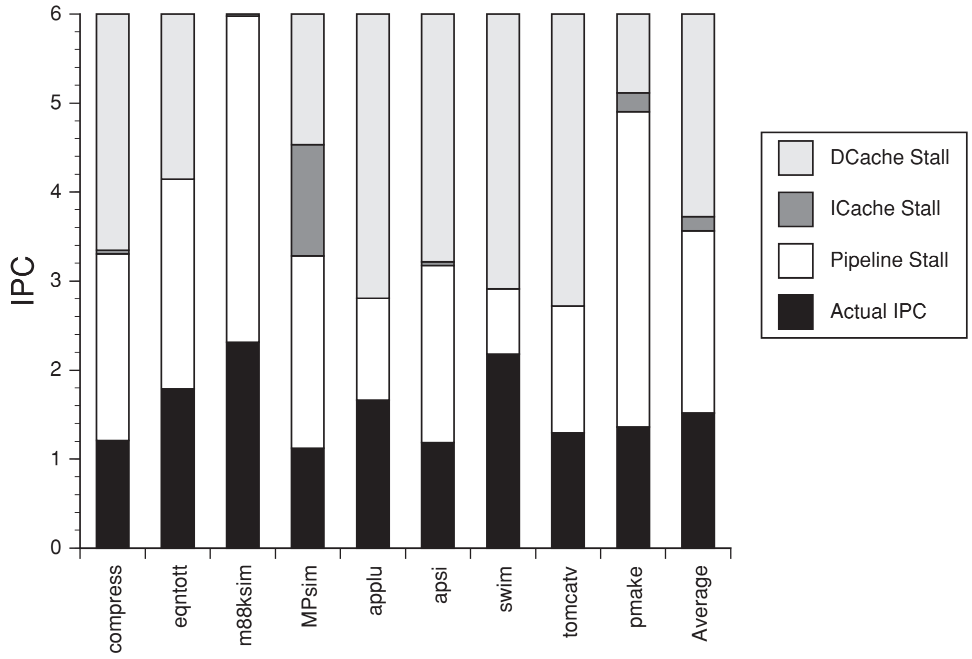

FIGURE 1.8: IPC breakdown for the 6-issue processor.

TABLE 1.5: Performance of the full 4 × 2-issue CMP

PROGRAM	I-CACHE % MPCI	D-CACHE % MPCI	L2 CACHE % MPCI
compress	0.0	3.5	1.0
eqntott	0.6	5.4	1.2
m88ksim	2.3	3.3	0.0
MPsim	4.8	2.5	3.4
applu	0.0	2.1	1.8
apsi	2.7	6.9	2.0
swim	0.0	1.2	1.5
tomcatv	0.0	7.8	2.5
pmake	2.4	4.6	0.7
Average	1.4	4.1	1.6

Table 1.5 shows cache miss rates for the CMP microarchitecture given in MPCI. To reduce miss-rate effects caused by the idle loop and spinning due to synchronization, the number of completed instructions are those of the original uniprocessor code executing on one processor. Comparing Tables 1.3 and 1.5 shows that for *eqntott*, *m88ksim*, and *apsi* the MP microarchitecture has significantly higher data cache miss rates than the single 2-issue processor. This is due primarily to the high degree of communication present in these applications. Although *pmake* also exhibits an increase in the data cache miss rate, it is caused by process migration from processor to processor in the MP microarchitecture.

Finally, Fig. 1.9 shows the overall performance comparison between the SS and CMP microarchitectures. The performance is measured as the speedup of each microarchitecture relative to a single 2-issue processor running alone. On *compress*, an application with little parallelism, the CMP is able to achieve 75% of the SS performance, even though three of the four processors are idle. For applications with fine-grained parallelism and high communication, such as *eqntott*, *m88ksim*, and *apsi*, performance results on the CMP and SS are similar. Both architectures are able to exploit fine-grained parallelism, although in different ways. The SS microarchitecture relies on the dynamic extraction of ILP from a single thread of control, while the CMP takes advantage of a combination of some ILP and some fine-grained TLP. Both

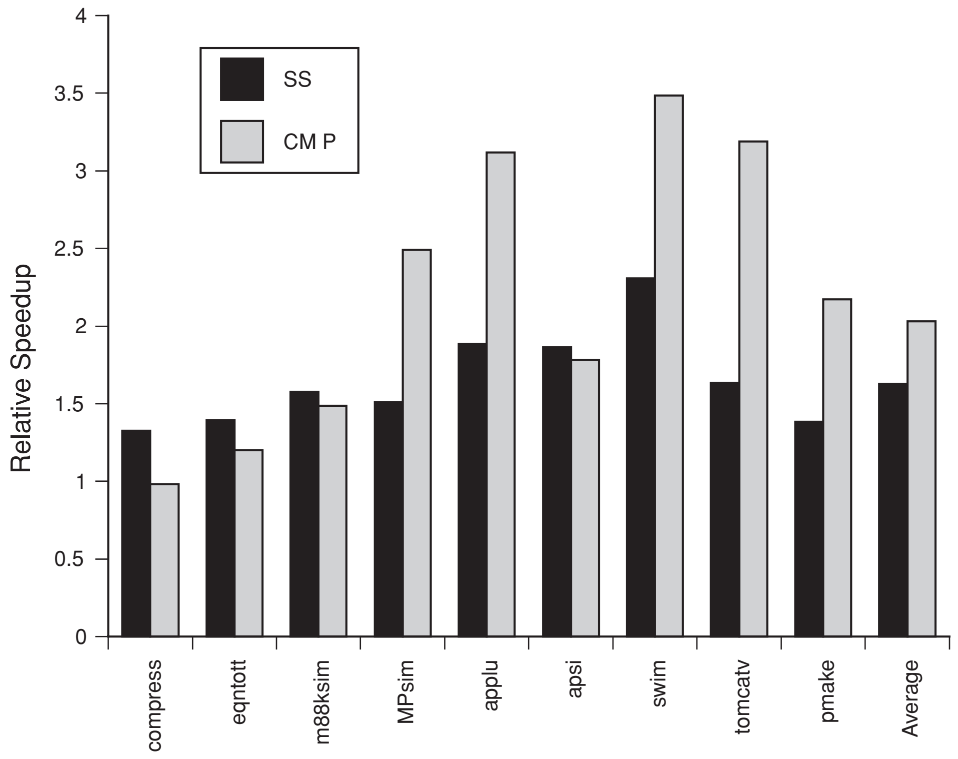

FIGURE 1.9: Performance comparison of SS and CMP.

of these approaches provide a 30–100% performance boost over a single 2-issue processor. However, it should be noted that this boost was completely "free" in the SS architecture, but required some programmer effort to extract TLP in the CMP case. Finally, applications with large amounts of parallelism allow the CMP microarchitecture to take advantage of coarse-grained parallelism and TLP, while the SS can only exploit ILP. For these applications, the CMP is able to significantly outperform the SS microarchitecture, whose ability to dynamically extract parallelism is limited by its single instruction window.

1.4 THIS BOOK: BEYOND BASIC CMPs

Now that we have made the case for why CMP architectures will predominate in the future, in the remainder of this book we examine several techniques that can be used to improve these multicore architectures beyond simple CMP designs that just glue discrete processors together and put them onto single pieces of silicon. The tight, on-chip coupling among processors in CMPs means that several new techniques can be used in multicore designs to improve both applications that are throughput-sensitive, such as most server applications, and ones that are latency-sensitive, such as typical desktop applications. These techniques are organized by the types of applications that they help support:

- In Chapter 2, we look at how CMPs can be designed to effectively address throughput-bound applications, such as our *pmake* example or commercial server workloads, where applications are already heavily threaded and therefore there is significant quantity of natural coarse-grained TLP for the various cores in the CMP to exploit. For these applications, the primary issues are quantitative: one must balance core size, numbers of cores, the amount of on-chip cache, and interconnect bandwidths between all of the on-chip components and to the outside world. Poor balance among these factors can have a significant impact on overall system performance for these applications.
- Chapter 3 focuses on the complimentary problem of designing a CMP for latency-bound general-purpose applications, such as *compress* in the previous example, which require extraction of fine-grained parallelism from sequential code. This chapter focuses on techniques for accelerating legacy uniprocessor code, more or less automatically, by extracting TLP from nominally sequential code. Given the large amount of latency-sensitive sequential code already in existence (and still being written!) and the finite number of engineers familiar with parallel programming, this is a key concern for fully utilizing the CMPs of the future.
- While automated parallelism extraction is helpful for running existing latency-bound applications on CMPs, real programmers will almost always be able to achieve better performance than any automated system, in part because they can adjust algorithms to expose additional parallelism, when necessary. However, conventional parallel programming models, which were originally designed to be implementable with the technology available on multichip multiprocessors of the past, have generally been painful to use. As a result, most programmers have avoided them. Architecture enhancements now possible with CMPs, such as the *transactional memory* described in Chapter 4, can help to simplify parallel programming and make it truly feasible for a much wider range of programmers.

Finally, in Chapter 5 we conclude with a few predictions about the future of CMP architecture design—or, more accurately, the future of all general-purpose microprocessor design, since it is now apparent that we are living in a multicore world.

REFERENCES

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, Apr. 19, 1965.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach,* 3rd edition. San Francisco, CA: Morgan Kaufmann, 2003.

[3] D. W. Wall, “Limits of instruction-level parallelism,” WRL Research Report 93/6, Digital Western Research Laboratory, Palo Alto, CA, 1993.

[4] K. Yeager et al., “R10 000 superscalar microprocessor,” presented at *Hot Chips VII*, Stanford, CA, 1995.

[5] B. A. Nayfeh, L. Hammond, and K. Olukotun, “Evaluating alternatives for a multiprocessor microprocessor,” in *Proceedings of 23rd Int. Symp. Computer Architecture*, Philadelphia, PA, 1996, pp. 66–77.

[6] S. Amarasinghe et al., “Hot compilers for future hot chips,” in *Hot Chips VII*, Stanford, CA, Aug. 1995. http://www.hotchips.org/archives/

[7] J. Ousterhout, “Why aren’t operating systems getting faster as fast as hardware?” in *Summer 1990 USENIX Conference*, June 1990, pp. 247–256.

[8] Standard Performance Evaluation Corporation, SPEC, http://www.spec.org, Warrenton, VA.