Chapter 1 Introduction

Integrated circuits containing multiple processor cores are the latest in a long series of technology advances driven by Moore's law, which states that the number of transistors on a single integrated circuit chip will double every 18-24 months. Multi-core chips will enable the next major step in delivering cost-efficient computing power to a broad range of computer systems, ranging from laptops to server farms that fill a large room. The widespread adoption of multiple cores in commodity integrated circuits also means, by implication, that practically all computer systems will be multiprocessor systems. And the study of computer system architecture will necessarily entail the study of multiprocessor systems.

Multiprocessor systems are not new, of course; systems containing multiple processors have been with us since the early days of computing. Multiprocessor systems originally were used only for applications with the highest computational demands. Over the years, their use has gradually become more widespread, and they are now commonly used in server systems. With the introduction of multi-core integrated circuits, however, they will become truly ubiquitous as their use will extend all the way down to desktop client and mobile computing systems. This means that there will be a wider range of multiprocessor applications, including many that have traditionally been done on uniprocessors. And, moving multiple processors onto a single chip shifts a number of engineering tradeoffs and places new challenges upon the hardware designer.

Historically, architecture-level performance increases have come through the use of parallelism in one form or another. Superscalar uniprocessors employ complex hardware to exploit high levels of instruction level parallelism (ILP). But, because it has a single program counter, a superscalar processor supports only one thread of execution. Extracting increasing levels of ILP from a single thread requires complexity (in terms of transistor counts, critical control paths, and intellectual complexity) that increases disproportionately when compared with the achieved performance benefits.

In contrast to the increasingly aggressive superscalar uniprocessors, multi-core integrated circuits have the potential to provide a much more cost-effective way of increasing chip-level computing power. A multi-core integrated circuit contains multiple program counters, all operating in parallel to fetch and execute instructions belonging to multiple threads. The multiple threads can be applied to the execution of a single program, for truly parallel operation, or the threads can be applied to different programs, thereby increasing system throughput.

Placing multiple cores on a chip also increases the opportunities for improved efficiency through hardware resource sharing, especially in the memory hierarchy. Shared cache memories will allow flexibility in cache management so that available cache resources can be balanced among competing computation threads based on their needs. But this will require innovative hardware mechanisms to provide the right balance of resources. Relieved of chip boundary constraints, designers will also have new opportunities to innovate by taking advantage of increased bandwidth and reduced latency when on-chip

subsystems communicate. At the system level, combining multi-core chips into larger multiprocessors will present additional opportunities for resource sharing, as well as challenges. Some of these challenges come from the scale of the systems and others from the need to interface on-chip memory hierarchies with off-chip memory while providing a coherent, consistent view of memory to software.

Multiple cores on a chip will add little additional system cost when compared with single core chips: motherboards, power supplies, and cooling requirements are little changed by multi-core chips, especially given that the cores themselves are likely to be more performance efficient when compared with the high performance wide issue superscalar chips of today.

More than anything else, the thing that has driven computer systems toward multi-core systems is demand for performance. This performance will enable new, more demanding applications and it can be used to improve performance and quality for existing applications. In the next section we will describe ways in which multiple processors can be used to provide higher performance. Then, in the following section we will summarize some of the primary application areas for multi-core systems and comment on the impact that multi-core ICs will have. Then, we will turn to important performance, cost, and architecture issues that will help shape multi-core systems.

1.1 Using Multiple Processors: Programming Models

To illustrate the ways that multiprocessors can improve performance, we first consider, at a high level, the ways that software can take advantage of multiple processors. We refer to these high level paradigms as *programming models* – these are the basic methods for expressing algorithms. A number of parallel programming models have been proposed and used, but here we will focus on three that are used most commonly in practice.

The first programming model doesn't involve explicit software parallelism at all. Conventional *multiprogramming* supports multiple single-threaded programs on the same system (Figure 1). The operating system schedules the programs onto available processors. On a multiprocessor system, a number of independent programs can execute in parallel, thereby increasing aggregate system throughput.

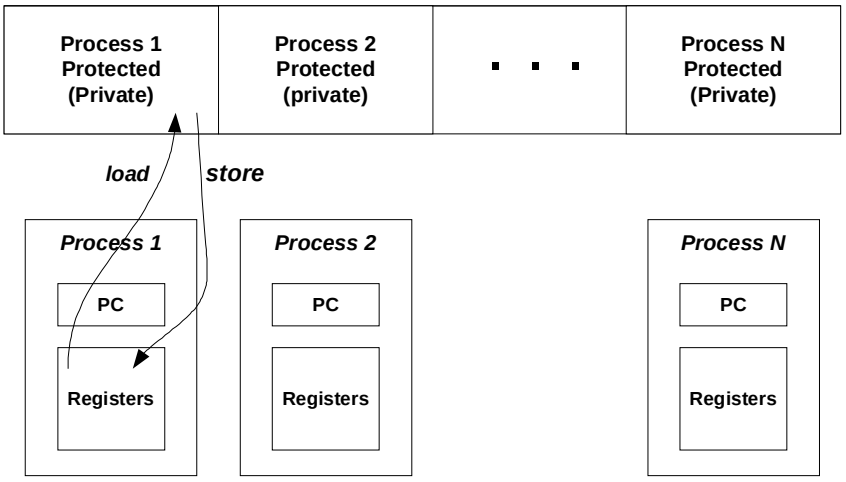


Figure 1. In a multiprogrammed system, multiple, independent programs execute in parallel.

We refer to this as *throughput* parallelism, and not only is aggregate system throughput improved, but shared system resources are used more efficiently, including memory, interconnection networks, and

the I/O system. The goals are improved aggregate throughput and efficiency, and these goals may, in some cases, come at the expense of single thread performance. There may be instances when individual programs run slower than if they were running alone on a non-shared uniprocessor system.

With *program parallelism*, a single program is written in such a way that the computational work can be divided among the multiple processors. Then, the single program is assigned to multiple processors, thereby decreasing the total time required for the program's execution. In practice, there are two primary programming models for achieving program parallelism: shared memory and message passing.

With the shared memory programming model (Figure 4), parallel threads of execution all have access to a shared memory address space. Consequently, they can communicate by storing and loading values into the shared memory. The threads may also have private data areas for variables that only they use; often at least part of this private area is configured as a stack.

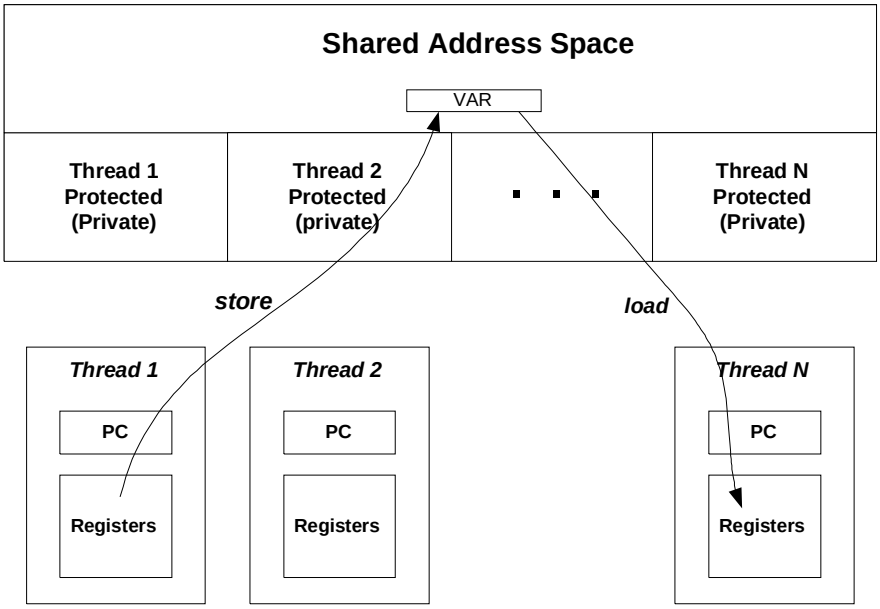


Figure 2. The shared memory programming model communicates data values through a single address space that is shared among multiple threads of execution.

In the message passing programming model (Figure 3), each of the threads of execution has its own memory address space that only it can access. Communication among the threads then takes place via explicit sends and receives of messages. The messages may be communicated via networking hardware. Alternatively, the message passing model can be implemented on a system that implements physically shared memory. In this case, messages may be passed through buffers in the shared memory region.

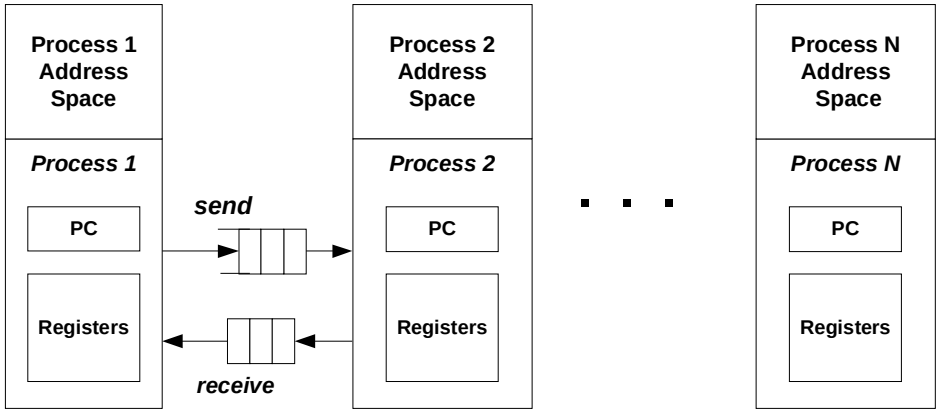


Figure 3. The message passing programming model communicates data by using explicit send and receive operations.

1.2 Applications of Multiprocessor Systems

The move to multi-core systems will yield enhanced computing platforms for a wide range of applications. This section briefly summarizes some of them. This is by no means an exhaustive list, but it will illustrate the breadth of applications for multi-core systems. To understand the applications of multi-core systems, we will divide the applications broadly into those that exploit throughput parallelism and those that exploit program parallelism.

1.2.1 Commercial

Today, commercial applications are the biggest user of multiprocessor systems. These systems are primarily servers of one kind or another. They provide a number of users (sometimes a very large number of users) with a requested service. Examples include web servers that provide online entertainment, retail services, or access to general information, and transaction servers as one might find in a bank or airline reservation system.

Throughput parallelism is of primary interest in these servers; in many of applications, a given service request only occupies one processor at a time. For example, with online transaction processing (OLTP) as might be done at a bank, a customer's individual transaction would normally be handled by a single processor (and the computing load would be fairly light at that). In other applications, however, multiple processors may be marshaled to perform a service cooperatively. For example, if a very large database spread over a large number of disks is being searched, the search can be performed in parallel with each processor searching a subset of the data base. This is an example of program parallelism, but the separate processors run more-or-less independently for long periods of time, only interacting, for example, when combining results of their searches.

Much of the work in a commercial server involves data movement, and tasks such as searching and sorting, with relatively simple operations being performed on the data. These applications are often disk I/O and network intensive. There are other applications, however, where there is a significant amount of number crunching. Examples here are stock portfolio analysis and decision support. Stock portfolio analysis can best be done by a compute server (See Section 1.2.4 below). Decision support often involves "data mining" where large amounts of data are searched and analyzed in order to make higher level marketing and sales decisions, for example.

Based on current trends, there will likely be two categories of commercial servers. First, there will be small servers, where the components fit on a single board and fit in a relatively small package. Second, these smaller servers will be interconnected via local area networks to form computing clusters. In general, these clusters can be quite large (Figure 4).

Figure 4. A commercial server farm composed of clustered servers can fill a large room.

1.2.2 Desktop/Office

Desktop systems, as the name suggests, sit on (or possibly under) a single user's desk and are used principally by that one user. These personal computers (PCs) typically act as the clients for the servers described in the preceding section, and they are used for general purpose computing in the office. Most PC application programs have not been re-written to exploit program parallelism. So, when using currently available software, most desktop applications that would benefit from a multi-core system would do so via throughput parallelism. Even so, many users have at least a few applications that are run simultaneously and can therefore benefit from throughput parallelism; for example, one might be doing a virus scan simultaneously with viewing an internet video.

Because a single user is involved, however, the number of programs actively executing at any given time is likely to be small. As multi-core systems come into widespread use on the desktop, the more compute intensive applications are likely to be programmed to take advantage of program parallelism. This is one of the big challenges presented by multi-core systems, however. Unlike some commercial server applications, such as the database search example given above, the personal productivity programs running concurrently on a desktop tend to have parallelism that is less regular, making them more difficult to parallelize, and limiting the number of parallel processors that can be efficiently used.

1.2.3 Multi-media/Home

People use personal computers in the home, sometimes in similar ways to those in the office. But there is also a significant, and increasing, amount of home computing that is focused on entertainment. Entertainment applications, including games, are increasingly leaving the desktop PC for more special purpose entertainment systems and game consoles. For these more media intensive applications both program parallelism and throughput parallelism are available. As entertainment systems acquire functionality, the use of classic desktop systems is likely to diminish in many homes.

Special purpose entertainment computers, despite being considered special purpose, will have very high sales volumes, and will likely be a significant driver for multi-core systems. In these systems, the multiple cores may not be identical, but will likely include a combination of general purpose and special purpose cores. The special purpose cores will be designed solely to exploit program parallelism. Throughput parallelism will then be spread over the general purpose and special purpose cores.

Historically, entertainment systems have been standalone, but entertainment is rapidly evolving into a client-server model. Remote media servers will have clients in the home; the home client(s) may be individual computers or entertainment hubs, themselves small multi-threaded servers.

1.2.4 Science and Engineering

Science and engineering computer applications are among the oldest. Consequently, these applications motivated the first multiprocessor systems. Scientific and engineering applications are often compute-intensive and involve large numbers of floating point calculations. Today these applications range from drug development to automobile crash testing to weather forecasting. These systems have some characteristics in common with commercial servers, but in scientific and engineering applications there is more emphasis on program parallelism rather than throughput processing. In very large compute servers there are multiple users so both throughput and program parallelism take place simultaneously.

When compared with commercial servers, however, the sales volumes for compute servers are significantly lower. Consequently, they are likely to employ the same multi-core chips as the commercial servers. This, in turn, will affect the system level architectures that will be used. This means that, for example, small compute servers will likely be the same as small commercial servers. Large scale computing clusters employ very high speed networks to interconnect the same board-level servers as used in commercial applications. The high speed networks facilitate data sharing required by parallel applications.

Performance has provided the impetus for multi-core systems, but cost considerations, in addition to performance, shape their architecture. In the next sections, we discuss performance and cost trends and tradeoffs and then, in subsequent sections, explain how they shape multi-core chip and system architectures.

1.3 Performance

There are a number of metrics used for measuring performance, but the basis for virtually all of them is the time required to carry out a computation. Speedup, for example, is the time it takes to perform a computation (e.g., a benchmark program) on a baseline system divided by the time on a target system. A shorter execution time on the target system means a higher speedup. As another example, computation rate measured as instructions per second uses time as the denominator; the less time, the more instructions per second. Measures such as speedups and rates are useful at a higher level, and are often applied to entire programs. Because we will be working at the hardware implementation level,

however, it is more useful to consider performance at a correspondingly lower, more detailed level, with time as a performance measure.

The time required to execute a given program is a function of both software properties and the computer hardware that is executing it. First, we consider performance properties of the major computer system components, both separately and in combination. Then, we consider important performance-related software properties. In the following section, Section 1.4, we will take an engineering perspective and consider the interaction of hardware and software and the ways they will shape future chip designs.

1.3.1 System Components

We are primarily interested in computer systems consisting of a set of processors connected to a memory hierarchy through interconnection networks (Figure 5). In addition to the components shown in the figure, there are various I/O devices that store and transmit data as well as interact with humans. The primary feature in the figure is the memory hierarchy consisting of processor registers, multiple levels of cache memories, main memory, and secondary storage (disks). The memories used in the memory hierarchy span a range of sizes. Typical memory sizes and access latencies are given on the right side of Figure 5. The relationship between size and access latency leads to the use of memory hierarchies with small, fast memories nearer the processor, and larger, slower memories farther away. Recently used data (or data close to recently used data) naturally migrates to a level of the memory hierarchy closer to the processor, while less recently used data migrates away from the processor.

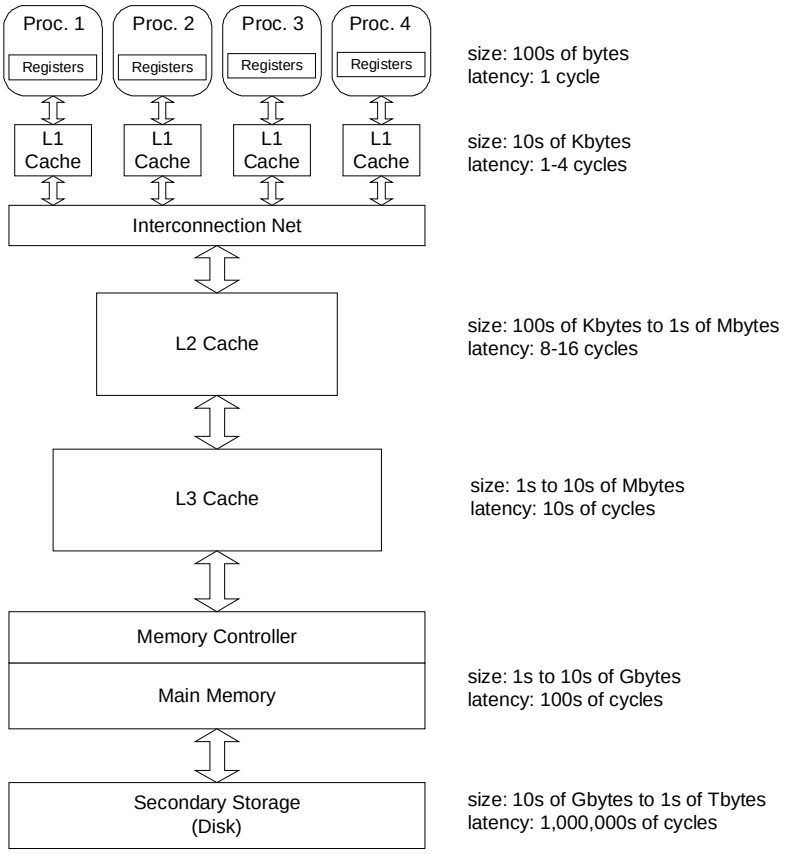


Figure 5. A multiprocessor computer system, illustrating the memory hierarchy.

The interconnected processors and memory subsystems work in concert to execute computer applications by communicating and operating on data and instructions. The execution of programs is under the control of a program counter, contained in a processor, which defines a *thread* of execution. This thread consists of the dynamic sequence of instructions as they are fetched from memory and executed.

Storage, interconnection structures, and processors provide resources whose capabilities can be measured as *capacities* (sizes) and/or *bandwidths*. These, in turn, help determine the performance measure we are most interested in: the *latency*, or time required to perform a portion of a given computation. In a multiprocessor system, capacities and bandwidths represent resources that can be allocated to, and shared among, multiple execution threads. These resources are associated with distinct physical elements (transistors and/or wires). Latency, in contrast, is a measure of the time required to perform a function and is not a resource that can be allocated or shared.

STORAGE

A storage subsystem, or memory resource, has both capacity and a bandwidth. Storage capacity is simply the number of bytes that a memory can hold, and bandwidth is the rate at which the data can be accessed from the memory. The latency of a memory and is the time required for a single access and is a function of the memory size. This is illustrated, in general terms, in Figure 5, where different levels of the memory hierarchy have different sizes and latencies. A memory also has a data width, which is the number of bytes that can be read or written in parallel. Normally, a memory cannot be pipelined (although logic before and after the memory often can), so the bandwidth is the width divided by the access time.

INTERCONNECTION NETWORKS

Interconnection networks provide bandwidth, i.e., the ability to transfer data at a certain rate. In Figure 5, only one interconnection network is explicitly labeled, but in a real system, there is a network at each level of the memory hierarchy. In some cases the interconnection network may be as simple as a point-to-point bus. In general, an interconnection network may contain a number of switching stages (Figure 6), with time being spent on wires between the switches and in the switches, themselves. For simplicity, we assume that the interconnection network operates synchronously with a cycle time, which may or may not be the same as the processor's cycle time. The unloaded latency, *L*, between two ends of the network is the minimum time required to move a data item from one end to the other. As with a memory, an interconnection network also has a width; in this case, the width is the number of bits or bytes that can be transferred in a single cycle. The bandwidth is, therefore, the width divided by the network's cycle time.

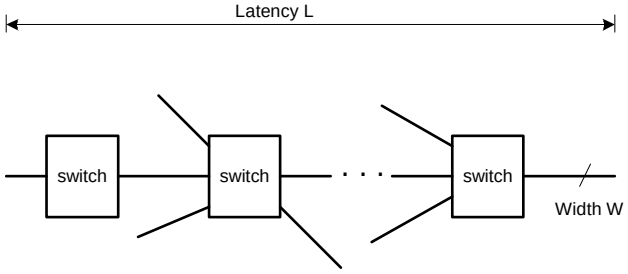


Figure 6. A section of an interconnection network with an unloaded latency L and width W.

In a network, it is often the case that a block of data being transferred is larger than the width of the network's data links. In this case, it takes multiple cycles to transfer the block across a link. Assuming the network is fully pipelined, the data block size is B and the width is W, then it takes B/W cycles to place the data block onto the network (or remove it from the network at the other end). If the end-to-end latency of the network is L then, the total latency for transferring the data is $B/W+L$ cycles. Note that the latency as we have just defined it is for an unloaded system. That is, where there is no contention for any of the network resources. When there is contention, for example, if two different transfers need to use the same link at the same time, the total latency also includes arbitration time (to determine which of the contending requests gets to use the link) and queuing delay (time waiting in buffers while other accesses are serviced). In a heavily used interconnection network, the loaded latency can be much longer than the unloaded latency.

PROCESSORS

Processors provide a *computation bandwidth* that involves more than the simple access and movement of data as with interconnection networks and memories. Rather, computation bandwidth is the rate at which instructions can be fetched, decoded, and executed. Computation bandwidth is measured as instructions per cycle (IPC). Today's superscalar processors employ extensive instruction buffering and out-of-order instruction issuing and are capable of executing several instructions per cycle (2 to 6) under ideal conditions; that is, where all branch instructions are correctly predicted and accesses to the caches always hit. Execution of a block of instructions has a latency that is a function of the computational bandwidth. If a group of N instructions are to be executed, and the computational bandwidth is D, then N/D is the latency for executing the instructions (under ideal conditions).

SYSTEM PERFORMANCE

In a computer system, the capacities, bandwidths, and latencies of the subsystems are related, and together they determine the overall system performance (measured as a latency). To better appreciate the relationships, we use a simple model for computer performance; refer to Eyerman et al. [4] for more details. The model considers the time-related hardware activities that take place as a single thread of execution proceeds (see Figure 7).

The upper line in Figure 7 traces the computational rate (number of instructions executed per cycle) as a function of time. As long as data and instructions are found in registers and the L1 cache(s), instruction execution can proceed at approximately the maximum computational bandwidth (assuming the processor is a well-balanced superscalar design). Consequently, the latency for performing the computation of N instructions is N/D as given above. Then, the example thread encounters a branch misprediction, and begins executing wrong-path instructions. Eventually, the branch is resolved, the pipeline is flushed, and correct-path instructions are fetched and executed. Collectively, the branch misprediction results in a period of time when no good instructions are executed.

After the thread continues computation, there is a cache miss; for the sake of our example, assume the cache data must be retrieved from main memory. When this happens, the address is sent to memory over an address bus; the latency for doing so can be computed as $B/W+L$ as given above. Typically, an address bus has a low value for B/W because the address width is relatively small compared with the address bus width. Then, there is an interval where memory is busy accessing a cache line's worth of data; this requires a period of time determined by the memory access latency. After the data is read from memory, there is a burst of activity on the data lines of the bus as the data is sent back to the

processor; here, *B/W* may be a few cycles as the cache line width is typically larger than the data bus width. Upon receiving the data, the processor again begins computing at (or near) its maximum rate.

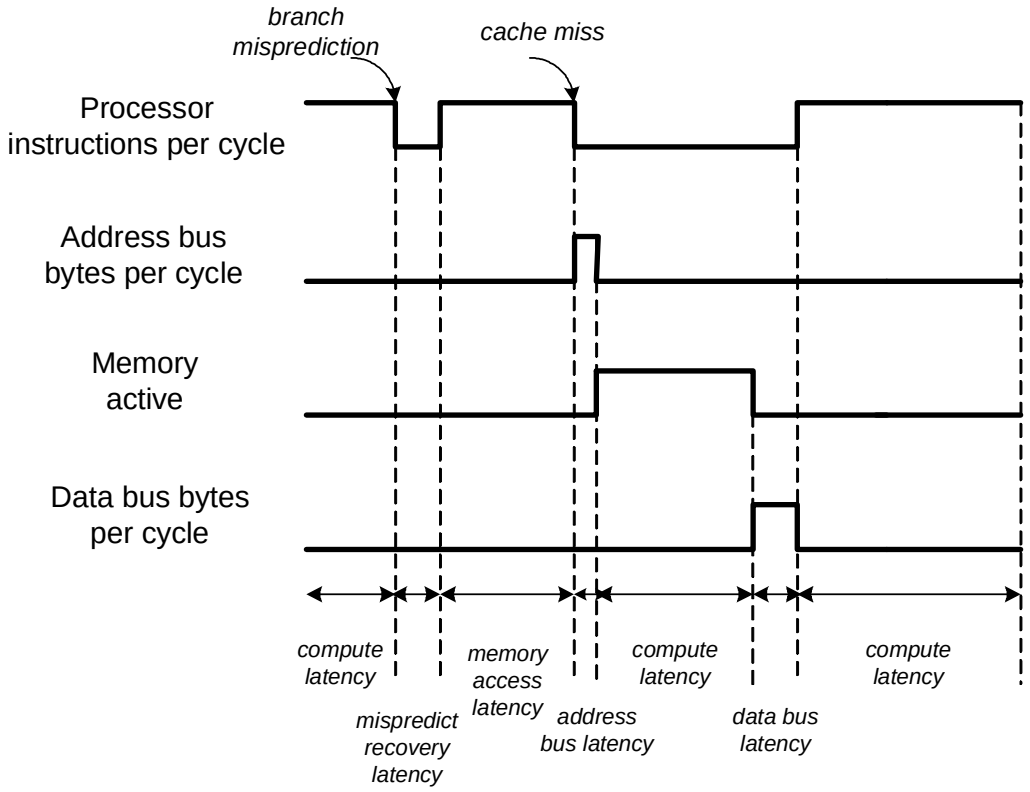


Figure 7. Dynamic system activity as a thread executes.

Although simple, this model is fairly accurate; certainly accurate enough to serve our purposes. It basically determines the total execution time (the performance metric we are interested in) by adding the latencies for computation, interconnection delays, and memory accesses. It assumes a well-balanced superscalar processor where hardware resources are adequate to sustain the maximum computational bandwidth under ideal conditions. A significant complication, not shown in our example, is the overlapping of latencies due to a burst of data cache misses in close proximity. When this happens, the handling of the multiple misses can be overlapped so that the total latency is approximately the same as the latency for a single, isolated miss.

From this example, we can make a couple of observations. First, program execution time consists of a series of time intervals, some of which perform computation, while others are spent performing no useful computation as when branch mispredictions or cache misses are resolved. Second, many of the hardware resources are active only intermittently – for example, the data path to/from memory and the memory, itself. Similar bursts of activity are present (but of different durations of course) for accesses to other elements in the memory hierarchy.

1.3.2 Program Characteristics

The performance model given in the preceding subsection ties together latency and bandwidth – higher bandwidths lead to reduced latencies. Now, we consider the effect of program characteristics on performance. These effects support observations regarding computational bandwidth and bring storage capacity and into the overall performance picture.

PROGRAM ILP

First, consider time intervals in Figure 7 when the processor is executing instructions. During these intervals, it can perform at full computational bandwidth, *provided* 1) there is enough parallelism in the instruction stream, and 2) that the processor has the hardware resources that allow a "window" of instructions to be inspected to extract this parallelism. The amount instruction level parallelism (ILP) is a program property. For a modern superscalar processor with modest issue widths (up to 6, but typically 3 or 4) there is usually enough inherent ILP that for reasonable hardware cost, one can construct an out-of-order superscalar processor to exploit it. This is illustrated in Figure 8, taken from a paper by Michaud et al. [11]. This figure, plotted on a log-log scale shows that for a number of benchmark programs ILP between 4 and 8 can be extracted from a window of size 16 to 64. There is more on this in Chapter 3 when superscalar cores are discussed.

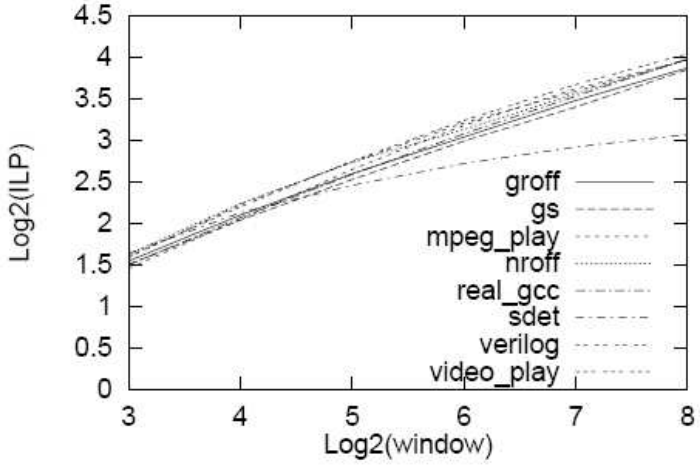


Figure 8. Figure 9 taken from Michaud et al.

CACHE WORKING SETS

Next, consider the time intervals in Figure 7 when cache misses and branch mispredictions are being handled. We lump branch prediction in with cache memories because they both involve prediction of future behavior based on past history. For convenience, we refer to cache misses and branch mispredictions collectively as *miss events*. Referring again to Figure 7, when a miss event occurs, there is an interval where performance is related to latency of handling the cache miss or resolving the branch misprediction. We will consider caches first and then branch prediction.

An important aspect of cache behavior is related to the data (or instructions) that are accessed during a given interval of time. A program's *working set* is the set of data (or instructions) that have been accessed during the most recent time interval. The size of the interval depends on the situation, but computer engineers often refer to a "working set" in a less formal way to mean the set of data or in-

structions that are actively being accessed (with an exact interval not being stated). The point is that if a program's working set fits within the size of a given cache, then most cache access will hit. On the other hand, if the working set does not fit in the cache, then the miss rate is likely to be significantly higher.

Working set effects can be observed by measuring cache miss rates for a range of cache sizes. See Figure 9 for typical program behavior. The solid line illustrates the actual miss rate. The miss rate is often rather irregular because of the structure of many programs (loop structure, for example). Moving from right to left on the graph, the sizes of data sets for a certain part of the program (or loop nest) will fit into a cache until it goes below a certain size, then it no longer fits, and there is a large jump in miss rate. However, if one smoothes out the irregularities (see the dotted line), the behavior is generally quadratic; i.e. the miss rate decreases inversely with the square root of the cache size [5].

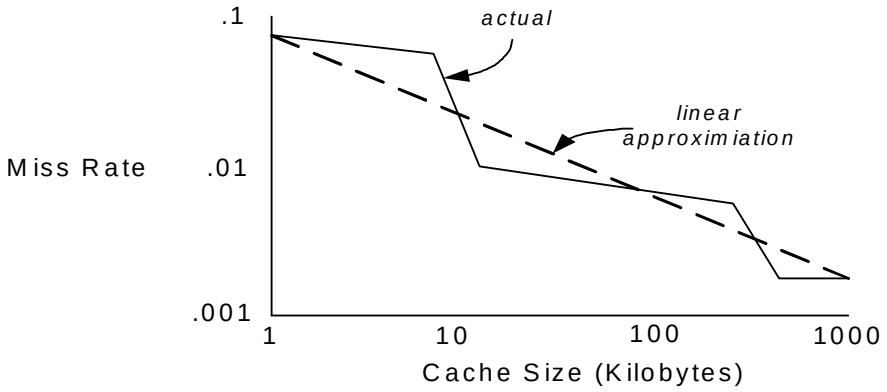


Figure 9. Cache miss rate as a function of cache size that illustrates typical working set effects.

Another important factor is that the working set of a program my change over time, as the program passes through phases of execution, and different programs may have different working set behavior. During some program phases, the working set fits in the cache and the miss rate is low; at other times it does not fit in the cache, and the miss rate is much higher.

Conditional branches also exhibit working set-like behavior. That is, over a given time interval, a set of conditional branch instructions appear repeatedly. Consequently, branches are often predicted via tables that hold history data for a working set of branch instructions. As with caches, the objective is to build prediction tables large enough to capture the branch instruction working set, but for branches, capturing a working set alone is not enough. A branch predictor also has to deal with the twists and turns that control flow paths often take due to changing data and control variables (such as loop counters); this is done by considering the past history of branch outcomes (both the branch under consideration and branches that closely precede it in time). Consequently, even if a branch history table can hold the complete working set, there will still be mispredicted branches due to inherent unpredictability of data values and control structures.

So, to tie cache and predictor capacities to performance, we conclude that, generally speaking, storage structures with larger capacities lead to fewer miss events. The fewer the number of miss events, then the fewer the number of time intervals when the processor is idle. However, the advantage of larger capacities can be offset by longer access latencies. This relationship, as well as others, is described in more detail in the next section.

1.4 Engineering Future Systems

As chip technology continues to follow Moore's law, a major concern of computer engineers is improving performance by using available transistors in the most effective way. Based on our above discussion on application demands, performance characteristics, and program properties, we will now consider a number of basic design alternatives for future systems.

1.4.1 Design Alternative 1: Build More Powerful Processors

This general approach has been the object of intense research and engineering activity for a number of years. The objective is to improve performance by reducing the time it takes to perform active computation. This can be done by executing more instructions per cycle, which is the motivation for superscalar processors that attempt to exploit ILP. This approach, by itself, has significant limitations, however. These limitations can best be understood by considering what is known as *Amdahl's law*[8]. Although Amdahl's law is usually discussed when parallelism is applied at a higher level (for example, in systems with many parallel processors), it also applies to ILP as we are considering here. Any book on multiprocessors would be lacking if it didn't discuss Amdahl's law sooner or later, so we do it sooner, rather than later.

Amdahl's law applies to systems where there is parallel activity (as when instruction are actively executed in Figure 7) interspersed with serial activity (as when a mispredicted branches or cache misses are being handled). The parallel computation activity can be sped up by adding resources (going to wider superscalar processors, for example). However, the serial (miss) activity receives no speedup from the increased computational parallelism; i.e., using a wider superscalar processor does not reduce the latency for handling a cache miss or a branch misprediction.

Consider a baseline system that uses a one-wide processor capable of sustaining steady state behavior of one instruction per cycle as long as there are no miss events. If, when the baseline processor executes a program, let *f* be the fraction of cycles where instructions are executed and *1-f* be the remaining fraction of cycles when no useful instructions are being executed due to a a stall for a cache miss, or perhaps a branch misprediction. Then, going to an *n*-wide superscalar processor can only speed up active instruction execution, not the handling miss events. The speedup due an *n*-wide superscalar is:

$$S_n = time_1 / time_n = time_1 / \ (f*time_1/n + (1-f)*time_1\ ) = 1/(f/n + 1 - f)\ .$$

This equation is a statement of Amdahl's law in the form of an equation. If, for example, the baseline one-wide processor spends half its cycles executing instructions and the other half of its cycles handling misses and mispredictions, then the speedup for wider superscalar processors is illustrated in Figure 10.

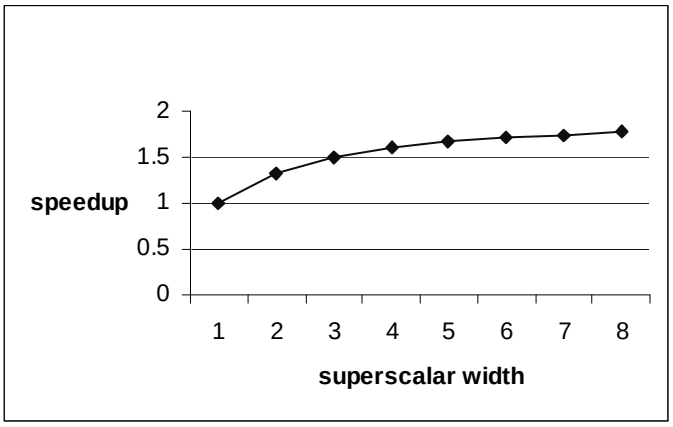


Figure 10. Amdahl's law applied to superscalar processors -- speedup from superscalar processing as a function of superscalar width.

We see from the figure that going to wider superscalar processors does improve performance, but there is a point of diminishing returns. In this example, the overall speedup increases very little for superscalar widths much larger than four. In fact, even with infinite issue width, where executing instructions takes zero time, the speedup is only two because of the time required to handle cache misses and branch mispredictions.

To make matters worse, increasing the ILP comes at a transistor cost that is superlinear with respect to superscalar processor width. That is, with each increase in width, a disproportionately large number of transistors are required. The reason is that a number of the microarchitecture structures that support superscalar processing grow quadratically as the issue width grows linearly. These include the instruction issue buffer and reorder buffer, among others. This is illustrated below in Section 1.4.3. The combination of diminishing performance gains and superlinear costs has caused superscalar processor widths to level off (at about four as just noted). There has been no significant increase in superscalar issue widths in several years.

Power considerations are also important. Power consumption is either dynamic, due to switching signals, or static, due to inherent transistor characteristics. Static power is a function of transistor counts and is always consumed, regardless of switching activity. The quadratic growth of superscalar structures tends to burn both static and dynamic power that is superlinear with respect to increases in performance.

As an alternative to wider superscalar processors, pipelines can also be made progressively deeper; that is, there are more pipeline stages, each doing less work per clock cycle. Deeper pipelines tend to increase throughput under ideal conditions. However, because of cache misses and branch mispredictions, Amdahl's law is also a limiter when deeper pipelines are used. Moreover, because of the interaction between miss latencies and the overheads for latches that separate pipeline stages, this approach for improving performance can actually result in reduced performance when pipeline depth exceeds a certain optimal point. A more detailed explanation of this phenomenon can be found in Eyerman et al. [4] and Hartstein and Puzak [6].

The conclusion that has been reached by many computer designers is that superscalar widths and pipeline depths have been pushed about as far as is practical. From the microarchitecture perspective, us-

ing increasing transistor budgets to build higher performance uniprocessors has reached a point of diminished returns.

1.4.2 Design Choice 2: Build Larger On-Chip Caches (and Predictors)

With this approach for improving performance, the objective is to reduce the overall time spent on miss events (branch mispredictions and cache misses). This can be done either by reducing their number or by reducing their effect.

With respect to branch mispredictions, relatively little can be done to reduce their effect when they occur. There have been research proposals for simultaneously fetching and executing instructions from multiple branch paths, but this tends to be costly in practice due to the very high demand for instruction fetch resources. One can also attempt to reduce the number of branch predictions, but it appears that improving branch predictors has also reached a point of diminished returns, and the cost of improved branch predictors tends to be superlinear in cost versus the reduction of mispredictions.

Turning to caches, one can reduce the number of cache misses, but this also introduces complexity and yields additional design tradeoffs. First, there is the approximate inverse quadratic relationship between cache capacity and miss rate [5]; hence, the cache size must be squared to produce a factor of two reduction in miss rate. This alone points to diminished benefits of simply making caches larger. In part, this occurs because there are some cache misses that can't really be eliminated: the compulsory misses that happen the very first time a data item is referenced.

Second, we run into the phenomenon illustrated earlier when discussing memory hierarchies: increasing the cache size makes it slower. The results of a study are reported by Hartstein et al. {7}. They show that the latency for a cache grows as the square root of it size (plus a constant for cache access operations external to the memory, itself. This is illustrated in Figure 11, taken from the aforementioned reference. So, for example, increasing the L2 cache size may reduce the number of L2 misses, but it will slow down the handling of L1 misses that hit in the L2. The performance loss due to longer L1 hit latencies offsets the gain due to fewer L1 misses.

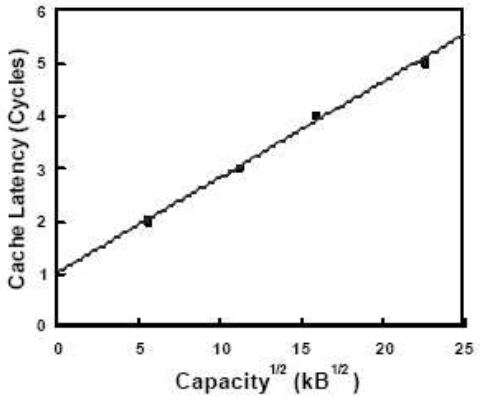


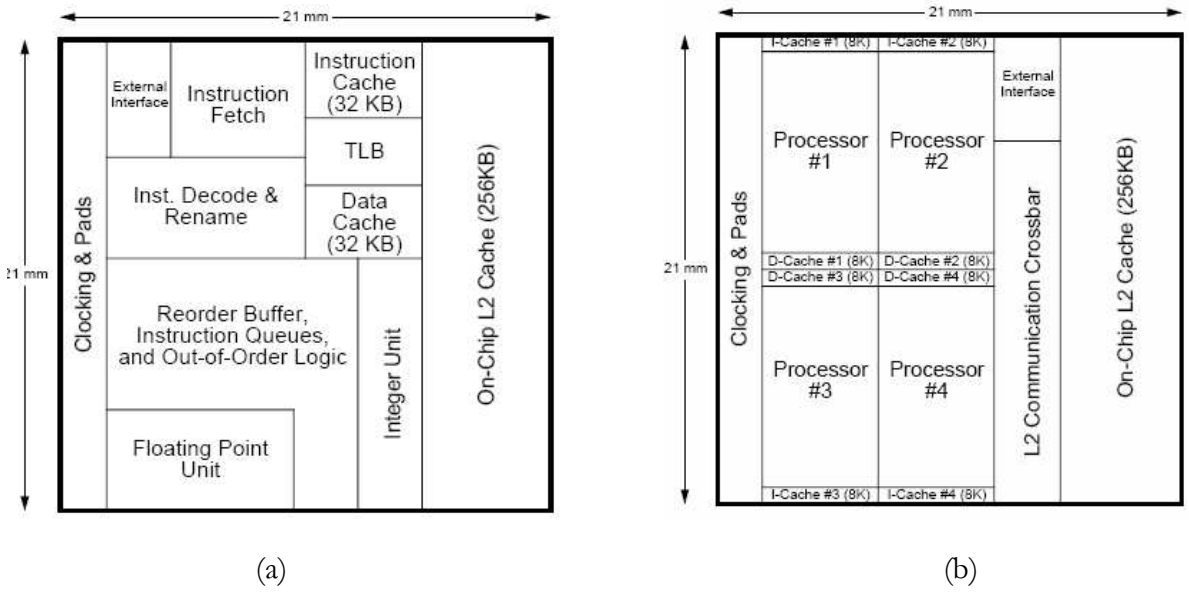
Figure 11. Cache latency increases as the square root of cache size. Figure taken from Hartstein et al., 2007 [7].

Finally, there are other techniques that can be applied to reduce the effects of cache misses, such as prefetching, but here benefits are very program dependent, and can sometimes cause slowdown. Performance losses can occur from cache pollution where potentially useful cache lines are removed to make room for prefetched lines that are never used. Prefetching also consumes memory and interconnection bandwidth, and in some cases, this can offset performance gains.

The bottom line is that improvements in caching and branch prediction are still the subject of research, but significant improvements have become very difficult achieve. A reasonable conclusion is that increasing cache and predictor sizes will yield relatively small performance improvements. Although on-chip caches and predictors are likely to increase in size somewhat, this alone is not a very effective way of using increasing transistor budgets.

1.4.3 Design Choice 3: Place Multiple Cores on a Chip

Placing multiple cores on a chip provides raw computational bandwidth that is of roughly linear cost in terms of both area and power consumption. That is, placing *n* cores on a chip provides a factor of *n* improvement in raw computational bandwidth. In an early paper advocating multi-core chips with simpler processors as an alternative to more powerful uniprocessor chips Olukotun et al. [2] analyzed the area requirements and performance of both approaches. Chip floorplans, taken from that paper are shown in Figure 12. In Figure 12a is the floorplan for a projected 6-way superscalar processor, and Figure 12b is for a multi-core chip with 4 2-way superscalar processors. Both designs incorporate the same amount of cache memory; the difference is in the processor cores. The section of the 6-way superscalar processor labeled "Reorder Buffer, Instruction Queues, and Out-of-Order Logic" contains the portions of the superscalar processor that grow quadratically with the issue width. In a 6-way superscalar processor this logic alone consumes more area than an entire 2-way core.



(a) (b)

Figure 12. Floorplans for chips containing a) a single 6-way superscalar processor and b) four 2-way superscalar cores. Figures taken from Olukotun et al. [2].

Of course, translating raw computational power into realized computational power is not simple. If the application domain is one where multiple threads of execution are readily available, then much of the raw computational bandwidth can be realized. Many servers are in such an application domain,

where throughput parallelism tends to dominate. With throughput parallelism, the serial miss event latencies for the multiple cores are essentially processed in parallel.

If the performance objective is to make a single program run faster, as is the case in many client applications, then realizing the potential of multiple cores is much more difficult. It requires programming (or re-programming) using a parallel programming model. Then, overheads for thread communication and synchronization, as well as serial code sections, can diminish the achieved performance of multiple cores (the classic application of Amdahl's Law). These issues will be discussed in greater detail in later chapters.

The bottom line is that multi-core chips have been adopted by all the microprocessor vendors. They provide an effective way of using increasing transistor budgets to improve raw computational bandwidth and of overlapping miss event latencies among multiple threads. Increasing the number of cores also increases the benefit of using more transistors for on-chip caches because multiple threads of execution also support multiple working sets. However, there are a number of challenges in achieving the potential of multi-core chips. These will be discussed in later chapters and are a central theme of this book. Moreover, there are important engineering choices within the domain of multi-core systems. These also will also be studied in detail in later chapters of the book. Here, we briefly outline two of the more important.

One of the important engineering choices is: how powerful should the cores be? Should they be complex superscalar cores that improve single thread performance? Or should they be simpler cores that maximize aggregate throughput for a larger number of threads. The answer boils down to the application domain and the importance of single program performance versus throughput. A related engineering problem is the balance between the chip real estate devoted to cores versus the area devoted to cache hierarchy.

Another important set of engineering decisions revolve around the sharing of on-chip resources. Placing multiple cores on a chip allows both on-chip memory and interconnection resources to be shared among the multiple executing threads. Because caches and interconnection structures are used intermittently, multiple cores allow more efficient use of these structures if they are shared. On the other hand, sharing does increase contention for resources, and, therefore latency increases, especially under heavy load. Deciding which on-chip resources should be shared and which should be private to a given thread is an important design problem. A related problem that occurs when resources are shared is dealing with the contention (and interference) among the threads so that overall performance is optimized and/or resources are shared in a fair manner.

Sharing can also extend to resources within a single processor core. Because architected state, i.e., the program counter and registers, defines a thread, by replicating the architected resources within a single core, multiple threads can share the core's resources efficiently. Important design decisions in the realm of multi-threaded cores involve the allocation of shared resources and thread-switching policies.

1.5 Cost Considerations

Engineering does not involve performance alone – it involves balancing a number of factors and one of the most important is cost. And cost considerations have a lot to do with determining the structure of multi-core systems.

1.5.1 System Costs

There are two major categories of costs: non-recurring costs that are paid once for the product and can be spread over the volume sold, and per-unit costs. Hence, the cost per unit shipped is $N/v + M$. Where N is non-recurring costs, v is the sales volume, and M is the per-unit cost. Some of the individual costs are broken down further follow

Design/engineering costs -- these are primarily non-recurring costs; a large part of these costs are for engineers' salaries (as well as overheads such as office space, computing support for CAD, etc.)

Production costs – these costs tend to be per-unit, but even here there are significant advantages to high volumes, because production lines can be made more efficient due to economies of scale.

Materials costs –- these also tend to be per-unit, but there are advantages to volumes. For example, suppliers often give volume discounts (because they are working with the same cost formula).

Maintenance/service costs – this tilts more in the direction of a non-recurring costs, although, again there are some cost advantages to supporting larger volumes.

Clearly, considering the above costs, volumes are very important both directly in the case of amortizing non-recurring costs, and less directly in economies of scale achieved for per-unit costs. Volume is a function of both the size of a market and a company's market share. Lower costs tend to increase market share, which, in turn increase volumes and further reduce costs; among other things, this cycle has led to a very small number of players in the general purpose microprocessor market. Another outcome is that the high volume markets (e.g., for PCs) heavily influence the design of microprocessors. The smaller volume markets often make-do with the high volume parts.

1.5.2 Market Considerations

Earlier in the chapter we discussed a number of computer application areas which can also be considered to be as *markets*. Therefore, a key aspect of microprocessor (and system) design is whether these markets should be provided with individual application-directed microprocessors, or whether some (or all) of them should share a common microprocessor design. Historically, the desktop client microprocessors and server microprocessors have been essentially the same. A server is made by combining multiple client chips. Many entertainment systems are simply PCs with high performance graphics chips added to the motherboard.

As integration levels continue, however, there will be an increasing tension between single microprocessor designs that span multiple markets, and individual market-specific designs. And this tension will be an important consideration in the direction that multi-core chip designs will take. For example, as graphics processing is integrated on the same chip as general purpose processing, will there be separate chips to address the entertainment market and the office market? Or will they use the same chip? Will there be server chips with the emphasis on throughput for many independent threads and separate client chips with emphasis on single-thread performance (and throughput for a small number of threads)?

The pressure toward separate microprocessor designs is that the performance needs and power requirements of the individual markets can be better addressed by specialized designs. The pressure toward a very small number of general designs that span markets comes from the high volumes that will be produced. Also, as a compromise, this market-based tradeoff may lead to a single core design, with

the core being placed on different chip designs in different combinations. For example, many copies of a given core may be placed on a chip for server applications, and a smaller number of the same core, combined with a graphics processor, may be used for client applications. This compromise, at least, amortizes the some of the non-recurring design maintenance costs.

An important shaper of multi-core systems is the degree to which achieving the performance objective derives from program parallelism versus throughput parallelism. Hence, there is also a tension between computer systems that are designed for markets (and application areas) that can exploit program parallelism, and systems that are designed to support the more widely available throughput parallelism. Here, some historical perspective may be enlightening.

When computers were few and very expensive, they were constructed with very large numbers of integrated circuits, each containing a small number of transistors, and high part volumes could be generated even within a single computer system. The same integrated circuit (or discrete transistors) could be used in a multitude of places within the same computer. For example, if only a few NAND gates were on a chip, then the whole computer could be designed with one part type. With MSI, where a MUX might be on a single chip, there were high volumes of MUX parts. As ICs became more complex, however, they became more specialized. This meant fewer parts in a computer, but it also meant that a given part type would be used in fewer places. In this case, volumes have to come from the numbers of computers being sold – a single computer no longer generates high part volumes all by itself.

Eventually, with the microprocessor, an entire processor was absorbed onto the chip, and this, while being economical, reduced the flexibility of the chip. Sales volumes were critical, and the volumes favored uniprocessor ICs; originally desktop uniprocessors (workstations and PCs), and more recently throughput-oriented multiprocessor servers composed of multiple uniprocessor chips. This led to integrated circuits targeted at these markets – where the volumes were very high. Consequently, the processors and systems targeted at throughput became much less expensive than high performance computer systems that attempted to exploit more program parallelism, but used specialized parts, from ICs to boards, to chassis. This divergence continued until a point was reached where the commodity processors were so cheap compared with large scale supercomputers, that it became cost effective to use large numbers of the microprocessors, very inefficiently, to tackle many large parallel problems. This trend was called the "attack of the killer micros"; supercomputers -- high performance systems with processors targeted at program parallelism -- were the target of the attack.

This trend remains to this day. As there is increasing integration, flexibility is lost. Because applications having relatively little program parallelism have higher volumes, integrated circuits for the most part favor them (at least for general purpose computing – for graphics, there are exceptions). Its not that there aren't customers for parallel computers (versus throughput computers), there certainly are. It's just that there are not enough of them to demand the volumes required to make them commodity parts.

This trend continues with muti-core systems. Thus far, they are targeted at throughput applications. Its not that they can't be used for high performance single application parallel performance; its just that they are not optimized for those applications. The dichotomy between throughput optimized systems and parallelism optimized systems influence the architecture of systems made of multi-core chips, and consequently, content of this book. By choosing to focus on multiprocessors constructed of multi-core chips we are focusing more on throughput processing than parallel processing. That is, we limit

discussion of multiprocessor systems that are specialized specifically for parallel processing; rather, the primary place we consider parallel processing is in large scale computer servers (clusters) where the network connecting computers in the cluster may be the only specialized component.

1.5.3 Price/Cost Points

There is also an important non-linear relationship between system cost and performance. That is, if we plot system cost as a function of performance, we often see "bumps" in performance caused by adding DRAM, or multiple processor chips, for example. There are also bumps in cost caused by the need to move up to higher packaging levels, to a larger power supply, to more advanced cooling, etc. and these bumps tend to come in discrete steps. This nonlinear cost/performance behavior has created cost points (actually price points) in the marketplace for both desktop and server systems.

To take the desktop example, the step up from a single motherboard to multiple motherboards would add significantly to cost, so desktop systems use a single motherboard. In desktop systems, there is a single processor chip; using multiple chips means a bump up for processor costs (as well as the socket, more cooling, etc.). A big part of cost is DRAM costs. There is typically a minimum number of DRAM chips to provide sufficient bandwidth, and adding more DRAM chips causes a another bump up in cost and performance. Over the years, the number of DRAM chips in a desktop system has remained relatively constant; memory size increases have come from increasing per-chip memory density. Similar observations can be said about servers systems where, today, there is typically a single motherboard containing, a small number of processor sockets, power, and cooling for multiple processor chips and large DRAM memories.

The cost and price points haven't changed much over the years for either desktops or servers. For example, Figure 13 shows the price of PCs plotted over nearly 25 years [10]. There is an interesting rise and fall in later 80s and early 90s, but overall the price has changed relatively little, in contrast to performance and storage capacity which have increased by many orders of magnitude (note the increase in clock frequency on a log scale). Microprocessor-based servers have also had relatively constant price points. The reason is that most of the cost components have remained more-or-less the same: systems tend to have the same number of processor chips, memory chips, similar physical packaging, etc. Most of the technology advances have been directed at giving much better performance/storage capacity at similar costs.

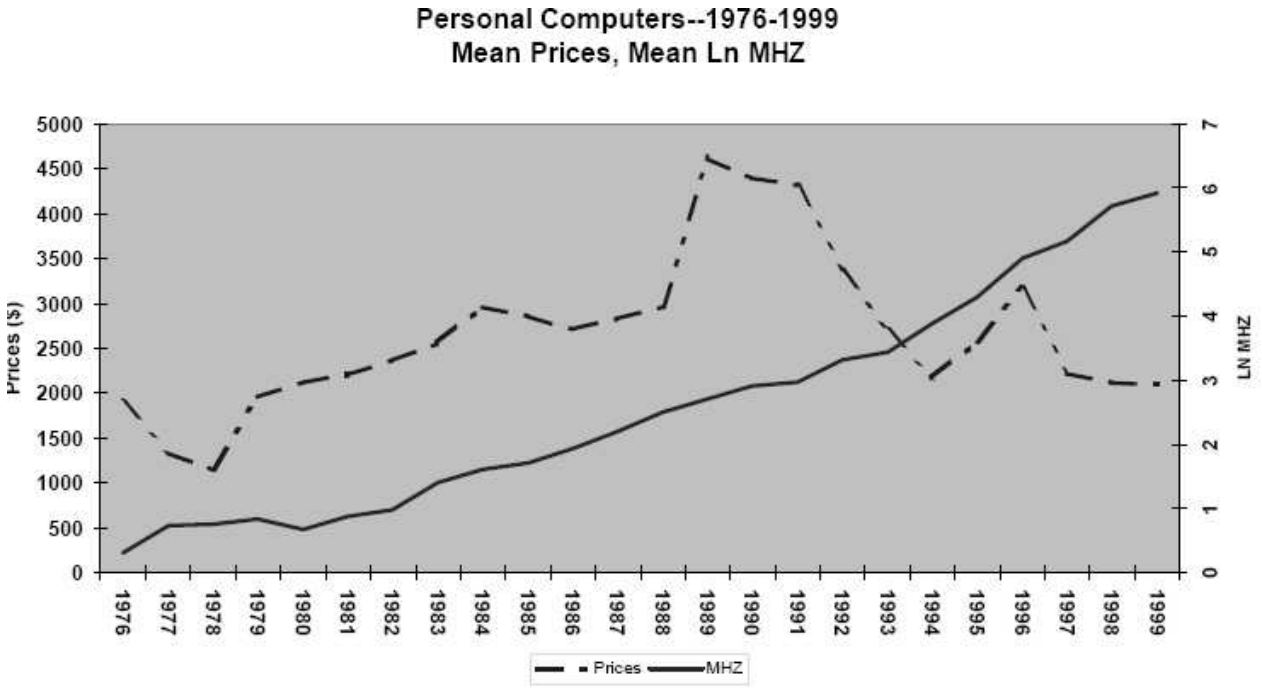


Figure 13. PC price and MHz over time (from Berndt et al., 2000)

Now, given that the cost points are strongly related to chip counts, either directly or indirectly, with the processor chip being one of the more significant, if multiple processor cores fit on a single chip, then it is possible to provide desktop and server systems with more processors, while not changing the cost points in any significant way. (This also assumes that power and cooling requirements do not change significantly). That is, a desktop system can go from a uniprocessor to a 4-way multiprocessor with little additional cost. Similarly, an entry level server can go from four processors to sixteen with little additional cost.

1.6 Multiprocessor System Architectures

Now we consider some basic principles of multiprocessor system architecture; these principles will be expanded upon throughout the remainder of the book. As mentioned earlier, a key shaper of system architecture is the sharing of resources, memory hierarchy resources in particular. There are efficiencies to sharing the hardware. In the case of main memory, hardware sharing also supports the commonly used shared memory programming model where parallel threads of execution communicate via loads and stores to variables in the shared memory.

To better understand some of the tradeoffs when sharing memory resources among multiple threads, let's first consider a simplified, ideal system consisting only of processors, an interconnection network, and one level of memory. Keep in mind that the same types of tradeoffs occur at all levels of the hierarchy (although they may be traded differently at the different levels of the hierarchy).

Ideally a shared memory system would look like Figure 14 – this is called the PRAM (Parallel Random Access Memory) model. In a perfect world all of memory would be accessible in a single cycle (unit latency) by all processors with not contention. That is, all the processors can access memory in parallel (and even the same memory location). Of course, it isn't a perfect world, and this ideal cannot be achieved. Therefore, some compromises must be made, to deal with physical realities and cost issues.

Different types of compromises can be, and are, made in real systems. Compromises can be made with respect to latency (both the amount of latency and uniformity) as well as the amount of resource contention.

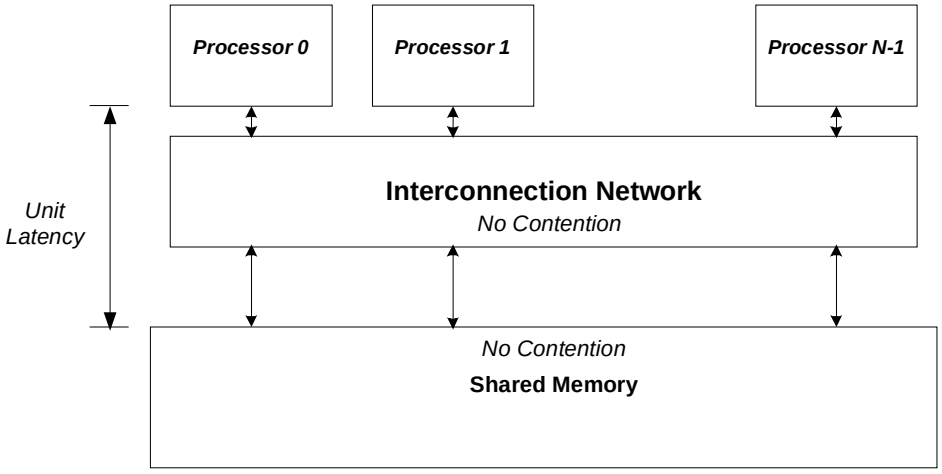


Figure 14. A PRAM multiprocessor.

First of all, it is impractical to remove all contention for resources. However, to reduce the effects of contention in the memory system, a memory can be interleaved. (See Figure 15). That is, memory is implemented with multiple banks, each of which holds a portion of the total address space, and the banks can be accessed in parallel. As long as accesses go to different banks, they can proceed in parallel; otherwise, there is some contention and one access must have to wait for the other. Then, an interconnection network, with uniform latency, connects the processors to memory. Because of the symmetry in memory access system, this is commonly called a Symmetric Multiprocessor, or SMP.

The interconnection network can be constructed with parallel buses, for example, or with more complex multi-layered switching networks, with varying degrees of contention (and cost). At the extreme, one can construct a full cross-bar switch where every input of the network has a path to every output so that there is not contention within the crossbar (although multiple inputs may contend for the same output).

After these steps have been taken, the processor-to-memory latency is larger than a single cycle in practical systems. Furthermore, the latency tends to increase as the number of sharers increases. If shared bus(es) are used, latency goes up as the number of sharers increases (which increases the bus loads and bus length). This is in addition to any latency caused by contention, and contention also grows as processors are added to the bus. With more advanced, scalable interconnection networks containing multiple stages and links, the latency typically increases roughly as the logarithm of the number of sharers. Even with crossbar switches, there is increased latency simply due to the longer wire lengths caused by a physically larger system. With today's technology, an SMP system as illustrated in Figure 15 is practical for relatively small numbers multiprocessors.

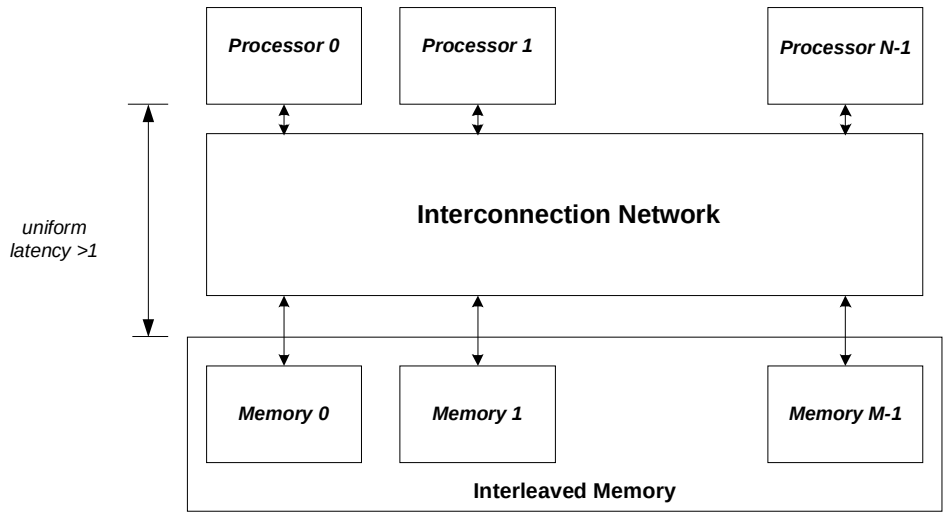


Figure 15. In a Symmetric Multiprocessor (SMP) memory is shared among a number of processors with uniform latency.

The primary advantage of this system is that it maintains uniform latency, even though it may be uniformly large. From the perspective of software development, uniform access means that the programmer (or hardware in the case of a cache memory) can place the data anywhere in the memory without regard to performance; the performance will be more-or-less the same.

An alternative approach is to make memory access latencies non-uniform, and have low latency for some memory access and high latency for others. This approach leads to non-uniform memory access (NUMA) architectures, illustrated in Figure 16. There are two basic types of NUMA architectures. In one type, all of memory is part of a common address space, supported in hardware. This means that any processor can access any memory location, but with different latencies, depending on where the memory is located. This approach supports the shared memory programming model, but does add a complication to parallel software development: the placement of data in memory can affect performance and is one more thing for the programmer to worry about. However, effective use of cache memories can sometimes alleviate performance issues related to data placement. In summary, shared memory NUMA systems allow the design of very large, scalable multiprocessor systems at the expense of sophisticated, scalable interconnection networks and the problem of data placement.

In the second type of NUMA system, the memories have different address spaces, directly accessible only by local processor(s). The interconnection network is typically a software-visible local network, and remote memories are accessed via explicit send/receive messages over the network. These systems are referred to as *Distributed Memory* systems, because the memory space is logically (as well as physically) distributed. The primary advantage of this system is that it can combine small, off-the-shelf computer systems and network hardware to perform a larger multiprocessor system.

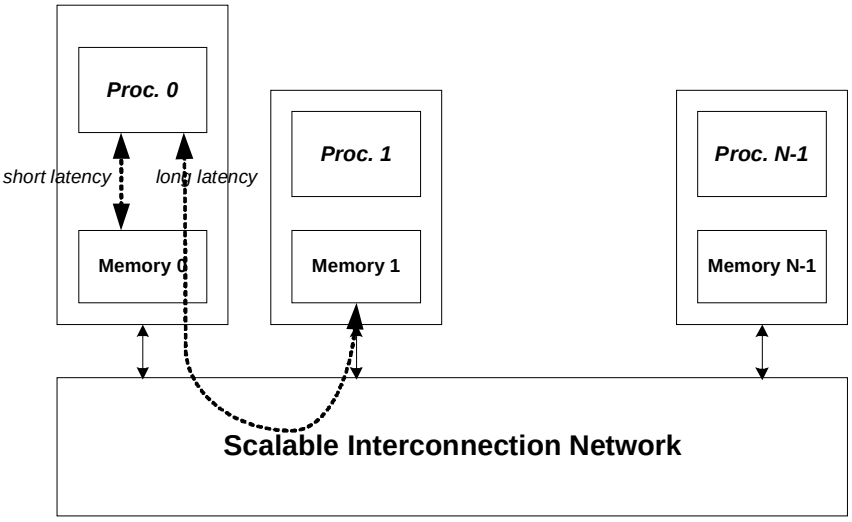


Figure 16. In a non-uniform memory access (NUMA) system, the latency for accessing may be different for a local memory than for a remote one.

An important compromise solution that combines some of the advantages of SMPs and distributed memory systems combines small SMPs with a local network (Figure 17). These systems are commonly referred to as *Clusters*, although the term is sometimes applied to distributed memory systems made up of uniprocessors as in Figure 16. Small SMPs provide advantages of hardware sharing and shared memory programming at one level, with scalability to large systems at another level. Also illustrated in the figure is another advantage of clustered systems – system storage, especially for large databases, can be shared among all the processors in the system.

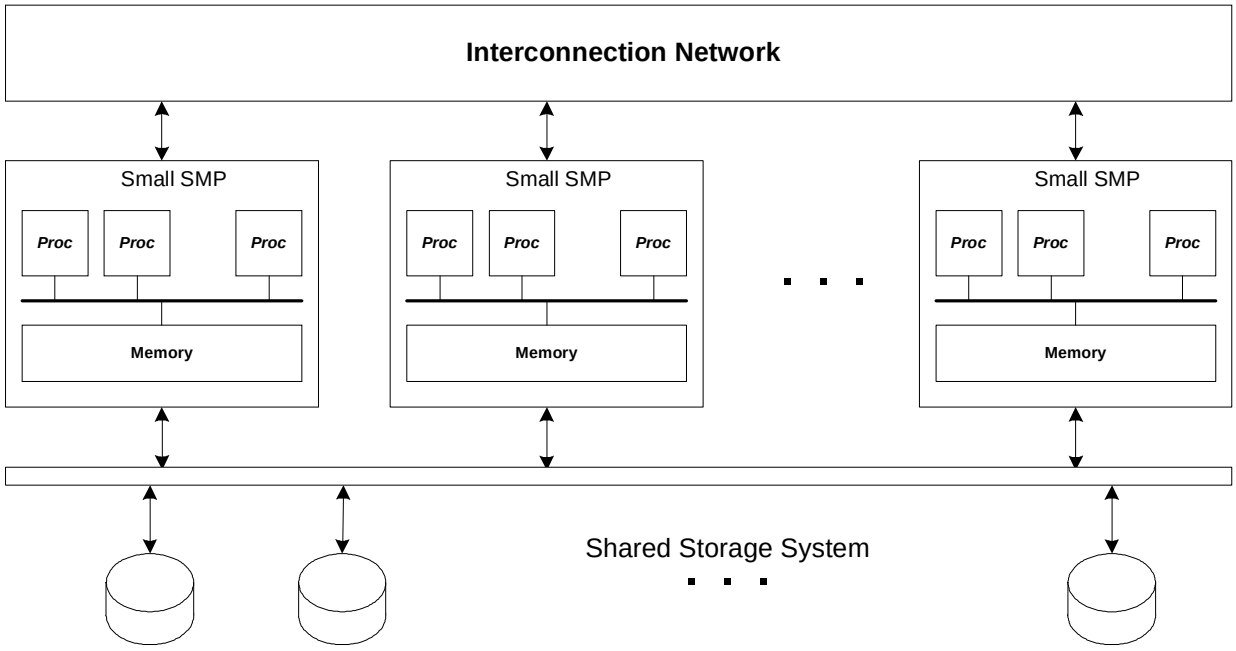


Figure 17. A cluster system combines small SMPs to form a large multiprocessor system. Cluster systems often share storage for large databases.

Now, let's throw packaging considerations into the mix. By *packaging*, we mean the physical/mechanical structures(s) that hold computing elements. Packaging is usually done in a hierarchical fashion. First, computing elements consisting of a large number of transistors are placed on an integrated circuit chip. Then, multiple chips may be placed on a circuit board (or they may first be combined into a multi-chip module (MCM) which is then placed on a board). Multiple boards can be placed in a chassis, and multiple chassis can be interconnected and situated together in a single room.

Latencies, i.e., interconnection delays, as well as power consumption tend to increase significantly at package boundaries, for example, when going from one chip to another or from some board to another. Furthermore, the width of a datapath may be more constrained at a package boundary. For example, the number of pins on a chip may limit the datapath widths, and therefore limit the bandwidth going on/off the chip.

To consider the effect of packaging on system architecture and resource sharing, consider a system of recent vintage where a single processor, along with its L1 and L2 cache are packaged on the same chip (see Figure 18a). One could opt for a shared, non-uniform L2 cache, which would add to complexity, or one could simply use separate non-shared L2 caches. A large L3 cache, is packaged separately, and, because all the processors must cross a package boundary to access it, the processor chips would be given uniform access to the shared L3 cache.

In contrast, if multiple cores with their caches are packaged on the same chip (see Figure 18b), then it would be much easier to implement a single, shared uniform access L2 cache on the chip, with a shared external L3 cache, again with uniform access delay. Note that in this example, we keep the L1 caches separate even though they are on the same chip. Although this leads to non-optimal resource sharing, it is often done for performance reasons. The L1 cache should have very low latency for good performance, and, as such, the L1 cache is usually deeply embedded within the processor core – it is essentially designed as if it is part of the processor rather than a separate memory. Here the trade-off is that low latency is more important than efficient sharing.

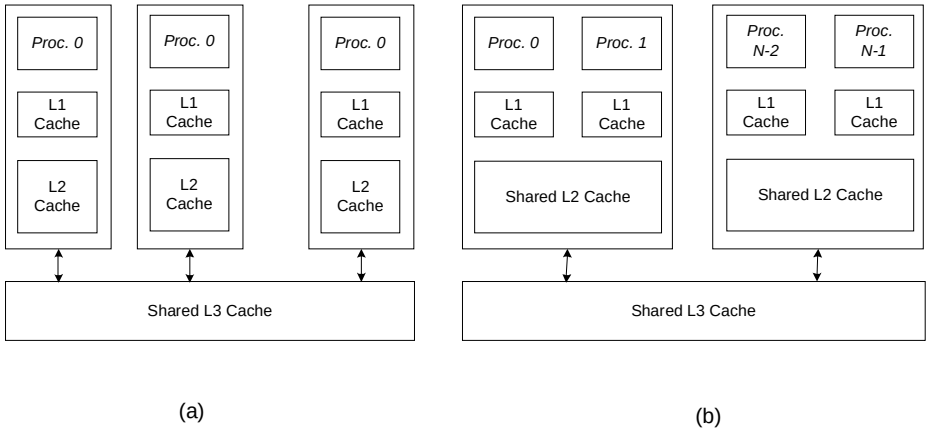


Figure 18. Packaging can affect sharing. a) A multiprocessor with single processors per package sharing memory resources (L3 cache) external to the package. b) multiple processors on the same package uniformly share memory resource internal to the package.

1.7 Summary: Multi-core MPs

We will now combine the performance characteristics, resource sharing, and cost considerations to arrive at the system architectures that (at least currently) are appropriate for multi-core chips.

Client systems contain a single, multi-core chip. Initially, these systems have a shared L2 cache on the same chip. The low cost desktop systems have no off-chip L3 ache; higher cost systems may have an off-chip L3 cache. Main memory consists of a number of chips dictated by price points, as it has always been. The display, disk cost etc., are pretty much the same as for single core chips. Really, the only thing that changes with multi-core chips is that a multi-core chip is put into a single microprocessor socket rather than a single-core chip. The microarchitecture issues, then, are related to the way the cores and caches are organized within the confines of a single chip. As technology advances, the number of cores on a chip can be expected to increase, and, at some point, it will likely become advantageous to place an L3 cache on the chip (because in the desktop regime there may be diminished returns from adding cores beyond some point).

Servers -- Servers, at least the relatively small ones, are SMPs that fit on a single board. The board contains a small number of sockets for microprocessor chips. Hence, as with client systems, the number of cores per board increases by virtue of multiple cores per chip. There may be a shared L3 cache external to the processor caches (as in Figure 18b). Here the main architecture issue is the larger scale of system and on-chip versus off-chip communication latencies and bandwidths, as well as the sharing of the external cache memory.

In the server domain, where throughput is often a primary performance consideration, multi-threaded processors come into play. Here the tradeoff is between single-thread performance versus aggregate performance. At one extreme is a large number of very simple cores that support large numbers of threads very efficiently. At the other is a small number of superscalar cores.

Clusters (**Large Servers)** – These are multi-board systems. Because of volume considerations, however, they tend to be interconnected versions of the small servers. Because each board has its own main memory, there is a choice between systems with NUMA and a common address space, or clustered systems with separate address spaces. If sharing is used, then design complexity is introduced, and this complexity would have to be paid at the individual board level, thereby increasing the cost of the small servers. Consequently, most of these systems do not share main memory; rather, they use the cluster approach.

The bandwidth of the cluster's LAN depends to some extent on the amount of program parallelism versus throughput parallelism that is expected in the application software. In a database or web server, the LAN will likely be an off-the shelf network. In the case of a compute server, the boards may be clustered w/ special packaging and a higher performance network; possibly, even a proprietary network.

For large servers, even though the total number of servers sold may be small when compared with desktops, the total number of boards is still be relatively large. Also, networking small servers does not take advantage of another form of resource sharing: power and cooling. Hence, in some servers the boards may be specially designed to share power supplies, cooling technology, etc. Then, power and cooling can be supplied at the chassis level, with the boards plugging into a backplane for power supplies and interconnection. Systems of this type are called "blade" servers because the individual server boards come in thin packages.

Entertainment – These are heterogeneous MP systems where media processing is placed on the same chip as a general purpose core. Here, the sales volumes are very high, and the system can take on a special-purpose nature, especially for graphics and other media processing.

1.8 The Rest of the Book

In this book, we consider the spectrum of systems based on multi-core integrated circuits. The next chapter looks at the major levels of abstraction that sit above the hardware. These start at programming models and go down to the instruction set architecture. We focus on instructions and other functionality related to multi-threaded execution. Then in chapter 3, cores are the topic of discussion – in a book on multi-core systems, it is appropriate to consider the design options for cores. Both in-order and out-of-order superscalar cores are described, as well as both single and multi-threaded cores Chapter 4 discusses memory implementations. This includes both main memory and cache memories. Then, in the next three chapters we look at system architectures. Chapter 5 focuses on client systems as a vehicle for talking about on-chip organization of multi-core systems. Chapter 6 focuses on servers and considers the interconnected multi-core chips. Chapter 7 discusses larger scale servers, or clusters, that are implemented by connecting shared memory servers with communication networks. These three chapters include numerous case studies of real systems. Finally, Chapter 8 (or an appendix) delves more deeply into memory coherence implementations and explores transactional memory – an approach intended to enable better multi-threaded software.

1.9 References

1. D. Kuck and B. Kumar, "A System Model for Computer Performance Evaluation," ACM SIGMETRICS 1976, pp. 187-199.
2. K. Olukotun, et al., "The Case for a Single-Chip Multiprocessor," ASPLOS-7, October 1996.
3. J. D. Davis, et al., "Maximizing CMT Throughput with Mediocre Cores," In Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, Sep. 2005, pages 51-62.
4. S. Eyerman, et al. "A Mechanistic Model for Superscalar Processors," 2007.
5. A. Hartstein et al., "On the Nature of Cache Miss Behavior: Is It √2 ?" Journal of Instruction Level Parallelism, July 2006, pp. 1-21.
6. A. Hartstein and T. Puzak, "The Optimum Pipeline Depth for a Microprocessor," 29th Int. Symp. on Computer Architecture, June 2002, pp. 7-13.
7. A Hartstein et al., "Optimal Memory Hierarchy", unpublished paper, 2007.
8. G. Amdahl, "The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," Spring Joint Computer Conference, 1967, pp. 483-485.
9. I. Tuomi, "The Lives and Death of Moore's Law," *First Monday*, Nov. 2002.
10. E. Berndt et al., "Price and Quality of Desktop and Mobile Personal Computers: A Quarter Century of Progress," National Bureau of Economic Research Summer Institute 2000, Nov. 2000.
11. P. Michaud, A. Seznec, S. Jourdan, "Exploring Instruction-Fetch Bandwidth Requirement in Wide-Issue Superscalar Processors", *International Conference on Parallel Architectures and Compilation Techniques*, Oct. 12-16, 1999.

12. P. Michaud, A. Seznec, S. Jourdan, "An Exploration of Instruction Fetch Requirement in Out-of-Order Superscalar Processors", *International Journal of Parallel Programming*, Vol.29 No.1, Feb. 2001.