CS2354 ADVANCED COMPUTER ARCHITECTURE L T P C
3 0 0 3

UNIT I INSTRUCTION LEVEL PARALLELISM 9
ILP - Concepts and challenges - Hardware and software approaches – Dynamic scheduling - Speculation - Compiler techniques for exposing ILP - Branch prediction.

UNIT II MULTIPLE ISSUE PROCESSORS 9
VLIW & EPIC - Advanced compiler support - Hardware support for exposing parallelism - Hardware versus software speculation mechanisms - IA 64 and Itanium processors - Limits on ILP.

UNIT III MULTIPROCESSORS AND THREAD LEVEL PARALLELISM 9
Symmetric and distributed shared memory architectures - Performance issues - Synchronization - Models of memory consistency - Introduction to Multithreading.

UNIT IV MEMORY AND I/O 9
Cache performance - Reducing cache miss penalty and miss rate - Reducing hit time - Main memory and performance - Memory technology. Types of storage devices - Buses - RAID - Reliability, availability and dependability - I/O performance measures - Designing an I/O system.

UNIT V MULTI-CORE ARCHITECTURES 9
Software and hardware multithreading - SMT and CMP architectures - Design issues - Case studies - Intel Multi-core architecture - SUN CMP architecture - heterogeneous multi-core processors - case study: IBM Cell Processor.

TOTAL: 45 PERIODS

TEXT BOOK:
1. John L. Hennessey and David A. Patterson, “Computer architecture - A quantitative approach”, Morgan Kaufmann / Elsevier Publishers, 4th. Edition, 2007.

REFERENCES:
1. David E. Culler, Jaswinder Pal Singh, “Parallel computing architecture: A hardware/software approach” , Morgan Kaufmann /Elsevier Publishers, 1999.
2. Kai Hwang and Zhi.Wei Xu, “Scalable Parallel Computing”, Tata McGraw Hill, New Delhi, 2003.

UNIT I

INSTRUCTION LEVEL PARALLELISM

ILP - Concepts and challenges - Hardware and software approaches – Dynamic scheduling - Speculation - Compiler techniques for exposing ILP - Branch prediction.

1.1 Fundamentals of Computer Design

Computer technology has made incredible progress in the roughly from last 55 years. This rapid rate of improvement has come both from advances in the technology used to build computers and from innovation in computer design. During the first 25 years of electronic computers, both forces made a major contribution; but beginning in about 1970, computer designers became largely dependent upon integrated circuit technology.

During the 1970s, performance continued to improve at about 25% to 30% per year for the mainframes and minicomputers that dominated the industry.

The late 1970s after invention of microprocessor the growth roughly increased 35% per year in performance. This growth rate, combined with the cost advantages of a mass-produced microprocessor, led to an increasing fraction of the computer business. In addition, two significant changes are observed in computer industry.

- First, the virtual elimination of assembly language programming reduced the need for object-code compatibility.
- Second, the creation of standardized, vendor-independent operating systems, such as UNIX and its clone, Linux, lowered the cost and risk of bringing out a new architecture.

These changes made it possible to successfully develop a new set of architectures, called RISC (Reduced Instruction Set Computer) architectures. In the early 1980s. The RISC-based machines focused the attention of designers on two critical performance techniques, the exploitation of instruction-level parallelism and the use of caches. The combination of architectural and organizational enhancements has led to 20 years of sustained growth in performance at an annual rate of over 50%. Figure 1.1 shows the effect of this difference in performance growth rates.

The effect of this dramatic growth rate has been twofold.

- First, it has significantly enhanced the capability available to computer users. For many applications, the highest performance microprocessors of today outperform the supercomputer of less than 10 years ago.
- Second, this dramatic rate of improvement has led to the dominance of micro-

processor-based computers across the entire range of the computer design.

1.1.1 Technology Trends

The changes in the computer applications space over the last decade have dramatically changed the metrics. Desktop computers remain focused on optimizing cost-performance as measured by a single user, servers focus on availability, scalability, and throughput cost-performance, and embedded computers are driven by price and often power issues.

If an instruction set architecture is to be successful, it must be designed to survive rapid changes in computer technology. An architect must plan for technology changes that can increase the lifetime of a computer.

The following Four implementation technologies changed the computer industry:

Integrated circuit logic technology

Transistor density increases by about 35% per year, and die size increases 10% to 20% per year. The combined effect is a growth rate in transistor count on a chip of about 55% per year.

Semiconductor DRAM:

Density increases by between 40% and 60% per year and Cycle time has improved very slowly, decreasing by about one-third in 10 years. Bandwidth per chip increases about twice as fast as latency decreases. In addition, changes to the DRAM interface have also improved the bandwidth.

Magnetic disk technology:

it is improving more than 100% per year. Prior to 1990, density increased by about 30% per year, doubling in three years. It appears that disk technology will continue the faster density growth rate for some time to come. Access time has improved by one-third in 10 years.

Network technology:

Network performance depends both on the performance of switches and on the performance of the transmission system, both latency and bandwidth can be improved, though recently bandwidth has been the primary focus. For many years, networking technology appeared to improve slowly: for example, it took about 10 years for Ethernet technology to move from 10 Mb to 100 Mb. The increased importance of networking has led to a faster rate of progress with 1 Gb Ethernet becoming available about five years after 100 Mb.

These rapidly changing technologies impact the design of a microprocessor that may, with speed and technology enhancements, have a lifetime of five or more years.

Scaling of Transistor Performance, Wires, and Power in Integrated Circuits

Integrated circuit processes are characterized by the feature size, which is decreased from 10 microns in 1971 to 0.18 microns in 2001. Since a transistor is a 2-dimensional object, the density of transistors increases quadratically with a linear decrease in feature size. The increase in transistor performance, this combination of scaling factors leads to a complex interrelationship between transistor performance and process feature size.

First approximation, transistor performance improves linearly with decreasing feature size. In the early days of microprocessors, the higher rate of improvement in density was used to quickly move from 4-bit, to 8bit, to 16-bit, to 32-bit microprocessors. More recently, density improvements have supported the introduction of 64-bit microprocessors as well as many of the innovations in pipelining and caches.

The signal delay for a wire increases in proportion to the product of its resistance and capacitance. As feature size shrinks wires get shorter, but the resistance and capacitance per unit length gets worse. Since both resistance and capacitance depend on detailed aspects of the process, the geometry of a wire, the loading on a wire, and even the adjacency to other structures. In the past few years, wire delay has become a major design limitation for large integrated circuits and is often more critical than transistor switching delay. Larger and larger fractions of the clock cycle have been consumed by the propagation delay of signals on wires. In 2001, the Pentium 4 broke new ground by allocating two stages of its 20+ stage pipeline just for propagating signals across the chip.

Power also provides challenges as devices are scaled. For modern CMOS microprocessors, the dominant energy consumption is in switching transistors. The energy required per transistor is proportional to the product of the load capacitance of the transistor, the frequency of switching, and the square of the voltage. As we move from one process to the next, the increase in the number of transistors switching and the frequency with which they switch, dominates the decrease in load capacitance and voltage, leading to an overall growth in power consumption.

1.1.2 Cost, Price and their Trends

In the past 15 years, the use of technology improvements to achieve lower cost, as well as increased performance, has been a major theme in the computer industry.

• Price is what you sell a finished good for,

• Cost is the amount spent to produce it, including overhead.

The Impact of Time, Volume, Commodification, and Packaging

The cost of a manufactured computer component decreases over time even without major improvements in the basic implementation technology. The underlying principle that drives costs down is the learning curve manufacturing costs decrease over time. As an example of the learning curve in action, the price per megabyte of DRAM drops over the long term by 40% per year.

The Microprocessor prices also drop over time, but because they are less standardized than DRAMs, the relationship between price and cost is more complex. In a period of significant competition, price tends to track cost closely

The Volume is a second key factor in determining cost. Increasing volumes affect cost in several ways.

• First, they decrease the time needed to get down the learning curve, which is partly proportional to the number of systems (or chips) manufactured.

• Second, volume decreases cost, since it increases purchasing and manufacturing efficiency.

As a rule of thumb, some designers have estimated that cost decreases about 10% for each doubling of volume.

The Commodities are products that are sold by multiple vendors in large volumes and are essentially identical. Virtually all the products sold on the shelves of grocery stores are commodities, as are standard DRAMs, disks, monitors, and keyboards. In the past 10 years, much of the low end of the computer business has become a commodity business focused on building IBM-compatible PCs. There are a variety of vendors that ship virtually identical products and are highly competitive. Of course, this competition decreases the gap between cost and selling price, but it also decreases cost.

Cost of an Integrated Circuit:

The cost of packaged integrated circuit is

Cost of die + Cost of testing die + Costof packaging and final testCost of integrated circuit=Final test yield

The number of good chips per wafer requires first learning how many dies fit on a wafer and then learning how to predict the percentage of those that will work. From there it is simple to predict cost:

Cost of waferCost of die=Dies per wafer × Die yield

The number of dies per wafer is basically the area of the wafer divided by the area of the die. It can be more accurately estimated by

2×(Wafer diameter/2) × Wafer diameterDies per wafer=Die area2xDieAreaππ−

The first term is the ratio of wafer area (πr^2) to die area. The second compensates for the "square peg in a round hole" problem rectangular dies near the periphery of round wafers. Dividing the circumference (πd) by the diagonal of a square die is approximately the number of dies along the edge. For example, a wafer 30 cm (≈ 12 inch) in diameter produces π× 225 – (π × 30 ⁄ 1.41) = 640 1-cm dies.

Cost Versus Price—Why They Differ and By How Much

Cost goes through a number of changes before it becomes price, and the computer designer should understand how a design decision will affect the potential selling price. For example, changing cost by $1000 may change price by $3000 to $4000.

The relationship between price and volume can increase the impact of changes in cost, especially at the low end of the market. Typically, fewer computers are sold as the price increases. Furthermore, as volume decreases, costs rise, leading to further increases in price.

Direct costs refer to the costs directly related to making a product. These include labor costs, purchasing components, scrap (the leftover from yield), and warranty. Direct cost typically adds 10% to 30% to component cost.

The next addition is called the gross margin, the company's overhead that cannot be billed directly to one product. This can be thought of as indirect cost. It includes the company's research and development (R&D), marketing, sales, manufacturing equipment maintenance, building rental, cost of financing, pretax profits, and taxes. When the component costs are added to the direct cost and gross margin,

Average selling price is the money that comes directly to the company for each product sold. The gross margin is typically 10% to 45% of the average selling price, depending on the uniqueness of the product. Manufacturers of low-end PCs have lower gross margins for several reasons. First, their R&D expenses are lower. Second, their cost of sales is lower, since they use indirect distribution by mail, the Internet, phone order, or retail store) rather than salespeople. Third, because their products are less unique, competition is more intense, thus forcing lower prices and often lower profits, which in turn lead to a lower gross margin.

List price and average selling price are not the same. One reason for this is that companies offer volume discounts, lowering the average selling price. As personal computers became commodity products, the retail mark-ups have dropped significantly, so list price and average selling price have closed.

1.2 Measuring and Reporting Performance

The computer user is interested in reducing response time(the time between the start and the completion of an event) also referred to as execution time. The manager of a large data processing center may be interested in increasing throughput(the total amount of work done in a given time).

Even execution time can be defined in different ways depending on what we count. The most straightforward definition of time is called wall-clock time, response time, or elapsed time, which is the latency to complete a task, including disk accesses, memory accesses, input/output activities, operating system overhead

Choosing Programs to Evaluate Performance

A computer user who runs the same programs day in and day out would be the perfect candidate to evaluate a new computer. To evaluate a new system the user would simply compare the execution time of her workload—the mixture of programs and operating system commands that users run on a machine.

There are five levels of programs used in such circumstances, listed below in decreasing order of accuracy of prediction.

1. Real applications— Although the buyer may not know what fraction of time is spent on these programs, she knows that some users will run them to solve real problems. Examples are compilers for C, text-processing software like Word, and other applications like Photoshop. Real applications have input, output, and options that a user can select when running the program. There is one major downside to using real applications as benchmarks: Real applications often encounter portability problems arising from dependences on the operating system or compiler. Enhancing portability often means modifying the source and sometimes eliminating some important activity, such as interactive graphics, which tends to be more system-dependent.

2. Modified (or scripted) applications—In many cases, real applications are used as the building block for a benchmark either with modifications to the application or with a script that acts as stimulus to the application. Applications are modified for two primary reasons: to enhance portability or to focus on one particular aspect of system performance. For example, to create a CPU-oriented benchmark, I/O may be removed or restructured to minimize its impact on execution time. Scripts are used to reproduce interactive behavior, which might occur on a desktop system, or to simulate complex multiuser interaction, which occurs in a server system.

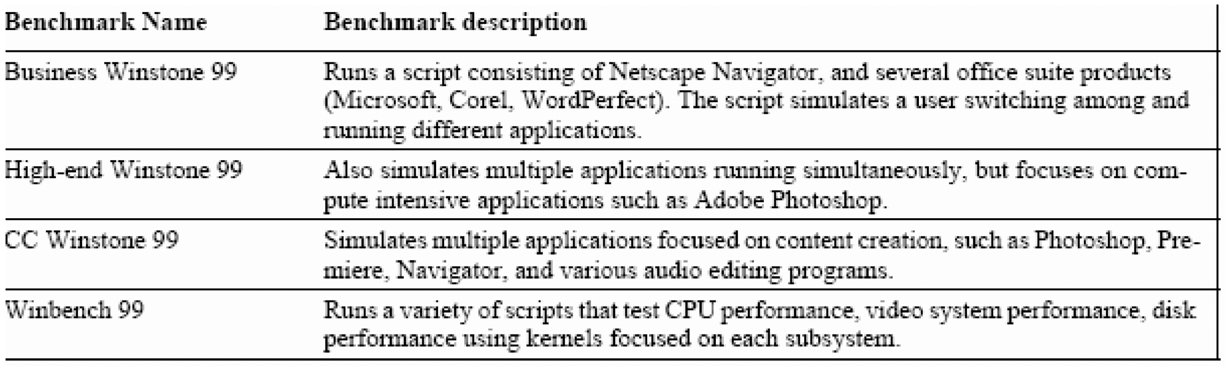
3. Kernels—Several attempts have been made to extract small, key pieces from real programs and use them to evaluate performance. Livermore Loops and Linpack are the best known examples. Unlike real programs, no user would run kernel programs, for they exist solely to evaluate performance. Kernels are best used to isolate performance of individual features of a machine to explain the reasons for differences in performance of real programs.

4. Toy benchmarks—Toy benchmarks are typically between 10 and 100 lines of code and produce a result the user already knows before running the toy program. Programs like Sieve of Eratosthenes, Puzzle, and Quicksort are popular because they are small, easy to type, and run on almost any computer. The best use of such programs is beginning programming assignments

5. Synthetic benchmarks—Similar in philosophy to kernels, synthetic benchmarks try to match the average frequency of operations and operands of a large set of programs. Whetstone and Dhrystone are the most popular synthetic benchmarks.

Benchmark Suites

Recently, it has become popular to put together collections of benchmarks to try to measure the performance of processors with a variety of applications. One of the most successful attempts to create standardized benchmark application suites has been the SPEC (Standard Performance Evaluation Corporation), which had its roots in the late 1980s efforts to deliver better benchmarks for workstations. Just as the computer industry has evolved over time, so has the need for different benchmark suites, and there are now SPEC benchmarks to cover different application classes, as well as other suites based on the SPEC model. Which is shown in figure



Benchmark Name	Benchmark description
Business Winstone 99	Runs a script consisting of Netscape Navigator, and several office suite products (Microsoft, Corel, WordPerfect). The script simulates a user switching among and running different applications.
High-end Winstone 99	Also simulates multiple applications running simultaneously, but focuses on compute intensive applications such as Adobe Photoshop.
CC Winstone 99	Simulates multiple applications focused on content creation, such as Photoshop, Premiere, Navigator, and various audio editing programs.
Winbench 99	Runs a variety of scripts that test CPU performance, video system performance, disk performance using kernels focused on each subsystem.

Desktop Benchmarks

Desktop benchmarks divide into two broad classes: CPU intensive benchmarks and graphics intensive benchmarks intensive CPU activity). SPEC originally created a benchmark set focusing on CPU performance (initially called SPEC89), which has evolved into its fourth generation: SPEC CPU2000, which follows SPEC95, and SPEC92.

Although SPEC CPU2000 is aimed at CPU performance, two different types of graphics benchmarks were created by SPEC: SPEC viewperf is used for benchmarking systems supporting the OpenGL graphics library, while SPECapc consists of applications that make extensive use of graphics. SPECviewperf measures the 3D rendering performance of systems running under OpenGL using a 3-D model and a series of OpenGL calls that transform the model. SPECapc consists of runs of three large applications:

1. Pro/Engineer: a solid modeling application that does extensive 3-D rendering. The input script is a model of a photocopying machine consisting of 370,000 triangles.

2. SolidWorks 99: a 3-D CAD/CAM design tool running a series of five tests varying from I/O intensive to CPU intensive. The largetest input is a model of an assembly line consisting of 276,000 triangles.

3. Unigraphics V15: The benchmark is based on an aircraft model and covers a wide spectrum of Unigraphics functionality, including assembly, drafting, numeric control machining, solid modeling, and optimization. The inputs are all part of an aircraft design.

Server Benchmarks

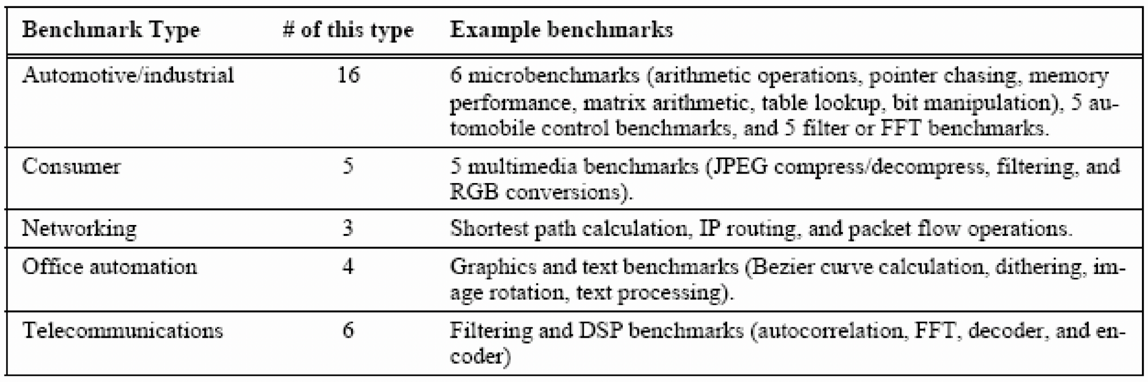
Just as servers have multiple functions, so there are multiple types of benchmarks. The simplest benchmark is perhaps a CPU throughput oriented benchmark. SPEC CPU2000 uses the SPEC CPU benchmarks to construct a simple throughput benchmark where the processing rate of a multiprocessor can be measured by running multiple copies (usually as many as there are CPUs) of each SPEC CPU benchmark and converting the CPU time into a rate. This leads to a measurement called the SPECRate. Other than SPECRate, most server applications and benchmarks have significant I/O activity arising from either disk or network traffic, including benchmarks for file server systems, for web servers, and for database and transaction processing systems. SPEC offers both a file server benchmark (SPECSFS) and a web server benchmark (SPECWeb). SPECSFS (see http://www.spec.org/osg/sfs93/) is a benchmark for measuring NFS (Network File System) performance using a script of file server requests; it tests the performance of the I/O system (both disk and network I/O) as well as the CPU. SPECSFS is a throughput oriented benchmark but with important response time requirements.

Transaction processing benchmarks measure the ability of a system to handle transactions, which consist of database accesses and updates. All the TPC benchmarks measure performance in transactions per second. In addition, they include a response-time requirement, so that throughput performance is measured only when the response time limit is met. To model real-world systems, higher transaction rates are also associated with larger systems, both in terms of users and the data base that the transactions are applied to. Finally, the system cost for a benchmark system must also be included, allowing accurate comparisons of cost-performance.

Embedded Benchmarks

Benchmarks for embedded computing systems are in a far more nascent state than those for either desktop or server environments. In fact, many manufacturers quote Dhrystone performance, a benchmark that was criticized and given up by desktop systems more than 10 years ago! As mentioned earlier, the enormous variety in embedded applications, as well as differences in performance requirements (hard real-time, soft real-time, and overall cost-performance), make the use of a single set of benchmarks unrealistic.

In practice, many designers of embedded systems devise benchmarks that reflect their application, either as kernels or as stand-alone versions of the entire application. For those embedded applications that can be characterized well by kernel performance, the best standardized set of benchmarks appears to be a new benchmark set: the EDN Embedded Microprocessor Benchmark Consortium (or EEMBC–pronounced embassy). The EEMBC benchmarks fall into five classes: automotive/industrial, consumer, networking, office automation, and telecommunications Figure shows the five different application classes, which include 34 benchmarks.



Benchmark Type	# of this type	Example benchmarks
Automotive/industrial	16	6 microbenchmarks (arithmetic operations, pointer chasing, memory performance, matrix arithmetic, table lookup, bit manipulation), 5 automobile control benchmarks, and 5 filter or FFT benchmarks.
Consumer	5	5 multimedia benchmarks (JPEG compress/decompress, filtering, and RGB conversions).
Networking	3	Shortest path calculation, IP routing, and packet flow operations.
Office automation	4	Graphics and text benchmarks (Bezier curve calculation, dithering, image rotation, text processing).
Telecommunications	6	Filtering and DSP benchmarks (autocorrelation, FFT, decoder, and encoder)

FIGURE 1.13 The EEMBC benchmark suite, consisting of 34 kernels in five different classes. See www.eembc.org for more information on the benchmarks and for scores.

1.3. Quantitative Principles of Computer Design

The most important and pervasive principle of computer design is to make the common case fast In applying this simple principle, we have to decide what the frequent case is and how much performance can be improved by making that case faster. A fundamental law, called Amdahl's Law, can be used to quantify this principle.

Amdahl's Law

The performance gain that can be obtained by improving some portion of a computer can be calculated using Amdahl's Law. Amdahl's Law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used. Amdahl's Law defines the speedup that can be gained by using a particular feature.

Speedup is the Ratio

$$\text{Speedup}= \frac{\text{Performance for entire task using enhancement when possible}}{\text{Performance for entire task without using enhancement}}$$

Alternatively,

Execution Time for entire task without using enhancement Speedup=Execution Time for entire task using enhancement when possible

Speedup tells us how much faster a task will run using the machine with the enhancement as opposed to the original machine. Amdahl's Law gives us a quick way to find the speedup from some enhancement, which depends on two factors:

1. The fraction of the computation time in the original machine that can be converted to take advantage of the enhancement—For example, if 20 seconds of the execution time of a program that takes 60 seconds in total can use an enhancement, the fraction is 20/60. This value, which we will call Fractionenhanced, is always less than or equal to 1.

2. The improvement gained by the enhanced execution mode; that is, how much faster the task would run if the enhanced mode were used for the entire program— This value is

the time of the original mode over the time of the enhanced mode: If the enhanced mode takes 2 seconds for some portion of the program that can completely use the mode, while the original mode took 5 seconds for the same portion, the improvement is 5/2. We will call this value, which is always greater than 1, Speedupenhanced.

The execution time using the original machine with the enhanced mode will be the time spent using the unenhanced portion of the machine plus the time spent using the enhancement:

Amdahl's Law can serve as a guide to how much an enhancement will improve performance and how to distribute resources to improve cost/performance.

The CPU Performance Equation

Essentially all computers are constructed using a clock running at a constant rate. These discrete time events are called ticks, clock ticks, clock periods, clocks, cycles, or clock cycles. Computer designers refer to the time of a clock period by its duration (e.g., 1 ns) or by its rate (e.g., 1 GHz). CPU time for a program can then be expressed

CPU Time = CPU Clock Cycles Per a Program X Clock Cycle Time

In addition to the number of clock cycles needed to execute a program, we can also count the number of instructions executed—the instruction path length or instruction count (IC). If we know the number of clock cycles and the instruction count we can calculate the average number of clock cycles per instruction (CPI).

CPI is computed as

$$\text{CPI} = \frac{\text{CPU Clock Cycles Per a Program}}{\text{Instruction Count}}$$

This allows us to use CPI in the execution time formula:

CPU time = Instruction count X Clock Cycle Time X Cycles per Instruction

Principle of Locality

Locality of reference means: Programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code. An implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past.

Locality of reference also applies to data accesses, though not as strongly as to code accesses. Two different types of locality have been observed. Temporal locality states that recently accessed items are likely to be accessed in the near future. Spatial locality says that items whose addresses are near one another tend to be referenced close together in time.

Advantage of Parallelism

Advantage of parallelism is one of the most important methods for improving performance. We give three brief examples, which are expounded on in later chapters. Our first example is the use of parallelism at the system level. To improve the throughput performance on a typical server benchmark, such as SPECWeb or TPC, multiple processors and multiple disks can be used. The workload of handling requests can then be spread among the CPUs or disks resulting in improved throughput. This is the reason that scalability is viewed as a valuable asset for server applications.

At the level of an individual processor, taking advantage of parallelism among instructions is critical to achieving high performance. This can be done to do this is through pipelining. The basic idea behind pipelining is to overlap the execution of instructions, so as to reduce the total time to complete a sequence of instructions. Viewed from the perspective of the CPU performance equation, we can think of pipelining as reducing the CPI by allowing instructions that take multiple cycles to overlap.

A key insight that allows pipelining to work is that not every instruction depends on its immediate predecessor, and thus, executing the instructions completely or partially in parallel may be possible.

Parallelism can also be exploited at the level of detailed digital design. For example set associative caches use multiple banks of memory that are typical searched in parallel to find a desired item. Modern ALUs use carry-lookahead, which uses parallelism to speed the process of computing sums from linear in the number of bits in the operands to logarithmic.

1.4 Instruction-Level Parallelism: Concepts and Challenges:

Instruction-level parallelism (ILP) is the potential overlap the execution of instructions using pipeline concept to improve performance of the system. The various techniques that are used to increase amount of parallelism are reduces the impact of data and control hazards and increases processor ability to exploit parallelism

There are two approaches to exploiting ILP.

1. Static Technique – Software Dependent

2. Dynamic Technique – Hardware Dependent

The simplest and most common way to increase the amount of parallelism is loop-level parallelism. Here is a simple example of a loop, which adds two 1000- element arrays, that is completely parallel:

```
for (i=1;i<=1000; i=i+1) x[i] = x[i] + y[i];
```

CPI (Cycles per Instruction) for a pipelined processor is the sum of the base CPI and all contributions from stalls:

Pipeline CPI = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls

The ideal pipeline CPI is a measure of the maximum performance attainable by the implementation. By reducing each of the terms of the right-hand side, we minimize the overall pipeline CPI and thus increase the IPC (Instructions per Clock).

Technique	Reduces
Forwarding and bypassing	Potential data hazard stalls
Delayed branches and simple branch scheduling	Control hazard stalls
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences
Dynamic scheduling with renaming	Data hazard stalls and stalls from anti dependences and output dependences
Dynamic branch prediction	Control stalls
Issuing multiple instructions per cycle	Ideal CPI
Speculation	Data hazard and control hazard stalls
Dynamic memory disambiguation	Data hazard stalls with memory
Loop unrolling	Control hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Compiler dependence analysis	Ideal CPI, data hazard stalls
Compiler speculation	Ideal CPI, data, control stalls

1.4.1 Various types of Dependences in ILP.

Data Dependence and Hazards:

To exploit instruction-level parallelism, determine which instructions can be executed in parallel. If two instructions are parallel, they can execute simultaneously in a pipeline without causing any stalls. If two instructions are dependent they are not parallel and must be executed in order.

There are three different types of dependences: data dependences (also called true data dependences), name dependences, and control dependences.

Data Dependences:

An instruction j is data dependent on instruction i if either of the following holds:

• Instruction i produces a result that may be used by instruction j, or
• Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.

The second condition simply states that one instruction is dependent on another if there exists a chain of dependences of the first type between the two instructions. This dependence chain can be as long as the entire program.

For example, consider the following code sequence that increment a vector of values in memory (starting at 0(R1) and with the last element at 8(R2)) by a scalar in register F2:

```
Loop: L.D        F0,0(R1) ;       F0=array element
      ADD.D      F4,F0,F2 ;       add scalar in F2
      S.D        F4,0(R1) ;       store result
      DADDUI     R1,R1,#-8 ;      decrement pointer 8 bytes
      BNE        R1,R2,LOOP ; branch R1!=zero
```

The dependence implies that there would be a chain of one or more data hazards between the two instructions. Executing the instructions simultaneously will cause a processor with pipeline interlocks to detect a hazard and stall, thereby reducing or eliminating the overlap. Dependences are a property of programs.

Whether a given dependence results in an actual hazard being detected and whether that hazard actually causes a stall are properties of the pipeline organization. This difference is critical to understanding how instruction-level parallelism can be exploited.

The presence of the dependence indicates the potential for a hazard, but the actual hazard and the length of any stall is a property of the pipeline. The importance of the data dependences is that a dependence

(1) indicates the possibility of a hazard,

(2) Determines the order in which results must be calculated, and

(3) Sets an upper bound on how much parallelism can possibly be exploited.

Name Dependences

The name dependence occurs when two instructions use the same register or memory location, called a name, but there is no flow of data between the instructions associated with that name.

There are two types of name dependences between an instruction i that precede instruction j in program order:

• An antidependence between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. The original ordering must be preserved to ensure that i reads the correct value.

• An output dependence occurs when instruction i and instruction j write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j.

Both anti-dependences and output dependences are name dependences, as opposed to true data dependences, since there is no value being transmitted between the instructions. Since a name dependence is not a true dependence, instructions involved in a name dependence can execute simultaneously or be reordered, if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict.

This renaming can be more easily done for register operands, where it is called register renaming. Register renaming can be done either statically by a compiler or dynamically by the hardware. Before describing dependences arising from branches, let's examine the relationship between dependences and pipeline data hazards.

Control Dependences:

A control dependence determines the ordering of an instruction, i, with respect to a branch instruction so that the instruction i is executed in correct program order. Every

Instruction, except for those in the first basic block of the program, is control dependent on some set of branches, and, in general, these control dependences must be preserved to preserve program order. One of the simplest examples of a control dependence is the dependence of the statements in the "then" part of an if statement on the branch. For example, in the code segment:

```
if p1 { S1;
};
if p2 { S2;}
```

S1 is control dependent on p1, and S2is control dependent on p2 but not on p1. In general, there are two constraints imposed by control dependences:

1. An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch. For example, we cannot take an instruction from the then-portion of an if-statement and move it before the ifstatement.

2. An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch. For example, we cannot take a statement before the if-statement and move it into the then-portion.

Control dependence is preserved by two properties in a simple pipeline, First, instructions execute in program order. This ordering ensures that an instruction that occurs before a branch is executed before the branch. Second, the detection of control or branch hazards ensures that an instruction that is control dependent on a branch is not executed until the branch direction is known.

1.4.2 Data Hazard and various hazards in ILP.

Data Hazards

A hazard is created whenever there is dependence between instructions, and they are close enough that the overlap caused by pipelining, or other reordering of instructions, would change the order of access to the operand involved in the dependence.

Because of the dependence, preserve order that the instructions would execute in, if executed sequentially one at a time as determined by the original source program. The goal of both our software and hardware techniques is to exploit parallelism by preserving program order only where it affects the outcome of the program. Detecting and avoiding hazards ensures that necessary program order is preserved.

Data hazards may be classified as one of three types, depending on the order of read and write accesses in the instructions.

Consider two instructions i and j, with i occurring before j in program order. The possible data hazards are **RAW (read after write)** — j tries to read a source before i write it, so j incorrectly gets the old value. This hazard is the most common type and corresponds to true data dependence. Program order must be preserved to ensure that j receives the value from i. In the simple common five-stage static pipeline a load instruction followed by an integer ALU instruction that directly uses the load result will lead to a RAW hazard.

WAW (write after write)

j tries to write an operand before it is written by i. The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination. This hazard corresponds to output dependence. WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled. The classic five-stage integer pipeline writes a register only in the WB stage and avoids this class of hazards.

WAR (write after read)

j tries to write a destination before it is read by i, so i incorrectly gets the new value. This hazard arises from antidependence. WAR hazards cannot occur in most static issue pipelines even deeper pipelines or floating point pipelines because all reads are early (in ID) and all writes are late (in WB). A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline, and other instructions that read a source late in the pipeline or when instructions are reordered.

1.5. Dynamic Scheduling

Overcoming Data Hazards with Dynamic Scheduling:

The Dynamic Scheduling is used handle some cases when dependences are unknown at a compile time. In which the hardware rearranges the instruction execution to reduce the stalls while maintaining data flow and exception behavior.

It also allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline. Although a dynamically scheduled processor cannot change the data flow, it tries to avoid stalling when dependences, which could generate hazards, are present.

Dynamic Scheduling:

A major limitation of the simple pipelining techniques is that they all use in-order instruction issue and execution: Instructions are issued in program order and if an instruction is stalled in the pipeline, no later instructions can proceed. Thus, if there is a dependence between two closely spaced instructions in the pipeline, this will lead to a hazard and a stall. If there are multiple functional units, these units could lie idle. If instruction j depends on a long-running instruction i, currently in execution in the pipeline, then all instructions after j must be stalled until i is finished and j can execute.

For example, consider this code:

```
DIV.D    F0,F2,F4
ADD.D    F10, F0, F8
SUB.D    F12, F8, F14
```

Out-of-order execution introduces the possibility of WAR and WAW hazards, which do not exist in the five-stage integer pipeline and its logical extension to an in-order floating-point pipeline.

Out-of-order completion also creates major complications in handling exceptions. Dynamic scheduling with out-of-order completion must preserve exception behavior in the sense that exactly those exceptions that would arise if the program were executed in strict program order actually do arise.

Imprecise exceptions can occur because of two possibilities:

1. The pipeline may have already completed instructions that are later in program order than the instruction causing the exception, and

2. The pipeline may have not yet completed some instructions that are earlier in program order than the instruction causing the exception.

To allow out-of-order execution, we essentially split the ID pipe stage of our simple five-stage pipeline into two stages:

1. Issue—Decode instructions, check for structural hazards.

2. Read operands—Wait until no data hazards, then read operands.

In a dynamically scheduled pipeline, all instructions pass through the issue stage in order (inorder issue); however, they can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order.

Score-boarding is a technique for allowing instructions to execute out-of-order when there are sufficient resources and no data dependences; it is named after the CDC 6600 scoreboard, which developed this capability. We focus on a more sophisticated technique, called Tomasulo's algorithm that has several major enhancements over scoreboarding.

1.5.1 Dynamic Scheduling Using Tomasulo's Approach :

This scheme was invented by RobertTomasulo, and was first used in the IBM 360/91. it uses register renaming to eliminate output and anti-dependencies, i.e. WAW and WAR hazards. Output and anti-dependencies are just name dependencies; there is no actual data dependence

Tomasulo's algorithm implements register renaming through the use of what are called reservation stations. Reservation stations are buffers which fetch and store instruction operands as soon as they are available.

In addition, pending instructions designate the reservation station that will provide their input. Finally, when successive writes to a register overlap in execution, only the last one is actually used to update the register. As instructions are issued, the register specifies for pending operands are renamed to the names of the reservation station, which provides register renaming.

The basic structure of a Tomasulo-based MIPS processor, including both the floating-point unit and the load/store unit.

Instructions are sent from the instruction unit into the instruction queue from which they are issued in FIFO order.

The reservation stations include the operation and the actual operands, as well as information used for detecting and resolving hazards. Load buffers have three functions: hold the components of the effective address until it is computed, track outstanding loads that are waiting on the memory, and hold the results of completed loads that are waiting for the CDB.

Similarly, store buffers have three functions: hold the components of the effective until it is computed, hold the destination memory addresses of outstanding stores that are waiting for the data value to store, and hold the address and value to store until the memory unit is available.

All results from either the FP units or the load unit are put on the CDB, which goes to the FP register file as well as to the reservation stations and store buffers. The FP adders implement addition and subtraction, and the FP multipliers do multiplication and division.

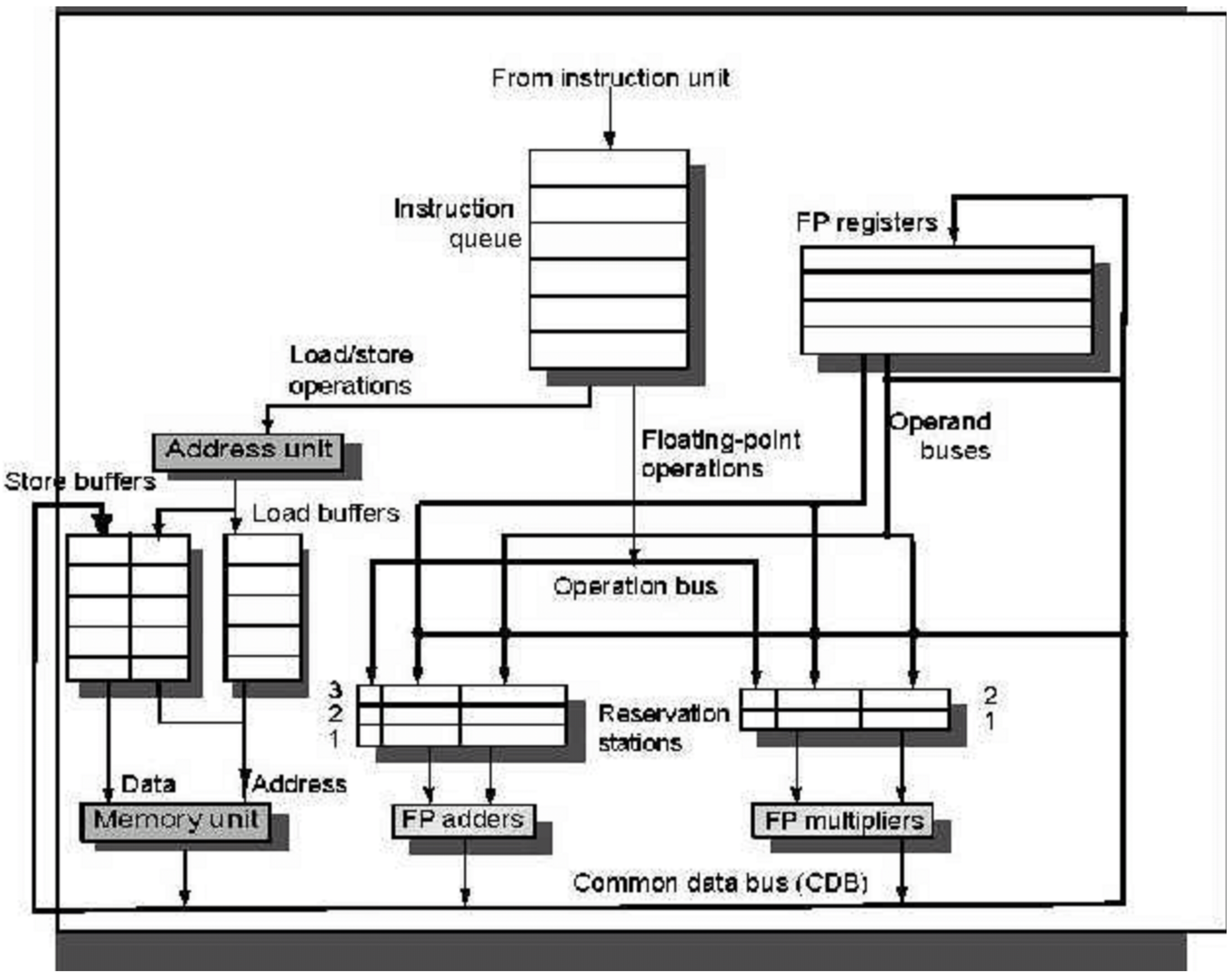


FIGURE 1.1 The basic structure of a MIPS floating point unit using Tomasulo's algorithm.

There are only three steps in Tomasulo's Aprroach :

1. **Issue—Get the next instruction from the head of the instruction queue. If there is a matching reservation station that is empty, issue the instruction to the station with the operand values (renames registers)**

2. Execute (EX) — When all the operands are available, place into the corresponding reservation stations for execution. If operands are not yet available, monitor the common data bus (CDB) while waiting for it to be computed.

3. Write result (WB)—When the result is available, write it on the CDB and from there into the registers and into any reservation stations (including store buffers) waiting for this result. Stores also write data to memory during this step: When both the address and data value are available, they are sent to the memory unit and the store completes.

Each reservation station has six fields:

- Op—The operation to perform on source operands S1 and S2.
- Qj, Qk—The reservation stations that will produce the corresponding source operand; a value of zero indicates that the source operand is already available in Vj or Vk, or is unnecessary.
- Vj, Vk—The value of the source operands. Note that only one of the V field or the Q field is valid for each operand. For loads, the Vk field is used to the offset from the instruction.
- A–used to hold information for the memory address calculation for a load or store.
- Busy—Indicates that this reservation station and its accompanying functional unit are occupied.

1.6 Hardware speculation

Hardware-based speculation combines three key ideas: dynamic branch prediction to choose which instructions to execute, speculation to allow the execution of instructions before the control dependences are resolved and dynamic scheduling to deal with the scheduling of different combinations of basic blocks.

Hardware-based speculation follows the predicted flow of data values to choose when to execute instructions. This method of executing programs is essentially a data-flow execution: operations execute as soon as their operands are available.

The approach is implemented in a number of processors (PowerPC 603/604/G3/G4, MIPS R10000/R12000, Intel Pentium II/III/ 4, Alpha 21264, and AMD

K5/K6/Athlon), is to implement speculative execution based on Tomasulo's algorithm.

The key idea behind implementing speculation is to allow instructions to execute out of order but to force them to commit in order and to prevent any irrevocable action until an instruction commits.

In the simple single-issue five-stage pipeline we could ensure that instructions committed in order, and only after any exceptions for that instruction had been detected, simply by moving writes to the end of the pipeline.

Here are the four steps involved in instruction execution:

1. Issue—Get an instruction from the instruction queue. Issue the instruction if there is an empty reservation station and an empty slot in the ROB, send the operands to the reservation station if they available in either the registers or the ROB for execution. If either all reservations are full or the ROB is full, then instruction issue is stalled until both have available entries. This stage is sometimes called dispatch in a dynamically scheduled processor.

2. Execute—If one or more of the operands is not yet available, monitor the CDB (common data bus) while waiting for the register to be computed. When both operands are available at a reservation station, execute the operation.

3. Write result—When the result is available, write it on the CDB and from the CDB into the ROB, as well as to any reservation stations waiting for this result. If the value to be stored is not available yet, the CDB must be monitored until that value is broadcast, at which time the Value field of the ROB entry of the store is updated.

4. Commit—There are three different sequences of actions at commit depending on whether the committing instruction is: a branch with an incorrect prediction, a store, or any other instruction (normal commit). The normal commit case occurs when an instruction reaches the head of the ROB and its result is present in the buffer; at this point, the processor updates the register with the result and removes the instruction from the ROB.

Committing a store is similar except that memory is updated rather than a result register. When a branch with incorrect prediction reaches the head of the ROB, it indicates that the speculation was wrong. The ROB is flushed and execution is restarted at the correct successor of the branch. If the branch was correctly predicted, the branch is finished. Some machines call this commit phase completion or graduation.

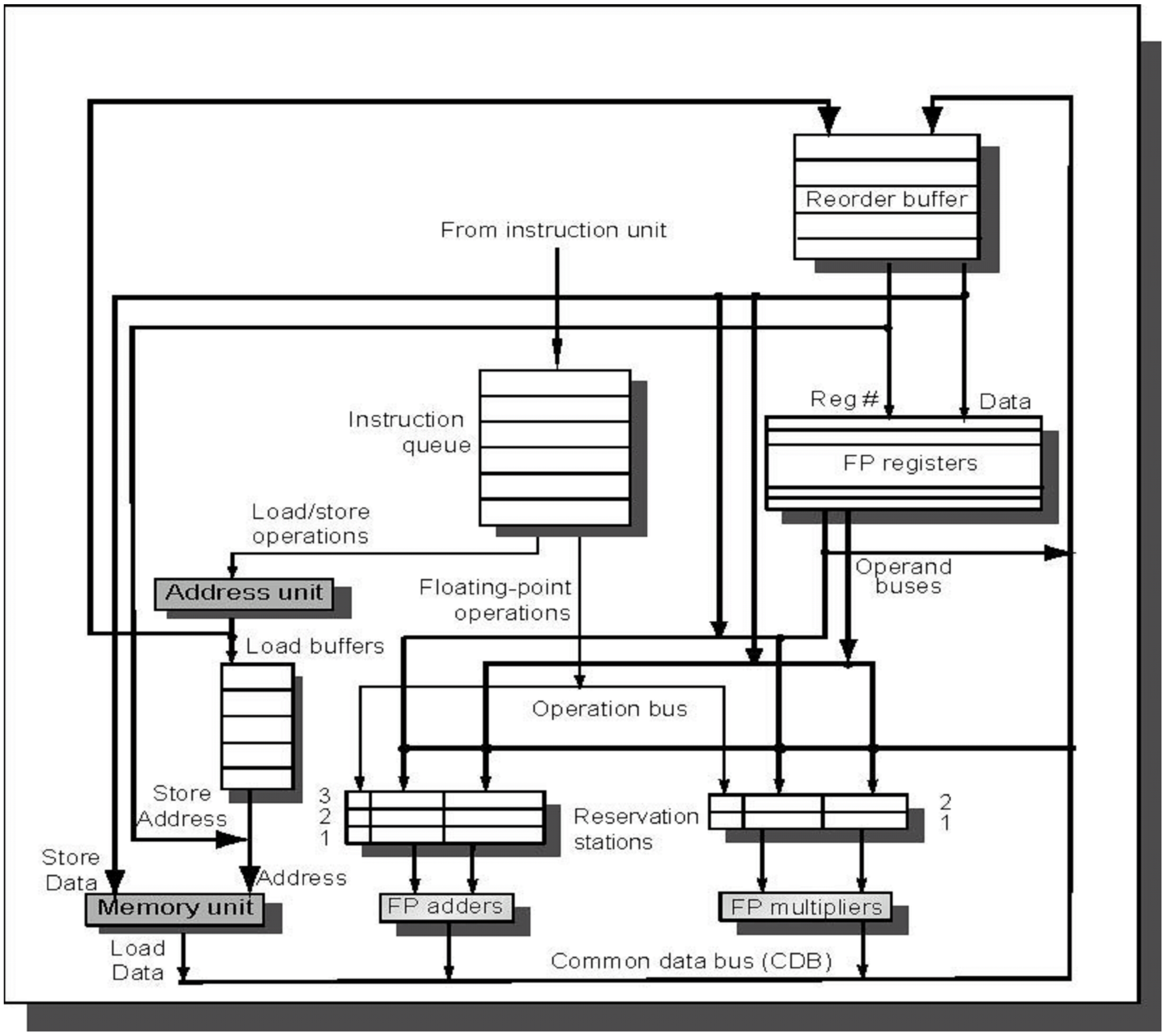


FIGURE 1.6 The basic structure of a MIPS FP unit using Tomasulo's algorithm and extended to handle speculation.

1.7 Compiler Techniques for Exposing ILP

1.7.1 Basic Pipeline Scheduling and Loop Unrolling

To avoid a pipeline stall, a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction.

A compiler's ability to perform this scheduling depends both on the amount of ILP available in the program and on the latencies of the functional units in the pipeline. Throughout this chapter we will assume the FP unit latencies shown in Figure 1.7

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

FIGURE 1.7 Latencies of FP operations used in this chapter.

The first column shows the originating instruction type. The second column is the type of the consuming instruction. The last column is the number of intervening clock cycles needed to avoid a stall.

We will rely on an example similar to the one we used in the last chapter, adding a scalar to a vector:

```
for (i=1000; i>0; i=i–1)
        x[i] = x[i] + s;
```

This loop is parallel by noticing that the body of each iteration is independent. The first step is to translate the above segment to MIPS assembly language. In the following code segment, R1is initially the address of the element in the array with the highest address, and F2 contains the scalar value, s. Register R2 is precomputed, so that 8(R2) is the last element to operate on. The straightforward MIPS code, not scheduled for the pipeline, looks like this:

```
Loop:  L.D       F0,0(R1)       ;F0=array element
       ADD.D     F4,F0,F2       ;add scalar in F2
       S.D       F4,0(R1)       ;store result
       DADDUI    R1,R1,#-8      ;decrement pointer
                                ;8 bytes (per DW)
       BNE       R1,R2,Loop     ;branch R1!=zero
```

Let’s start by seeing how well this loop will run when it is scheduled on a simple pipeline for MIPS with the latencies

```
                                    Clock cycle issued

Loop:  L.D           F0,0(R1)           1
Stall                                   2
ADD.D                F4,F0,F2           3
stall                                   4
stall                                   5
S.D                  F4,0(R1)           6
DADDUI               R1,R1,#-8          7
stall                                   8
```

```
        BNE          R1,R2,Loop      9
        stall                        10
```

This code requires 10 clock cycles per iteration. We can schedule the loop to obtain only one stall:

```
Loop:  L.D          F0,0(R1)
       DADDUI       R1,R1,#-8
       ADD.D        F4,F0,F2
       stall
       BNE          R1,R2,Loop    ;delayed branch
       S.D          F4,8(R1)      ;altered & interchanged with DADDUI
```

Execution time has been reduced from 10 clock cycles to 6.The stall after ADD.D is for the use by the S.D.

In the above example, we complete one loop iteration and store back one array element every 6 clock cycles, but the actual work of operating on the array element takes just 3 (the load, add, and store) of those 6 clock cycles. The remaining 3 clock cycles consist of loop overhead—the DADDUI and BNE—and a stall. To eliminate these 3 clock cycles we need to get more operations within the loop relative to the number of overhead instructions.

A simple scheme for increasing the number of instructions relative to the branch and overhead instructions is loop unrolling. Unrolling simply replicates the loop body multiple times, adjusting the loop termination code.

Loop unrolling can also be used to improve scheduling. Because it eliminates the branch, it allows instructions from different iterations to be scheduled together. In this case, we can eliminate the data use stall by creating additional independent instructions within the loop body. If we simply replicated the instructions when we unrolled the loop, the resulting use of the same registers could prevent us from effectively scheduling the loop. Thus, we will want to use different registers for each iteration, increasing the required register count.

In real programs we do not usually know the upper bound on the loop. Suppose it is n, and we would like to unroll the loop to make k copies of the body. Instead of a single unrolled loop, we generate a pair of consecutive loops. The first executes (n mod k) times and has a body that is the original loop. The second is the unrolled body surrounded by an outer loop that iterates (n/k) times. For large values of n, most of the execution time will be spent in the unrolled loop body.

1.7.2 Summary of the Loop Unrolling and Scheduling Example

To obtain the final unrolled code we had to make the following decisions and transformations:

Determine that it was legal to move the S.D after the DADDUI and BNE, and find the amount to adjust the S.D offset.

Determine that unrolling the loop would be useful by finding that the loop iterations were independent, except for the loop maintenance code.

Use different registers to avoid unnecessary constraints that would be forced by using the same registers for different computations.

Eliminate the extra test and branch instructions and adjust the loop termination and iteration code.

Determine that the loads and stores in the unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent. This transformation requires analyzing the memory addresses and finding that they do not refer to the same address. Unrolling is the growth in code size that results.

1.8. Branch Prediction method

1.8.1 Static Branch Prediction method

Static branch predictors are used in processors where the expectation is that branch behavior is highly predictable at compile-time; static prediction can also be used to assist dynamic predictors.

An architectural feature that supports static branch predication, namely delayed branches. Delayed branches expose a pipeline hazard so that the compiler can reduce the penalty associated with the hazard. The effectiveness of this technique partly depends on whether we correctly guess which way a branch will go. Being able to accurately predict a branch at compile time is also helpful for scheduling data hazards. Loop unrolling is on branches. Loop unrolling is on simple example of this; another example arises from conditional selection branches.

Consider the following code segment:

```
    LD       R1, 0(R2)
    DSUBU R1, R1, R3
    BEQZ    R1, L
    OR       R4, R5, R6
    DADDU   R10, R4, R3
L:  DADDU    R7, R8, R9
```

The dependence of the DSUBU and BEQZ on the LD instruction means that a stall will be needed after the LD. Suppose this branch was almost always taken and that the value of R7 was not needed on the fall-through path. Then we could increase the speed of the program by moving the instruction DADD R7,R8,R9 to the position after the LD. If the branch was rarely taken and that the value of R4 was not needed on the taken path, then we could contemplate moving the OR instruction after the LD.

To perform these optimizations, we need to predict the branch statically when we compile the program. There are several different methods to statically predict branch behavior. The simplest scheme is to predict a branch as taken. This scheme has an average misprediction rate that is equal to the untaken branch frequency, which for the SPEC programs is 34%.

Unfortunately, the misprediction rate ranges from not very accurate (59%) to highly accurate (9%).

A better alternative is to predict on the basis of branch direction, choosing backward-going branches to be taken and forward-going branches to be not taken. For some programs and compilation systems, the frequency of forward taken branches may be significantly less than 50%, and this scheme will do better than just predicting all branches as taken. In the SPEC programs, however, more than half of the forward-going branches are taken. Hence, predicting all branches as taken is the better approach.

A still more accurate technique is to predict branches on the basis of profile information collected from earlier runs. The key observation that makes this worthwhile is that the behavior of branches is often bimodally distributed; that is, an individual branch is often highly biased toward taken or untaken.

1.8.2 Reduce Branch Costs with Dynamic Hardware Prediction

1.8.2.1 Basic Branch Prediction and Branch-Prediction Buffers

- The simplest dynamic branch-prediction scheme is a branch-pr ediction buffer or branch history table.
- A branch-prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction.
- The memory contains a bit that says whether the branch was recently taken or not. if the prediction is correct—it may have been put there by another branch that has the same low-order address bits.
- The prediction is a hint that is assumed to be correct, and fetching begins in the predicted direction. If the hint turns out to be wrong, the prediction bit is inverted and stored back.
- The performance of the buffer depends on both how often the prediction is for the branch of interest and how accurate the prediction is when it matches.
- This simple one-bit prediction scheme has a performance shortcoming: Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken.

The two bits are used to encode the four states in the system. In a counter implementation, the counters are incremented when a branch is taken and decremented when it is not taken; the counters saturate at 00 or 11.

One complication of the two-bit scheme is that it updates the prediction bits more often than a one-bit predictor, which only updates the prediction bit on a mispredict. Since we typically read the prediction bits on every cycle, a two-bit predictor will typically need both a read and a write access port.

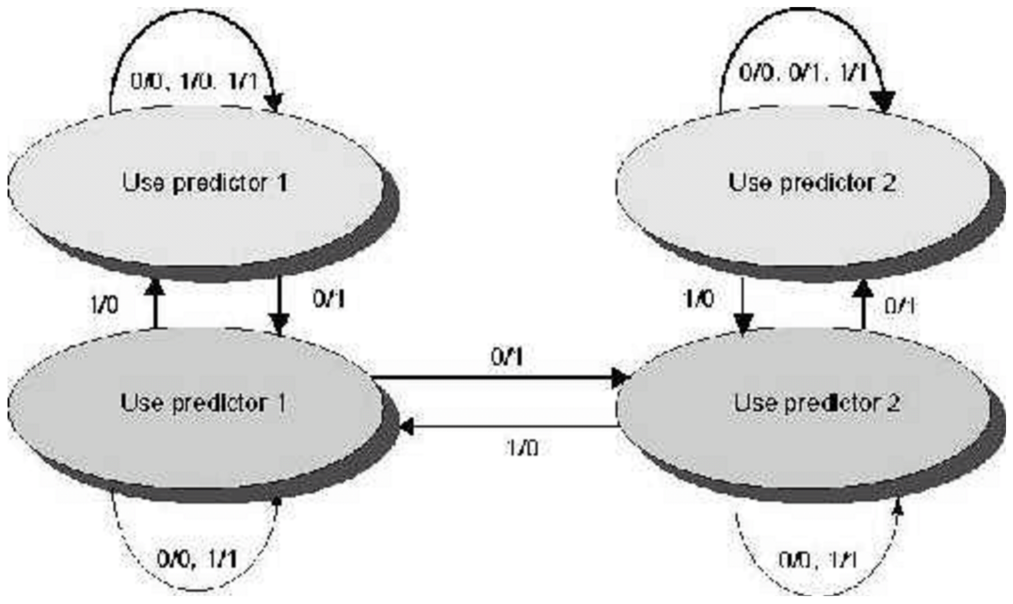


FIGURE 1.2 The states in a two-bit prediction scheme.

The two-bit scheme is actually a specialization of a more general scheme that has an n-bit saturating counter for each entry in the prediction buffer. With an n-bit counter, the counter can take on values between 0 and $2^n - 1$: when the counter is greater than or equal to one half of its maximum value (2^{n-1}), the branch is predicted as taken; otherwise, it is predicted untaken.

To exploit more ILP, the accuracy of our branch prediction becomes critical, this problem in two ways: by increasing the size of the buffer and by increasing the accuracy of the scheme we use for each prediction.

1.8.2.2 Correlating Branch Predictors:

These two-bit predictor schemes use only the recent behavior of a single branch to predict the future behavior of that branch. It may be possible to improve the prediction accuracy if we also look at the recent behavior of other branches rather than just the branch we are trying to predict.

Consider a small code fragment from the SPEC92 benchmark

```
if (aa==2)
aa=0;
if (bb==2)
bb=0;
if (aa!=bb) {
```

Here is the MIPS code that we would typically generate for this code fragment assuming that aa and bb are assigned to registers R1 and R2:

```
DSUBUI      R3, R1, #2
```

```
        BNEZ       R3, L1; branch b1 (aa! =2)
        DADD       R1, R0, R0; aa=0
L1:     DSUBUI     R3, R2, #2
        BNEZ       R3, L2; branch b2 (bb! =2)
        DADD       R2, R0, R0; bb=0
L2:     DSUBU      R3, R1, R2; R3=aa-bb
        BEQZ       R3, L3; branch b3 (aa==bb)
```

Let's label these branches b1, b2, and b3. The key observation is that the behavior of branch b3 is correlated with the behavior of branches b1 and b2. Clearly, if branches b1 and b2 are both not taken (i.e., the if conditions both evaluate to true and aa and bb are both assigned 0), then b3 will be taken, since aa and bb are clearly equal. A predictor that uses only the behavior of a single branch to predict the outcome of that branch can never capture this behavior.

Branch predictors that use the behavior of other branches to make a prediction are called correlating predictors or two-level predictors.

1.8.2.3 Tournament Predictors: Adaptively Combining Local and Global Predictors

The primary motivation for correlating branch predictors came from the observation that the standard 2-bit predictor using only local information failed on some important branches and that by adding global information, the performance could be improved. Tournament predictors take this insight to the next level, by using multiple predictors, usually one based on global information and one based on local information, and combining them with a selector. Tournament predictors can achieve both better accuracy at medium sizes (8Kb-32Kb) and also make use of very large numbers of prediction bits effectively.

Tournament predictors are the most popular form of multilevel branch predictors. A multilevel branch predictor use several levels of branch prediction tables together with an algorithm for choosing among the multiple predictors; Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors. The four states of the counter dictate whether to use predictor 1 or predictor 2. The state transition diagram is shown in Figure 1.3

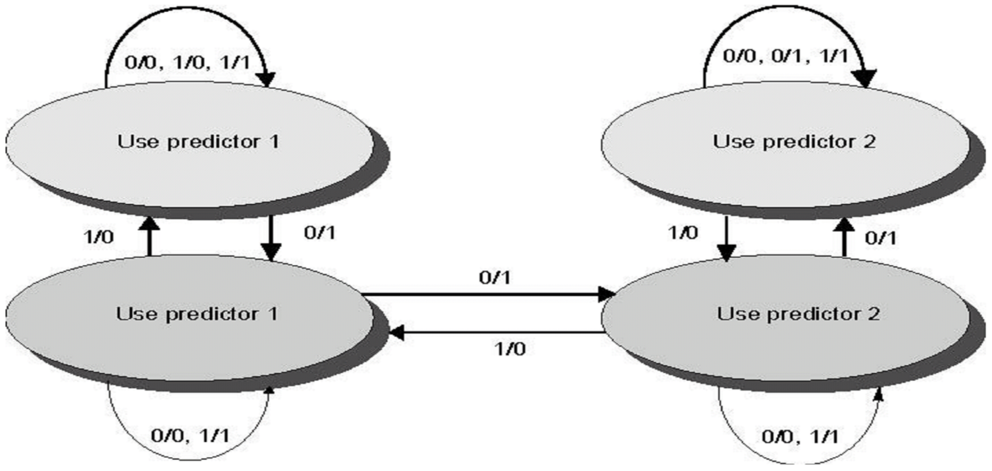


FIGURE 1.3 The state transition diagram for a tournament predictor has four states corresponding to which predictor to use

1.8.2.4 High Performance Instruction Delivery

Branch Target Buffers

A branch-prediction cache that stores the predicted address for the next instruction after a branch is called a branch-target buffer or branch-target cache.

For the classic, five-stage pipeline, a branch-prediction buffer is accessed during the ID cycle, so that at the end of ID we know the branch-target address (since it is computed during ID), the fall-through address (computed during IF), and the prediction. Thus, by the end of ID we know enough to fetch the next predicted instruction. For a branch-target buffer, we access the buffer during the IF stage using the instruction address of the fetched instruction, a possible branch, to index the buffer. If we get a hit, then we know the predicted instruction address at the end of the IF cycle, which is one cycle earlier than for a branch-prediction buffer.

Because we are predicting the next instruction address and will send it out before decoding the instruction, we must know whether the fetched instruction is predicted as a taken branch. Figure 1.4 shows what the branch-target buffer looks like. If the PC of the fetched instruction matches a PC in the buffer, then the corresponding predicted PC is used as the next PC.

If a matching entry is found in the branch-target buffer, fetching begins immediately at the predicted PC. Note that the entry must be for this instruction, because the predicted PC will be sent out before it is known whether this instruction is even a branch. If we did not check whether the entry matched this PC, then the wrong PC would be sent out for instructions that were not branches, resulting in a slower processor. We only need to store the predicted-taken branches in the branch-target buffer, since an untaken branch follows the same strategy as a non branch.

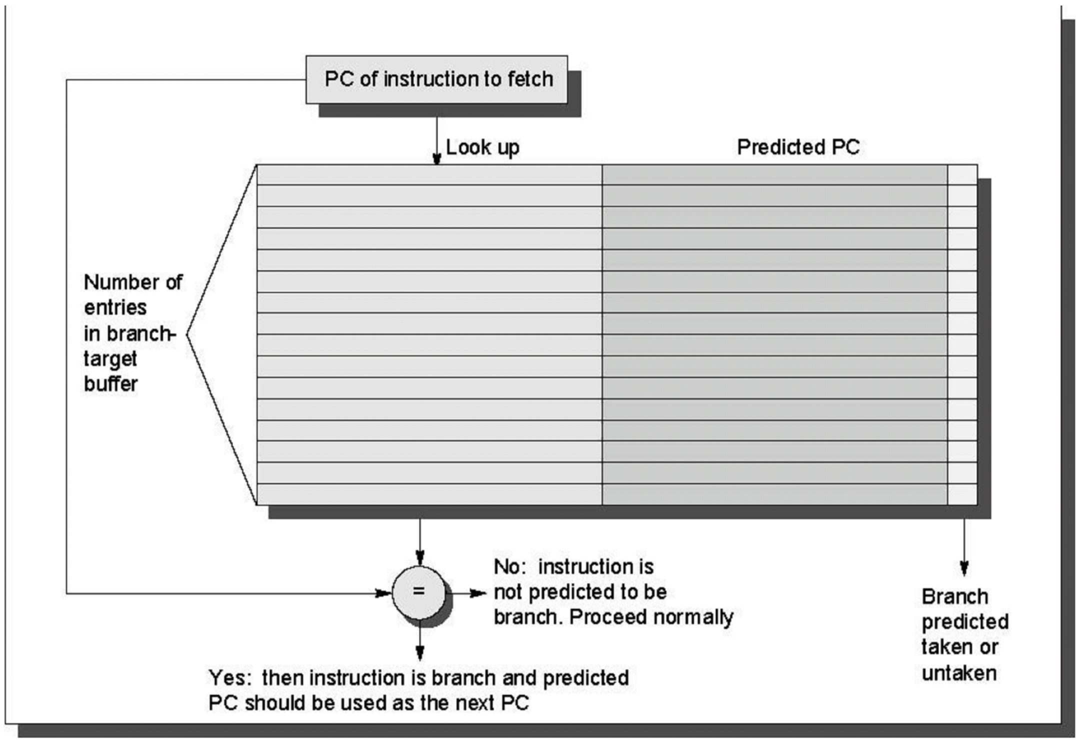

PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
No: instruction is not predicted to be branch. Proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken

FIGURE 1.4 A branch-target buffer.

Figure 1.5 shows the steps followed when using a branch-target buffer and where these steps occur in the pipeline. From this we can see that there will be no branch delay if a branch-prediction entry is found in the buffer and is correct.

The recent designs have used an integrated instruction fetch unit that integrates several functions:

1. Integrated branch prediction: the branch predictor becomes part of the instruction fetch unit and is constantly predicting branches, so to drive the fetch pipeline.
2. Instruction prefetch: to deliver multiple instructions per clock, the instruction fetch unit will likely need to fetch ahead. The unit autonomously manages the prefetching of instructions, integrating it with branch prediction.
3. Instruction memory access and buffering:.The instruction fetch unit also provides buffering, essentially acting as an on-demand unit to provide instructions to the issue stage as needed and in the quantity needed

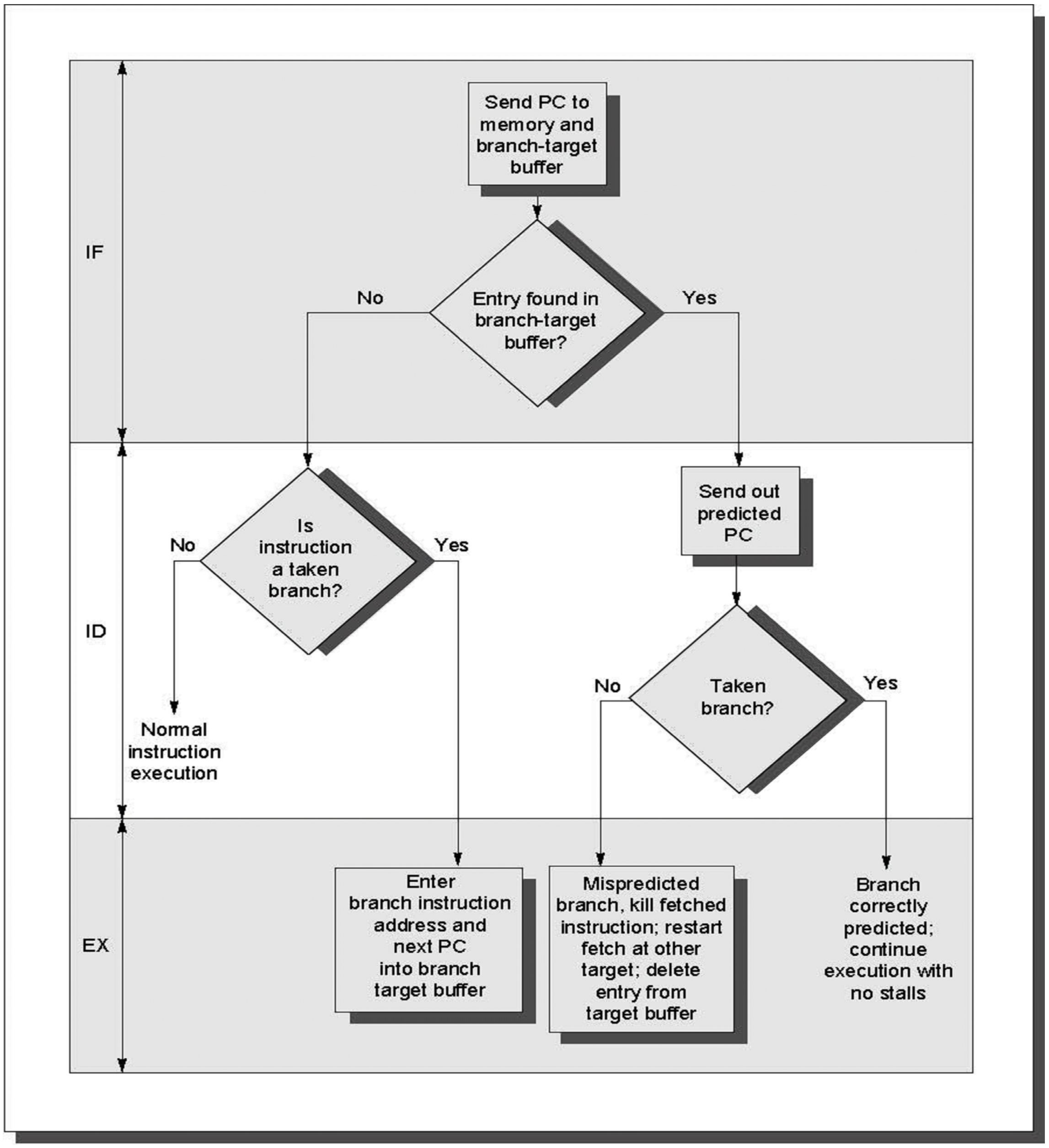

IF
ID
EX
Send PC to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
Is instruction a taken branch?
No
Yes
Normal instruction execution
Send out predicted PC
Taken branch?
No
Yes
Enter branch instruction address and next PC into branch target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls

FIGURE 1.5 The steps involved in handling an instruction with a branch-target buffer
Integrated Instruction Fetch Units

Return Address Predictors:

The concept of a small buffer of return addresses operating as a stack is used to predict the return address. This structure caches the most recent return addresses: pushing a return address on the stack at a call and popping one off at a return. If the cache is sufficiently large, it will predict the returns perfectly.

UNIT II MULTIPLE ISSUE PROCESSORS

VLIW & EPIC - Advanced compiler support - Hardware support for exposing parallelism - Hardware versus software speculation mechanisms - IA 64 and Itanium processors - Limits on ILP.

2. 1 VLIW Approach

The compiler may be required to ensure that dependences within the issue packet cannot be present or, at a minimum, indicate when a dependence may be present.

The first multiple-issue processors that required the instruction stream to be explicitly organized to avoid dependences. This architectural approach was named VLIW, standing for Very Long Instruction Word, and denoting that the instructions, since they contained several instructions, were very wide (64 to 128 bits, or more).

The basic architectural concepts and compiler technology are the same whether multiple operations are organized into a single instruction, or whether a set of instructions in an issue packet is preconfigured by a compiler to exclude dependent operations (since the issue packet can be thought of as a very large instruction). Early VLIWs were quite rigid in their instruction formats and effectively required recompilation of programs for different versions of the hardware.

VLIWs use multiple, independent functional units. Rather than attempting to issue multiple, independent instructions to the units, a VLIW packages the multiple operations into one very long instruction, or requires that the instructions in the issue packet satisfy the same constraints. we will assume that multiple operations are placed in one instruction, as in the original VLIW approach. Since the burden for choosing the instructions to be issued simultaneously falls on the compiler, the hardware in a superscalar to make these issue decisions is unneeded.

Since this advantage of a VLIW increases as the maximum issue rate grows, we focus on a wider-issue processor. Indeed, for simple two issue processors, the overhead of a superscalar is probably minimal. Because VLIW approaches make sense for wider processors, we choose to focus our example on such architecture.

For example, a VLIW processor might have instructions that contain five operations, including: one integer operation (which could also be a branch), two floating-point operations, and two memory references. The instruction would have a set of fields for each functional unit— perhaps 16 to 24 bits per unit, yielding an instruction length of between 112 and 168 bits.

To keep the functional units busy, there must be enough parallelism in a code sequence to fill the available operation slots. This parallelism is uncovered by unrolling loops and scheduling the code within the single larger loop body. If the unrolling generates straighline code, then local scheduling techniques, which operate on a single basic block, can be used. If finding and exploiting the parallelism requires scheduling code across branches, a substantially more complex global scheduling algorithm must be used.

Global scheduling algorithms are not only more complex in structure, but they must deal with significantly more complicated tradeoffs in optimization, since moving code across branches is expensive. Trace scheduling is one of these global scheduling techniques developed specifically for VLIWs.

Suppose we have a VLIW that could issue two memory references, two FP operations, and one integer operation or branch in every clock cycle. Show an unrolled version of the loop x[i] = x[i] +s (see page 223 for the MIPS ode) for such a processor. Unroll as many times as necessary to eliminate any stalls. Ignore the branch-delay slot.

The code is shown in Figure 2.1.The loop has been unrolled to make seven copies of the body, which eliminates all stalls (i.e., completely empty issue cycles), and runs in 9 cycles. This code yields a running rate of seven results in 9 cycles, or 1.29 cycles per result, nearly twice as fast as the two-issue superscalar that used unrolled and scheduled code.

Memory reference 1	**Memory reference 2**	**FP operation 1**	**FP operation 2**	**Integer operation/branch**
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D -8(R1),F8	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D -24(R1),F16			
S.D F20,-32(R1)	S.D -40(R1),F24			DADDUI R1,R1,#-56
S.D F28,8(R1)				BNE R1,R2,Loop

FIGURE 2.1 VLIW instructions that occupy the inner loop and replace the unrolled sequence.

For the original VLIW model, there are both technical and logistical problems. The technical problems are the increase in code size and the limitations of lock-step operation. Two different elements combine to increase code size substantially for a VLIW. First, generating enough operations in a straight-line code fragment requires ambitiously unrolling loops (as earlier examples) thereby increasing code size. Second, whenever instructions are not full, the unused functional units translate to wasted bits in the instruction encoding.

we saw that only about 60% of the functional units were used, so almost half of each instruction was empty. In most VLIWs, an instruction may need to be left completely empty if no operations can be scheduled.

Early VLIWs operated in lock-step; there was no hazard detection hardware at all. This structure dictated that a stall in any functional unit pipeline must cause the entire processor to stall, since all the functional units must be kept synchronized. Although a compiler may be able to schedule the deterministic functional units to prevent stalls, predicting which data accesses will encounter a cache stall and scheduling them is very difficult.

Hence, caches needed to be blocking and to cause all the functional units to stall. As the issue rate and number of memory references becomes large, this synchronization restriction becomes unacceptable. In more recent processors, the functional units operate more independently, and the compiler is used to avoid hazards at issue time, while hardware checks allow for unsynchronized execution once instructions are issued.

Binary code compatibility has also been a major logistical problem for VLIWs. In a strict VLIW approach, the code sequence makes use of both the instruction set definition and the detailed pipeline structure, including both functional units and their latencies.

One possible solution to this migration problem and the problem of binary code compatibility in general, is object-code translation or emulation. This technology is developing quickly and could play a significant role in future migration schemes. Another approach is to temper the strictness of the approach so that binary compatibility is still feasible. This later approach is used in the IA-64 architecture.

The major challenge for all multiple-issue processors is to try to exploit large amounts of ILP. When the parallelism comes from unrolling simple loops in FP programs, the original loop probably could have been run efficiently on a vector processor.

2.2 EPIC (Explicitly Parallel Instruction Computing)

EPIC permits microprocessors to execute software instructions in parallel by using the compiler, rather than complex on-die circuitry, to control parallel instruction execution.This was intended to allow simple performance scaling without resorting to higher clock frequencies.

It was the basis for Intel and HP development of the Intel Itanium architecture Intel/HP EPIC/IA-64 Architecure. EPIC is ISA philosophy approach. Very closely related to but not the same as VLIW.IA-64 an ISA definition. Intel's new 64-bit ISA. An EPIC type ISA

Itanium

A processor implementation of an ISA. The first implementation of the IA-64 ISA EPIC. EPIC design style Specifies ILP explicit in the machine code, that is, the parallelism is encoded directly into the instructions similarly to VLIW. A fully predicated instruction set. An inherent scalable instruction set. Many register Speculative execution of load instructions. EPIC instruction word contains three 41-bit instructions and a 5-bit control field.

EPIC design challenges

Develop architectures applicable to general-purpose computing. Find substantial parallelism in “difficult to parallellize” scalar programs. Provide compatibility across hardware generations. Support emerging applications. Compiler must find or create sufficient ILP. • Combine the best attributes of VLIW & superscalar RISC.Scale architectures for modern single-chip implementation

IA-64 EPIC Architecture

Instruction set architecture has128 64-bit integer registers + 128 82-bit floating points. Not separate register files per functional unit as in VLIW. Hardware checks dependencies (interlocks => binary compatibility over time).Predicated execution (select 1 out of 64 1-bit flags)

Instruction group is a sequence of consecutive instructions with no register data dependencies. All the instructions in a group could be executed in parallel, if sufficient hardware resources existed and if any dependence through memory were preserved

An instruction group can be arbitrarily long, but the compiler must explicitly indicate the boundary between one instruction group and another by placing a stop between 2 instructions that belong to different groups.IA-64 EPIC instructions are encoded in bundles, which are 128 bits wide

2.2.1 IA-64 EPIC vs VLIW

Similarities

- Compiler generated wide instructions
- Static detection of dependencies
- ILP encoded in the binary
- Large number of architected registers

Differences

- Instructions in a bundle can have dependencies
- Hardware interlock between dependent instructions
- Accommodates varying number of functional units and latencies
- Allows dynamic scheduling and functional unit binding
- Code size is reduced
- The same code can be executed on different processor implementations (ex: different number of functional units)
- Compiler detects ILP and indicates when an instruction cannot be executed in parallel with its
- successors

2.3 Hardware Support for Exposing More Parallelism at Compiler Time

Techniques such as loop unrolling, software pipelining, and trace scheduling can be used to increase the amount of parallelism available when the behavior of branches is fairly predictable at compile time. When the behavior of branches is not well known, compiler techniques alone may not be able to uncover much ILP. In such cases, the control dependences may severely limit the amount of parallelism that can be exploited. Similarly, potential dependences between memory reference instructions could prevent code movement that would increase available ILP. This section introduces several techniques that can help overcome such limitations.

The first is an extension of the instruction set to include conditional or predicated instructions. Such instructions can be used to eliminate branches converting a control dependence into a data dependence and potentially improving performance.

Hardware speculation with in-order commit preserved exception behavior by detecting and raising exceptions only at commit time when the instruction was no longer speculative. To enhance the ability of the compiler to speculatively move code over branches, while still preserving the exception behavior, we consider several different methods, which either include explicit checks for exceptions or techniques to ensure that only those exceptions that should arise are generated.

Finally, the hardware speculation schemes provided support for reordering loads and stores, by checking for potential address conflicts at runtime. To allow the compiler to reorder loads and stores when it suspects they do not conflict, but cannot be absolutely certain, a mechanism for checking for such conflicts can be added to the hardware. This mechanism permits additional opportunities for memory reference speculation.

2.3.1 Conditional or Predicated Instructions

The concept behind conditional instructions is quite simple: An instruction refers to a condition, which is evaluated as part of the instruction execution. If the condition is true, the instruction is executed normally; if the condition is false, the execution continues as if the instruction was a no-op. The most common example of such an instruction is conditional move, which moves a value from one register to another if the condition is true. Such an instruction can be used to completely eliminate a branch in simple code sequences.

Consider the following code:

```
if (A==0) {S=T;}
```

Assuming that registers R1, R2, and R3 hold the values of A, S, and T, respectively,

The straightforward code using a branch for this statement is

```
BNEZ R1, L ADDU R2, R3, R0
```

L: Using a conditional move that performs the move only if the third operand is equal to zero, we can implement this statement in one instruction:

```
CMOVZ R2, R3, R1
```

The conditional instruction allows us to convert the control dependence present in the branch-based code sequence to data dependence. For a pipelined processor, this moves the place where the dependence must be resolved from near the front of the pipeline, where it is resolved for branches, to the end of the pipeline where the register write occurs.

One obvious use for conditional move is to implement the absolute value function: A = abs (B), which is implemented as if (B<0) {A = - B;) else {A=B;}. This if statement can be implemented as a pair of conditional moves, or as one unconditional move (A=B) and one conditional move (A= - B).

In the example above or in the compilation of absolute value, conditional moves are used to change a control dependence into a data dependence. This enables us to eliminate the branch and possibly improve the pipeline behavior.

Conditional moves are the simplest form of conditional or predicated instructions, and although useful for short sequences, have limitations. In particular, using conditional move to eliminate branches that guard the execution of large blocks of code can be in efficient, since many conditional moves may need to be introduced.

To remedy the in efficiency of using conditional moves, some architectures support full predication, whereby the execution of all instructions is controlled by a predicate. When the predicate is false, the instruction becomes a no-op. Full predication allows us to simply convert large blocks of code that are branch dependent. For example, an if-then-else statement within a loop can be entirely converted to predicated execution, so that the code in the then-case executes only if the value of the condition is true, and the code in the else-case executes only if the value of the condition is false. Predication is particularly valuable with global code scheduling, since it can eliminate nonloop branches, which significantly complicate instruction scheduling.

Predicated instructions can also be used to speculatively move an instruction that is time-critical, but may cause an exception if moved before a guarding branch. Although it is possible to do this with conditional move, it is more costly.

Predicated or conditional instructions are extremely useful for implementing short alternative control flows, for eliminating some unpredictable branches, and for reducing the overhead of global code scheduling. Nonetheless, the usefulness of conditional instructions is limited by several factors:

- Predicated instructions that are annulled (i.e., whose conditions are false) still take some processor resources. An annulled predicated instruction requires fetch resources at a minimum, and in most processors functional unit execution time.
- Predicated instructions are most useful when the predicate can be evaluated early. If the condition evaluation and predicated instructions cannot be separated (because of data dependences in determining the condition), then a conditional instruction may result in a stall for a data hazard. With branch prediction and speculation, such stalls can be avoided, at least when the branches are predicted accurately.
- The use of conditional instructions can be limited when the control flow involves more than a simple alternative sequence. For example, moving an instruction across multiple branches requires making it conditional on both branches, which requires two conditions to be specified or requires additional instructions to compute the controlling predicate.
- Conditional instructions may have some speed penalty compared with unconditional instructions. This may show up as a higher cycle count for such instructions or a slower clock rate overall. If conditional instructions are more expensive, they will need to be used judiciously

For these reasons, many architectures have included a few simple conditional instructions (with conditional move being the most frequent), but only a few architectures include conditional versions for the majority of the instructions. The MIPS, Alpha, Power-PC, SPARC and Intel x86 (as defined in the Pentium processor) all support conditional move. The IA-64 architecture supports full predication for all instructions.

2.3.2 Compiler Speculation with Hardware Support

Many programs have branches that can be accurately predicted at compile time either from the program structure or by using a profile. In such cases, the compiler may want to speculate either to improve the scheduling or to increase the issue rate. Predicated instructions provide one method to speculate, but they are really more useful when control dependences can be completely eliminated by if-conversion. In many cases, we would like to move speculated instructions not only before branch, but before the condition evaluation, and predication cannot achieve this.

As pointed out earlier, to speculate ambitiously requires three capabilities:

1. The ability of the compiler to find instructions that, with the possible use of register renaming, can be speculatively moved and not affect the program data flow,

2. The ability to ignore exceptions in speculated instructions, until we know that such exceptions should really occur, and

3. The ability to speculatively interchange loads and stores, or stores and stores, which may have address conflicts.

The first of these is a compiler capability, while the last two require hardware support.

2.3.3 Hardware Support for Preserving Exception Behavior

There are four methods that have been investigated for supporting more ambitious speculation without introducing erroneous exception behavior:

1 The hardware and operating system cooperatively ignore exceptions for speculative instructions.

2 Speculative instructions that never raise exceptions are used, and checks are introduced to determine when an exception should occur.

3 A set of status bits, called poison bits, are attached to the result registers written by speculated instructions when the instructions cause exceptions. The poison bits cause a fault when a normal instruction attempts to use the register.

4 A mechanism is provided to indicate that an instruction is speculative and the hardware buffers the instruction result until it is certain that the instruction is no longer speculative.

To explain these schemes, we need to distinguish between exceptions that indicate a program error and would normally cause termination, such as a memory protection violation, and those that are handled and normally resumed, such as a page fault. Exceptions that can be resumed can be accepted and processed for speculative instructions just as if they were normal instructions.

If the speculative instruction should not have been executed, handling the unneeded exception may have some negative performance effects, but it cannot cause incorrect execution. The cost of these exceptions may be high, however, and some processors use hardware support to avoid taking such exceptions, just as processors with hardware speculation may take some exceptions in speculative mode, while avoiding others until an instruction is known not to be speculative.

Exceptions that indicate a program error should not occur in correct programs, and the result of a program that gets such an exception is not well defined, except perhaps when the program is running in a debugging mode. If such exceptions arise in speculated instructions, we cannot take the exception until we know that the instruction is no longer speculative.

In the simplest method for preserving exceptions, the hardware and the operating system simply handle all resumable exceptions when the exception occurs and simply return an undefined value for any exception that would cause termination.

A second approach to preserving exception behavior when speculating introduces speculative versions of instructions that do not generate terminating exceptions.

A third approach for preserving exception behavior tracks exceptions as they occur but postpones any terminating exception until a value is actually used, preserving the occurrence of the exception, although not in a completely precise fashion.

The fourth and final approach listed above relies on a hardware mechanism that operates like a reorder buffer. In such an approach, instructions are marked by the compiler as speculative and include an indicator of how many branches the instruction was speculatively moved across and what branch action (taken/not taken) the compiler assumed.

All instructions are placed in a reorder buffer when issued and are forced to commit in order, as in a hardware speculation approach. The reorder buffer tracks when instructions are ready to commit and delays the “write back” portion of any speculative instruction. Speculative instructions are not allowed to commit until the branches they have been speculatively moved over are also ready to commit, or, alternatively, until the corresponding sentinel is reached.

2.3.4 Hardware Support for Memory Reference Speculation

Moving loads across stores is usually done when the compiler is certain the addresses do not conflict. To allow the compiler to undertake such code motion, when it cannot be absolutely certain that such a movement is correct, a special instruction to check for address conflicts can be included in the architecture. The special instruction is left at the original location of the load instruction (and acts like a guardian) and the load is moved up across one or more stores.

When a speculated load is executed, the hardware saves the address of the accessed memory location. If a subsequent store changes the location before the check instruction, then the speculation has failed. If the location has not been touched then the speculation is successful. Speculation failure can be handled in two ways.

If only the load instruction was speculated, then it suffices to redo the load at the point of the check instruction Ifadditional instructions that depended on the load were also speculated, then a fix-up sequence that re-executes all the speculated instructions starting with the load is needed

2.4 Hardware versus Software Speculation Mechanisms

Hardware Speculation	Software Speculation
Dynamic runtime disambiguation of memory addresses is done using Tomasulo's algorithm. This disambiguation allows us to move loads past stores at runtime.	Dynamic runtime disambiguation of memory addresses is difficult to do at compile time for integer programs that contain pointers
Hardware-based speculation works better when control flow is unpredictable, and when hardware-based branch prediction is superior to software-based branch prediction done at compile time.	Hardware-based branch prediction is superior than software-based branch prediction done at compile time.
Hardware-based speculation maintains a completely precise exception model even for speculated instructions	Software-based approaches have added special support to allow this as well.
Hardware-based speculation does not require compensation or bookkeeping code.	Software-based speculation require compensation or Bookkeeping
The ability to see further in the code is very poor in Hardware based speculation	Compiler-based approaches may benefit from the ability to see further in the code sequence, resulting in better code scheduling than a purely hardware-driven approach.

2.5 IA-64 ARCHITECTURE

It is a RISC-style, register-register instruction set architecture.Designed to support compiler-based exploitation of ILP.

2.5.1 COMPONENTS OF IA-64 REGISTER STATE

- 128 64-Bit general purpose registers
- 128 82-Bit floating-point register (provides 2 extra bits over std. 80-bit IEEE format)
- 64 1-Bit predicate registers
- 8 64-Bit branch registers, used for indirect branches
- Various registers used for system control, memory mapping, performance counters, etc.

2.5.2 REGISTER MECHANISM

- 0-31 → Accessible Registers
- 32-128 → Used as a register stack
- CFM → Set of registers to be used by a given procedure. (Current FraMe Point)
- Integer register
- Floating point register
- Predicate register

2.5.3 REGISTER

A frame is created for a called procedure, by renaming the registers in hardware.

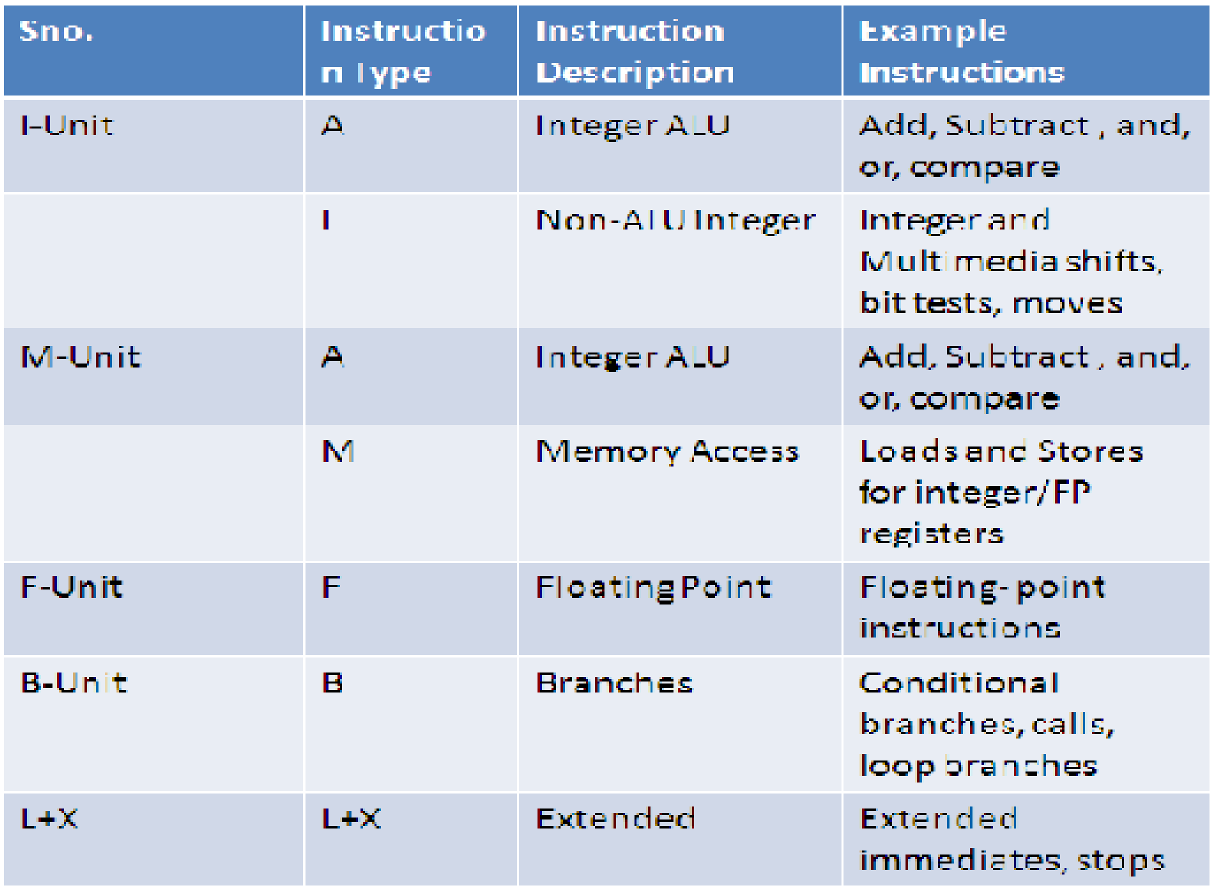
- Frame has local area and output area parts.
- The "alloc" instruction specifies the size of these areas.

To handle the over flow of the register stack, special h/w called the register stack engine is used.

2.5.4 INSTRUCTION FORMAT

- Supports for both Explicit Parallelism and Implicit Parallelism
- Benefits of VLIW approach-implicit parallelism among operations in an instruction and fixed formatting of the operation fields.
- It can be achieved by relying on the compiler to detect ILP and schedule instructions into parallel instruction slots.

2.5.4 FIVE EXECUTION UNIT SLOTS



Sno.	Instruction Type	Instruction Description	Example Instructions
I-Unit	A	Integer ALU	Add, Subtract, and, or, compare
	I	Non-ALU Integer	Integer and Multimedia shifts, bit tests, moves
M-Unit	A	Integer ALU	Add, Subtract, and, or, compare
	M	Memory Access	Loads and Stores for integer/FP registers
F-Unit	F	Floating Point	Floating-point instructions
B-Unit	B	Branches	Conditional branches, calls, loop branches
L+X	L+X	Extended	Extended immediates, stops

2.5.5 BENEFITS OF IA-64

1) Implicit parallelism
    - ➢ By placing instructions into instruction groups

2) Ease of Instruction decode
    - ➢ By bundle

3) Predication

4) Speculation

5) Memory Reference

2.5.6 Instruction Group:

- ➢ It is a sequence of consecutive instructions with no register data dependences among them.
- ➢ If sufficient hardware resources existed and if any dependences through memory were preserved, then all the instructions in a group could be executed in parallel.

2.5.7 Bundles

Each bundle consists of a 5-bit template field and three instructions, each 41 bits in length. Template field specifies what types of execution units each instruction in the bundle requires**.**

2.5.8 Predication

- ➢ An instruction is predicated by a predicate register, whose identity is placed in the lower 6 bits of each instruction field.
- ➢ Predicate registers are the set using compare and test instructions
- ➢ It allows multiple comparisons to be done in one instruction.

2.5.9 Speculation:

- ➢ It supports for control speculation
- ➢ That is deals with deferring exception for speculated instruction, memory reference speculation and thus supports speculation of load instructions.

2.5.10 Memory reference

- ➢ It uses the concept of advanced loads
- ➢ Advanced load is a load that has been speculatively moved above store instructions on which it is potentially dependent
- ➢ The instruction ld.a is used for advanced load, which is to speculatively perform a load.
- ➢ Execution creates an entry called ALAT (Advanced Load Address Table).
- ➢ ALAT stores both the register destination of the load and the address of the accessed memory location.

- When a store is executed, an associative lookup against the active ALAT entries is performed.

2.6 Limitations of ILP

2.6.1 The Hardware Model

An ideal processor is one where all artificial constraints on ILP are removed. The only limits on ILP in such a processor are those imposed by the actual data flows either through registers or memory.

The assumptions made for an ideal or perfect processor are as follows:

1. Register renaming—There are an infinite number of virtual registers available and hence all WAW and WAR hazards are avoided and an unbounded number of instructions can begin execution simultaneously.

2. Branch prediction—Branch prediction is perfect. All conditional branches are predicted exactly.

3. Jump prediction—All jumps (including jump register used for return and computed jumps) are perfectly predicted. When combined with perfect branch prediction, this is equivalent to having a processor with perfect speculation and an unbounded buffer of instructions available for execution.

4. Memory-address alias analysis—All memory addresses are known exactly and a load

can be moved before a store provided that the addresses are not identical.

Assumptions 2 and 3 eliminate all control dependences. Likewise, assumptions 1 and 4 eliminate all but the true data dependences. Together, these four assumptions mean that any instruction in the of the program's execution can be scheduled on the cycle immediately following the execution of the predecessor on which it depends.

2.6.2Limitations on the Window Size and Maximum Issue Count

A dynamic processor might be able to more closely match the amount of parallelism uncovered by our ideal processor.

consider what the perfect processor must do:

1. Look arbitrarily far ahead to find a set of instructions to issue, predicting all branches perfectly.

2. Rename all register uses to avoid WAR and WAW hazards.

3. Determine whether there are any data dependencies among the instructions in the issue packet; if so, rename accordingly.

4. Determine if any memory dependences exist among the issuing instructions and handle them appropriately.

5. Provide enough replicated functional units to allow all the ready instructions to issue.

Obviously, this analysis is quite complicated. For example, to determine whether n issuing instructions have any register dependences among them, assuming all instructions are register-register and the total number of registers is unbounded, requires

$2n-2+2n-4+\ldots\ldots+2 = 2\sum_{i=1}^{n-1} i = [2(n-1)n]/2 = n^2 - n$

Comparisons. Thus, to detect dependences among the next 2000 instructions—the default size we assume in several figures—requires almost four million comparisons! Even issuing only 50 instructions requires 2450 comparisons. This cost obviously limits the number of instructions that can be considered for issue at once.

2.6.3The Effects of Realistic Branch and Jump Prediction:

Our ideal processor assumes that branches can be perfectly predicted: The outcome of any branch in the program is known before the first instruction is executed.

The five levels of branch prediction shown in these figures are

1. Perfect—All branches and jumps are perfectly predicted at the start of execution.
2. Tournament-based branch predictor—The prediction scheme uses a correlating two-bit predictor and a noncorrelating two-bit predictor together with a selector, which chooses the best predictor for each branch.
3. Standard two-bit predictor with 512 two-bit entries—In addition, we assume a 16-entry buffer to predict returns.
4. Static—A static predictor uses the profile history of the program and predicts that the branch is always taken or always not taken based on the profile.
5. None—No branch prediction is used, though jumps are still predicted. Parallelism is largely limited to within a basic block.

2.6.4 Limitations on ILP for Realizable Processors

The performance of processors an ambitious level of hardware support equal to or better than what is likely in the next five years. In particular we assume the following fixed attributes:

1. Up to 64 instruction issues per clock with no issue restrictions. As we discuss later, the practical implications of very wide issue widths on clock rate, logic complexity, and power may be the most important limitation on exploiting ILP.
2. A tournament predictor with 1K entries and a 16-entry return predictor. This predictor is fairly comparable to the best predictors in 2000; the predictor is not a primary bottleneck.
3. Perfect disambiguation of memory references done dynamically—this is ambitious but perhaps attainable for small window sizes (and hence small issue rates and load/store buffers) or through a memory dependence predictor.
4. Register renaming with 64 additional integer and 64 additional FP registers,exceeding largest number available on any processor in 2001 (41 and 41 in the Alpha 21264), but probably easily reachable within two or three years.

UNIT III

MULTIPROCESSORS AND THREAD LEVEL PARALLELISM

Symmetric and distributed shared memory architectures - Performance issues - Synchronization - Models of memory consistency - Introduction to Multithreading.

3.1 Symmetric Shared Memory Architectures

The Symmetric Shared Memory Architecture consists of several processors with a single physical memory shared by all processors through a shared bus which is shown below.

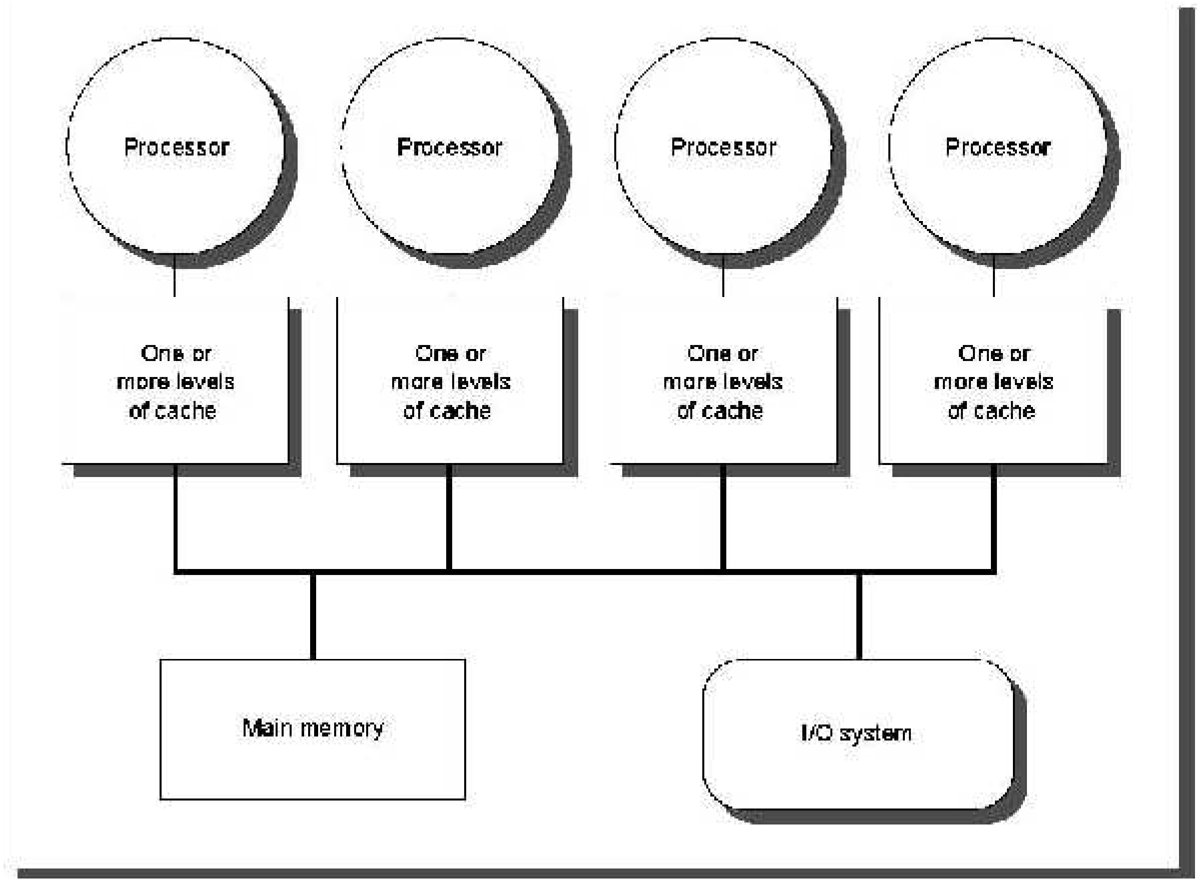

Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
I/O system

FIGURE 3.1 Symmetric Shared Memory Architecture

` Small-scale shared-memory machines usually support the caching of both shared and private data. Private data is used by a single processor, while shared data is used by multiple processors, essentially providing communication among the processors thro ugh reads and writes of the shared data. When a private item is cached, its location is migrated to the cache, reducing the average access time as well as the memory bandwidth required. Since no other processor uses the data, the program behavior is identical to that in a uniprocessor.

3.1.1 Cache Coherence in Multiprocessors:

Introduction of caches caused a coherence problem for I/O operations, The same problem exists in the case of multiprocessors, because the view of memory held by two different processors is through their individual caches.

The problem and shows how two different processors can have two different values for the same location. This difficulty is generally referred to as the cache-

Coherence problem.

		Cache contents for	**Cache contents for**	**Memory contents for**
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 into X	0	1	0

FIGURE 3.2 The cache-coherence problem for a single memory location (X), read and written by two processors (A and B).

We initially assume that neither cache contains the variable and that X has the value 1.We also assume a write-through cache; a write-back cache adds some additional but similar complications.

After the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not, and if B reads the value of X, it will receive 1!

Informally, we could say that a memory system is coherent if any read of a data item returns the most recently written value of that data item. This simple definition contains two different aspects of memory system behavior, both of which are critical to writing correct shared-memory programs.

The first aspect, called coherence, defines what values can be returned by a read. The second aspect, called consistency, determines when a written value will be returned by a read. Let's look at coherence first.

A memory system is coherent if

- A read by a processor, P, to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.
- A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.
- Writes to the same location are serialized: that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

Coherence and consistency are complementary: Coherence defines the behavior of reads and writes to the same memory location, while consistency defines the behavior of reads and writes with respect to accesses to other memory locations.

3.1.2 Basic Schemes for Enforcing Coherence

Coherent caches provide migration, since a data item can be moved to a local cache and used there in a transparent fashion. This migration reduces both the latency to access a shared data item that is allocated remotely and the bandwidth demand on the shared memory.

Coherent caches also provide replication for shared data that is being simultaneously read, since the caches make a copy of the data item in the local cache. Replication reduces both latency of access and contention for a read shared data item.

The protocols to maintain coherence for multiple processors are called cache-coherence protocols. There are two classes of protocols, which use different techniques to track the sharing status, in use:

Directory based—The sharing status of a block of physical memory is kept in just one location, called the directory; we focus on this approach in section 6.5, when we discuss scalable shared-memory architecture.

Snooping—Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, and no centralized state is kept. The caches are usually on a shared-memory bus, and all cache controllers monitor or snoop on the bus to determine whether or not they have a copy of a block that is requested on the bus.

3.1.3 Snooping Protocols

The method which ensures that a processor has exclusive access to a data item before it writes that item. This style of protocol is called a write invalidate protocol because it invalidates other copies on a write. It is by far the most common protocol, both for snooping and for directory schemes. Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: all other cached copies of the item are invalidated.

Since the write requires exclusive access, any copy held by the reading processor must be invalidated (hence the protocol name). Thus, when the read occurs, it misses in the cache and is forced to fetch a new copy of the data.

For a write, we require that the writing processor have exclusive access, preventing any other processor from being able to write simultaneously.

If two processors do attempt to write the same data simultaneously, one of them wins the race, causing the other processor's copy to be invalidated. For the other processor to complete its write, it must obtain a new copy of the data, which must now contain the updated value. Therefore, this protocol enforces write serialization.

Processor activity	**Bus activity**	**Contents of CPU A's cache**	**Contents of CPU B's cache**	**Contents of memory location X**
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

FIGURE 3.3 An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches.

The alternative to an invalidate protocol is to update all the cached copies of a data item when that item is written. This type of protocol is called a write update or writes broadcast protocol. Figure 6.8 shows an example of a write update protocol in operation. In the decade since these protocols were developed, invalidate has emerged as the winner for the vast majority of designs.

Processor activity	**Bus activity**	**Contents of CPU A's cache**	**Contents of CPU B's cache**	**Contents of memory location X**
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Write broadcast of X	1	1	1
CPU B reads X		1	1	1

FIGURE 3.4 An example of a write update or broadcast protocol working on a snooping bus for a single cache block (X) with write-back caches.

The performance differences between write update and write invalidate protocols arise from three characteristics:

- Multiple writes to the same word with no intervening reads require multiple write broadcasts in an update protocol, but only one initial invalidation in a write invalidate protocol.
- With multiword cache blocks, each word written in a cache block requires a write broadcast in an update protocol, although only the first write to any word in the block needs to generate an invalidate in an invalidation protocol. An invalidation protocol works on cache blocks, while an update protocol must work on individual words (or bytes, when bytes are written). It is possible to try to merge writes in a write broadcast scheme.
- The delay between writing a word in one processor and reading the written value in another processor is usually less in a write update scheme, since the written
- data are immediately updated in the reader's cache

3.1.4 Basic Implementation Techniques

The serialization of access enforced by the bus also forces serialization of writes, since when two processors compete to write to the same location, one must obtain bus access before the other. The first processor to obtain bus access will cause th e other processor's copy to be invalidated, causing writes to be strictly serialized. One implication of this scheme is that a write to a shared data item cannot complete until it obtains bus access.

For a write-back cache, however, the problem of finding the most recent data value is harder, since the most recent value of a data item can be in a cache rather than in memory. Happily, write-back caches can use the same snooping scheme both for caches misses and for writes: Each processor snoops every address placed on the bus. If a processor finds that it has a dirty copy of the requested cache block, it provides that cache block in response to the read request and causes the memory access to be aborted.

Since write-back caches generate lower requirements for memory bandwidth, they are greatly preferable in a multiprocessor, despite the slight increase in complexity. Therefore, we focus on implementation with write-back caches.

The normal cache tags can be used to implement the process of snooping, and the valid bit for each block makes invalidation easy to implement. Read misses, whether generated by an invalidation or by some other event, are also straightforward since they simply rely on the snooping capability. For writes we'd like to know whether any other copies of the block are cached, because, if there are no other cached copies, then the write need not be placed on the bus in a write-back cache. Not sending the write reduces both the time taken by the write and the required bandwidth.

3.2 Distributed Shared-Memory Architectures.

There are several disadvantages in Symmetric Shared Memory architectures.

- First, compiler mechanisms for transparent software cache coherence are very limited.
- Second, without cache coherence, the multiprocessor loses the advantage of being able to fetch and use multiple words in a single cache block for close to the cost of fetching one word.
- Third, mechanisms for tolerating latency such as prefetch are more useful when they can fetch multiple words, such as a cache block, and where the fetched data remain coherent; we will examine this advantage in more detail later.

These disadvantages are magnified by the large latency of access to remote memory versus a local cache. For these reasons, cache coherence is an accepted requirement in small-scale multiprocessors.

For larger-scale architectures, there are new challenges to extending the cache-coherent shared-memory model. Although the bus can certainly be replaced with a more scalable interconnection network and we could certainly distribute the memory so that the memory bandwidth could also be scaled, the lack of scalability of the snooping coherence scheme needs to be addressed is known as Distributed Shared Memory architecture.

The first coherence protocol is known as a directory protocol. A directory keeps the state of every block that may be cached. Information in the directory includes which caches have copies of the block, whether it is dirty, and so on.

To prevent the directory from becoming the bottleneck, directory entries can be distributed along with the memory, so that different directory accesses can go to different locations, just as different memory requests go to different memories. A distributed directory retains the characteristic that the sharing status of a block is always in a single known location. This property is what allows the coherence protocol to avoid broadcast.

Figure 3.5 shows how our distributed-memory multiprocessor looks with the directories

Added to each node.

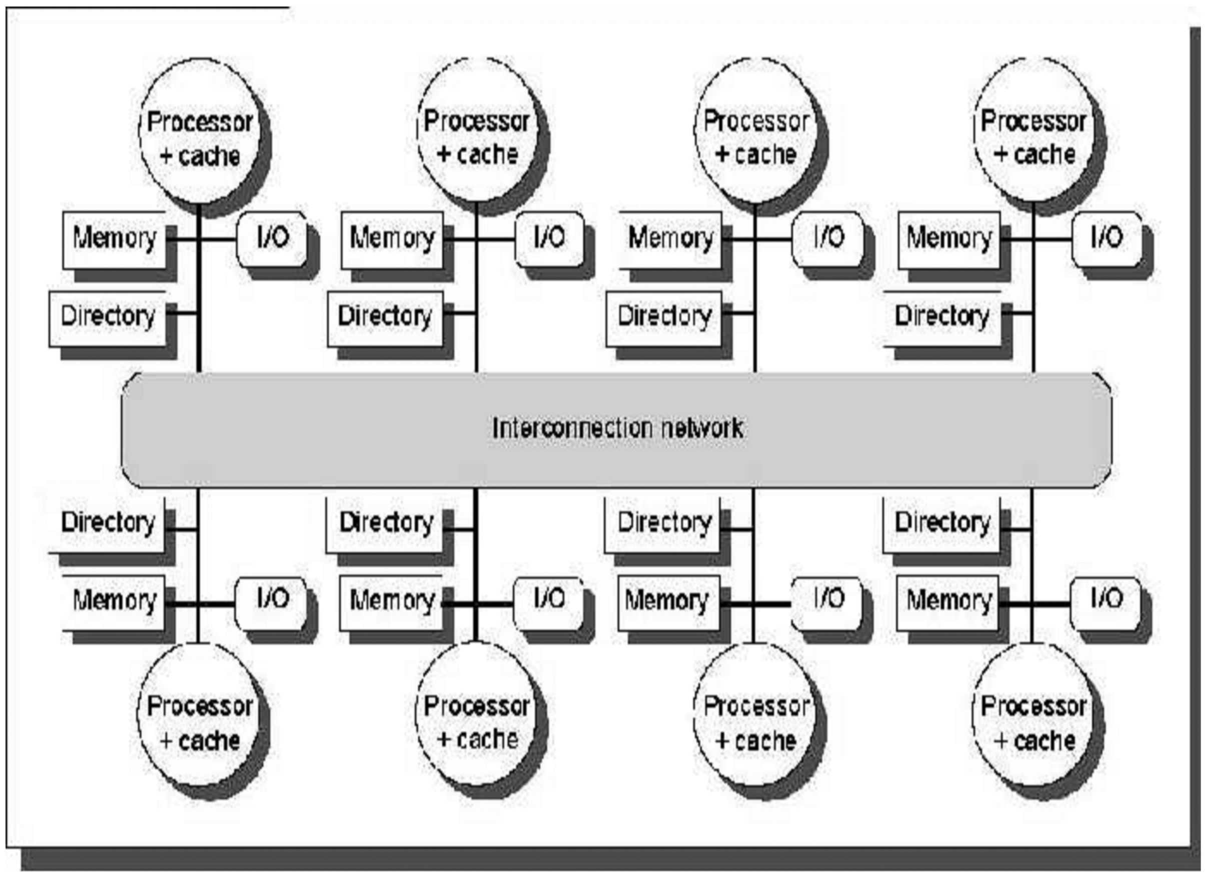


FIGURE 3.5 A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor

3.2.1 Directory-Based Cache-Coherence Protocols: The Basics

There are two primary operations that a directory protocol must implement:

- handling a read miss and handling a write to a shared, clean cache block. (Handling a write miss to a shared block is a simple combination of these two.)
- To implement these operations, a directory must track the state of each cache block.

In a simple protocol, these states could be the following:

- Shared—One or more processors have the block cached, and the value in memory is up to date (as well as in all the caches)
- Uncached—No processor has a copy of the cache block

Exclusive—Exactly one processor has a copy of the cache block and it has written the block, so the memory copy is out of date. The processor is called the owner of the block.

In addition to tracking the state of each cache block, we must track the processors that have copies of the block when it is shared, since they will need to be invalidated on a write.

The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor has a copy of that block. We can also use the bit vector to keep track of the owner of the block when the block is in the exclusive state. For efficiency reasons, we also track the state of each cache block at the individual caches.

A catalog of the message types that may be sent between the processors and the directories. Figure 6.28 shows the type of messages sent among nodes. The local node is the node where a request originates.

The home node is the node where the memory location and the directory entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known.

For example, the high-order bits may provide the node number, while the low-order bits provide the offset within the memory on that node. The local node may also be the home node. The directory must be accessed when the home node is the local node, since copies may exist in yet a third node, called a remote node.

A remote node is the node that has a copy of a cache block, whether exclusive (in which case it is the only copy) or shared. A remote node may be the same as either the local node or the home node. In such cases, the basic protocol does not change, but interprocessor messages may be replaced with intraprocessor messages.

Message type	**Source**	**Destination**	**Message contents**	**Function of this message**
Read miss	Local cache	Home directory	P, A	Processor P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Processor P has a write miss at address A; — request data and make P the exclusive owner.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.

Fetch/invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write back	Remote cache	Home directory	A, D	Write back a data value for address A.

FIGURE 3.6 The possible messages sent among nodes to maintain coherence are shown with the source and destination node, the contents (where P =requesting processor number), A=requested address, and D=data contents), and the function of the message.

3.3 Synchronization and various Hardware Primitives

3.3.1 Synchronization

Synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions. The efficient spin locks can be built using a simple hardware synchronization instruction and the coherence mechanism.

3.3.2 Basic Hardware Primitives

The key ability we require to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to atomically read and modify a memory location. Without such a capability, the cost of building basic synchronization primitives wil l be too high and will increase as the processor count increases.

There are a number of alternative formulations of the basic hardware primitives, all of which provide the ability to atomically read and modify a location, together with some way to tell if the read and write were performed atomically.

These hardware primitives are the basic building blocks that are used to build a wide variety of user-level synchronization operations, including things such as locks and barriers.

One typical operation for building synchronization operations is the atomic exchange, which interchanges a value in a register for a value in memory.

Use this to build a basic synchronization operation, assume that we want to build a

Simple lock where the value 0 is used to indicate that the lock is free and a 1 is used to indicate that the lock is unavailable.

A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock. The value returned from the exchange instruction is 1 if some other processor had already claimed access and 0 otherwise. In the latter case, the value is also changed to be 1, preventing any competing exchange from also retrieving a 0.

There are a number of other atomic primitives that can be used to implement synchronization. They all have the key property that they read and update a memory value in such a manner that we can tell whether or not the two operations executed atomically. One operation, present in many older multiprocessors, is test-and-set, which tests a value and sets it if the value passes the test. For example, we could define an operation that tested for 0 and set the value to 1, which can be used in a fashion similar to how we used atomic exchange.

Another atomic synchronization primitive is fetch-and-increment: it returns the value of a memory location and atomically increments it. By using the value 0 to indicate that the synchronization variable is unclaimed, we can use fetch-and-increment, just as we used exchange. There are other uses of operations like fetch-and-increment.

3.3.3 Implementing Locks Using Coherence

We can use the coherence mechanisms of a multiprocessor to implement spin locks: locks that a processor continuously tries to acquire, spinning around a loop until it succeeds. Spin locks are used when a programmer expects the lock to be held for a very short amount of time and when she wants the process of locking to be low latency when the lock is available. Because spin locks tie up the processor, waiting in a loop for the lock to become free, they are inappropriate in some circumstances.

The simplest implementation, which we would use if there were no cache coherence, supports cache coherence, we can maintain the lock value coherent an implementation where the proca tight loop) could be done on a lo would keep the lock variables in memory. A processor could continually try to acquire the lock using an atomic operation, say exchange, and test whether the exchange returned the lock as free. To release the lock, the processor simply stores the value 0 to the lock. Here is the code sequence to lock a spin lock whose address is in R1 using an atomic exchange:

```
        DADDUI  R2,R0,#1
lockit: EXCH    R2,0(R1) ; atomic exchange
        BNEZ    R2,lockit ; already locked?
```

If our multiprocessor supports cache coherence, we can cache the locks using the coherence mechanism to maintain the lock value coherently. Caching locks has two advantages. First, it allows an implementation where the process of “spinning” (trying to test and acquire the lock in a tight loop) could be done on a local cached copy rather than requiring a global memory access on each attempt to acquire the lock.

The second advantage comes from the observation that there is often locality in lock accesses: that is, the processor that used the lock last will use it again in the near future. In such cases, the lock value may reside in the cache of that processor, greatly reducing the time to acquire the lock..

3.3.4 Synchronization Performance Challenges

Barrier Synchronization

One additional common synchronization operation in programs with parallel loops is a barrier. A barrier forces all processes to wait until all the processes reach the barrier and then releases all of the processes. A typical implementation of a barrier can be done with two spin locks: one used to protect a counter that tallies the processes arriving at the barrier and one used to hold the processes until the last process arrives at the barrier.

Synchronization Mechanisms for Larger-Scale Multiprocessors

Software Implementations

The major difficulty with our spin-lock implementation is the delay due to contention when many processes are spinning on the lock. One solution is to artificially delay processes when they fail to acquire the lock. The best performance is obtained by increasing the delay exponentially whenever the attempt to acquire the lock fails.

Figure 3.7 shows how a spin lock with exponential back-off is implemented. Exponential back- off is a common technique for reducing contention in shared resources, including access to shared networks and buses. This implementation still attempts to preserve low latency when contention is small by not delaying the initial spin loop. The result is that if many processes are waiting, the back-off does not affect the processes on their first attempt to acquire the lock. We could also delay that process, but the result would be poorer performance when the lock was in use by only two processes and the first one happened to find it locked.

```
        ADDUI    R3,R0,#1 ;    R3 = initial delay
lockit: LL       R2,0(R1) ;    load linked
        BNEZ     R2,lockit ;   not available-spin
        DADDUI   R2,R2,#1 ;    get locked value
        SC       R2,0(R1) ;    store conditional
        BNEZ     R2,gotit ;    branch if store succeeds
        DSLL     R3,R3,#1 ;    Increase delay by factor of 2
        PAUSE    R3 ;          delays by value in R3
        J        lockit
        gotit: use data protected by lock
```

FIGURE 3.7 A spin lock with exponential back-off.

Another technique for implementing locks is to use queuing locks. Queuing locks work

by constructing a queue of waiting processors; whenever a processor frees up the lock, it causes the next processor in the queue to attempt access. This eliminates contention for a lock when it is freed. We show how queuing locks operate in the next section using a hardware implementation, but software implementations using arrays can achieve most of the same benefits Before we look at hardware primitives,

3.3.5 Hardware Primitives

In this section we look at two hardware synchronization primitives. The first primitive deals with locks, while the second is useful for barriers and a number of other user-level operations that require counting or supplying distinct indices. In both cases we can create a hardware primitive where latency is essentially identical to our earlier version, but with much less serialization, leading to better scaling when there is contention.

The major problem with our original lock implementation is that it introduces a large amount of unneeded contention. For example, when the lock is released all processors generate both a read and a write miss, although at most one processor can successfull y get the lock in the unlocked state. This sequence happens on each of the 20 lock/unlock sequences.

We can improve this situation by explicitly handing the lock from one waiting processor to the next. Rather than simply allowing all processors to compet e every time the lock is released, we keep a list of the waiting processors and hand the lock to one explicitly, when its turn comes. This sort of mechanism has been called a queuing lock. Queuing locks can be implemented either in hardware, or in software using an array to

keep track of the waiting processes.

3.4 Multithreading exploiting TLP.

3.4.1 Multithreading: Exploiting Thread-Level Parallelism within a Processor

Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion. To permit this sharing, the processor must duplicate the independent state of each thread. For example, a separate copy of the register file, a separate PC, and a separate page table are required for each thread.

There are two main approaches to multithreading.

- Fine-grained multithreading switches between threads on each instruction, causing the execution of multiples threads to be interleaved. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that time.
- Coarse-grained multithreading was invented as an alternative to fine-grained multithreading. Coarse-grained multithreading switches threads only on costly stalls, such as level two cache misses. This change relieves the need to have thread-switching be essentially free and is much less likely to slow the processor down, since instructions from other threads will only be issued, when a thread encounters a costly stall.

3.4.2 Simultaneous Multithreading: Converting Thread-Level Parallelism into Instruction-Level Parallelism:

Simultaneous multithreading (SMT) is a variation on multithreading that uses the resources of a multiple issue, dynamically-scheduled processor to exploit TLP at the same time it exploits ILP. The key insight that motivates SMT is that modern multipleissue processors often have more functional unit parallelism available than a single thread can effectively use. Furthermore, with register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability.

Figure 6.44 conceptually illustrates the differences in a processor's ability to exploit the resources of a superscalar for the following processor configurations:

n a superscalar with no multithreading support,

n a superscalar with coarse-grained multithreading,

n a superscalar with fine-grained multithreading, and

n a superscalar with simultaneous multithreading.

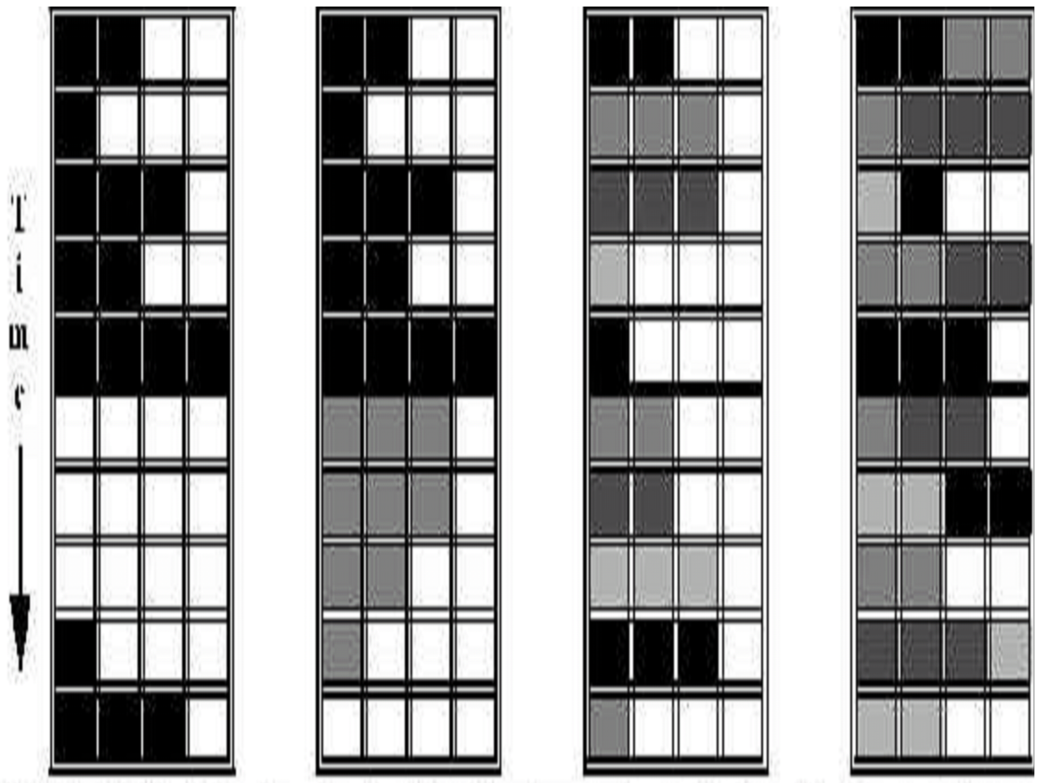


FIGURE 6.44 This illustration shows how these four different approaches use the issue slots of a superscalar processor. The horizontal dimension represents the instruction issue capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding issue slot is unused in that clock cycle. The shades of grey and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support.

In the superscalar without multithreading support, the use of issue slots is limited by a lack of ILP.

In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor.In the fine-grained case, the interleaving of threads eliminates fully empty slots. Because only one thread issues instructions in a given clock cycle.

In the SMT case, thread-level parallelism (TLP) and instruction-level parallelism (ILP) are exploited simultaneously; with multiple threads using the issue slots in a single clock cycle.

Figure 6.44 greatly simplifies the real operation of these processors it does illustrate the potential performance advantages of multithreading in general and SMT in particular.

3.4.3 Design Challenges in SMT processors

There are a variety of design challenges for an SMT processor, including:

- Dealing with a larger register file needed to hold multiple contexts,
- Maintaining low overhead on the clock cycle, particularly in critical steps such as instruction issue, where more candidate instructions need to be considered, and in instruction completion, where choosing what instructions to commit may be challenging, and
- Ensuring that the cache conflicts generated by the simultaneous execution of multiple threads do not cause significant performance degradation.

In viewing these problems, two observations are important. In many cases, the potential performance overhead due to multithreading is small, and simple choices work well enough. Second, the efficiency of current super-scalars is low enough that there is room for significant improvement, even at the cost of some overhead.

UNIT IV

MEMORY AND I/O

Cache performance - Reducing cache miss penalty and miss rate - Reducing hit time - Main memory and performance - Memory technology. Types of storage devices - Buses - RAID - Reliability, availability and dependability - I/O performance measures - Designing an I/O system.

4.1 Cache Performance

The average memory access time is calculated as follows

Average memory access time = hit time + Miss rate x Miss Penalty.

Where Hit Time is the time to deliver a block in the cache to the processor (includes time to determine whether the block is in the cache), Miss Rate is the fraction of memory references not found in cache (misses/references) and Miss Penalty is the additional time required because of a miss the average memory access time due to cache misses predicts processor performance.

First, there are other reasons for stalls, such as contention due to I/O devices using memory and due to cache misses

Second, The CPU stalls during misses, and the memory stall time is strongly correlated to average memory access time.

CPU time = (CPU execution clock cycles + Memory stall clock cycles) × Clock cycle time

There are 17 cache optimizations into four categories:

1 Reducing the miss penalty: multilevel caches, critical word first, read miss before write miss, merging write buffers, victim caches;

2 Reducing the miss rate larger block size, larger cache size, higher associativity, pseudo-associativity, and compiler optimizations;

3 Reducing the miss penalty or miss rate via parallelism: nonblocking caches, hardware prefetching, and compiler prefetching;

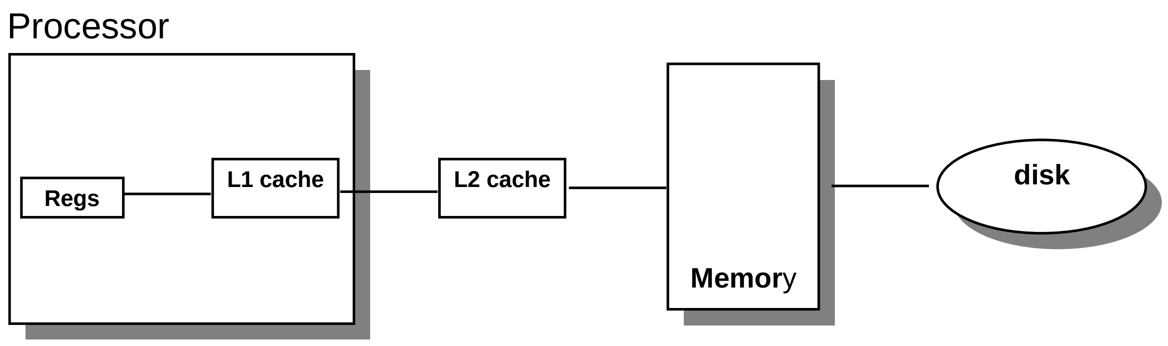
4 Reducing the time to hit in the cache: small and simple caches, avoiding address translation, and pipelined cache access.

4.1.1 Techniques for Reducing Cache Miss Penalty

There are five optimizations techniques to reduce miss penalty.

i) First Miss Penalty Reduction Technique: Multi-Level Caches

The First Miss Penalty Reduction Technique follows the Adding another level of cache between the original cache and memory. The first-level cache can be small enough to match the clock cycle time of the fast CPU and the second-level cache can be large enough to capture many accesses that would go to main memory, thereby the effective miss penalty.



The definition of average memory access time for a two-level cache. Using the subscripts L_1 and L_2 to refer, respectively, to a first-level and a second-level cache, the formula is

Average memory access time = Hit time_{L1} + Miss rate_{L1} × Miss penalty_{L1} and

Miss penalty_{L1} = Hit time_{L2} + Miss rate_{L2} × Miss penalty_{L2}

so Average memory access time = Hit time_{L1} + Miss rate_{L1}× (Hit time_{L2} + Miss rate_{L2} × Miss penalty_{L2})

Local miss rate

This rate is simply the number of misses in a cache divided by the total number of memory accesses to this cache. As you would expect, for the first-level cache it is equal to Miss rate_{L1} and for the second-level cache it is Miss rate_{L2}

Global miss rate

The number of misses in the cache divided by the total num-ber of memory accesses generated by the CPU. Using the terms above, the global miss rate for the first-level cache is still just Miss rate_{L1} but for the second-level cache it is Miss rate_{L1} × Miss rate_{L2}.

This local miss rate is large for second level caches because the first-level cache skims the cream of the memory accesses. This is why the global miss rate is the more useful measure: it indicates what fraction of the memory accesses that leave the CPU go all the way to memory.

Here is a place where the misses per instruction metric shines. Instead of confusion about local or global miss rates, we just expand memory stalls per instruction to add the impact of a second level cache.

Average memory stalls per instruction = Misses per instruction_{L1}× Hit time_{L2} + Misses per instruction_{L2} × Miss penalty_{L2}.

We can consider the parameters of second-level caches. The foremost difference between the two levels is that the speed of the first-level cache affects the clock rate of the CPU, while the speed of the second-level cache only affects the miss penalty of the first-level cache.

The initial decision is the size of a second-level cache. Since everything in the first-level cache is likely to be in the second-level cache, the second-level cache should be much bigger than the first. If second-level caches are just a little bigger, the local miss rate will be high.

Figures 5.1 and 5.2 show how miss rates and relative execution time change with the size of a second-level cache for one design.

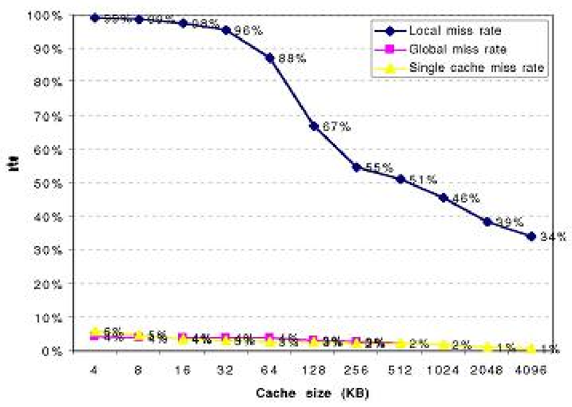


FIGURE 5.1

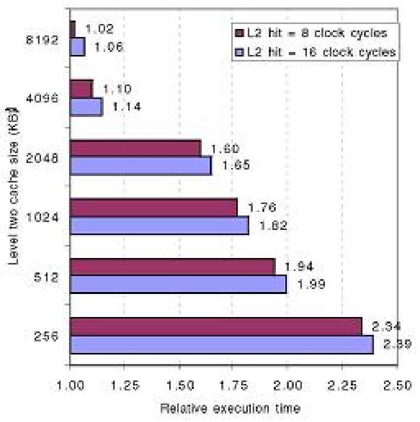


FIGURE 5.2

ii) Second Miss Penalty Reduction Technique: Critical Word First and Early Restart

Multilevel caches require extra hardware to reduce miss penalty, but not this second technique. It is based on the observation that the CPU normally needs just one word of the block at a time. This strategy is impatience: Don't wait for the full block to be loaded before sending the requested word and restarting the CPU.

Here are two specific strategies:

Critical word first

Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Critical-word-first fetch is also called wrapped fetch and requested word first.

Early restart

Fetch the words in normal order, but as soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution.

Generally these techniques only benefit designs with large cache blocks, since the benefit is low unless blocks are large. The problem is that given spatial locality, there is more than random chance that the next miss is to the remainder of the block. In such cases, the effective miss penalty is the time from the miss until the second piece arrives.

iii) Third Miss Penalty Reduction Technique: Giving Priority to Read Misses over Writes

This optimization serves reads before writes have been completed. We start with looking at complexities of a write buffer. With a write-through cache the most important improvement is a write buffer of the proper size. Write buffers, however, do complicate memory accesses in that they might hold the updated value of a location needed on a read miss. The simplest way out of this is for the read miss to wait until the write buffer is empty.

The alternative is to check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue. Virtually all desktop and server processors use the latter approach, giving reads priority over writes.

The cost of writes by the processor in a write-back cache can also be reduced. Suppose a read miss will replace a dirty memory block. Instead of writing the dirty block to memory, and then reading memory, we could copy the dirty block to a buffer, then read memory, and then write memory.

This way the CPU read, for which the processor is probably waiting, will finish sooner. Similar to the situation above, if a read miss occurs, the processor can either stall until the buffer is empty or check the addresses of the words in the buffer for conflicts.

iv) Fourth Miss Penalty Reduction Technique: Merging Write Buffer

This technique also involves write buffers, this time improving their efficiency. Write through caches rely on write buffers, as all stores must be sent to the next lower level of the hierarchy. As mentioned above, even write back caches use a simple buffer when a block is replaced.

If the write buffer is empty, the data and the full address are written in the buffer, and the write is finished from the CPU's perspective; the CPU continues working while the write buffer prepares to write the word to memory.

If the buffer contains other modified blocks, the addresses can be checked to see if the address of this new data matches the address of the valid write buffer entry. If so, the new data are combined with that entry, called write merging.

If the buffer is full and there is no address match, the cache (and CPU) must wait until the buffer has an empty entry. This optimization uses the memory more efficiently since multiword writes are usually faster than writes performed one word at a time.

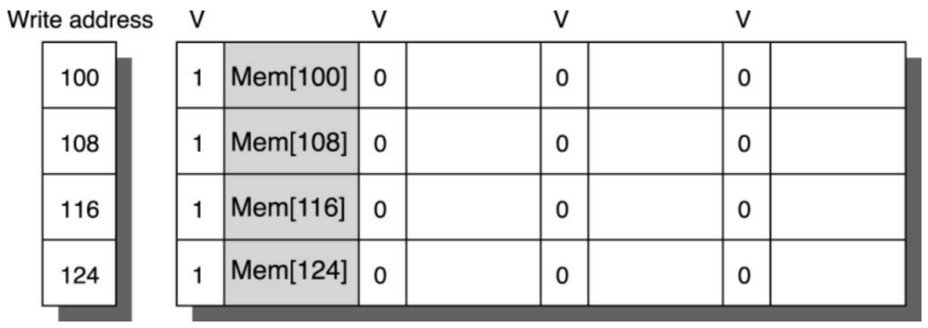


FIGURE 5.3

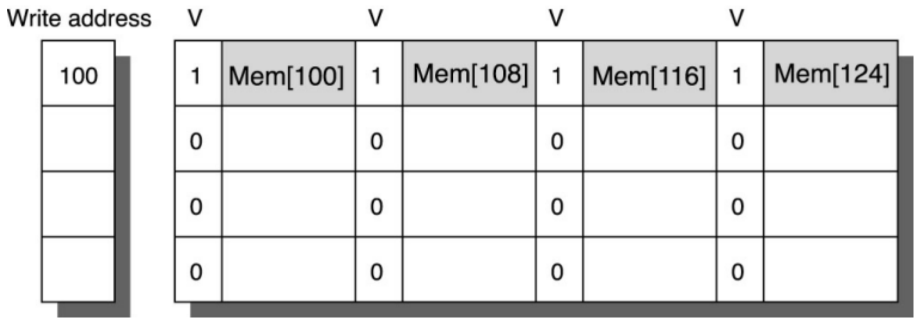


Figure 5.3

The optimization also reduces stalls due to the write buffer being full. Figure 5.12 shows a write buffer with and without write merging. Assume we had four entries in the write buffer, and each entry could hold four 64-bit words. Without this optimization, four stores to sequential addresses would fill the buffer at one word per entry, even though these four words when merged exactly fit within a single entry of the write buffer.

Figure 5.3 & 5.4 To illustrate write merging, the write buffer on top does not use it while the write buffer on the bottom does. The four writes are merged into a single buffer entry with write merging; without it, the buffer is full even though three-fourths of each entry is wasted. The buffer has four entries, and each entry holds four 64-bit words.

The address for each entry is on the left, with valid bits (V) indicating whether or not the next sequential eight bytes are occupied in this entry. (Without write merging, the words to the right in the upper drawing would only be used for instructions which wrote multiple words at the same time.)

v) Fifth Miss Penalty Reduction Technique: Victim Caches

One approach to lower miss penalty is to remember what was discarded in case it is needed again. Since the discarded data has already been fetched, it can be used again at small cost.

Such “recycling” requires a small, fully associative cache between a cache and its refill path. Figure 5.13 shows the organization. This victim cache contains only blocks that are discarded from a cache because of a miss “victims” and are checked on a miss to see if they have the desired data before going to the next lower-level memory. If it is found there, the victim block and cache block are swapped. The AMD Athlon has a victim cache with eight entries.

Jouppi [1990] found that victim caches of one to five entries are effective at reducing misses, especially for small, direct-mapped data caches. Depending on the program, a four-entry victim cache might remove one quarter of the misses in a 4-KB direct-mapped data cache.

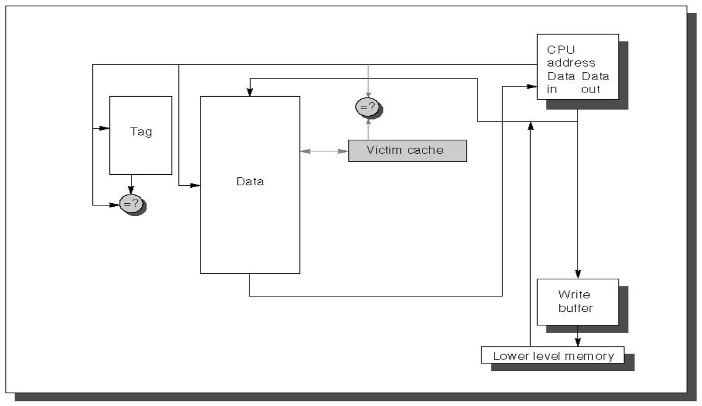


FIGURE 5.13 Placement of victim cache in the memory hierarchy. Although it reduces miss penalty, the victim cache is aimed at reducing the damage done by conflict misses, described in the next section. Jouppi [1990] found the four-entry victim cache could reduce the miss penalty for 20% to 95% of conflict misses.

Summary of Miss Penalty Reduction Techniques

The first technique follows the proverb "the more the merrier": assuming the principle of locality will keep applying recursively, just keep adding more levels of increasingly larger caches until you are happy. The second technique is impatience: it retrieves the word of the block that caused the miss rather than waiting for the full block to arrive. The next technique is preference. It gives priority to reads over writes since the processor generally waits for reads but continues after launching writes.

The fourth technique is companion-ship, combining writes to sequential words into a single block to create a more efficient transfer to memory. Finally comes a cache equivalent of recycling, as a victim cache keeps a few discarded blocks available for when the fickle primary cache wants a word that it recently discarded. All these techniques help with miss penalty, but multilevel caches is probably the most important.

4.1.2 Techniques to reduce miss rate

The classical approach to improving cache behavior is to reduce miss rates, and there are five techniques to reduce miss rate. we first start with a model that sorts all misses into three simple categories:

Compulsory

The very first access to a block cannot be in the cache, so the block must be brought into the cache. These are also called cold start misses or first reference misses.

Capacity

If the cache cannot contain all the blocks needed during execution of a program, capacity misses (in addition to compulsory misses) will occur be-cause of blocks being discarded and later retrieved.

Conflict

If the block placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur be-cause a block may be discarded and later retrieved if too many blocks map to its set. These misses are also called collision misses or interference misses. The idea is that hits in a fully associative cache which become misses in an N-way set associative cache are due to more than N requests on some popular sets.

i) First Miss Rate Reduction Technique: Larger Block Size

The simplest way to reduce miss rate is to increase the block size. Figure 5.4 shows the trade-off of block size versus miss rate for a set of programs and cache sizes. Larger block sizes will reduce compulsory misses. This reduction occurs because the principle of locality has two components: temporal locality and spatial locality. Larger blocks take advantage of spatial locality.

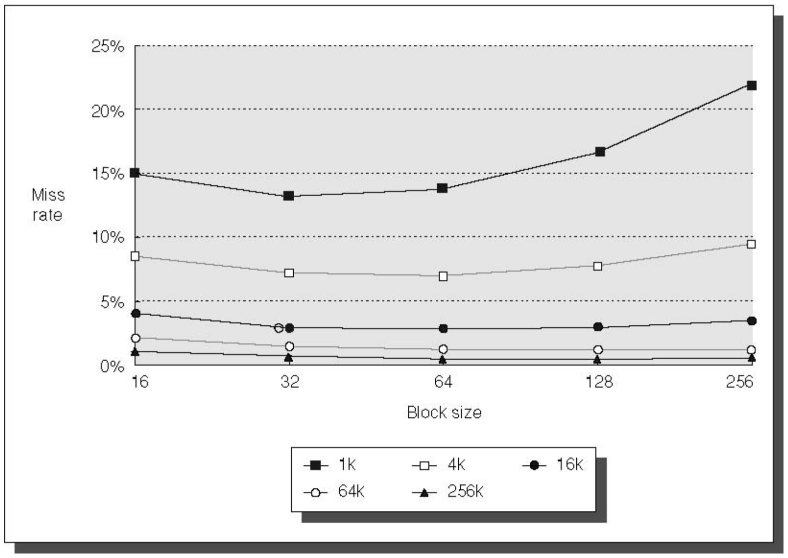


FIGURE 5.4 Miss rate versus block size for five different-sized caches.

At the same time, larger blocks increase the miss penalty. Since they reduce the number of blocks in the cache, larger blocks may increase conflict misses and even capacity misses if the cache is small. Clearly, there is little reason to increase the block size to such a size that it increases the miss rate. There is also no benefit to reducing miss rate if it increases the average memory access time. The increase in miss penalty may outweigh the decrease in miss rate.

ii) Second Miss Rate Reduction Technique: Larger caches

The obvious way to reduce capacity misses in the above is to increases capacity of the cache. The obvious drawback is longer hit time and higher cost. This technique has been especially popular in off-chip caches: The size of second or third level caches in 2001 equals the size of main memory in desktop computers.

iii) Third Miss Rate Reduction Technique: Higher Associativity:

Generally the miss rates improves with higher associativity. There are two general rules of thumb that can be drawn. The first is that eight-way set associative is for practical purposes as effective in reducing misses for these sized caches as fully associative. You can see the difference by comparing the 8-way entries to the capacity miss, since capacity misses are calculated using fully associative cache.

The second observation, called the 2:1 cache rule of thumb and found on the front inside cover, is that a direct-mapped cache of size N has about the same miss rate as a 2-way set-associative cache of size N/2. This held for cache sizes less than 128 KB.

iv) Fourth Miss Rate Reduction Technique: Way Prediction and Pseudo-Associative Caches

In way-prediction, extra bits are kept in the cache to predict the set of the next cache access. This prediction means the multiplexer is set early to select the desired set, and only a single tag comparison is performed that clock cycle. A miss results in checking the other sets for matches in subsequent clock cycles.

The Alpha 21264 uses way prediction in its instruction cache. (Added to each block of the instruction cache is a set predictor bit. The bit is used to select which of the two sets to try on the next cache access. If the predictor is correct, the instruction cache latency is one clock cycle. If not, it tries the other set, changes the set predictor, and has a latency of three clock cycles.

In addition to improving performance, way prediction can reduce power for embedded applications. By only supplying power to the half of the tags that are expected to be used, the MIPS R4300 series lowers power consumption with the same benefits.

A related approach is called pseudo-associative or column associative. Accesses proceed just as in the direct-mapped cache for a hit. On a miss, however, before going to the next lower level of the memory hierarchy, a second cache entry is checked to see if it matches there. A simple way is to invert the most significant bit of the index field to find the other block in the "pseudo set."

Pseudo-associative caches then have one fast and one slow hit time—corresponding to a regular hit and a pseudo hit—in addition to the miss penalty. Figure 5.20 shows the relative times. One danger would be if many fast hit times of the direct-mapped cache became slow hit times in the pseudo-associative cache. The performance would then be degraded by this optimization. Hence, it is important to indicate for each set which block should be the fast hit and which should be the slow one. One way is simply to make the upper one fast and swap the contents of the blocks. Another danger is that the miss penalty may become slightly longer, adding the time to check another cache entry.

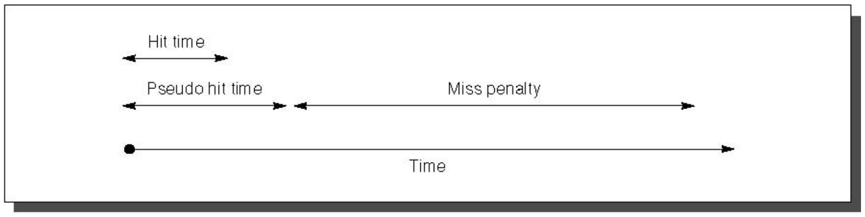


FIGURE 5.20 Relationship between a regular hit time, pseudo hit time, and miss penalty. Basically, pseudoassociativity offers a normal hit and a slow hit rather than more misses.

v) Fifth Miss Rate Reduction Technique: Compiler Optimizations

This final technique reduces miss rates without any hardware changes. This magical reduction comes from optimized software. The increasing performance gap between processors and main memory has inspired compiler writers to scrutinize the memory hierarchy to see if compile time optimizations can improve performance. Once again research is split between improvements in instruction misses and improvements in data misses.

Code can easily be rearranged without affecting correctness; for example, reordering the procedures of a program might reduce instruction miss rates by reducing conflict misses. Reordering the instructions reduced misses by 50% for a 2-KB direct-mapped instruction cache with 4-byte blocks, and by 75% in an 8-KB cache.

Another code optimization aims for better efficiency from long cache blocks. Aligning basic blocks so that the entry point is at the beginning of a cache block decreases the chance of a cache miss for sequential code.

Loop Interchange:

Some programs have nested loops that access data in memory in non-sequential order. Simply exchanging the nesting of the loops can make the code access the data in the order it is stored. Assuming the arrays do not fit in cache, this technique reduces misses by improving spatial locality; reordering maximizes use of data in a cache block before it is discarded.

```
            /* Before */
for (j = 0; j < 100; j = j+1)
for (i = 0; i < 5000; i = i+1)
    x[i][j] = 2 * x[i][j];

            /* After */
 for (i = 0; i < 5000; i = i+1)
 for (j = 0; j < 100; j = j+1)
     x[i][j] = 2 * x[i][j];
```

This optimization improves cache performance without affecting the number of instructions executed.

Blocking:

This optimization tries to reduce misses via improved temporal locality. We are again dealing with multiple arrays, with some arrays accessed by rows and some by columns. Storing the arrays row by row (row major order) or column by column (column major order) does not solve the problem because both rows and columns are used in every iteration of the loop. Such orthogonal accesses mean the transformations such as loop interchange are not helpful.

Instead of operating on entire rows or columns of an array, blocked algorithms operate on submatrices or blocks. The goal is to maximize accesses to the data loaded into the cache before the data are replaced.

Summary of Reducing Cache Miss Rate

This section first presented the three C's model of cache misses: compulsory, capacity, and conflict. This intuitive model led to three obvious optimizations: larger block size to reduce compulsory misses, larger cache size to reduce capacity misses, and higher associativity to reduce conflict misses. Since higher associativity may affect cache hit time or cache power consumption, way prediction checks only a piece of the cache for hits and then on a miss checks the rest. The final technique is the favorite of the hardware designer, leaving cache optimizations to the compiler.

4.2 Virtual memory & techniques for fast address translation

Virtual memory divides physical memory into blocks (called page or segment) and allocates them to different processes. With virtual memory, the CPU produces virtual addresses that are translated by a combination of HW and SW to physical addresses, which accesses main memory. The process is called memory mapping or address translation.Today, the two memory-hierarchy levels controlled by virtual memory are DRAMs and magnetic disks

Virtual Memory manages the two levels of the memory hierarchy represented by main memory and secondary storage. Figure 5.31 shows the mapping of virtual memory to physical memory for a program with four pages.

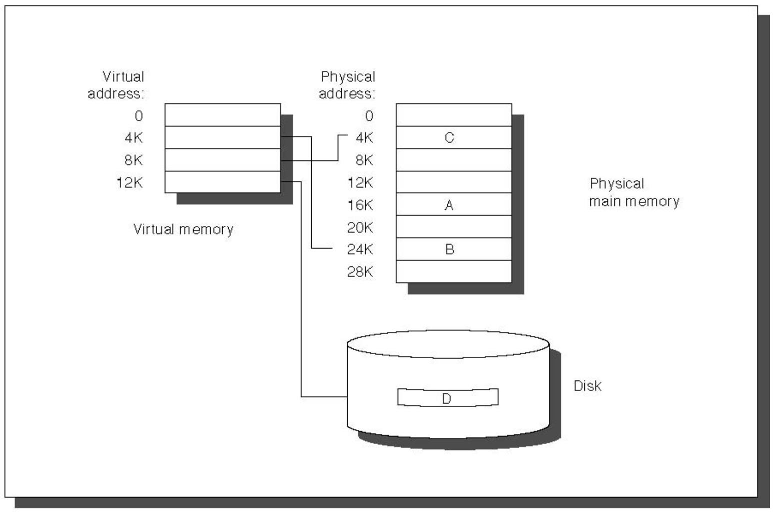
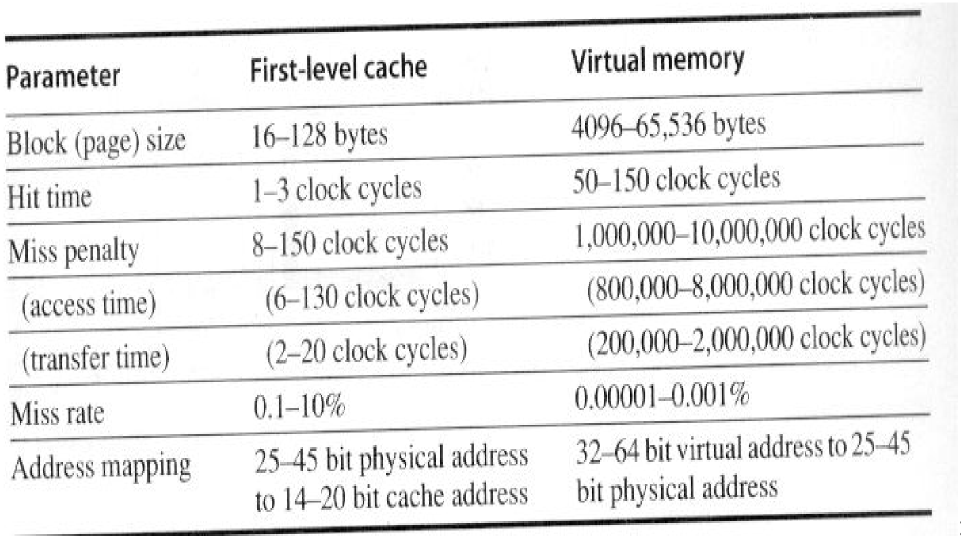


FIGURE 5.31 The logical program in its contiguous virtual address space is shown on the left. It consists of four pages A, B, C, and D. The actual location of three of the blocks is in physical main memory and the other is located on the disk.

There are further differences between caches and virtual memory beyond those quantitative ones mentioned in Figure 5.32



Parameter	First-level cache	Virtual memory
Block (page) size	16–128 bytes	4096–65,536 bytes
Hit time	1–3 clock cycles	50–150 clock cycles
Miss penalty	8–150 clock cycles	1,000,000–10,000,000 clock cycles
(access time)	(6–130 clock cycles)	(800,000–8,000,000 clock cycles)
(transfer time)	(2–20 clock cycles)	(200,000–2,000,000 clock cycles)
Miss rate	0.1–10%	0.00001–0.001%
Address mapping	25–45 bit physical address to 14–20 bit cache address	32–64 bit virtual address to 25–45 bit physical address

:

Virtual memory also encompasses several related techniques. Virtual memory systems can be categorized into two classes: those with fixed-size blocks, called pages, and those with variable-size locks, called segments. Pages are typically fixed at 4096 to 65,536 bytes, while segment size varies. The largest segment supported on any processor ranges from 2^{16} bytes up to 2^{32} bytes; the smallest segment is 1 byte. Figure 5.33 shows how the two approaches might divide code and data.

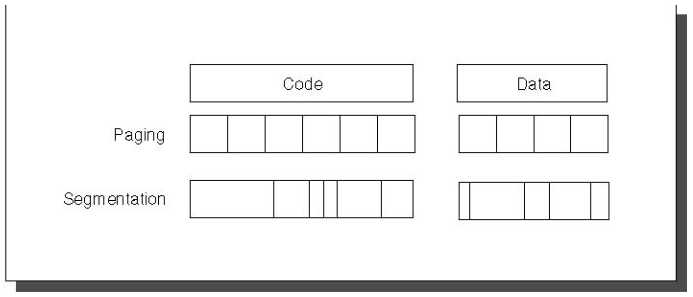


FIGURE 5.33 Example of how paging and segmentation divide a program.

The block can be placed anywhere in main memory. Both paging and segmentation rely on a data structure that is indexed by the page or segment number. This data structure contains the physical address of the block. For segmentation, the offset is added to the segment's physical address to obtain the final physical address. For paging, the offset is simply concatenated to this physical page address (see Figure 5.35).

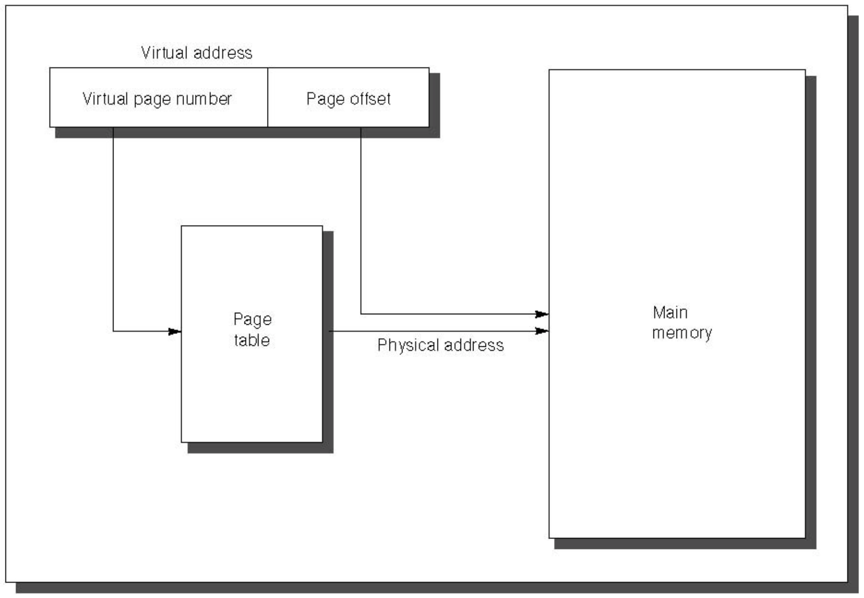


FIGURE 5.35 The mapping of a virtual address to a physical address via a page table.

This data structure, containing the physical page addresses, usually takes the form of a page table. Indexed by the virtual page number, the size of the table is the number of pages in the virtual address space. Given a 32-bit virtual address, 4-KB pages, and 4 bytes per page table entry, the size of the page table would be $(2^{32}/2^{12}) \times 2^{2} = 2^{22}$ or 4 MB.

To reduce address translation time, computers use a cache dedicated to these address translations, called a translation look-aside buffer, or simply translation buffer. They are described in more detail shortly.

With the help of Operating System and LRU algorithm pages can be replaced whenever page fault occurs.

4.2.1 Techniques for Fast Address Translation

Page tables are usually so large that they are stored in main memory, and some-times paged themselves. Paging means that every memory access logically takes at least twice as long, with one memory access to obtain the physical address and a second access to get the data. This cost is far too dear.

One remedy is to remember the last translation, so that the mapping process is skipped if the current address refers to the same page as the last one. A more general solution is to again rely on the principle of locality; if the accesses have locality, then the address translations for the accesses must also have locality. By keeping these address translations in a special cache, a memory access rarely re-quires a second access to translate the data. This special address translation cache is referred to as a translation look-aside buffer or TLB, also called a translation buffer or TB.

A TLB entry is like a cache entry where the tag holds portions of the virtual address and the data portion holds a physical page frame number, protection field, valid bit, and usually a use bit and dirty bit. To change the physical page frame number or protection of an entry in the page table, the operating system must make sure the old entry is not in the TLB; otherwise, the system won't be-have properly. Note that this dirty bit means the corresponding page is dirty, not that the address translation in the TLB is dirty nor that a particular block in the data cache is dirty. The operating system resets these bits by changing the value in the page table and then invalidating the corresponding TLB entry. When the entry is reloaded from the page table, the TLB gets an accurate copy of the bits.

Figure 5.5 shows the Alpha 21264 data TLB organization, with each step of a translation labeled. The TLB uses fully associative placement; thus, the translation begins (steps 1 and 2) by sending the virtual address to all tags. Of course, the tag must be marked valid to allow a match. At the same time, the type of memory access is checked for a violation (also in step 2) against protection infor-mation in the TLB.

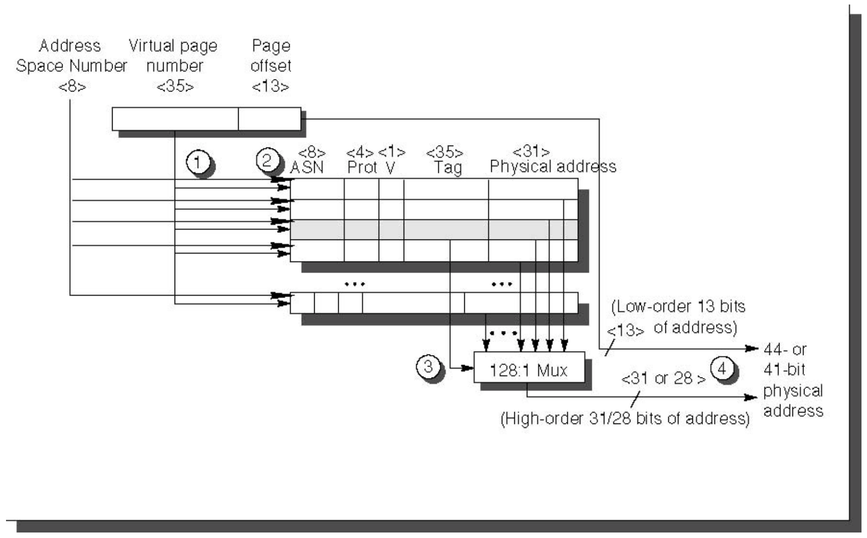


FIGURE 5.4 Operation of the Alpha 21264 data TLB during address translation.

Selecting a Page Size

The most obvious architectural parameter is the page size. Choosing the page is a question of balancing forces that favor a larger page size versus those favoring a smaller size. The following favor a larger size:

- The size of the page table is inversely proportional to the page size; memory (or other resources used for the memory map) can therefore be saved by making the pages bigger.
- A larger page size can allow larger caches with fast cache hit times.
- Transferring larger pages to or from secondary storage, possibly over a network, is more efficient than transferring smaller pages.
- The number of TLB entries are restricted, so a larger page size means that more memory can be mapped efficiently, thereby reducing the number of TLB misses.

Virtual memory protection

Multiprogramming forces to worry about usage of virtual memory. So Protection is required for virtual memory concept. The responsibility for maintaining correct process behavior is shared by designers of the computer and the operating system. The computer designer must ensure that the CPU portion of the process state can be saved and restored. The operating system designer must guarantee that processes do not interfere with each others' computations.

The safest way to protect the state of one process from another would be to copy the current information to disk. However, a process switch would then take seconds—far too long for a time-sharing environment. This problem is solved by operating systems partitioning main memory so that several different processes have their state in memory at the same time.

Protecting Processes

The simplest protection mechanism is a pair of registers that checks every ad-dress to be sure that it falls between the two limits, traditionally called base and bound. An address is valid if Base ≤ Address ≤ Bound

In some systems, the address is considered an unsigned number that is always added to the base, so the limit test is just (Base + Address) ≤ Bound

If user processes are allowed to change the base and bounds registers, then users can't be protected from each other. The operating system, however, must be able to change the registers so that it can switch processes. Hence, the computer designer has three more responsibilities in helping the operating system designer protect processes from each other:

- Provide at least two modes, indicating whether the running process is a user process or an operating system process. This latter process is sometimes called a kernel process, a supervisor process, or an executive process.
- Provide a portion of the CPU state that a user process can use but not write. This state includes the base/bound registers, a user/supervisor mode bit(s), and the exception enable/disable bit. Users are prevented from writing this state because the operating system cannot control user processes if users can change the address range checks, give themselves supervisor privileges, or disable exceptions.
- Provide mechanisms whereby the CPU can go from user mode to supervisor mode and vice versa. The first direction is typically accomplished by a system call, implemented as a special instruction that transfers control to a dedicated location in supervisor code space. The PC is saved from the point of the sys-tem call, and the CPU is placed in supervisor mode. The return to user mode is like a subroutine return that restores the previous user/supervisor mode.

4.2.2 A Paged Virtual Memory Example: The Alpha Memory Management and the 21264 TLB

The Alpha architecture uses a combination of segmentation and paging, providing protection while minimizing page table size. With 48-bit virtual addresses, the 64-bit address space is first divided into three segments: seg0 (bits 63 - 47 = 0...00), kseg (bits 63 - 46 = 0...10), and seg1 (bits 63 to 46 = 1...11). kseg is re-served for the operating system kernel, has uniform protection for the whole space, and does not use memory management.

User processes use seg0, which is mapped into pages with individual protection. Figure 5.38 shows the layout of seg0 and seg1. seg 0 grows from address 0 upward, while seg1 grows downward to 0. This approach provides many advantages: segmentation divides the address space and conserves page table space, while paging provides virtual memory, relocation, and protection.

The Alpha uses a three-level hierarchical page table to map the address space to keep the size reasonable. Figure 5.5 shows address translation in the Alpha. The addresses for each of these page tables come from three "level" fields, labeled level1, level2, and level3. Address translation starts with adding the level1 address field to the page table base register and then reading memory from this location to get the base of the second-level page table.

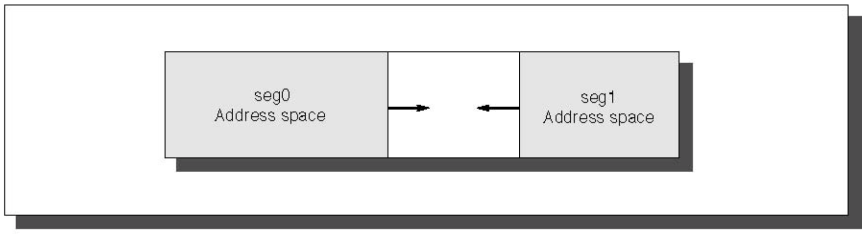


FIGURE 5.38 The organization of seg0 and seg1 in the Alpha. User processes live in seg0, while seg1 is used for portions of the page tables. seg0 includes a downward growing stack, text and data, and an upward growing heap.

The level2 address field is in turn added to this newly fetched address, and memory is accessed again to determine the base of the third page table. The level3 address field is added to this base address, and memory is read using this sum to (finally) get the physical address of the page being referenced. This address is concatenated with the page offset to get the full physical address. Each page table in the Alpha architecture is constrained to fit within a single page.

The first three levels (0, 1, and 2) use physical addresses that need no further translation, but Level 3 is mapped virtually. These normally hit the TLB, but if not, the table is accessed a second time with physical addresses.

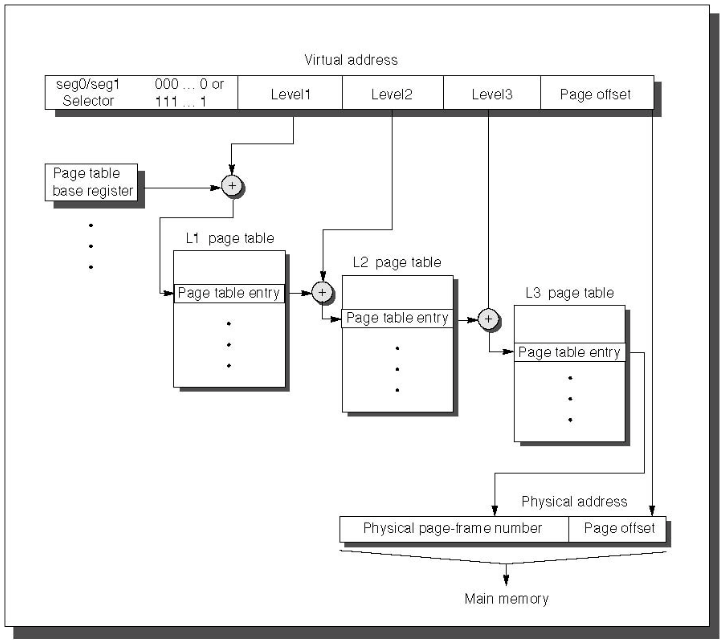


FIGURE 5.5 The mapping of an Alpha virtual address.

The Alpha uses a 64-bit page table entry (PTE) in each of these page tables. The first 32 bits contain the physical page frame number, and the other half includes the following five protection fields:

Valid—Says that the page frame number is valid for hardware translation

User read enable—Allows user programs to read data within this page

Kernel read enable—Allows the kernel to read data within this page

User write enable—Allows user programs to write data within this page

Kernel write enable—Allows the kernel to write data within this page

In addition, the PTE has fields reserved for systems software to use as it pleases. Since the Alpha goes through three levels of tables on a TLB miss, there are three potential places to check protection restrictions. The Alpha obeys only the bottom-level PTE, checking the others only to be sure the valid bit is set.

4.2.3 A Segmented Virtual Memory Example: Protection in the Intel Pentium

The original 8086 used segments for addressing, yet it provided nothing for virtual memory or for protection. Segments had base registers but no bound registers and no access checks, and before a segment register could be loaded the corresponding segment had to be in physical memory.

Intel's dedication to virtual memory and protection is evident in the successors to the 8086 (today called IA-32), with a few fields extended to support larger addresses. This protection scheme is elaborate, with many details carefully designed to try to avoid security loopholes.

The first enhancement is to double the traditional two-level protection model: the Pentium has four levels of protection. The innermost level (0) corresponds to Alpha kernel mode and the outermost level (3) corresponds to Alpha user mode. The IA-32 has separate stacks for each level to avoid security breaches between the levels.

The IA-32 divides the address space, al-lowing both the operating system and the user access to the full space. The IA-32 user can call an operating system routine in this space and even pass parameters to it while retaining full protection. This safe call is not a trivial action, since the stack for the operating system is different from the user's stack. Moreover, the IA-32 allows the operating system to maintain the protection level of the called routine for the parameters that are passed to it. This potential loophole in protection is prevented by not allowing the user process to ask the operating system to access something indirectly that it would not have been able to access itself. (Such security loopholes are called Trojan horses.)

Adding Bounds Checking and Memory Mapping

The first step in enhancing the Intel processor was getting the segmented addressing to check bounds as well as supply a base. Rather than a base address, as in the 8086, segment registers in the IA-32 contain an index to a virtual memory data structure called a descriptor table. Descriptor tables play the role of page tables in the Alpha. On the IA-32 the equivalent of a page table entry is a segment descriptor.

It contains fields found in PTEs:

A present bit—equivalent to the PTE valid bit, used to indicate this is a valid translation

A base field—equivalent to a page frame address, containing the physical address of the first byte of the segment

An access bit—like the reference bit or use bit in some architectures that is helpful for replacement algorithms

An attributes field—specifies the valid operations and protection levels for operations that use this segment

There is also a limit field, not found in paged systems, which establishes the upper bound of valid offsets for this segment. Figure 5.41 shows examples of IA-32 segment descriptors.

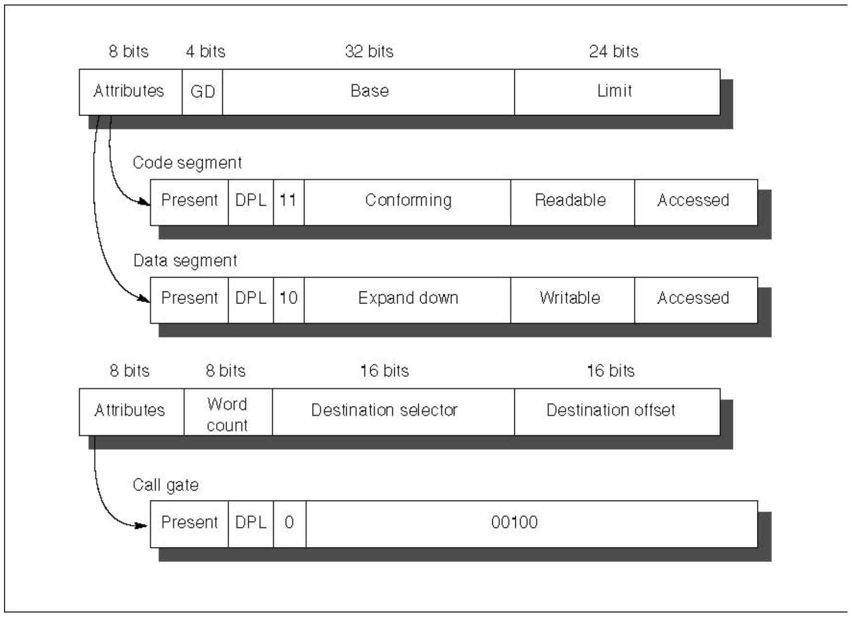


FIGURE 5.41 The IA-32 segment descriptors are distinguished by bits in the attributes field. Base, limit, present, readable, and writable are all self-explanatory.

IA-32 provides an optional paging system in addition to this segmented addressing. The upper portion of the 32-bit address selects the segment descriptor and the middle portion is an index into the page table selected by the descriptor.

Adding Sharing and Protection

To provide for protected sharing, half of the address space is shared by all processes and half is unique to each process, called global address space and local address space, respectively. Each half is given a descriptor table with the appropriate name. A descriptor pointing to a shared segment is placed in the global descriptor table, while a descriptor for a private segment is placed in the local descriptor table.

4.3 Storage Systems

Types of Storage Devices

There are various types of Storage devices such as magnetic disks, magnetic tapes, automated tape libraries, CDs, and DVDs.

The First Storage device magnetic disks have dominated nonvolatile storage since 1965. Magnetic disks play two roles in computer systems:

- Long Term, nonvolatile storage for files, even when no programs are running
- A level of the memory hierarchy below main memory used as a backing store for virtual memory during program execution.

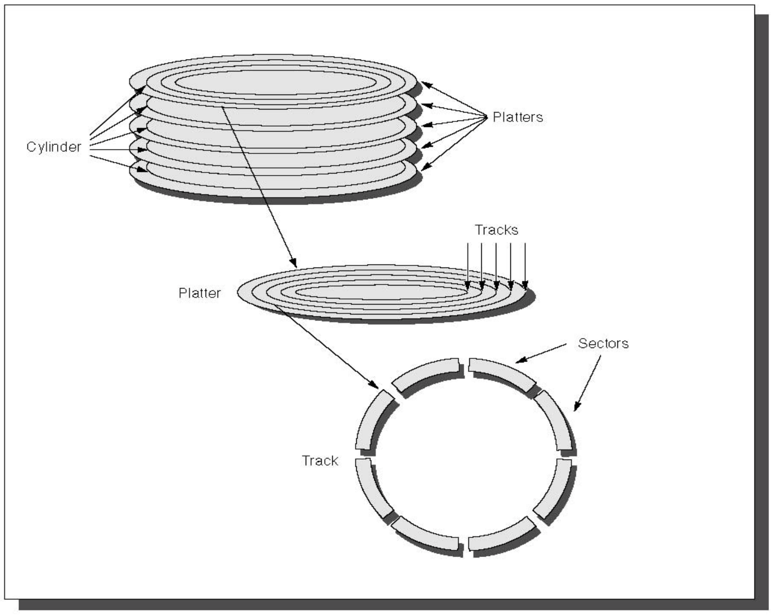


FIGURE 7.1 Disks are organized into platters, tracks, and sectors. Both sides of a platter are coated so that information can be stored on both surfaces. A cylinder refers to a track at the same position on every platter.

A magnetic disk consists of a collection of platters (generally 1 to 12), rotating on a spindle at 3,600 to 15,000 revolutions per minute (RPM). These platters are metal or glass disks covered with magnetic recording material on both sides, so 10 platters have 20 recording surfaces.

The disk surface is divided into concentric circles, designated tracks. There are typically 5,000 to 30,000 tracks on each surface. Each track in turn is divided into sectors that contain the information; a track might have 100 to 500 sectors. A sector is the smallest unit that can be read or written. IBM mainframes allow users to select the size of the sectors, although most systems fix their size, typically at 512 bytes of data. The sequence recorded on the magnetic media is a sector number, a gap, the information for that sector including error correction code, a gap, the sector number of the next sector, and so on.

To read and write information into a sector, a movable arm containing a read/ write head is located over each surface. To read or write a sector, the disk controller sends a command to move the arm over the proper track. This operation is called a seek, and the time to move the arm to the desired track is called seek time.

Average seek time is the subject of considerable misunderstanding. Disk manufacturers report minimum seek time, maximum seek time, and average seek time in their manuals. The first two are easy to measure, but the average was open to wide interpretation.

The time for the requested sector to rotate under the head is the rotation latency or rotational delay. The average latency to the desired information is obviously halfway around the disk; if a disk rotates at 10,000 revolutions per minute (RPM), the average rotation time is therefore

Average Rotation Time = 0.5/10,000RPM = 0.5/(10,000/60)RPM = 3.0ms

The next component of disk access, transfer time, is the time it takes to transfer a block of bits, typically a sector, under the read-write head. This time is a function of the block size, disk size, rotation speed, recording density of the track, and speed of the electronics connecting the disk to computer. Transfer rates in 2001 range from 3 MB per second for the 3600 RPM, 1-inch drives to 65 MB per second for the 15000 RPM, 3.5-inch drives.

The Future of Magnetic Disks

The disk industry has concentrated on improving the capacity of disks. Improvement in capacity is customarily expressed as improvement in areal density, measured in bits per square inch:

Areal Density = (Tracks/Inch) on a disk surface X (Bits/Inch) on a track

Through about 1988 the rate of improvement of areal density was 29% per year, thus doubling density every three years. Between then and about 1996, the rate improved to 60% per year, quadrupling density every three years and matching the traditional rate of DRAMs. From 1997 to 2001 the rate increased to 100%, or doubling every year. In 2001, the highest density in commercial products is 20 billion bits per square inch, and the lab record is 60 billion bits per square inch.

Optical Disks:

One challenger to magnetic disks is optical compact disks, or CDs, and its successor, called Digital Video Discs and then Digital Versatile Discs or just DVDs. Both the CD-ROM and DVD-ROM are removable and inexpensive to manufacture, but they are read-only mediums. These 4.7-inch diameter disks hold 0.65 and 4.7 GB, respectively, although some DVDs write on both sides to double their capacity. Their high capacity and low cost have led to CD-ROMs and DVD-ROMs replacing floppy disks as the favorite medium for distributing software and other types of computer data.

The popularity of CDs and music that can be downloaded from the WWW led to a market for rewritable CDs, conveniently called CD-RW, and write once CDs, called CD-R. In 2001, there is a small cost premium for drives that can record on CD-RW. The media itself costs about $0.20 per CD-R disk or $0.60 per CD-RW disk. CD-RWs and CD-Rs read at about half the speed of CD-ROMs and CD-RWs and CD-Rs write at about a quarter the speed of CD-ROMs.

Magnetic Tape:

Magnetic tapes have been part of computer systems as long as disks because they use the similar technology as disks, and hence historically have followed the same density improvements. The inherent cost/performance difference between disks and tapes is based on their geometries:

- Fixed rotating platters offer random access in milliseconds, but disks have a limited storage area and the storage medium is sealed within each reader.
- Long strips wound on removable spools of “unlimited” length mean many tapes can be used per reader, but tapes require sequential access that can take seconds.

One of the limits of tapes had been the speed at which the tapes can spin without breaking or jamming. A technology called helical scan tapes solves this problem by keeping the tape speed the same but recording the information on a diagonal to the tape with a tape reader that spins much faster than the tape is moving. This technology increases recording density by about a factor of 20 to 50. Helical scan tapes were developed for low-cost VCRs and camcorders, which brought down the cost of the tapes and readers.

Automated Tape Libraries

Tape capacities are enhanced by inexpensive robots to automatically load and store tapes, offering a new level of storage hierarchy. These nearline tapes mean access to terabytes of information in tens of seconds, without the intervention of a human operator.

Flash Memory

Embedded devices also need nonvolatile storage, but premiums placed on space and power normally lead to the use of Flash memory instead of magnetic recording. Flash memory is also used as a rewritable ROM in embedded systems, typically to allow software to be upgraded without having to replace chips. Applications are typically prohibited from writing to Flash memory in such circumstances.

Like electrically erasable and programmable read-only memories (EEPROM), Flash memory is written by inducing the tunneling of charge from transistor gain to a floating gate. The floating gate acts as a potential well which stores the charge, and the charge cannot move from there without applying an external force. The primary difference between EEPROM and Flash memory is that Flash restricts write to multi-kilobyte blocks, increasing memory capacity per chip by reducing area dedicated to control. Compared to disks, Flash memories offer low power consumption (less than 50 milliwatts), can be sold in small sizes, and offer read access times comparable to DRAMs. In 2001, a 16 Mbit Flash memory has a 65 ns access time, and a 128 Mbit Flash memory has a 150 ns access time.

4.4 Buses : Connecting I/O Devices to CPU/Memory

Buses were traditionally classified as CPU-memory buses or I/O buses. I/O buses may be lengthy, may have many types of devices connected to them, have a wide range in the data bandwidth of the devices connected to them, and normally follow a bus standard. CPU-memory buses, on the other hand, are short, generally high speed, and matched to the memory system to maximize memory-CPU bandwidth. During the design phase, the designer of a CPU-memory bus knows all the types of devices that must connect together, while the I/O bus designer must accept devices varying in latency and bandwidth capabilities. To lower costs, some computers have a single bus for both memory and I/O devices. In the quest for higher I/O performance, some buses are a hybrid of the two. For example, PCI is relatively short, and is used to connect to more traditional I/O buses via bridges that speak both PCI on one end and the I/O bus protocol on the other. To indicate its intermediate state, such buses are sometimes called mezzanine

Bus Design Decisions

The design of a bus presents several options, as Figure 7.8 shows. Like the rest of the computer system, decisions depend on cost and performance goals. The first three options in the figure are clear—separate address and data lines, wider data lines, and multiple-word transfers all give higher performance at more cost.

Option	High performance	Low cost
Bus width	Separate address and data lines	Multiplex address and data lines
Data width	Wider is faster (e.g., 64 bits)	Narrower is cheaper (e.g., 8 bits)
Transfer size	Multiple words have less bus overhead	Single-word transfer is simpler
Bus masters	Multiple (requires arbitration)	Single master (no arbitration)
Split transaction?	Yes—separate request and reply packets get higher bandwidth (need multiple masters)	No—continuous connection is cheaper and has lower latency
Clocking	Synchronous	Asynchronous

The next item in the table concerns the number of bus masters. These devices can initiate a read or write transaction; the CPU, for instance, is always a bus master. A bus has multiple masters when there are multiple CPUs or when I/O devices can initiate a bus transaction. With multiple masters, a bus can offer higher bandwidth by using packets, as opposed to holding the bus for the full transaction. This technique is called split transactions.

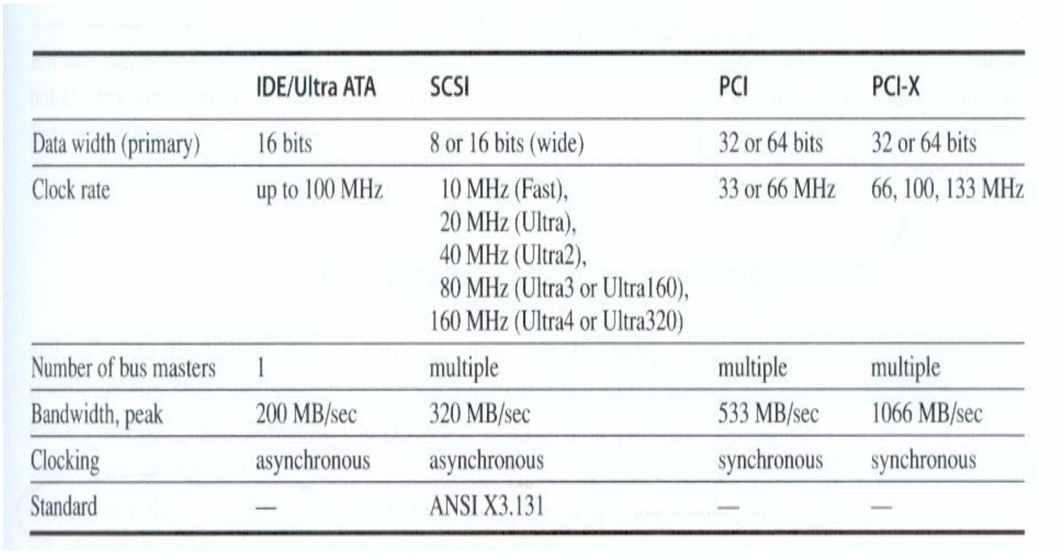
The final item in Figure 7.8, clocking, concerns whether a bus is synchronous or asynchronous. If a bus is synchronous, it includes a clock in the control lines and a fixed protocol for sending address and data relative to the clock. Since little or no logic is needed to decide what to do next, these buses can be both fast and inexpensive.

Bus Standards

Standards that let the computer designer and I/O-device designer work independently play a large role in buses. As long as both designers meet the requirements, any I/O device can connect to any computer. The I/O bus standard is the document that defines how to connect devices to computers.

- The Good
    - Let the computer and I/O-device designers work independently
    - Provides a path for second party (e.g. cheaper) competition
- The Bad
    - Become major performance anchors
    - Inhibit change
    - How to create a standard
- Bottom-up
    - Company tries to get standards committee to approve it's latest philosophy in hopes that they'll get the jump on the others (e.g. S bus, PC-AT bus, ...)
    - De facto standards
- Top-down
    - Design by committee (PCI, SCSI, ...)

Some sample bus designs are shown below



	IDE/Ultra ATA	SCSI	PCI	PCI-X
Data width (primary)	16 bits	8 or 16 bits (wide)	32 or 64 bits	32 or 64 bits
Clock rate	up to 100 MHz	10 MHz (Fast), 20 MHz (Ultra), 40 MHz (Ultra2), 80 MHz (Ultra3 or Ultra160), 160 MHz (Ultra4 or Ultra320)	33 or 66 MHz	66, 100, 133 MHz
Number of bus masters	1	multiple	multiple	multiple
Bandwidth, peak	200 MB/sec	320 MB/sec	533 MB/sec	1066 MB/sec
Clocking	asynchronous	asynchronous	synchronous	synchronous
Standard	—	ANSI X3.131	—	—

Interfacing Storage Devices to the CPU

The I/O bus is connected to the main memory bus is shown in figure 7.15

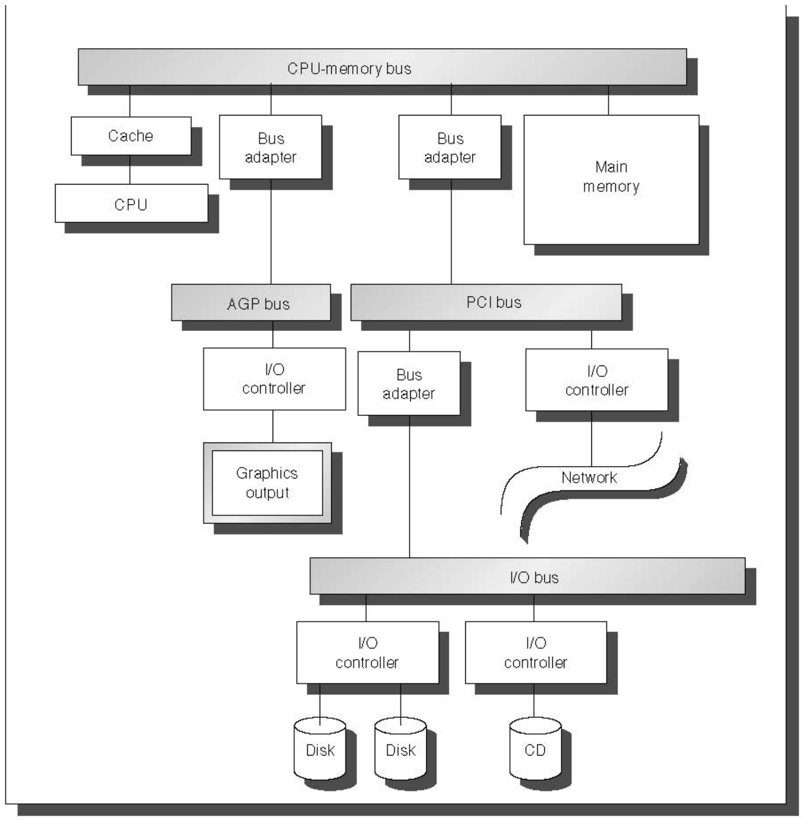


FIGURE 7.15 A typical interface of I/O devices and an I/O bus to the CPU-memory bus.

Processor interface with i/o bus can be done with two techniques one using interrupts and second using memory mapped I/O

- I/O Control Structures
    - Polling
    - Interrupts
    - DMA
    - I/O Controllers
    - I/O Processors

The simple interface, in which the CPU periodically checks status bits to see if it is time for the next I/O operation, is called polling.

Interrupt-driven I/O, used by most systems for at least some devices, allows the CPU to work on some other process while waiting for the I/O device. For example, the LP11 has a mode that allows it to interrupt the CPU whenever the done bit or error bit is set. In general-purpose applications, interrupt-driven I/O is the key to multitasking operating systems and good response times.

The drawback to interrupts is the operating system overhead on each event. In real-time applications with hundreds of I/O events per second, this overhead can be intolerable. One hybrid solution for real-time systems is to use a clock to periodically interrupt the CPU, at which time the CPU polls all I/O devices

The DMA hardware is a specialized processor that transfers data between memory and an I/O device while the CPU goes on with other tasks. Thus, it is external to the CPU and must act as a master on the bus. The CPU first sets up the DMA registers, which contain a memory address and number of bytes to be transferred. More sophisticated DMA devices support scatter/gather, whereby a DMA device can write or read data from a list of separate addresses. Once the DMA transfer is complete, the DMA controller interrupts the CPU. There may be multiple DMA devices in a computer system.

4.5 RAID : Redundant Arrays of Inexpensive Disks

An innovation that improves both dependability and performance of storage systems is disk arrays. One argument for arrays is that potential throughput can be increased by having many disk drives and, hence, many disk arms, rather than one large drive with one disk arm. Although a disk array would have more faults than a smaller number of larger disks when each disk has the same reliability, dependability can be improved by adding redundant disks to the array to tolerate faults. That is, if a single disk fails, the lost information can be reconstructed from redundant information.

The only danger is in having another disk fail between the time the first disk fails and the time it is replaced (termed mean time to repair, or MTTR). Since the mean time to failure (MTTF) of disks is tens of years, and the MTTR is measured in hours, redundancy can make the measured reliability of 100 disks much higher than that of a single disk. These systems have become known by the acronym RAID, stand-ing originally for redundant array of inexpensive disks, although some have re-named it to redundant array of independent disks

The several approaches to redundancy have different overhead and performance. Figure 7.17 shows the standard RAID levels. It shows how eight disks of user data must be supplemented by redundant or check disks at each RAID level. It also shows the minimum number of disk failures that a system would survive.

RAID level	Minimum number of Disk faults survived	Example Data disks	Corresponding Check disks	Corporations producing RAID products at this level
0 Non-redundant striped	0	8	0	Widely used
1 Mirrored	1	8	8	EMC, Compaq (Tandem), IBM
2 Memory-style ECC	1	8	4	
3 Bit-interleaved parity	1	8	1	Storage Concepts
4 Block-interleaved parity	1	8	1	Network Appliance
5 Block-interleaved distributed parity	1	8	1	Widely used
6 P+Q redundancy	2	8	2	

FIGURE 7.17 RAID levels, their fault tolerance, and their overhead in redundant disks.

No Redundancy (RAID 0)

This notation is refers to a disk array in which data is striped but there is no redundancy to tolerate disk failure. Striping across a set of disks makes the collection appear to software as a single large disk, which simplifies storage management. It also improves performance for large accesses, since many disks can operate at once. Video editing systems, for example, often stripe their data.

RAID 0 something of a misnomer as there is no redundancy, it is not in the original RAID taxonomy, and striping predates RAID. However, RAID levels are often left to the operator to set when creating a storage system, and RAID 0 is often listed as one of the options. Hence, the term RAID 0 has become widely used.

Mirroring (RAID 1)

This traditional scheme for tolerating disk failure, called mirroring or shadowing, uses twice as many disks as does RAID 0. Whenever data is written to one disk, that data is also written to a redundant disk, so that there are always two copies of the information. If a disk fails, the system just goes to the "mirror" to get the desired information. Mirroring is the most expensive RAID solution, since it requires the most disks.

The RAID terminology has evolved to call the former RAID 1+0 or RAID 10 ("striped mirrors”) and the latter RAID 0+1 or RAID 01 (“mirrored stripes”).

Bit-Interleaved Parity (RAID 3)

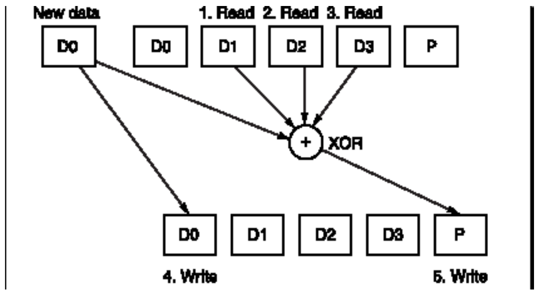
The cost of higher availability can be reduced to 1/N, where N is the number of disks in a protection group. Rather than have a complete copy of the original data for each disk, we need only add enough redundant information to restore the lost information on a failure. Reads or writes go to all disks in the group, with one extra disk to hold the check information in case there is a failure. RAID 3 is popular in applications with large data sets, such as multimedia and some scientific codes.

Parity is one such scheme. Readers unfamiliar with parity can think of the redundant disk as having the sum of all the data in the other disks. When a disk fails, then you subtract all the data in the good disks from the parity disk; the remaining information must be the missing information. Parity is simply the sum modulo two. The assumption behind this technique is that failures are so rare that taking longer to recover from failure but reducing redundant storage is a good trade-off.

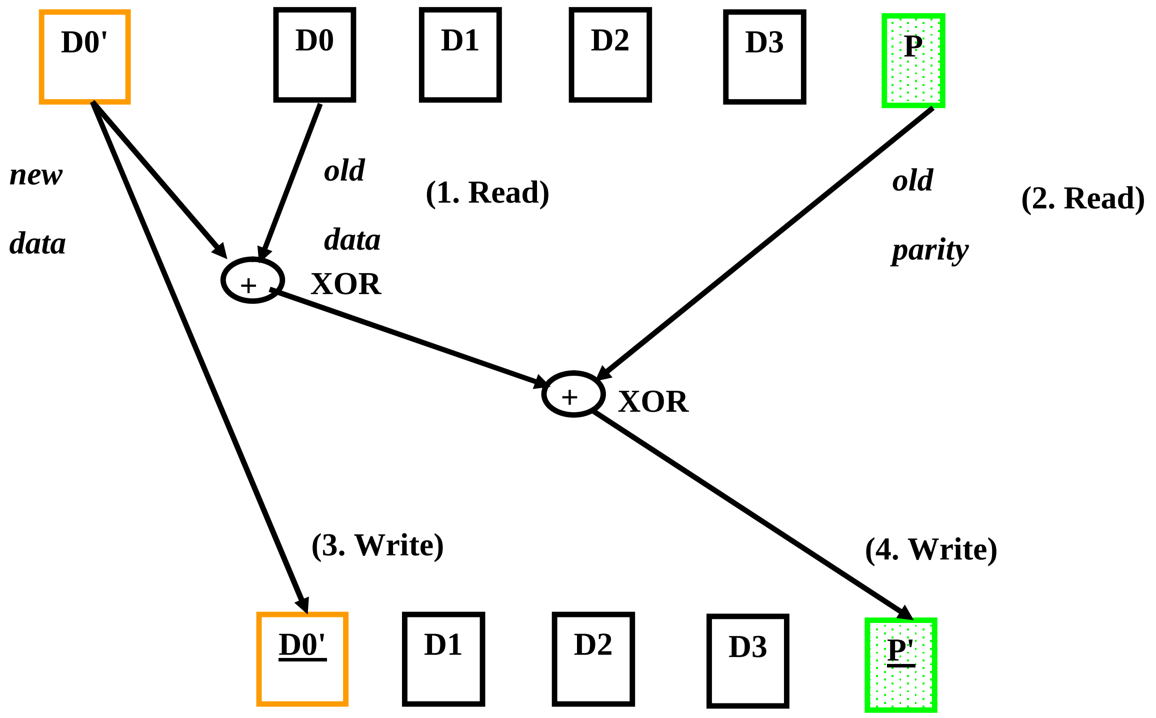
Block-Interleaved Parity and Distributed Block-Interleaved Parity (RAID 4 and RAID 5)

In RAID 3, every access went to all disks. Some applications would prefer to do smaller accesses, allowing independent accesses to occur in parallel. That is the purpose of the next RAID levels. Since error-detection information in each sector is checked on reads to see if data is correct, such “small reads” to each disk can occur independently as long as the minimum access is one sector.

Writes are another matter. It would seem that each small write would demand that all other disks be accessed to read the rest of the information needed to recalculate the new parity, as in Figure 7.18. A “small write” would require reading the old data and old parity, adding the new information, and then writing the new parity to the parity disk and the new data to the data disk.



RAID 4 efficiently supports a mixture of large reads, large writes, small reads, and small writes. One drawback to the system is that the parity disk must be updated on every write, so it is the bottleneck for back-to-back writes. To fix the parity-write bottleneck, the parity information can be spread throughout all the disks so that there is no single bottleneck for writes. The distributed parity organization is RAID 5.



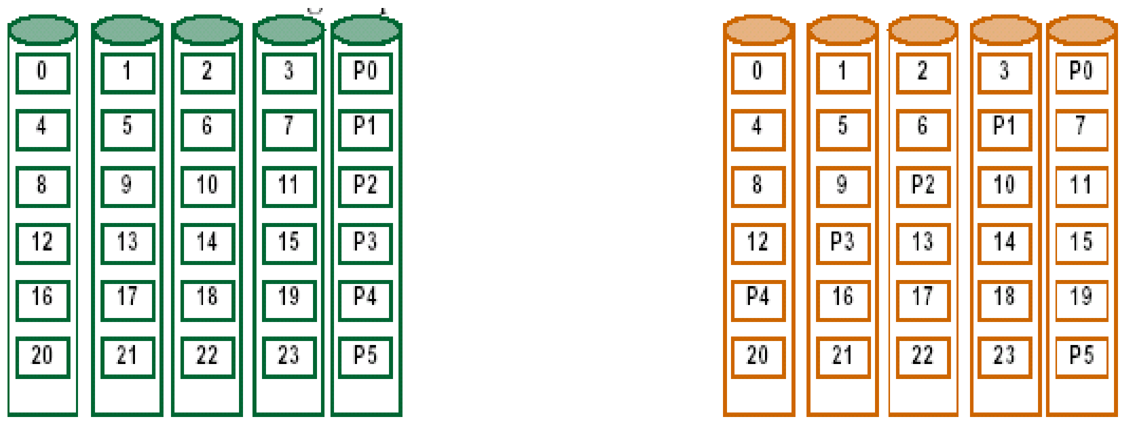


Figure 4.19

Figure 7.19 shows how data are distributed in RAID 4 vs. RAID 5. As the organization on the right shows, in RAID 5 the parity associated with each row of data blocks is no longer restricted to a single disk. This organization allows multiple writes to occur simultaneously as long as the stripe units are not located in the same disks. For example, a write to block 8 on the right must also access its parity block P2, thereby occupying the first and third disks. A second write to block 5 on the right, implying an update to its parity block P1, accesses the second and fourth disks and thus could occur at the same time as the write to block 8. Those same writes to the organization on the left would result in changes to blocks P1 and P2, both on the fifth disk, which would be a bottleneck.

P+Q redundancy (RAID 6)

Parity based schemes protect against a single, self-identifying failures. When a single failure is not sufficient, parity can be generalized to have a second calculation over the data and another check disk of information. Yet another parity block is added to allow recovery from a second failure. Thus, the storage overhead is twice that of RAID 5. The small write shortcut of Figure 7.18 works as well, ex-cept now there are six disk accesses instead of four to update both P and Q information.

Errors and Failures in Real Systems

Publications of real error rates are rare for two reasons. First academics rarely have access to significant hardware resources to measure. Second industrial, researchers are rarely allowed to publish failure information for fear that it would be used against their companies in the marketplace. Below are four exceptions.

Berkeley's Tertiary Disk

The Tertiary Disk project at the University of California created an art-image server for the Fine Arts Museums of San Francisco. This database consists of high quality images of over 70,000 art works. The database was stored on a clus-ter, which consisted of 20 PCs containing 368 disks connected by a switched Ethernet. It occupied in seven 7-foot high racks.

Component	Total in System	Total Failed	% Failed
SCSI Controller	44	1	2.3%
SCSI Cable	39	1	2.6%
SCSI Disk	368	7	1.9%
IDE Disk	24	6	25.0%
Disk Enclosure - Backplane	46	13	28.3%
Disk Enclosure - Power Supply	92	3	3.3%
Ethernet Controller	20	1	5.0%
Ethernet Switch	2	1	50.0%
Ethernet Cable	42	1	2.3%
CPU/Motherboard	20	0	0%

FIGURE 7.20 Failures of components in Tertiary Disk over eighteen months of operation.

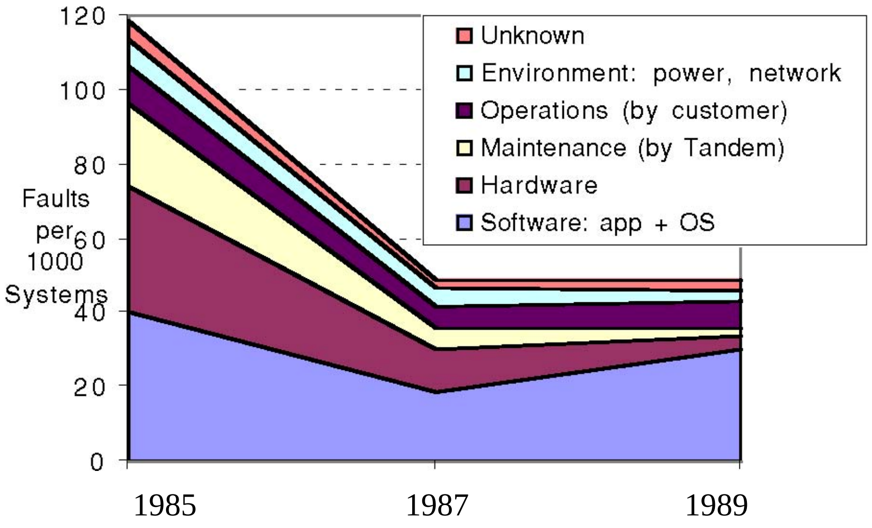
Figure 7.20 shows the failure rates of the various components of Tertiary Disk. In advance of building the system, the designers assumed that data disks would be the least reliable part of the system, as they are both mechanical and plentiful. As Tertiary Disk was a large system with many redundant components, it had the potential to survive this wide range of failures. Components were connected and mirrored images were placed no single failure could make any image unavailable. This strategy, which initially appeared to be overkill, proved to be vital.

This experience also demonstrated the difference between transient faults and hard faults. Transient faults are faults that come and go, at least temporarily fixing themselves. Hard faults stop the device from working properly, and will continue to misbehave until repaired.

Tandem

The next example comes from industry. Gray [1990] collected data on faults for Tandem Computers, which was one of the pioneering companies in fault tolerant computing. Figure 7.21 graphs the faults that caused system failures between 1985 and 1989 in absolute faults per system and in percentage of faults encoun-tered. The data shows a clear improvement in the reliability of hardware and maintenance.

Disks in 1985 needed yearly service by Tandem, but they were re-placed by disks that needed no scheduled maintenance. Shrinking number of chips and connectors per system plus software's ability to tolerate hardware faults reduced hardware's contribution to only 7% of failures by 1989. And when hardware was at fault, software embedded in the hardware device (firmware) was often the culprit. The data indicates that software in 1989 was the major source of reported outages (62%), followed by system operations (15%).



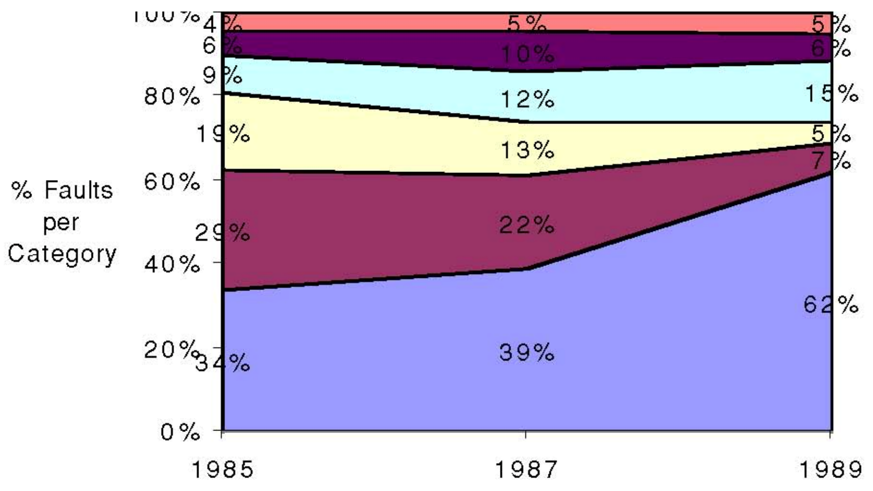


FIGURE 7.21 Faults in Tandem between 1985 and 1989. Gray [1990] collected these data for the fault tolerant Tandem computers based on reports of component failures by customers.

The problems with any such statistics are that these data only refer to what is reported; for example, environmental failures due to power outages were not reported to Tandem because they were seen as a local problem.

VAX

The next example is also from industry. Murphy and Gent [1995] measured faults in VAX systems. They classified faults as hardware, operating system, system management, or application/networking. Figure 7.22 shows their data for 1985 and 1993. They tried to improve the accuracy of data on operator faults by having the system automatically prompt the operator on each boot for the reason for that reboot. They also classified consecutive crashes to the same fault as operator fault. Note that the hardware/operating system went from causing 70% of the failures in 1985 to 28% in 1993. Murphy and Gent expected system management to be the primary dependability challenge in the future.

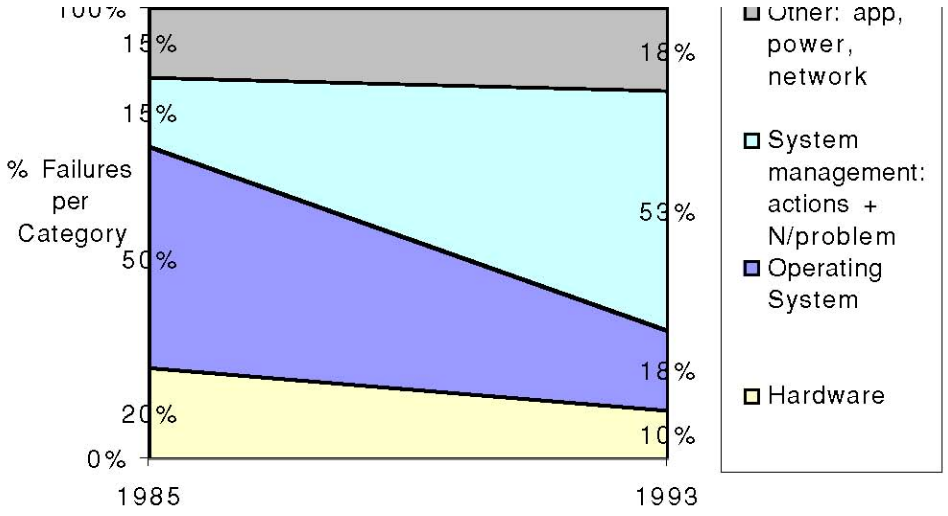


FIGURE 7.22 Causes of system failures on Digital VAX systems between 1985 and 1993 collected by Murphy and Gent [1995]. System management crashes include having several crashes for the same problem, suggesting that the problem was difficult for the operator to diagnose. It also included operator actions that directly resulted in crashes, such as giving parameters bad values, bad configurations, and bad application installation.

FCC

The final set of data comes from the government. The Federal Communications Commission (FCC) requires that all telephone companies submit explanations when they experience an outage that affects at least 30,000 people or lasts thirty minutes. These detailed disruption reports do not suffer from the self-reporting problem of earlier figures, as investigators determine the cause of the outage rather than operators of the equipment. Kuhn [1997] studied the causes of outages between 1992 and 1994 and Enriquez [2001] did a follow-up study for the first half of 2001. In addition to reporting number of outages, the FCC data includes the number of customers affected and how long they were affected. Hence, we can look at the size and scope of failures, rather than assuming that all are equally important. Figure 7.23 plots the absolute and relative number of customer-outage minutes for those years, broken into four categories:

- Failures due to exceeding the network's capacity (overload).
- Failures due to people (human).
- Outages caused by faults in the telephone network software (software).
- Switch failure, cable failure, and power failure (hardware).

These four examples and others suggest that the primary cause of failures in large systems today is faults by human operators. Hardware faults have declined due to a decreasing number of chips in systems, reduced power, and fewer connectors. Hardware dependability has improved through fault tolerance techniques such as RAID. At least some operating systems are considering reliability implications before new adding features, so in 2001 the failures largely occur elsewhere.

4.6 Benchmarks of storage performance and availability

Transaction Processing Benchmarks

Transaction processing (TP, or OLTP for on-line transaction processing) is chiefly concerned with I/O rate: the number of disk accesses per second, as opposed to data rate, measured as bytes of data per second. TP generally involves changes to a large body of shared information from many terminals, with the TP system guaranteeing proper behavior on a failure. Suppose, for example, a bank's computer fails when a customer tries to withdraw money. The TP system would guarantee that the account is debited if the customer received the money and that the account is unchanged if the money was not received. Airline reservations systems as well as banks are traditional customers for TP.

This report led to the Transaction Processing Council, which in turn has led to seven benchmarks since its founding.

The TPC benchmarks were either the first, and in some cases still the only ones, that have these unusual characteristics:

Price is included with the benchmark results. The cost of hardware, software, and five-year maintenance agreements is included in a submission, which en-ables evaluations based on price-performance as well as high performance.

Benchmark	Data Size (GB)	Performance Metric	Date of First Results
A: Debit Credit (retired)	0.1 to 10	transactions per second	July, 1990
B: Batch Debit Credit (retired)	0.1 to 10	transactions per second	July, 1991
C: Complex Query OLTP	100 to 3000 (minimum 0.07 * tpm)	new order transactions per minute	September, 1992
D: Decision Support (retired)	100, 300, 1000	queries per hour	December, 1995
H: Ad hoc decision support	100, 300, 1000	queries per hour	October, 1999
R: Business reporting decision support	1000	queries per hour	August, 1999
W: Transactional web benchmark	≈ 50, 500	web interactions per second	July, 2000

FIGURE 7.31 Transaction Processing Council Benchmarks. The summary results include both the performance metric and the price-performance of that metric. TPC-A, TPC-B, and TPC-D were retired.

The data set generally must scale in size as the throughput increases. The benchmarks are trying to model real systems, in which the demand on the sys-tem and the size of the data stored in it increase together. It makes no sense, for example, to have thousands of people per minute access hundreds of bank ac-counts.

The benchmark results are audited. Before results can be submitted, they must be approved by a certified TPC auditor, who enforces the TPC rules that try to make sure that only fair results are submitted. Results can be challenged and disputes resolved by going before the TPC council.

Throughput is the performance metric but response times are limited. For ex-ample, with TPC-C, 90% of the New-Order transaction response times must be less than 5 seconds.

An independent organization maintains the benchmarks. Dues collected by TPC pay for an administrative structure including a Chief Operating Office. This organization settles disputes, conducts mail ballots on approval of changes to benchmarks, hold board meetings, and so on.

SPEC System-Level File Server (SFS) and Web Benchmarks:

The SPEC benchmarking effort is best known for its characterization of processor performance, but has created benchmarks for other fields as well. In 1990 seven companies agreed on a synthetic benchmark, called SFS, to evaluate systems running the Sun Microsystems network file service NFS. This benchmark was upgraded to SFS 2.0 (also called SPEC SFS97) to include support for NSF version 3, using TCP in addition to UDP as the transport protocol, and making the mix of operations more realistic.

Figure 7.32 shows average response time versus throughput for four systems. Unfortunately, unlike the TPC benchmarks, SFS does not normalize for different price configurations. The fastest system in Figure 7.32 has 7 times the number of CPUs and disks as the slowest system, but SPEC leaves it to you to calculate price versus performance. As performance scaled to new heights, SPEC discovered bugs in the benchmark that impact the amount of work done during the measurement periods. Hence, it was retired in June 2001.

SPEC WEB is a benchmark for evaluating the performance of World Wide Web servers. The SPEC WEB99 workload simulates accesses to a web service provider, where the server supports home pages for several organizations. Each home page is a collection of files ranging in size from small icons to large docu-ments and images, with some files being more popular than others. The workload defines four sizes of files and their frequency of activity:

less than 1 KB, representing an small icon: 35% of activity

1 to 10 KB: 50% of activity

10 to 100 KB: 14% of activity

100 KB to 1 MB: representing a large document and image,1% of activity

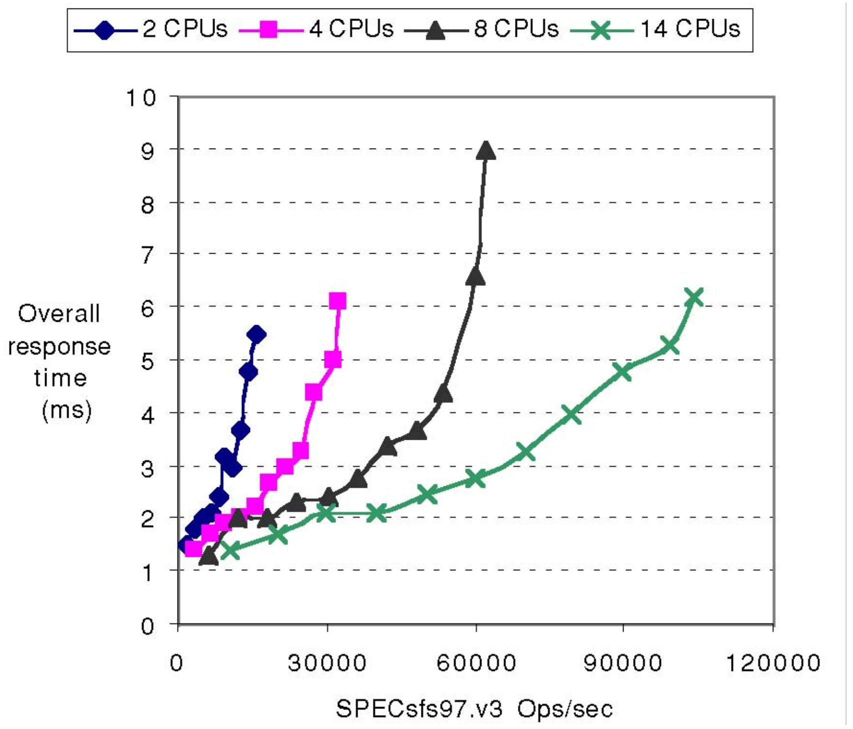


Figure 7.33 shows results for Dell computers. The performance result represents the number of simultaneous connections the web server can support using the predefined workload. As the disk system is the same, it appears that the large memory is used for a file cache to reduce disk I/O.

System Name	Result	CPUs	Result/ CPU	HTTP Version/OS	Pentium III	DRAM
PowerEdge 2400/667	732	1	732	IIS 5.0/Windows 2000	667 MHz EB	2 GB
PowerEdge 2400/667	1270	1	1270	TUX 1.0/Red Hat Linux 6.2	667 MHz EB	2 GB
PowerEdge 4400/800	1060	2	530	IIS 5.0/Windows 2000	800 MHz EB	4 GB
PowerEdge 4400/800	2200	2	1100	TUX 1.0/Red Hat Linux 6.2	800 MHz EB	4 GB
PowerEdge 6400/700	1598	4	400	IIS 5.0/Windows 2000	700 MHz Xeon	8 GB
PowerEdge 6400/700	4200	4	1050	TUX 1.0/Red Hat Linux 6.2	700 MHz Xeon	8 GB

FIGURE 7.33 SPEC WEB99 results in 2000 for Dell computers. Each machine uses five 9GB, 10,000 RPM disks except the fifth system, which had seven disk. The first four have 256 KB of L2 cache while the last two have 2 MB of L2 cache.

4.7 Design and I/O System in Five Easy Pieces

The art of I/O system design is to find a design that meets goals for cost, dependability, and variety of devices while avoiding bottlenecks to I/O performance. Avoiding bottlenecks means that components must be balanced between main memory and the I/O device, because performance and hence effective cost/performance can only be as good as the weakest link in the I/O chain. Finally, storage must be dependable, adding new constraints on proposed designs.

In designing an I/O system, analyze performance, cost, capacity, and availability using varying I/O connection schemes and different numbers of I/O devices of each type. Here is one series of steps to follow in designing an I/O system. The answers for each step may be dictated by market requirements or simply by cost, performance, and availability goals.

1. List the different types of I/O devices to be connected to the machine, or list the standard buses that the machine will support.
2. List the physical requirements for each I/O device. Requirements include size, power, connectors, bus slots, expansion cabinets, and so on.
3. List the cost of each I/O device, including the portion of cost of any controller needed for this device.
4. List the reliability of each I/O device.
5. Record the CPU resource demands of each I/O device.

This list should include

- Clock cycles for instructions used to initiate an I/O, to support operation of an I/O device (such as handling interrupts), and complete I/O
- CPU clock stalls due to waiting for I/O to finish using the memory, bus, or cache
- CPU clock cycles to recover from an I/O activity, such as a cache flush

List the memory and I/O bus resource demands of each I/O device. Even when the CPU is not using memory, the bandwidth of main memory and the I/O bus is limited.

The final step is assessing the performance and availability of the different ways to organize these I/O devices. Performance can only be properly evaluated with simulation, though it may be estimated using queuing theory. Reliability can be calculated assuming I/O devices fail independently and are that MTTFs are exponentially distributed. Availability can be computed from reliability by estimating MTTF for the devices, taking into account the time from failure to repair.

Cost/performance goals affect the selection of the I/O scheme and physical design. Performance can be measured either as megabytes per second or I/Os per second, depending on the needs of the application. For high performance, the only limits should be speed of I/O devices, number of I/O devices, and speed of memory and CPU. For low cost, the only expenses should be those for the I/O devices themselves and for cabling to the CPU. Cost/performance design, of course, tries for the best of both worlds. Availability goals depend in part on the cost of unavailability to an organization.

To make these ideas clearer, the next dozen pages go through five examples. Each looks at constructing a disk array with about 2 terabytes of capacity for user data with two sizes of disks. To offer a gentle introduction to I/O design and evaluation, the examples evolve in realism.

To try to avoid getting lost in the details, let's start with an overview of the five examples:

- Naive cost-performance design and evaluation: The first example calculates cost-performance of an I/O system for the two types of disks. It ignores dependability concerns, and makes the simplifying assumption of allowing 100% utilization of I/O resources. This example is also the longest.
- Availability of the first example: The second example calculates the poor availability of this naive I/O design.
- Response times of the first example: The third example uses queuing theory to calculate the impact on response time of trying to use 100% of an I/O resource.
- More realistic cost-performance design and evaluation: Since the third example shows the folly of 100% utilization, the fourth example changes the design to obey common rules of thumb on utilization of I/O resources. It then evaluates cost-performance.
- More realistic design for availability and its evaluation: Since the second example shows the poor availability when dependability is ignored, this final example uses a RAID 5 design. It then calculates availability and performance.

UNIT V

MULTI-CORE ARCHITECTURES

Software and hardware multithreading - SMT and CMP architectures - Design issues - Case studies - Intel Multi-core architecture - SUN CMP architecture - heterogeneous multi-core processors - case study: IBM Cell Processor

5.1 Multi-threading

The ability of an operating system to execute different parts of a program, called threads, simultaneously. The programmer must carefully design the program in such a way that all the threads can run at the same time without interfering with each other

Advantages of Multi-threading

If a thread can not use all the computing resources of the CPU (because instructions depend on each other's result), running another thread permits to not leave these idle. If several threads work on the same set of data, they can actually share its caching, leading to better cache usage or synchronization on its values.

If a thread gets a lot of cache misses, the other thread(s) can continue, taking advantage of the unused computing resources, which thus can lead to faster overall execution, as these resources would have been idle if only a single thread was executed

5.1.1 Two levels of thread

Two levels of thread User level(for user thread) Kernel level(for kernel thread)

User Threads

User threads are supported above the kernel and are implemented by a thread library at the user level. The library provides support for thread creation, scheduling, and management with no support from the kernel.Because the kernel is unaware of user-level threads, all thread creation and scheduling are done in user space without the need for kernel intervention.

User-level threads are generally fast to create and manage User-thread libraries include POSIX Pthreads,Mach C-threads,and Solaris 2 UI-threads.

Kernel Threads

Kernel threads are supported directly by the operating system: The kernel performs thread creation, scheduling, and management in kernel space. Because thread management is done by the operating system, kernel threads are generally slower to create and manage than are user threads. Most operating systems-including Windows NT, Windows 2000, Solaris 2, BeOS, and Tru64 UNIX (formerly Digital UN1X)-support kernel threads

Multi-threading Models

There are three models for thread libraries, each with its own trade-offs

- Many threads on one LWP (many-to-one)
- One thread per LWP (one-to-one)
- Many threads on many LWPs (many-to-many)

Many-to-one

The many-to-one model maps many user-level threads to one kernel thread. Advantages: Totally portable More efficient Disadvantages: cannot take advantage of parallelism The entire process is block if a thread makes a blocking system call Mainly used in language systems, portable libraries like solaris 2

One-to-one

The one-to-one model maps each user thread to a kernel thread. Advantages: allows parallelism Provide more concurrency Disadvantages: Each user thread requires corresponding kernel thread limiting the number of total threads Used in LinuxThreads and other systems like Windows 2000,Windows NT

Many-to-many

The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads. Advantages: Can create as many user thread as necessary Allows parallelism Disadvantages: kernel thread can the burden the performance Used in the Solaris implementation of Pthreads (and several other Unix implementations)?

5.2 SMT and CMP Architectures

Instruction-level parallelism(ILP)

- ➢ Wide-issue Superscalar processors (SS)
  - ➢ Four or more instruction per cycle
  - ➢ Executing a single program or thread
  - ➢ Attempts to find multiple instructions to issue each cycle.
  - ➢ Out-of-order execution => instructions are sent to execution units based on instruction dependencies rather than program order

Thread-level parallelism(TLP)

- ➢ Fine-grained multithreaded superscalars(FGMS)
  - ➢ Contain hardware state for several threads
  - ➢ Executing multiple threads
  - ➢ On any given cycle a processor executes instructions from one of the threads
- ➢ Multiprocessor(MP)
  - ➢ Performance improved by adding more CPUs

Simultaneous Multithreading

The idea is issue multiple instructions from multiple threads each cycle

The Features are

- ➢ Fully exploit thread-level parallelism and instruction-level parallelism.
- ➢ Multiple functional units
  - ➢ Modern processors have more functional units available then a single thread can utilize.

- Register renaming and dynamic scheduling
  - Multiple instructions from independent threads can co-exist and co-execute.

Superscalar processor with no multithreading:

Only one thread is processed in one clock cycle

- Use of issue slots is limited by a lack of ILP.
- Stalls such as an instruction cache miss leaves the entire processor idle.

Fine grained Multithreading

Switches threads on every clock cycle

- Pro: hide latency of from both short and long stalls
- Con: Slows down execution of the individual threads ready to go. Only one thread issues inst. In a given clock cycle.

Course-grained multithreading:

Switches threads only on costly stalls (e.g., L2 stalls)

- Pros: no switching each clock cycle, no slow down for ready-to-go threads. Reduces no of completely idle clock cycles.
- Con: limitations in hiding shorter stalls

Simultaneous Multithreading:

Exploits TLP at the same time it exploits ILP with multiple threads using the issue slots in a single-clock cycle.

- issue slots is limited by the following factors:
  - Imbalances in the resource needs.
  - Resource availability over multiple threads.
  - Number of active threads considered.
  - Finite limitations of buffer.
  - Ability to fetch enough instructions from multiple threads.
  - Practical limitations of what instructions combinations can issue from one thread and multiple threads.

Performance Implications of SMT

- Single thread performance is likely to go down (caches, branch predictors, registers, etc. are shared) – this effect can be mitigated by trying to prioritize one thread
- While fetching instructions, thread priority can dramatically influence total throughput – a widely accepted heuristic (ICOUNT): fetch such that each thread has an equal share of processor resources
- With eight threads in a processor with many resources, SMT yields throughput improvements of roughly 2-4
- Alpha 21464 and Intel Pentium 4 are examples of SMT

Effectively Using Parallelism on a SMT Processor

Parallel workload					
threads	**SS**	**MP2**	**MP4**	**FGMT**	**SMT**
1	**3.3**	**2.4**	**1.5**	**3.3**	**3.3**
2	**--**	**4.3**	**2.6**	**4.1**	**4.7**
4	**--**	**--**	**4.2**	**4.2**	**5.6**
8	**--**	**--**	**--**	**3.5**	**6.1**

Instruction Throughput executing a parallel workload

Comparison of SMT vs Superscalar

SMT processors are compared to base superscalar processors in several key measures :

- Utilization of functional units.
- Utilization of fetch units.
- Accuracy of branch predictor.
- Hit rates of primary caches.
- Hit rates of secondary caches.

Performance improvement:

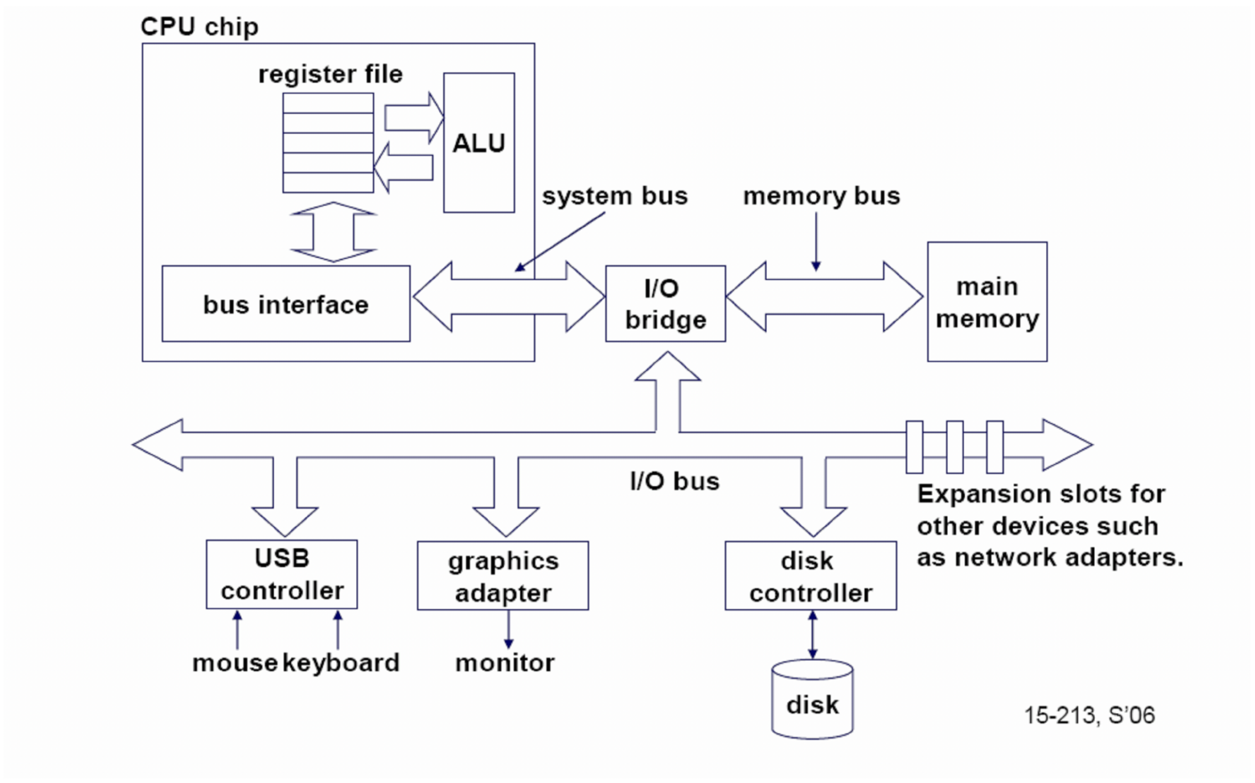
- Issue slots.
- Funtional units.
- Renaming registers**.**

5.2.1 CMP Architecture

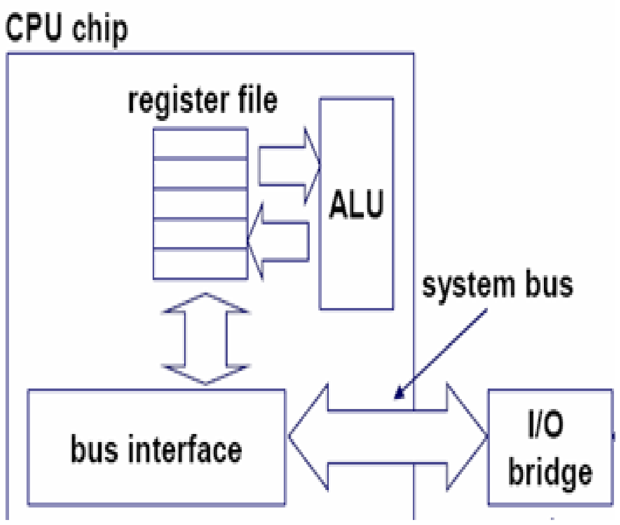
- Chip-level multiprocessing(CMP or multicore): integrates two or more independent cores(normally a CPU) into a single package composed of a single integrated circuit(IC), called a die, or more dies packaged, each executing threads independently.
- Every funtional units of a processor is duplicated.
- Multiple processors, each with a full set of architectural resources, reside on the same die
- Processors may share an on-chip cache or each can have its own cache

- Examples: HP Mako, IBM Power4
- Challenges: Power, Die area (cost)

Single core computer

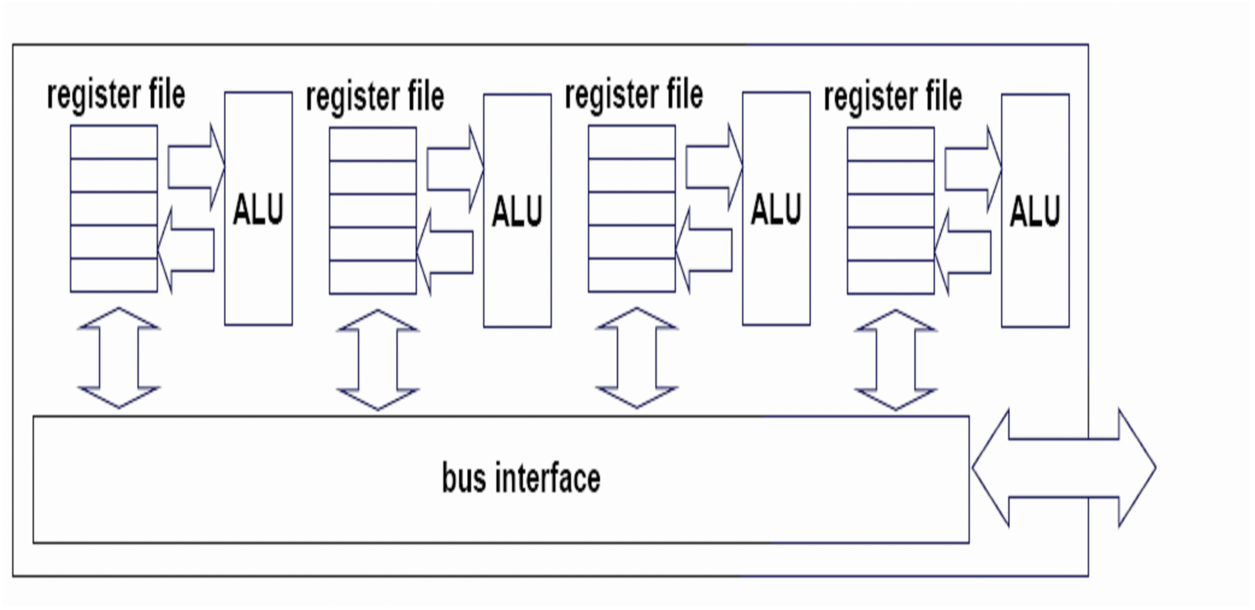


Single core CPU chip



Multi-core CPU chip

Core 1 Core 2 Core 3 Core 4



Chip Multithreading

Chip Multithreading = Chip Multiprocessing + Hardware Multithreading.

- Chip Multithreading is the capability of a processor to process multiple s/w threads simulataneous h/w threads of execution.
- CMP is achieved by multiple cores on a single chip or multiple threads on a single core.
- CMP processors are especially suited to server workloads, which generally have high levels of Thread-Level Parallelism(TLP).

CMP's Performance

- CMP's are now the only way to build high performance microprocessors , for a variety of reasons:
- Large uniprocessors are no longer scaling in performance, because it is only possible to extract a limited amount of parallelism from a typical instruction stream.
- Cannot simply ratchet up the clock speed on today's processors,or the power dissipation will become prohibitive.
- CMT processors support many h/w strands through efficient sharing of on-chip resources such as pipelines, caches and predictors.
- CMT processors are a good match for server workloads,which have high levels of TLP and relatively low levels of ILP.

SMT and CMP

- The performance race between SMT and CMP is not yet decided.
- CMP is easier to implement, but only SMT has the ability to hide latencies.
- A functional partitioning is not exactly reached within a SMT processor due to the centralized instruction issue.
  - A separation of the thread queues is a possible solution, although it does not remove the central instruction issue.
  - A combination of simultaneous multithreading with the CMP may be superior.
- Research : combine SMT or CMP organization with the ability to create threads with compiler support of fully dynamically out of a single thread.
  - Thread-level speculation
  - Close to multiscalar

5.3 DESIGN ISSUES:

SMT and CMP Architectures

They determine the performance measures of each processor in a precise manner. The issue slots usage limitations and its issues also determine the performance.Why Multithreading Today ILP is exhausted, TLP is in. Large performance gap between MEMORY and PROCESSOR. Too many transistors on chip. More existing MT applications today. Multiprocessors on a single chip. Long network latency, too

5.3.1DESIGN CHALLENGES OF SMT

Impact of fine grained scheduling on single thread performance?

A preferred thread approach sacrifices throughput and single threaded performance. Unfortunately with a preferred thread, the processor is likely to sacrifice some throughput

Reason for loss of throughput

Pipeline is less likely to have a mix of instructions from several threads resulting in a greater probability that either empty slots or a stall will occur

Design Challenges

Larger register file needed to hold multiple contexts.Not affecting clock cycle time, especially in

- Instruction issue- more candidate instructions need to be considered
- Instruction completion- choosing which instructions to commit may be challenging

Ensuring that cache and TLP conflicts generated by SMT do not degrade performance. There are mainly two observations

- Potential performance overhead due to multithreading is small
- Efficiency of current superscalar is low with the room for significant improvement

A SMT processor works well if Number of compute intensive threads does not exceed the number of threads supported in SMT. Threads have highly different charecteristics For eg; 1 thread doing mostly integer operations and another doing mostly floating point operations

It does not work well if Threads try to utilize the same functional units and for assignment problems

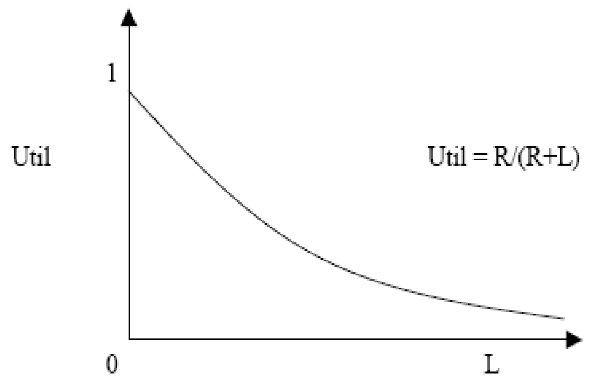
- Eg; a dual core processor system, each processor having 2 threads simultaneously
- 2 computer intensive application processes might end up on the same processor instead of different processors

The problem here is the operating system does not see the difference between the SMT and real processors !!!

Transient Faults

Faults that persist for a "short" duration. Cause is cosmic rays (e.g., neutrons).The effect is knock off electrons, discharge capacitor.The Solution is no practical absorbent for cosmic rays.1 fault per 1000 computers per year (estimated fault rate)

Processor Utilization vs. Latency



R = the run length to a long latency event

L = the amount of latency

Simultaneous & Redundantly Threaded Processor (SRT)

SRT = SMT + Fault Detection + Less hardware compared to replicated microprocessors SMT needs ~5% more hardware over uniprocessor SRT adds very little hardware overhead to existing SMT+ Better performance than complete replication better use of resources + Lower cost avoids complete replication

SRT Design Challenges

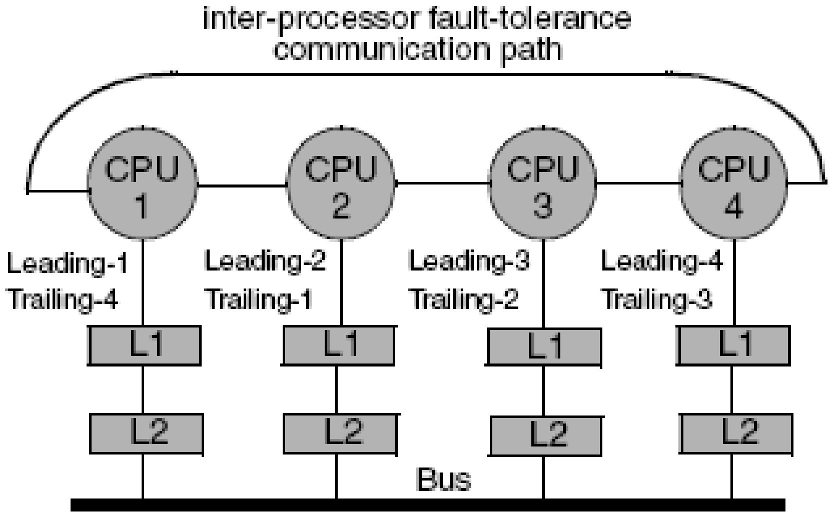
Lock stepping doesn't work because SMT may issue same instruction from redundant threads in different cycles. Must carefully fetch/schedule instructions from redundant threads since branch misprediction &cache miss will occur

Transient Fault Detection in CMPs

CRT borrows the detection scheme from the SMT-based simultaneously and Redundantly Threaded (SRT) processors and applies the scheme to CMPs.

- replicated two communicating threads (leading & trailing threads)

- Compare the results of the two.
- CRT executes the leading and trailing threads on different processors to achieve load balancing and to reduce the probability of a fault corrupting both threads



Detection is based on replication but to which extent?.Itreplicates register values (in register file in each core) but not memory values. The CRT's leading thread commits stores only after checking, so that memory is guaranteed to be correct.CRT compares only stores and uncached loads, but not register values, of the two threads.

An incorrect value caused by a fault propagates through computations and is eventually consumed by a store, checking only stores suffices for detection; other instructions commit without checking.

CRT uses a store buffer (StB) in which the leading thread places its committed store values and addresses. The store values and addresses of the trailing thread are compared against the StB entries to determine whether a fault has occurred. (one checked store reaches to the cache hierarchy)

Transient Fault Recovery for CMPs

Unlike CRT, CRTR must not allow any trailing instruction to commit before it is checked for faults, so that the register state of the trailing thread may be used for recovery. However, the leading thread in CRTR may commit register state before checking, as in CRT.

This asymmetric commit strategy allows CRTR to employ a long slack to absorb inter-processor latencies. As in CRT, CRTR commits stores only after checking. In addition to communicating branch outcomes, load addresses, load values, store addresses, and store values like CRT, CRTR also communicates register values.

Challenges with this approach

- I-Cache:
  Instruction bandwidth

- I-Cache misses:
  Since instructions are being grabbed from many different contexts, instruction locality is degraded and the I-cache miss rate rises.
- Register file access time:
  - Register file access time increases due to the fact that the regfile had to significantly increase in size to accommodate many separate contexts.
  - In fact, the HEP and Tera use SRAM to implement the regfile, which means longer access times.
- Single thread performance
  - Single thread performance significantly degraded since the context is forced to switch to a new thread even if none are available.
  - Very high bandwidth network, which is fast and wide
  - Retries on load empty or store full

To maximize SMT performance Issue slots, Functional units, Renaming registers

5.4 Case Studies

5.4.1 Multicore architecture

A multi-core design in which a single physical processor contains the core logic of more than one processor. Goal- enables a system to run more task simultaneously achieving greater overall performance

Hyper-threading or multicore?

Early PCs-capable of doing single task at a time. Later multi-threading tech., came into place. Intel's multi-threading called Hyper-threading

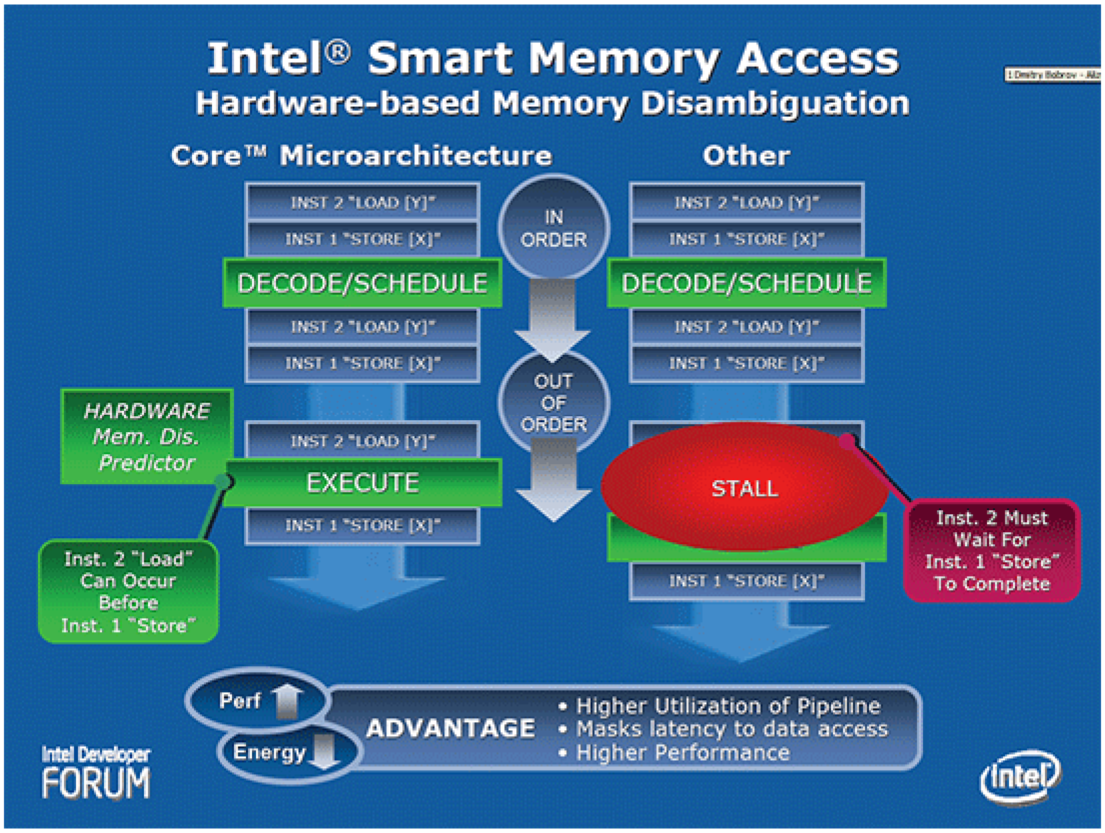
Multi-core processors

Each core has its execution pipeline. No limitation for the number of cores that can be placed in a single chip. Two cores run at slower speeds and lower temperatures. But the combined throughput > single processor. The fundamental relationship b/w freq. and power can be used to multiply the no. of cores from 2 to 4, 8 and even higher

Intel-multicore architecture

- Intel Turbo Boost Tech.
- Intel Hyper Threading Tech.
- Intel Core Microarchitecture.
- Intel Advanced Smart Cache.
- Intel Smart Memory Access.

Intel Smart Memory access


Intel® Smart Memory Access
Hardware-based Memory Disambiguation
Core™ Microarchitecture
Other
INST 2 "LOAD [Y]"
INST 1 "STORE [X]"
IN ORDER
DECODE/SCHEDULE
OUT OF ORDER
HARDWARE Mem. Dis. Predictor
EXECUTE
STALL
Inst. 2 "Load" Can Occur Before Inst. 1 "Store"
Inst. 2 Must Wait For Inst. 1 "Store" To Complete
Perf
Energy
ADVANTAGE
• Higher Utilization of Pipeline
• Masks latency to data access
• Higher Performance
Intel Developer FORUM
intel

Benefits

- Multi-core performance.
- Dynamic scalability.
- Design and performance scalability
- Intelligent performance on-demand
- Increased performance on Highly-threaded apps.
- Scalable shared memory.
- Multi-level shared cache.

5.5 IBM CELL PROCESSOR

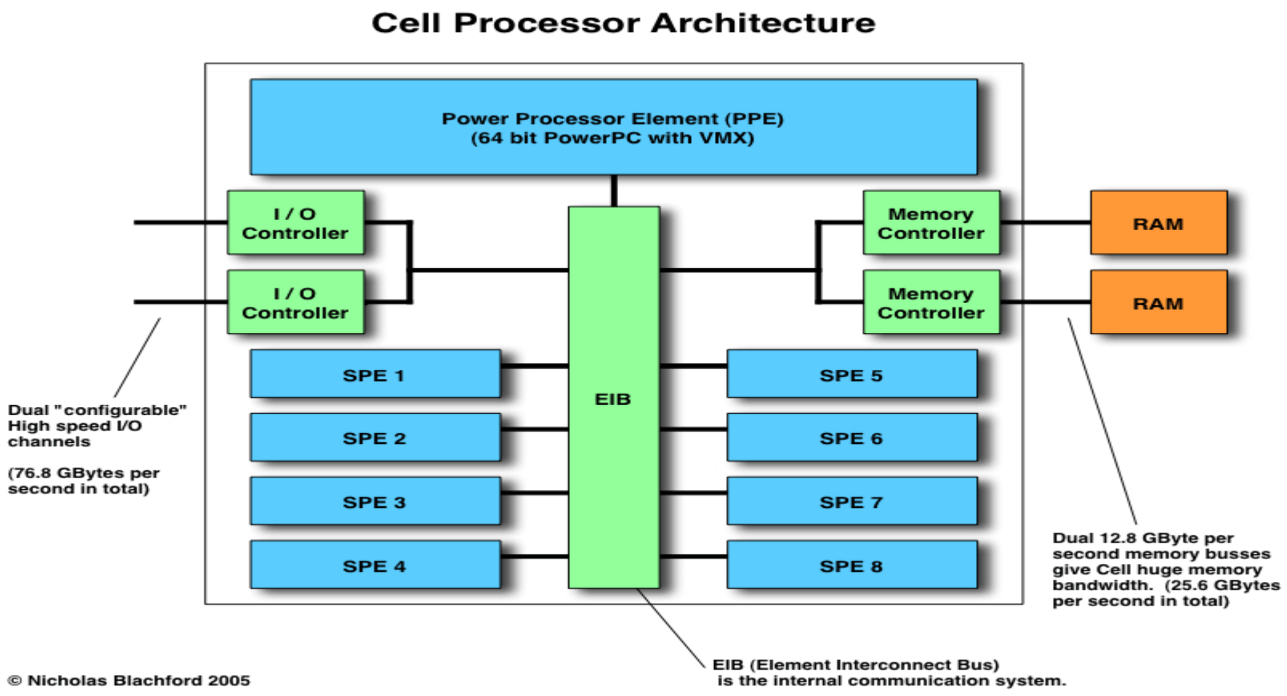
A chip with one PPC hyper-threaded core called PPE and eight specialized cores called SPEs.The challenge to be solved by the Cell was to put all those cores together on a single chip. This was made possible by the use of a bus with outstanding performance

The Cell processor can be split into four components**:**

- external input and output structures,
- the main processor called the Power Processing Element (PPE)

- eight fully-functional co-processors called the Synergistic Processing Elements, or SPEs,
- a specialized high-bandwidth circular data bus connecting the PPE, input/output elements and the SPEs, called the Element Interconnect Bus or EIB.

5.5.1 Overview of the architecture of a Cell chip


Cell Processor Architecture
Power Processor Element (PPE) (64 bit PowerPC with VMX)
I / O Controller
I / O Controller
Memory Controller
Memory Controller
RAM
RAM
EIB
SPE 1
SPE 2
SPE 3
SPE 4
SPE 5
SPE 6
SPE 7
SPE 8
Dual "configurable" High speed I/O channels
(76.8 GBytes per second in total)
Dual 12.8 GByte per second memory busses give Cell huge memory bandwidth. (25.6 GBytes per second in total)
EIB (Element Interconnect Bus) is the internal communication system.
© Nicholas Blachford 2005

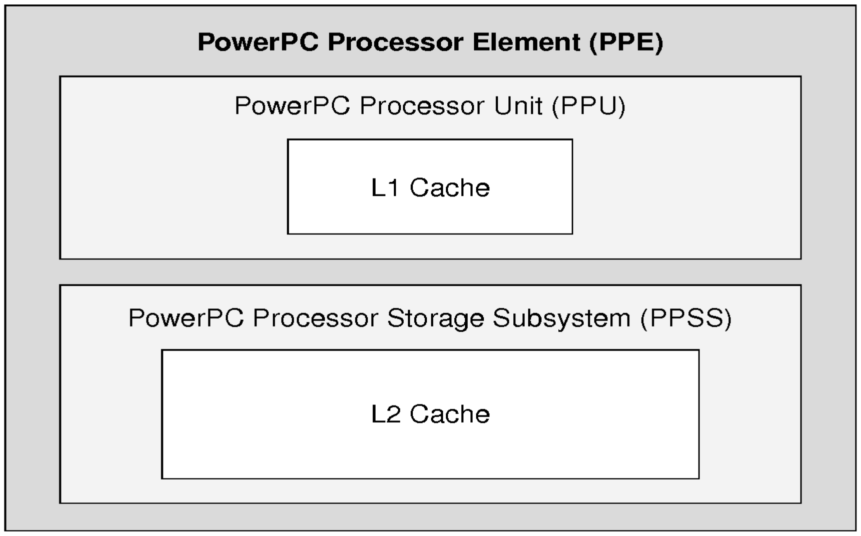
POWERPC PROCESSOR ELEMENT:(PPE)

- The PowerPC Processor Element, usually denoted as PPE is a dual-threaded powerpc processor version 2.02.
- This 64-bit RISC processor also has the Vector/SIMD Multimedia Extension.
- The PPE's role is crucial in the Cell architecture since it is on the one hand running the OS, and on the other hand controlling all other resources, including the SPEs .
- The PPE is made out of two main units:

  1: The Power Processor Unit

  2:The Power Processor Storage Subsystem (PPSS).

PPE Block diagram


PowerPC Processor Element (PPE)
PowerPC Processor Unit (PPU)
L1 Cache
PowerPC Processor Storage Subsystem (PPSS)
L2 Cache

PPU:

It is the processing part of the PPE and is composed of:

- A full set of 64-bit PowerPC registers.
- 32 128-bit vector multimedia registers.
- A 32KB L1 instruction cache.
- A 32KB L1 data cache.

All the common components of a ppc processors with vector/SIMD extensions (instruction control unit, load and store unit, fixed-Point integer unit, floating-point unit, vector unit, branch unit, virtual memory management unit).The PPU is hyper-threaded and supports 2 simultaneous threads.

PPSS

This handles all memory requests from the PPE and requests made to the PPE by other processors or I/O devices. It is composed of:

- A unified 512-KB L2 instruction and data cache.
- Various queues
- A bus interface unit that handles bus arbitration and pacing on the Element Interconnect Bus

SYNERGISTIC PROCESSOR ELEMENTS:SPE

Each Cell chip has 8 Synergistic Processor Elements. They are 128-bit RISC processor which are specialized for data-rich, compute-intensive SIMD applications. This consist of two main units.

1: The Synergistic Processor Unit (SPU)

2:The Memory Flow Controller (MFC)

The Synergistic Processor Unit (SPU):

This deals with instruction control and execution. It includes various components:

- A register file of 128 registers of 128 bits each.
- A unified instruction and data 256-kB Local Store (LS).
- A channel-and-DMA interface.
- As usual, an instruction-control unit, a load and store unit, two fixed-point units, a floating point unit.

The SPU implements a set of SIMD instructions, specific to the Cell. Each SPU is independent, and has its own program counter. Instructions are fetched in its own Local Store LS. Data are also loaded and stored in the LS

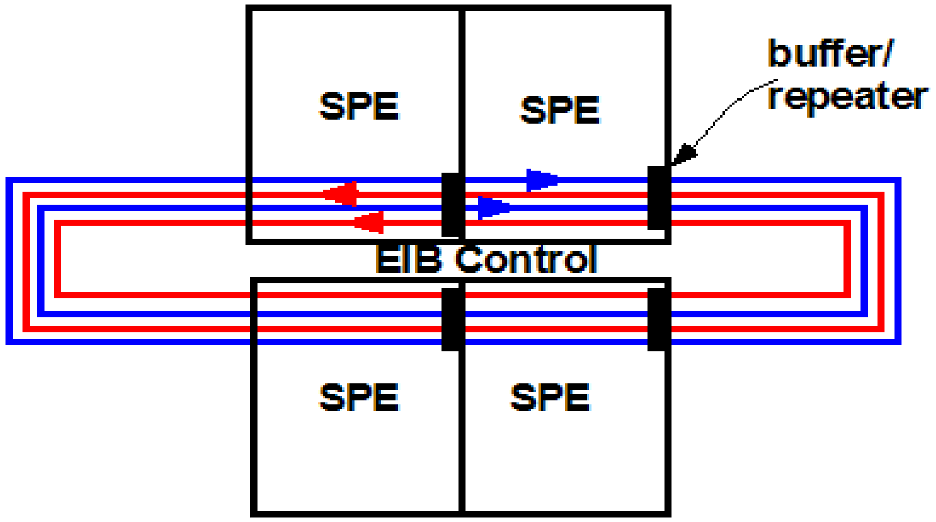
The Memory Flow Controller (MFC)

It is actually the interface between the SPU and the rest of the Cell chip.MFC interfaces the SPU with the EIB. In addition to a whole set of MMIO registers, this contains a DMA controller.

Bus design and communication among the Cell

1: The Element Interconnect Bus:

This bus makes it possible to link all parts of the chip. The EIB itself is made out of a 4-ring structure (two clockwise, and two counterclockwise) that is used to transfer data, and a tree structure used to carry commands. It is actually controlled by what is called the Data Arbitrer. This structure allows 8 simultaneous transactions on the bus.



2: Input/output interfaces:

The Memory Interface Controller (MIC).:

- It provides an interface between the EIB and the main storage.
- It currently supports two Rambus Extreme Data Rate (XDR) I/O (XIO) memory channels.

The Cell Broadband Engine Interface (BEI):

- This is the interface between the Cell and I/O devices,such as GPUs and various bridges.
- It supports two Rambus FlexIO external I/O channels.
- One of this channel only supports non-coherent transfers. The other supports either coherent or noncoherenT.

Key Attributes of Cell

- Cell is Multi-Core
- Cell is a Flexible Architecture
- Cell is a Broadband Architecture
- Cell is a Real-Time Architecture
- Cell is a Security Enabled Architecture

UNIT I

PART-A

1. Give few essential features of RISC architecture.

The RISC-based machines focused the attention of designers on two critical performance techniques, the exploitation of instruction level parallelism (initially through pipelining and later through multiple instruction issue) and the use of caches (initially in simple forms and later using more sophisticated organizations and optimizations).

The RISC-based computers raised the performance bar, forcing prior architectures to keep up or disappear.or/ both)

RISC architectures are characterized by a few key properties, which dramatically simplify their implementation:

• All operations on data apply to data in registers and typically change the entire register (32 or 64 bits per register).

• The only operations that affect memory are load and store operations that move data from memory to a register or to memory from a register, respectively.

Load and store operations that load or store less than a full register (e.g., a byte, 16 bits, or 32 bits) are often available.

• The instruction formats are few in number with all instructions typically being one size. These simple properties lead to dramatic simplifications in the implementation of pipelining, which is why these instruction sets were designed this way.

2. Power sensitive designs will avoid fixed field decoding. Why?

In RISC architecture, register specifiers are at a fixed location and decoding is done in parallel with reading registers. This technique is known as 'fixed field decoding'. In this method, we may read a register which we may not use. This doesn't help, but also doesn't hurt the performance. In case of power sensitive designs, it does waste energy for reading an unnecessary register.

3. Give the causes of structural hazards.

When a processor is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline. If some combination of instructions cannot be accommodated because of resource conflicts, the processor is said to have a *structural hazard.*

The most common instances of structural hazards arise when some functional unit is not fully pipelined. Then a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle. Another common way that structural hazards appear is when some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute. For example, a processor may have only one register-file write port, but under certain circumstances, the pipeline might want to perform two writes in a clock cycle. This will generate a structural hazard.

4. Give an example of result forwarding technique to minimize data hazard stalls. Is forwarding a software technique?

No, it is a hardware technique

Example:

Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined processor. Consider the pipelined execution of these instructions:

```
DADD    R1,R2,R3
DSUB    R4.R1.R5
AND     R6,R1,R7
OR      R8.R1.R9
XOR     R10.R1.R11
```

5. Give a sequence of code that has true dependence, anti-dependence and control dependence in it.

```
Loop:   L.D     F0,0(R1)    ;F0=array element
        ADD.D   F4,F0,F2    ;add scalar in F2
        S.D     F4,0(R1)    ;store result
        DADDUI  R1,R1,#-8   ;decrement pointer 8 bytes
        BNE     R1,R2,LOOP  ;branch R1!=R2
```

true dependence: Instrns 1,2 (R0)

antidependence: Instructions 3,4 (R1)

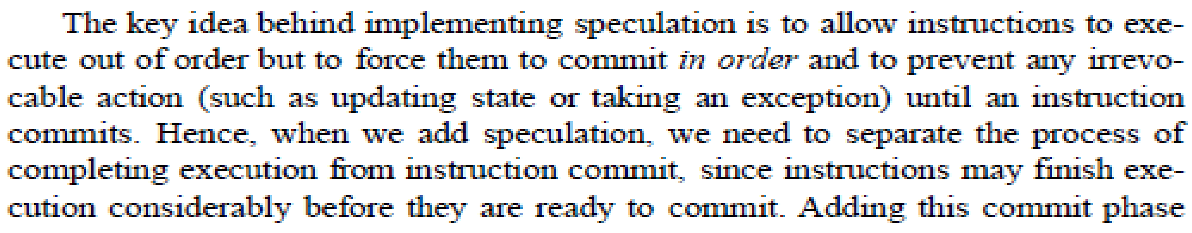
output dependence: Instructions 2,3 (F4); 4,5 (R1)

6. What is the flaw in 1-bit branch prediction scheme?

This simple 1-bit prediction scheme has a performance shortcoming: Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken, since the misprediction causes the prediction bit to be flipped.

To remedy this weakness, 2-bit prediction schemes are often used. In a 2-bit scheme, a prediction must miss twice before it is changed. Figure 2.4 shows the

7. What is the key idea behind the implementation of hardware speculation?



The key idea behind implementing speculation is to allow instructions to execute out of order but to force them to commit *in order* and to prevent any irrevocable action (such as updating state or taking an exception) until an instruction commits. Hence, when we add speculation, we need to separate the process of completing execution from instruction commit, since instructions may finish execution considerably before they are ready to commit. Adding this commit phase

8. What is trace scheduling? Which type of processors use this technique?

Trace scheduling is useful for processors with a large number of issues per clock, w here conditional or predicated execution is inappropriate or unsupported, and where simple loop unrolling may not be sufficient by itself to uncover enough ILP to keep the processor busy. Trace scheduling is a way to organize the global code motion process, so as to simplify the code scheduling by incurring the costs of possible code motion on the less frequent paths.

There are two steps to trace scheduling. The first step, called trace selection, tries to find a likely sequence of basic blocks whose operations will be put together into a smaller number of instructions; this sequence is called a trace. Loop unrolling is used to generate long traces, since loop branches are taken with high probability.

Once a trace is selected, the second process, called trace compaction, tries to squeeze the trace into a small number of wide instructions. Trace compaction is code scheduling; hence, it attempts to move operations as early as it can in a sequence (trace), packing the operations into as few wide instructions (or issue packets) as possible.

10 . Mention few limits on Instruction Level Parallelism.

1. Limitations on the Window Size and Maximum Issue Count
2. Realistic Branch and Jump Prediction
3. The Effects of Finite Registers
4. The Effects of Imperfect Alias Analysis

11. List the various data dependence.

- ➢ Data dependence
- ➢ Name dependence
- ➢ Control Dependence

12. What is Instruction Level Parallelism?

Pipelining is used to overlap the execution of instructions and improve performance. This potential overlap among instructions is called instruction level parallelism (ILP) since the instruction can be evaluated in parallel.

13. Give an example of control dependence?

if p1 {s1;}

if p2 {s2;}

S1 is control dependent on p1, and s2 is control dependent on p2

14. What is the limitation of the simple pipelining technique?

These technique uses in-order instruction issue and execution. Instructions are issued in program order, and if an instruction is stalled in the pipeline, no later instructions can proceed.

15. Briefly explain the idea behind using reservation station?

Reservation station fetches and buffers an operand as soon as available, eliminating the need to get the operand from a register.

16. Give an example for data dependence.

Loop: L.D F0,0(R1) ADD.D F4,F0,F2 S.D F4,0(R1) DADDUI R1,R1,#-8 BNE R1,R2, loop

17. Explain the idea behind dynamic scheduling?

In dynamic scheduling the hardware rearranges the instruction execution to reduce the stalls while maintaining data flow and exception behavior.

18. Mention the advantages of using dynamic scheduling?

It enables handling some cases when dependences are unknown at compile time and it simplifies the compiler. It allows code that was compiled with one pipeline in mind run efficiently on a different pipeline.

19. What are the possibilities for imprecise exception?

The pipeline may have already completed instructions that are later in program order than instruction causing exception. The pipeline may have not yet completed some instructions that are earlier in program order than the instructions causing exception.

20. What are multilevel branch predictors?

These predictors use several levels of branch-prediction tables together with an algorithm for choosing among the multiple predictors.

21. What are branch-target buffers?

To reduce the branch penalty we need to know from what address to fetch by end of IF (instruction fetch). A branch prediction cache that stores the predicted address for the next instruction after a branch is called a branch-target buffer or branch target cache.

22. Briefly explain the goal of multiple-issue processor?

The goal of multiple issue processors is to allow multiple instructions to issue in a clock cycle. They come in two flavors: superscalar processors and VLIW processors.

23. What is speculation?

Speculation allows execution of instruction before control dependences are resolved.

24. Mention the purpose of using Branch history table?

It is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not.

25. What are super scalar processors?

Superscalar processors issue varying number of instructions per clock and are either statically scheduled or dynamically scheduled.

26. Mention the idea behind hardware-based speculation?

It combines three key ideas: dynamic branch prediction to choose which instruction to execute, speculation to allow the execution of instructions before control dependences are resolved and dynamic scheduling to deal with the scheduling of different combinations of basic blocks.

27. What are the fields in the ROB?

Instruction type Destination field Value field Ready field

2**8. How many branch selected entries are in a (2,2) predictors that has a total of 8K bits in a prediction buffer?**

number of prediction entries selected by the branch = 8K number of prediction entries selected by the branch = 1K

29. What is the advantage of using instruction type field in ROB?

The instruction field specifies whether instruction is a branch or a store or a register operation

30. Mention the advantage of using tournament based predictors?

The advantage of tournament predictor is its ability to select the right predictor for right branch.

PART-B

1. What is instruction-level parallelism? Explain in details about the various dependences caused in ILP?

2. Explain in details about static branch prediction and dynamic branch prediction

3. Explain the techniques to overcome data hazards with dynamic scheduling?

4. Explain in detail the hardware based speculation for a MIPS processor

UNIT-II

PART - A

1. What is loop unrolling?

A simple scheme for increasing the number of instructions relative to the branch and overhead instructions is loop unrolling. Unrolling simply replicates the loop body multiple times, adjusting the loop termination code.

2. When static branch predictors are used?

They are used in processors where the expectation is that the branch behavior is highly predictable at compile time. Static predictors are also used to assists dynamic predictors.

3. Mention the different methods to predict branch behavior?

Predict the branch as taken Predict on basis of branch direction (either forward or backward) Predict using profile information collected from earlier runs.

4. Explain the VLIW approach?

They uses multiple, independent functional units. Rather than attempting to issue multiple, independent instructions to the units, a VLIW packages the multiple operations into one very long instruction.

5. Mention the techniques to compact the code size in instructions?

Using encoding techniques Compress the instruction in main memory and expand them when they are read into the cache or are decoded.

6. Mention the advantage of using multiple issue processor?

They are less expensive. They have cache based memory system. More parallelism.

7. What are loop carried dependence?

They focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations; such a dependence is called loop carried dependence. e.g for(i=1000;i>0;i=i-1) x[i]=x[i]+s;

8. Mention the tasks involved in finding dependences in instructions?

Good scheduling of code. Determining which loops might contain parallelism Eliminating name dependence

9. Use the G.C.D test to determine whether dependence exists in the following loop: for(i=1;i<=100;i=i+1) X[2*i+3]=X[2*i]*5.0;

Solution: a=2,b=3,c=2,d=0 GCD(a,c)=2 and d-b=-3 Since 2 does not divide -3, no dependence is possible.

10. What is software pipelining?

Software pipelining is a technique for reorganizing loops such that each iteration in the software pipelined code is made from instruction chosen from different iterations of the original loop.

11. What is global code scheduling?

Global code scheduling aims o compact code fragment with internal control structure into the shortest possible sequence that preserves the data and control dependence. Finding a shortest possible sequence is finding the shortest sequence for the critical path.

12. What is trace?

Trace selection tries to find a likely sequence of basic blocks whose operations will be put together into a smaller number of instructions; this sequence is called trace.

13. Mention the steps followed in trace scheduling?

Trace selection Trace compaction

14. What is superblock?

Superblocks are formed by a process similar to that used for traces, but are a form of extended basic block, which are restricted to a single entry point but allow multiple exits.

15. Mention the advantages of predicated instructions?

Remove control dependence Maintain data flow enforced by branch Reduce overhead of global code scheduling

16. Mention the limitations of predicated instructions?

They are useful only when the predicate can be evaluated early. Predicated instructions may have speed penalty.

17. What is poison bit?

Poison bits are a set of status bits that are attached to the result registers written by the speculated instruction when the instruction causes exceptions. The poison bits cause a fault hen a normal instruction attempts to use the register.

18. What are the disadvantages of supporting speculation in hardware?

Complexity Additional hardware resources required

19. Mention the methods for preserving exception behavior?

Ignore Exception Instructions that never raise exceptions are used Using poison bits Using hardware buffers

20. What is an instruction group?

It is a sequence of consecutive instructions with no register data dependence among them. All the instructions in the group could be executed in parallel. An instruction group can be arbitrarily long.

PART-B

1. Explain loop unrolling with an example? Loop unrolling technique Example

2. Discuss about the VLIW approach? VLIW approach basic idea

3. Explain the different techniques to exploit and expose more parallelism using compiler support?

4. Explain how hardware supports for exposing more parallelism at compile time?

5. Differentiate hardware and software speculation mechanisms?

6. Explain the limitations of ILP?

UNIT III

PART A

1. **What do you think are the reasons for the increasing importance of multi-processors?**

- A growing interest in servers and server performance
- A growth in data-intensive applications
- The insight that increasing performance on the desktop is less important (outside of graphics, at least)
- An improved understanding of how to use multiprocessors effectively, especially in server environments where there is significant natural thread-level parallelism
- The advantages of leveraging a design investment by replication rather than unique design—all multiprocessor designs provide such leverage

2. **Define process and thread in the context of multi-processors.**

With an MIMD, each processor is executing its own instruction stream. In many cases, each processor executes a different process. A process is a segment of code that may be run independently; the state of the process contains all the information necessary to execute that program on a processor. In a multiprogrammed environment, where the processors may be running independent tasks, each process is typically independent of other processes. It is also useful to be able to have multiple processors executing a single program and sharing the code and most of their address space. When multiple processes share code and data in this way, they are often called threads. Today, the term thread is often used in a casual way to refer to multiple loci of execution that may run on different processors, even when they do not share an address space. For example, a multithreaded architecture actually allows the simultaneous execution of multiple processes, with potentially separate address spaces, as well as multiple threads that share the same address space.

3. **What do you understand by grain size?. What is it's impact on parallelism?**

Although the amount of computation assigned to a thread, called the grain size, is important in considering how to exploit thread-level parallelism efficiently, the important qualitative distinction from instruction-level parallelism is that thread-level parallelism is identified at a high level by the software system and that the threads consist of hundreds to millions of instructions that may be executed in parallel.

Threads can also be used to exploit data-level parallelism, although the overhead is likely to be higher than would be seen in an SIMD computer. This overhead means that grain size must be sufficiently large to exploit the parallelism efficiently. For example, although a vector processor (see Appendix F) may be able to efficiently parallelize operations on short vectors, the resulting grain size when the parallelism is split among many threads may be so small that the overhead makes the exploitation of the parallelism prohibitively expensive.

4. Why Symmetric Shared memory architecture is called as UMA?

Because there is a single main memory that has a symmetric relationship to all processors and a uniform access time from any processor, these multiprocessors are most often called symmetric (shared-memory) multiprocessors (SMPs), and this style of architecture is sometimes called uniform memory access (UMA), arising from the fact that all processors have a uniform latency from memory, even if the memory is organized into multiple banks. This type of symmetric shared-memory architecture is currently by far the most popular organization.

5. List the two models available for communication in multi-processing environment.

Shared memory and Message passing multiprocesors

6. What are the challenges in parallel processing?

The first hurdle has to do with the limited parallelism available in programs, and the second arises from the relatively high cost of communications. Limitations in available parallelism make it difficult to achieve good speedups in any parallel processor.

The second major challenge in parallel processing involves the large latency of remote access in a parallel processor. In existing shared-memory multiprocessors, communication of data between processors may cost anywhere from 50 clock cycles (for multicores) to over 1000 clock cycles (for large-scale multiprocessors), depending on the communication mechanism, the type of interconnection network, and the scale of the multiprocessor. The effect of long communication delays is clearly substantial.

7. What do you understand by Cache coherence Problem? Give an example.

Unfortunately, caching shared data introduces a new problem because the view of memory held by two different processors is through their individual caches, which, without any additional precautions, could end up seeing two different values. Figure 4.3 illustrates the problem and shows how two different processors can have two different values for the same location. This difficulty is generally referred to as the cache coherence problem.

8. When can we say that the memory is coherent in a multi-processor system?

A memory system is coherent if

1. A read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.

2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.

3. Writes to the same location are serialized; that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

The first property simply preserves program order—we expect this property to be true even in uniprocessors. The second property defines the notion of what it means to have a coherent view of memory: If a processor could continuously read an old data value, we would clearly say that memory was incoherent.

9. Why serialization of reads and writes are important in an multi-processor environment?

The need for write serialization is more subtle, but important in an multi-processor environment. Suppose we did not serialize writes, and processor PI writes location X followed by P2 writing location X. Serializing the writes ensures that every processor will see the write done by P2 at some point. If we did not serialize the writes, it might be the case that some processor could see the write of P2 first and then see the write of PI, maintaining the value written by PI indefinitely. The simplest way to avoid such difficulties is to ensure that all writes to the same location are seen in the same order; this property is called write serialization.

10. Define memory coherence and consistency properties. Why are they important?

Informally, we could say that a memory system is coherent if any read of a data written value of that data item. This definition, although intuitively appealing, is vague and simplistic; the reality is much more complex. This simple definition contains two different aspects of memory system behavior, both of which are critical to writing correct shared-memory programs. The first aspect, called coherence, defines what values can be returned by a read. The second aspect, called consistency, determines when a written value will be returned by a read.

Coherence and consistency are complementary: Coherence defines the behavior of reads and writes to the same memory location, while consistency defines the behavior of reads and writes with respect to accesses to other memory locations. For now, make the following two assumptions. First, a write does not complete (and allow the next write to occur) until all processors have seen the effect of that write. Second, the processor does not change the order of any write with respect to any other memory access. These two conditions mean that if a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A. These restrictions allow the processor to reorder reads, but forces the processor to finish a write in program order.

11. List the two protocols used to track the status of the shared data block. How the status is maintained in both the schemes?

The protocols to maintain coherence for multiple processors are called cache coherence protocols. Key to implementing a cache coherence protocol is tracking the state of any sharing of a data block. There are two classes of protocols, which use different techniques to track the sharing status, in use:

a. Directory based—The sharing status of a block of physical memory is kept in just one location, called the directory; Directory-based coherence has slightly higher implementation overhead than snooping, but it can scale to larger processor counts. The Sun Tl design, uses directories, albeit with a central physical memory.

b. Snooping—Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, but no centralized state is kept. The caches are all accessible via some broadcast medium (a bus or switch), and all cache controllers monitor or snoop on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access.

12. What do you understand by write update protocol?

The alternative to an invalidate protocol is to update all the cached copies of a data item when that item is written. This type of protocol is called a write update or write broadcast protocol. Because a write update protocol must broadcast all writes to shared cache lines, it consumes considerably more bandwidth. For this reason, all recent multiprocessors have opted to implement a write invalidate protocol.

13. Which protocol is more suited for distributed shared memory architecture with large number of processors. Why?

The protocols to maintain coherence for multiple processors are called cache coherence protocols. Key to implementing a cache coherence protocol is tracking the state of any sharing of a data block. There are two classes of protocols, which use different techniques to track the sharing status, in use:

a. Directory based—The sharing status of a block of physical memory is kept in just one location, called the directory; Directory-based coherence has slightly higher implementation overhead than snooping, but it can scale to larger processor counts. The Sun Tl design, uses directories, albeit with a central physical memory.

b. Snooping—Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, but no centralized state is kept. The caches are all accessible via some broadcast medium (a bus or switch), and all cache controllers monitor or snoop on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access.

14.What is multi threading?

Multithreading allows multiple threads to share the functional uits of the single processor in an overlapping fashion.

15. What is fine grained multithreading?

It switches between threads on each instruction, causing the execution of multiple threads to be interleaved.

16. What is coarse grained multithreading?

It switches threads only on costly stalls. Thus it is much less likely to slow down the execution of an individual thread.

PART-B

1. Explain the concepts of centralized shared-memory and distributed-memory architecture for multi- processors with suitable block diagrams.

2. Discuss the various memory consistency models that are applicable for multi-processor systems

3. What are the metrics used to measure the performance of an I/O system? Discuss them with respect to Transaction Processing Benchmarks (6 marks).

4. Explain the basic implementation of Snooping Protocols with suitable transitions Diagrams

5. Explain the concepts of Multithreading.

UNIT IV

PART A

1. What is server utilization?

Mean number of tasks being serviced divided by service rate Server utilization = Arrival Rate/Server Rate The value should be between 0 and 1 otherwise there would be more tasks arriving than could be serviced.

2. What are the steps to design an I/O system?

- Naïve cost-performance design and evaluation
- Availability of naïve design
- Response time
- Realistic cost-performance, design and evaluation
- Realistic design for availability and its evaluation.

3. Briefly discuss about classification of buses?

I/O buses - These buses are lengthy ad have any types of devices connected to it. CPU memory buses – They are short and generally of high speed.

4. Explain about bus transactions?

Read transaction – Transfer data from memory Write transaction – Writes data to memory

5. What is the bus master?

Bus masters are devices that can initiate the read or write transaction. E.g CPU is always a bus master. The bus can have many masters when there are multiple CPU's and when the Input devices can initiate bus transaction.

6. Mention the advantage of using bus master?

It offers higher bandwidth by using packets, as opposed to holding the bus for full transaction.

7. What is spilt transaction?

The idea behind this is to split the bus into request and replies, so that the bus can be used in the time between request and the reply

8. **What do you understand by true sharing misses?**

True sharing misses are a type of coherent misses that arise from the communication of data through the cache coherence mechanism. In an invalidation based protocol, the first write by a processor to a shared cache block causes an invalidation to establish ownership of that block. Additionally, when another processor attempts to read a modified word in that cache block, a miss occurs and the resultant block is transferred. Both these misses are classified as true sharing misses since they directly arise from the sharing of data among processors.

9. **Assume that words xl and x2 are in the same cache block, which is in the shared state in the caches of both PI and P2. Assuming the following sequence of events, identify each miss as a true sharing miss, a false sharing miss, or a hit. Any miss that would occur if the block size were one word is designated a true sharing miss**.

Time	P1	P2
1	Write xl	
2		Read x2
3	Write xl	
4		Write x2
5	Read x2	

Here are classifications by time step:

1. This event is a true sharing miss, since xl was read by P2 and needs to be invalidated from P2.

2. This event is a false sharing miss, since x2 was invalidated by the write of xl in PI, but that value of xl is not used in P2.

3. This event is a false sharing miss, since the block containing xl is marked shared due to the read in P2, but P2 did not read xl. The cache block containing xl will be in the shared state after the read by P2; a write miss is required to obtain exclusive access to the block. In some protocols this will be handled as an upgrade request, which generates a bus invalidate, but does not transfer the cache block.

4. This event is a false sharing miss for the same reason as step 3.

5. This event is a true sharing miss, since the value being read was written by P2.

10. Giving priority to read misses over writes reduces miss penalty. How?

Giving priority to read misses over writes to reduce miss penalty—A write buffer is a good place to implement this optimization. Write buffers create hazards because they hold the updated value of a location needed on a read miss—that is, a read-after-write hazard through memory. One solution is to check the contents of the write buffer on a read miss. If there are no conflicts, and if the memory system is available, sending the read before the writes reduces the miss penalty. Most processors give reads priority over writes.

11. What is the impact of doubling associativity while doubling the cache size on the size of the index in Cache mapping?

For example, doubling associativity while doubling the cache size maintains the size of the index, since it is controlled by this formula:

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

A seemingly obvious alternative is to just use virtual addresses to access the cache, but this can cause extra overhead in the operating system.

12. List the Six basic optimizations of Cache?

- ➢ Larger block size to reduce miss rate Bigger caches to reduce miss rate Higher associativity to reduce miss rate
- ➢ Multilevel caches to reduce miss penalty
- ➢ Giving priority to read misses over writes to reduce miss penalty
- ➢ Avoiding address translation during indexing of the cache to reduce hit time

13. What is sequential inter-leaving? It is implemented in which level of the memory hierarchy?

Sequential inter-leaving is implemented at Cache level. It is one of the optimization technique used to improve cache performance.

Multibanked Caches are used to Increase Cache Bandwidth. Clearly, banking works best when the accesses naturally spread themselves across the banks, so the mapping of addresses to banks affects the behavior of the memory system. A simple mapping that works well is to spread the addresses of the block sequentially across the banks, called sequential interleaving. For example, if there are four banks, bank 0 has all blocks whose address modulo 4 is 0; bank 1 has all blocks whose address modulo 4 is 1; and so on. Figure 5.6 shows this interleaving.

PART B

1. Write Short notes on Compulsory, Capacity and conflict misses in Cahce.
2. Explain how write invalidate protocol maintain the coherence requirement?
3. Discuss various types of Coherence misses with examples
4. Explain the internal organization of a 64-bit DRAM Technology with suitable block diagram
5. Give the summary of the five standard RAID levels. (8 marks)
6. Discuss the Advanced Optimizations for improving Cache Performance. Give suitable Examples.
7. Write Short notes on Compiler Optimizations to reduce the miss rate.

UNIT V

PART A

1. **Define software multithreading**

   The ability of an operating system to execute different parts of a program, called threads, simultaneously. The programmer must carefully design the program in such a way that all the threads can run at the same time without interfering with each other

2. **What are the advantages of multithreading**

If a thread can not use all the computing resources of the CPU (because instructions depend on each other's result), running another thread permits to not leave these idle. If several threads work on the same set of data, they can actually share its caching, leading to better cache usage or synchronization on its values.

If a thread gets a lot of cache misses, the other thread(s) can continue, taking advantage of the unused computing resources, which thus can lead to faster overall execution, as these resources would have been idle if only a single thread was executed

3. **What are the two levels of thread?**

   - User Threads
   - Kernel Threads

4. **What are the multithreading models available?**

   - Many threads on one LWP (many-to-one)
   - One thread per LWP (one-to-one)
   - Many threads on many LWPs (many-to-many)

5. **What is Many-to-one model?**

   The many-to-one model maps many user-level threads to one kernel thread. Advantages: Totally portable More efficient Disadvantages: cannot take advantage of parallelism The entire process is block if a thread makes a blocking system call Mainly used in language systems, portable libraries like solaris 2

6. **What is One-to-one model?**

   The one-to-one model maps each user thread to a kernel thread. Advantages: allows parallelism Provide more concurrency Disadvantages: Each user thread requires corresponding kernel thread limiting the number of total threads Used in LinuxThreads and other systems like Windows 2000,Windows NT

7. **What is Many-to-many model?**

   The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads. Advantages: Can create as many user thread as necessary Allows parallelism Disadvantages: kernel thread can the burden the performance Used in the Solaris implementation of Pthreads (and several other Unix implementations)

8. **What are the factors will affect issue slot?**

   - Imbalances in the resource needs.

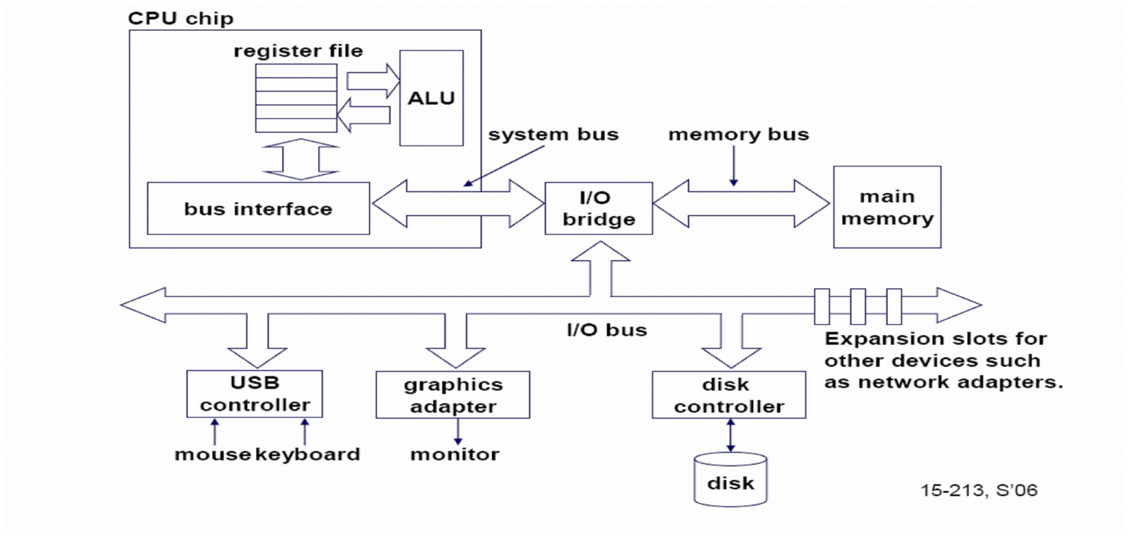
- Resource availability over multiple threads.
- Number of active threads considered.
- Finite limitations of buffer.
- Ability to fetch enough instructions from multiple threads.
- Practical limitations of what instructions combinations can issue from one thread and multiple threads.

9. Give Comparison of SMT vs Superscalar

SMT processors are compared to base superscalar processors in several key measures:

- Utilization of functional units.
- Utilization of fetch units.
- Accuracy of branch predictor.
- Hit rates of primary caches.
- Hit rates of secondary caches.

10. Draw the architecture of Single core computer



11. What are the design issues of SMT & CMP architectures?

They determine the performance measures of each processor in a precise manner. The issue slots usage limitations and its issues also determine the performance.Why Multithreading Today ILP is exhausted, TLP is in. Large performance gap between MEMORY and PROCESSOR. Too many transistors on chip. More existing MT applications today. Multiprocessors on a single chip. Long network latency, too

12. What is a Multi-core processor?

Each core has its execution pipeline. No limitation for the number of cores that can be placed in a single chip. Two cores run at slower speeds and lower temperatures. But the combined throughput > single processor. The fundamental relationship b/w freq. and power can be used to multiply the no. of cores from 2 to 4, 8 and even higher

13. What are the benefits of Intel Multi-core processor?

- Multi-core performance.
- Dynamic scalability.
- Design and performance scalability
- Intelligent performance on-demand
- Increased performance on Highly-threaded apps.
- Scalable shared memory.

14. What is a IBM cell processor?

A chip with one PPC hyper-threaded core called PPE and eight specialized cores called SPEs.The challenge to be solved by the Cell was to put all those cores together on a single chip. This was made possible by the use of a bus with outstanding performance

The Cell processor can be split into four components**:**

- external input and output structures,
- the main processor called the Power Processing Element (PPE)
- eight fully-functional co-processors called the Synergistic Processing Elements, or SPEs,
- a specialized high-bandwidth circular data bus connecting the PPE, input/output elements and the SPEs, called the Element Interconnect Bus or EIB.

PART B

1. Explain about Software and hardware multithreading
2. Give the explanation of SMT and CMP architectures with block diagram
3. What are the Design issues to be considered in SMT and CMP architecture?Explain
4. Explain about Intel Multi-core architecture
5. Give the explanation aboutIBM Cell Processor

GLOSSARY

1BP: 1-bit branch predictor

4 C's - compulsory Misses: the first time a block is accessed by the cache

4 C's - capacity misses: blocks must be evicted due to the size of the cache.

4 C's - coherence Miss: processors are accessing the same block. Processor A writes to the block. Even though Processor B has the block in its cache, it is a miss, because the block is no longer up-to-date.

4 C's - conflict Misses: associated with set associative and direct mapped caches - another data address needs the cache block and must replace the data currently in the cache.

ALAT: advance load table - stores advance information about load operations

Aliasing: in the BTB, when two addresses overlap with the same BTB entry, this is called aliasing. Aliasing should be kept to <1%.

ALU: arithmetic logic unit

AMAT: average memory access time

AMAT: Average Memory Access Time = hit time + miss rate * miss penalty

Amdahl's Law: an equation to determine the improvement of a system when only a portion of the system is improved.

Architectural registers: registers (Floating point and General Purpose) that are visible to the programmer.

ARF: architectural register file or retirement register file

Asynchronous Message Passing: a processor requests data, then continues processing instructions while message is retrieved.

BHT: branch history table - records if branch was taken or not taken.

Blocking cache: the cache services only one block at a time, blocking all other requests

BTB: branch target buffer - keeps track of what address was taken last time the processor encountered this instruction.

Cache: a collection of data duplicating original values stored elsewhere on a *computer*

Cache coherence definition #1: Definition #1 - A read R from address X on processor P1 returns the value written by the most recent write W to X on P1 if no other processor has written to X between W and R. http://www.numascale.com/cache-coherence.html

Cache coherence definition #2: Definition #2 - If P1 writes to X and P2 reads X after a sufficient time, and there are no other writes to X in between, P2's read returns the value written by P1's write.

Cache coherence definition #3: Definition #3 - Writes to the same location are serialized:two writes to location X are seen in the same order by all processors.

Cache hit: desired data is in the cache and is up-to-date

Cache miss: desired data is not in the cache or is dirty

Cache review: a cache review can be found at

Cache thrashing: when two or more addresses are competing for the same cache block. The processor is requesting both addresses, which results in each access evicting the previous access.

CDB: common data bus

Check pointing: store the state of the CPU before a branch is taken. Then if the branch is a misprediction, restore the CPU to correct state. Don't store to memory until it is determined this is the correct branch.

CISC Processor: complex instruction set

CMP: chip multiprocessor

Coarse multi-threading: the thread being processed changes every few clock cycles

Coherence: the consistency of shared resource data that ends up stored in multiple local caches.

Computer architecture: a set of disciplines that describes a computer system by specifying its parts and their relations

Consistency: order of access to different addresses

Control hazard: branching and jumps cannot be executed until the destination address is known

CPI: cycle per instruction

CPU: central processing unit

Dark Silicon: the gap between how many transistors are on a chip and how many you can use simultaneously. The simultaneous usage is determined by the power consumption of the chip.

Data hazard: the order of the program is changed which results in data commands being out of order, if the instructions are dependent - then there is a data hazard.

DDR SDRAM: double data rate synchronous dynamic RAM

Dependency chain: long series of dependent instructions in code

Directory protocols: information about each block state in the caches is stored in a common directory.

Distributed caches:distributed caching is a form of caching that allows the cache to span multiple servers so that it can grow in size and in transactional capacity

DRAM: dynamic random access memory

DSM: distributed shared memory - all processors can access all memory locations

Enterprise class: used for large scale systems that service enterprises

Error: defect that results in failure

Error forecasting: estimate presence, creation, and consequences of errors

Error removal: removing latent errors by verification

Exclusion property: each cache level will not contain any data held by a lower level cache

Explicit ILP: compiler decides which instruction to execute in parallel

Failure: the cause of an error

Fault avoidance: prevent an occurrence of faults by construction

Fault tolerance: prevent faults from becoming failures through redundancy

Faults: actual behavior deviates from specified behavior

FIFO: first in first out

Fine multi-threading: the thread being processed changes every cycle

FLOPS: floating point operations per second

Flynn's Taxonomy: classifications of parallel computer architecture, SISD, SIMD, MISD, MIMD

FPR: floating point register

FSB: front side bus

Geometric Mean: the nth root of the product of the numbers

Global miss rate: (the # of L2 misses)/(# of all memory misses)

GPR: general purpose register

Hit latency: time it takes to get data from cache. Includes the time to find the address in the cache and load it on the data lines

ILP instruction level programming **Inclusion property:** each level of cache will include all data from the lower level caches

IPC: instructions per cycle

Iron Law: execution time is the number of executed instructions N (write N in in the ExeTime for Single-Cycle), times the CPI (write x1), times the clock cycle time (write 2ns) so we get N*2ns (write =N*2ns) for single-cycle.

Iron Law: An instruction per program depends on source code, compiler technology, and ISA. CPI depends upon the ISA and the micro architecture. Time per cycle depends upon the micro architecture and the base technology.

Iron law of computer performance: relates cycles per instruction, frequency and number of instructions to computer performance

ISA: instruction set architecture

Itanium architecture: an explicit ILP architecture, six instructions can be executed per clock cycle

Itanium Processor: Intel family of 64-bit processors that uses the Itanium architecture

LFU: least frequently used

ll and sc: load link and store conditional, a method using two instructions ll and sc for ensuring synchronization.

local miss rate: # of L2 misses/ # of L1 misses

Locality principle: things that will happen soon are likely to be similar to things that just happened.

Loop interchange: used for nested loops. Interchange the order of the iterations of the loop, to make the accesses of the indexes closer to what is actually the layout in memory

LRU: least recently used

LSQ: load store queue http://www.cs.utah.edu/~manua/sim_doc/simics-micro-architectural-interface/topic15.html

MCB: memory conflict buffer - "Dynamic Memory Disambiguation Using the Memory Conflict Buffer", see also "Memory Disambiguation"

Memory dependence prediction: It is a technique employed by high performance out of order execution microprocessors

MEOSI Protocol: modified-exclusive-owner-shared-invalid protocol, the states of any cached block.

MESI Protocol: modified-exclusive-shared-invalid protocol, the states of any cached block.

Message Passing: a processor can only access its local memory. To access other memory locations is must send request/receive messages for data at other memory locations.

Meta-predictor: a predictor that chooses the best branch predictor for each branch.

MIMD: multiple instruction stream, multiple data streams

MISD: multiple instruction streams, single data stream

Miss latency: time it takes to get data from main memory. This includes the time it takes to check that it is not in the cache and then to determine who owns the data, and then send it to the CPU.

mobo: mother board

Moore's Law: Gordon E. Moore observed the number of transistors on an integrated circuit board doubles every two years.

MP: multiprocessing

MPKI: Misses per Kilo Instruction

MSI Protocol: modified-shared-invalid protocol, the states of any cached block.

MTPI: message transfer part interface

MTTF: mean time to failure

MTTR: mean time to repair

Multi-level caches: caches with two or more levels, each level larger and slower than the previous level.

mutex variable: mutually exclusive (mutex), a low level synchronization mechanism. A thread acquires the variable, then releases it upon completion of the task. During this period no other thread can acquire the mutex.

NMRU: not most recently used

Non-blocking caches: if there is a miss, the cache services the next request while waiting for memory

NUMA: non-uniform memory access, also called a distributed shared memory

OOO: out of order

OS: operating system

PAPT: physically addressed, physically tagged cache - the cache stores the data based on its physcial address

PC: program counter

PCI: peripheral component interconnect

Pentium Processor: x86 super scalar processor from Intel

Physical registers: registers, FP and GP that are not visible to the programmer

Pipeline burst cache:

Pipelined cache: a pipelined burst cache uses 3 clock cycles to transfer the first data set from a cache block, then 1 clock cycle to transfer each of the rest. The pipeline and the 'burst'. (3-1-1-1)

PIPT: physically indexed, physically tagged cache.

Power: Power = 1/2C *V^2 * f* Alpha

Power Architecture: performance optimization with enhanced RISC

Power vs Performance Equation:

Pre-fetch buffer: when getting data from memory, get all the data in the row and store it in a buffer.

Pre-fetching cache: instructions are fetched from memory before they are needed by the cpu "

Prescott Processor: Based on the Netburst architecture. It has a 31 stage pipeline in the core. The high penatly paid for mispredictions is supposedly offset with a Rapid Execution Engine. It also has a trace execution cache; this store decoded instructions and then reuses them instead of fetching and decoding again.

PRF: physical register file

Pseudo associative cache: an address is first searched in 1/2 of the cache. If it is not there, then it is searched in the other half of the cache

RAID: redundant array of independent disks

RAID 0: strips of data are stored on disks - alternating between disks. Each disk supplies a portion of the data, which usually improves performance. http://en.wikipedia.org/wiki/Raid-0#RAID_0

RAID 1: the data is replicated on another disk. Each disk contains the data. Which ever disk is free responds to the read request. The write request is written to one disk and then mirrored to the other disk(s).

RAID 2 and RAID 3: the data is striped on disks and Hamming codes or parity bits are used for error detection. RAID 2 and RAID 3 are not used in any current application

RAID 4: Data is striped in large blocks onto disks with a dedicated parity disk. It is used by the NetApp Company.

RAID 5: Data is striped in large blocks onto disks, but there is no dedicated parity disk. The parity for each block is stored on one of the data blocks.

RAR: read after read

RAS: return address stack

RAT: register alias table

RAT: *(another RAT in multiprocessing) register allocation table

RAW: read after write

RDRAM: direct random access memory

Relaxed consistency: some instructions can be performed ooo and still maintain consistency

Reliability: measure of continuous service accomplishment

Reservation stations: function unit buffers

RETO: return from interrupt

RF: register file

RISC Processor: reduced instruction set - simple instructions of the same size. Instructions are executed in one clock cycle

ROB: re-order buffer

RS: reservation station

RWX: read - write- execute permissions on files

SHARC processor: floating point processors designed for DSP applications

SIMD: singe instruction stream, multiple data streams

Simultaneous multi-threading: instructions from different threads are processed, even in the same cycle

SISD: single instruction stream, single data stream

SMP: symmetric multiprocessing

SMT: simultaneous multi threading

Snooping protocols: A broadcast network - caches for each processor watch the bus for addresses in their cache.

SPARC processor: Scalable Processor Architecture -a RISC instruction set processor

Spatial locality: if we access a memory location, nearby memory locations have a tendency to be accessed soon.

Speedup: how much faster a modified system is compared to the unmodified system.

SPR: special purpose registers - such as program counter, or status register

SRAM: static random access memory

Structural hazard: the pipeline contains two instructions attempting to access the same resource.

Super scalar architecture: the processor manages instruction dependencies at run-time. Executes more than one instruction per clock cycle using pipelines.

Synchronization: "a system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations for each individual processor appear in the order specified by the program." Quote by Leslie Lamport

Synchronous Message Passing: a processor requests data then waits until the data is received before continuing.

Tag: the part of the data address that is used to find the data in the cache. This portion of the address is unique so that it can be distinguished from other lines in the cache.

Temporal locality: if a program accesses a memory location, it tends to access the same location again very soon.

TLB: translation look aside buffer - a cache of translated virtual memory to physical memory addresses. TLB misses are very time consuming

Tomasulo's Algorithm: achieve high performance without using special compilers by using dynamic scheduling

Tournament predictor: a meta-predictor

Trace caches: sets of instructions are stored in a separate cache. These are instructions that have been decoded and executed. If there is a branch in the set, only the taken branch instructions are kept. If there is a misprediction the trace stops.

Trace scheduling: rearranging instructions for faster execution, the common cases are scheduled first

Tree, tournament, dissemination barriers: types of structures for barriers

UMA: uniform memory access - all memory locations have similar latencies

Vector registers: hold data for vector processing for SIMD

Victim cache: a cache that holds recently evicted blocks. The link is to a victim simulator (don't use the first trace program, need Firefox, java7).

VIPT: virtually indexed physically tagged

VIPT: virtually indexed physically tagged - the cache receives the virtual address as does the TLB (in parallel). If a hit, okay, if a miss, the physical address is already translated.

Virtual memory: Methods of allowing a process to access RAM memory independently of other processes. RAM memory is used by all processes, which means there is probably not enough RAM for all processes at the same time. The virtual address is used to point to an address, either RAM or Disk. The program sees it as the same; the TLB translates between request and the actual memory.

VIVT: virtually indexed virtually tagged

VIVT: virtually indexed, virtually tagged cache - the cache is virtually addressed by the cpu. On a cache miss the virtual address must be translated to the physical address

VLIW: very long instruction word

VPN/PPN: Virtual Physical Network, Physical page network

WAR: write after read

WAW: write after write

Way prediction: set associative caches are fast but use a lot of energy. To save energy, predict the address that will be selected. This reduces the energy wasted searching the entire cache.

Write invalidate: a write to shared data forces invalidation of all other cached copies

Write update: a write to shared data is broadcast to update copies

XEON: x86 processors that can operate together to form dual and quad core systems

Question Paper Code : 51355

B.E./B.Tech. DEGREE EXAMINATION, MAY/JUNE 2014.

Sixth Semester

Computer Science and Engineering

CS 2354/CS 64/10144 CS 604 — ADVANCED COMPUTER ARCHITECTURE

(Regulation 2008/2010)

Time : Three hours

Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. What are the possible types of data hazards?
2. What do you mean by branch prediction buffer?
3. What is VLIW?
4. List out the major parts of Itanium processor?
5. When is a memory said to be coherent in a multi-processor system?
6. Why do we need Synchronization?
7. Define the terms: access time, bandwidth
8. State the principle of locality. List the types of locality.
9. Write the benefits of multi-core architecture.
10. Why are design issues of SMT and CMP architectures important?

PART B — (5 × 16 = 80 marks)

11. (a) (i) Explain the three types of dependencies in instructions. (10)

(ii) Explain the function of Tomasulo's approach (6)

Or

(b) (i) List out the decisions and transformation to be considered to obtain unrolled code in loop unrolling and scheduling. (8)

12. (a) (i) Explain the problems with VLIW approach. (8)

(ii) Describe about compiler speculation with hardware support. (8)

Or

(b) (i) Compare hardware with software speculation mechanisms. (8)

(ii) Describe the various execution units in IA64 processors. (8)

13. (a) Explain the function of symmetric shared-memory architecture. Compare it with Distributed shared-memory architecture. (16)

Or

(b) (i) What is multithreading? Explain the approaches to multithreading. (8)

(ii) Explain the models of memory consistency. (8)

14. (a) (i) Discuss the techniques for reducing cache miss penalty. (8)

(ii) Briefly explain the types of storage devices. (8)

Or

(b) (i) Explain the various levels of RAID. (8)

(ii) List and explain the various I/O performance measures. (8)

15. (a) (i) Describe the two levels of threads. (8)

(ii) Discuss the factors that limits the issues in simultaneous multithreading. (8)

Or

(b) With a neat sketch, explain the architecture of IBM cell processor in detail. Highlight the features of it. (16)