UNIT I FUNDAMENTALS OF COMPUTER DESIGN AND ILP

Fundamentals of Computer Design – Measuring and Reporting Performance – Instruction Level Parallelism and its Exploitation – Concepts and Challenges –Exposing ILP - Advanced Branch Prediction - Dynamic Scheduling - Hardware-Based Speculation - Exploiting ILP - Instruction Delivery and Speculation - Limitations of ILP – Multithreading

1.1 Fundamentals of Computer Design

1.1.1 Introduction

Computer technology has made incredible progress in the roughly from last 55 years. This rapid improvement has come both from advances in 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.

- Virtual elimination of assembly language programming reduced the need for Object code compatibility
- Creation of standardized vendor-independent Operating Systems such as UNIX, Linux. These Changes made it possible to develop a new set of architectures with simpler instructions called RISC (Reduced Instruction Set Computer). RISC focuses on two techniques
- Exploitation of Instruction Level Parallelism
- Use of Caches

The RISC based computer raised the performance bar, forcing prior architectures to keep up or disappear.

1.1.2 Classes of Computers

1960 – Large Mainframes – Computers costing Billions of dollars and stored in Computer rooms with multiple operators

Applications: Business data processing, Large-scale Scientific Computing

1970 – Minicomputer – Smaller sized computer focused on applications in scientific Laboratories, sharing between multiple users. That decade also saw the emergence of Supercomputer, which were high performance compute for scientific computing.

1980 – Desktop computer – Based on microprocessor in the form of Personal computers and workstations. These computers replaced time sharing computers with servers. These servers are computers that provide larger scale services, reliable long term file storage and access and more computing power.

1990 – Internet and World Wide Web – Emergence of handheld computing devices, high performance of Digital consumer electronics

2000 – Embedded Computers – Computers are lodged in other devices and their presence is not immediately obvious

Desktop Computing: The first and still the largest market in dollar terms is desktop computing. Throughout this range in price, the desktop market tends to drive to optimize price performance. The combination of price and performance are the driving factors to customers and the computer designer. Hence the newest, high performance and cost effective processor often appears first in desktop computers

Servers: Servers provide large scale and reliable computing and file services and are mainly used in the large scale enterprise computing and web based services.

Characteristics:

- Dependability – Servers must operate 24 * 7 hours a week. Failure of server system is far more catastrophic than a failure of desktop. Enterprise will lose revenue if the server is unavailable.

- Scalability – as the business grows, the servers may have to provide more functionality / services. Thus the ability to scale up the computing capacity, memory, and storage and I/O bandwidth is crucial.
- Throughput – Overall performance of the server. Transaction completed per minute or web pages served per second are crucial for servers.

Embedded Computers: These are the fastest growing portion of the computer market. Simple embedded microprocessors are seen in washing machines, printers, network switches, handheld devices such as cell phones, smart card video games etc,. The primary goal is to meet the performance need at a minimum price rather than achieving high performance at high level. The other two characteristic requirement are to minimize the memory and power.

1.1.3 Defining Computer Architecture

Computer Architecture is often referred to as Instruction set Design. CA is the art of assembling or integrating all the required logical elements into a computing device.

Instruction Set Architecture

ISA refers to actual programmer visible instruction set. The ISA serves as boundary between the software and hardware. The seven dimensions of ISA are:

1. Class of ISA - Nearly all ISAs today are classified as general-purpose register architectures, where the operands are either registers or memory locations. The two popular versions of this class are *register-memory* ISAs, such as the 80x86, which can access memory as part of many instructions, and *load-store* ISAs such as ARM and MIPS, which can access memory only with load or store instructions.
2. Memory addressing - Virtually all desktop and server computers use byte addressing to access memory operands. An access to an object of size s bytes at byte address A is aligned if A mod $s = 0$.
3. Addressing modes - In addition to specifying registers and constant operands, addressing modes specify the address of a memory object. MIPS addressing modes are Register, Immediate (for constants), and Displacement, where a constant offset is added to a register to form the memory address. The 80x86 supports those three plus three variations of

displacement: no register (absolute), two registers (based indexed with displacement), and two registers where one register is multiplied by the size of the operand in bytes.

4. Types and sizes of operands - Like most ISAs, 80x86, ARM, and MIPS support operand sizes of 8-bit (ASCII character),

16-bit (Unicode character or half word),

32-bit (integer or word),

64-bit (double word or long integer), and

IEEE 754 floating point in 32-bit (single precision) and
64-bit(double precision).

The 80x86 also supports 80-bit floating point (extended double precision).

5. Operations – The general category of operations are:

data transfer,

arithmetic

logical,

control and

floating point.

6. Control and Flow instructions – All ISAs support conditional branches, unconditional jumps, procedure calls, and returns.

7. Encoding an ISA - There are two basic choices on encoding: *fixed length* and v*ariable length*. All ARM and MIPS instructions are 32 bits long, which simplifies instruction decoding. Number of registers and number of addressing modes have significant impact on the length of instruction as register field and addressing mode field can appear many times in single instruction.

1.1.4 Trends in 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

Performance Trends: Bandwidth or Latency is the total amount of work done in a given time. *latency* or *response time* is the time between the start and the completion of an event, such as milliseconds for a disk access.

1.1.5 Trends in Power in Integrated Circuits

Today, power is the biggest challenge facing the computer designer for nearly every class of computer. First, power must be brought in and distributed around the chip, and modern microprocessors use hundreds of pins and multiple interconnect layers just for power and ground. Second, power is dissipated as heat and must be removed.

For CMOS chips, the traditional dominant energy consumptions has been in switching transistors, also called dynamic power. The power required per transistor is proportional to the product of the

load capacitance of the transistor, the square of the voltage, and the frequency of switching, with watts being the unit:

$$\text{Power}_{\text{dynamic}} = \tfrac{1}{2} \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}_{\text{switched}}$$

$$\text{Energy}_{\text{dynamic}} = \text{Capacitive load} \times \text{Voltage}^2$$

1.1.6 Trends in Cost

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.

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

Cost of an Integrated Circuit

Although the costs of integrated circuits have dropped exponentially, the basic process of silicon manufacture is unchanged: A *wafer* is still tested and chopped into *dies* that are packaged.

$$cost\ of\ integrated\ circuit = \frac{Cost\ of\ die + Cost\ of\ testing\ die + Cost\ of\ packaging\ and\ final\ test}{Final\ test\ yield}$$

$$cost\ of\ die = \frac{Cost\ of\ wafer}{Dies\ per\ wafer \times Die\ yield}$$

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

$$Dies\ per\ wafer = \frac{\pi \times (wafer\ diameter/2)^2}{Die\ area} - \frac{\pi \times wafer\ diameter}{\sqrt{2 \times Die\ area}}$$

The first term is the ratio of wafer area ($\pi r2$) 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.

1.1.7 Dependability

Infrastructure providers started offering *service level agreements* (SLAs) or *service level objectives* (SLOs) to guarantee that their networking or power service would be dependable.

Systems alternate between two states of service with respect to an SLA:

1. *Service accomplishment*, where the service is delivered as specified
2. *Service interruption*, where the delivered service is different from the SLA

Failure = transition from state 1 to state 2

Restoration = transition from state 2 to state 1

Two main measures of dependability:

1. ***Module reliability*** is a measure of the continuous service accomplishment (or, equivalently, of the time to failure) from a reference initial instant.

Mean time to failure (MTTF) is a reliability measure.

FIT for *failures in time* = 1 / MTTF

Service interruption is measured as *mean time to repair* (MTTR).

Mean time between failures (MTBF) is simply the sum of MTTF + MTTR.

2. ***Module availability*** is a measure of the service accomplishment with respect to the alternation between the two states of accomplishment and interruption.

$$Module\ availability = \frac{MTTF}{MTTF + MTTR}$$

1.2 Measuring and Reporting Performance

The computer user is interested in reducing response time. **Execution time or response time** is the time between the start and the completion of an event. The manager of a large data processing center may be interested in increasing throughput. **Throughput** is the total amount of work done in a given time.

Performance is in units of things per sec. When we relate the performance of two different computers, say, X and Y. The phrase ―X is faster than Y‖ is used here to mean that the response time or execution time is lower on X than on Y for the given task. In particular, ―X is *n* times faster than Y‖ will mean:

$$\frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

Execution time is the reciprocal of performance, the following relationship holds:

$$n = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = \frac{\frac{1}{\text{Performance}_Y}}{\frac{1}{\text{Performance}_X}} = \frac{\text{Performance}_X}{\text{Performance}_Y}$$

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.

Benchmarks

The real applications are the best choice of benchmarks to evaluate the performance. The three types of benchmarks are:

- *Kernels*, which are small, key pieces of real applications
- *Toy programs*, which are 100-line programs from beginning programming assignments, such as quicksort.

- *Synthetic benchmarks*, which are fake programs invented to try to match the profile and behavior of real applications, such as Dhrystone

To make the process of evaluation a fair justice, the following points are to be followed.

1. No source code modifications are allowed.
2. Source code modifications are allowed but are essentially impossible. For example, database benchmarks rely on standard database programs that are tens of millions of lines of code. The database companies are highly unlikely to make changes to enhance the performance for one particular computer.
3. Source modifications are allowed, as long as the modified version produces the same output

Benchmark suites are the collections of benchmark applications, used to increase the predictability. 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.

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**.**

SPEC CPU2006 is aimed at processor performance, SPEC also has benchmarks for graphics and Java.

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 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. Transaction Processing Council (TPC) to try to create realistic and fair benchmarks for TP.

TPC-C Complex Query for Online Transaction Processing

TPC-H models adhoc decision support

TPC-W a transactional web benchmark

TPC App application server and web services benchmark

Embedded Benchmarks

Benchmarks for embedded computing systems are in a far more nascent state than those for either desktop or server environments. 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

Reporting Performance Results

The guiding principle of reporting performance measurements should be *reproducibility*—list everything another experimenter would need to duplicate the results. A SPEC benchmark report requires an extensive description of the computer and the compiler flags, as well as the publication of both the baseline and optimized results.

In addition to hardware, software, and baseline tuning parameter descriptions, a SPEC report contains the actual performance times, both in tabular form and as a graph. A TPC benchmark report is even more complete, since it must include results of a benchmarking audit and cost information.

Summarizing Performance Results

Likewise, consumers trying to choose a computer will rely on performance measurements from benchmarks, which hopefully are similar to the user's applications. In both cases, it is useful to have measurements for a suite of bench marks so that the performance of important applications is similar to that of one or more benchmarks in the suite and that variability in performance can be understood.

Once we have chosen to measure performance with a benchmark suite, we would like to be able to summarize the performance results of the suite in a **single number**. A straightforward approach to computing a summary result would be to compare the **arithmetic means of the execution times** of the programs in the suite. An alternative would be to add a **weighting factor** to each benchmark and use the weighted arithmetic mean as the single number to summarize performance.

The problem would then be how to pick weights; since each company might have their own favorite set of weights, which would make it hard to reach consensus. One approach is to use weights that make all programs execute an equal time on some reference computer, but this biases the results to the performance characteristics of the reference computer.

Rather than pick weights, we could normalize execution times to a reference computer by dividing the time on the reference computer by the time on the computer being rated, yielding a ratio

proportional to performance. SPEC uses this approach, calling the ratio the SPEC Ratio. For example, suppose that the SPECRatio of computer A on a benchmark was 1.25 times higher than computer B; then we would know:

$$1.25 = \frac{\text{SPECRatio}_A}{\text{SPECRatio}_B} = \frac{\dfrac{\text{Execution time}_{\text{reference}}}{\text{Execution time}_A}}{\dfrac{\text{Execution time}_{\text{reference}}}{\text{Execution time}_B}} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{\text{Performance}_A}{\text{Performance}_B}$$

Because a SPECRatio is a ratio rather than an absolute execution time, the mean must be computed using the *geometric* mean.

$$\text{Geometric mean} = \sqrt[n]{\prod_{i=1}^{n} sample_i}$$

The geometric mean ensures two important properties:

1. The geometric mean of the ratios is the same as the ratio of the geometric means.
2. The ratio of the geometric means is equal to the geometric mean of the performance ratios, which implies that the choice of the reference computer is irrelevant.

$$\frac{\text{Geometric mean}_A}{\text{Geometric mean}_B} = \frac{\sqrt[n]{\prod_{i=1}^{n} \text{SPECRatio A}_i}}{\sqrt[n]{\prod_{i=1}^{n} \text{SPECRatio B}_i}} = \sqrt[n]{\prod_{i=1}^{n} \frac{\text{SPECRatio A}_i}{\text{SPECRatio B}_i}}$$

$$= \sqrt[n]{\prod_{i=1}^{n} \frac{\dfrac{\text{Execution time}_{\text{reference}_i}}{\text{Execution time}_{A_i}}}{\dfrac{\text{Execution time}_{\text{reference}_i}}{\text{Execution time}_{B_i}}}} = \sqrt[n]{\prod_{i=1}^{n} \frac{\text{Execution time}_{B_i}}{\text{Execution time}_{A_i}}} = \sqrt[n]{\prod_{i=1}^{n} \frac{\text{Performance}_{A_i}}{\text{Performance}_{B_i}}}$$

That is, the ratio of the geometric means of the SPECRatios of A and B is the geometric mean of the performance ratios of A to B of all the benchmarks in the suite.

1.3 Quantitative Principles of Computer Design

While designing the computer, the advantage of the following points can be exploited to enhance the performance.

Parallelism – It is one of the most important methods for improving performance. One of the simplest ways to do this is through pipelining ie, to overlap the instruction execution to reduce the total time to complete an instruction sequence. Parallelism can also be exploited at the level of detailed digital design. Set associative caches use multiple banks of memory that are typically searched in parallel. Carry look ahead which uses parallelism to speed the process of computing.
Principle of locality – Program tends to reuse data and instructions they have used recently. The rule of thumb is that program spends 90% of its execution time in only 10% of code. With reasonable good accuracy prediction can be made to find what instruction and data the program will use in near future based on its accesses in the recent past.

Focus on the common case – While making a design trade off, favor the frequent case over the infrequent case. This principle applies when determining how to spend resources, since the impact of improvement is higher if the occurrence is frequent.

Amdahl's Law

Amdahl's Law states that how much a computation can be speed up by running part of it in parallel. 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.

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

Alternatively,

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

Speedup tells us how much faster a task will run using the computer with the enhancement as opposed to the original computer. speedup from some enhancement, depends on two factors:

1. *The fraction of the computation time in the original computer 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 $\text{Fraction}_{\text{enhanced}}$, 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, say, 2 seconds for a portion of the program, while it is 5 seconds in the original mode, the improvement is 5/2. This value is always greater than 1, $\text{Speedup}_{\text{enhanced}}$.

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

The overall speedup is the ratio of the execution times:

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

An important corollary of Amdahl's law is that if an enhancement is only usable for a fraction of a task then we can't speed up the task by more than the reciprocal of 1 minus that fraction. 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 Processor 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.*

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

or

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

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).

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

$$\text{CPU time} = \text{Instruction count} \times \text{Cycles per instruction} \times \text{Clock cycle time}$$

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

CPU time is *equally* dependent on these three characteristics; Processor performance depends on IC, CPI and clock rate or clock cycle. The 3 parameters are dependent on the following basic technologies:

- *Clock cycle time*—Hardware technology and organization
- *CPI*—Organization and instruction set architecture
- *Instruction count*—Instruction set architecture and compiler technology

1.4 Instruction-Level Parallelism: Concepts and Challenges

The potential of overlapping the execution of multiple instructions is called instructionlevel parallelism. 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 value of the 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).

1.4.1 Instruction-Level Parallelism

The simplest and most common way to increase the ILP is to exploit parallelism among iterations of a loop. This type of parallelism is often called *loop-level parallelism*. Here is a simple example of a loop that adds two 1000-element arrays and is completely parallel:

```
for (i=0; i<=999; i=i+1) x[i]
= x[i] + y[i];
```

Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little or no opportunity for overlap. An important alternative method for exploiting loop-

level parallelism is the use of SIMD in both vector processors and Graphics Processing Units (GPUs).

A SIMD instruction exploits data-level parallelism by operating on a small to moderate number of data items in parallel (typically two to eight). A vector instruction exploits data-level parallelism by operating on many data items in parallel using both parallel execution units and a deep pipeline.

1.4.2 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!=R2

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 anti dependence 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 if statement.

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.

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 fivestage 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 Basic Compiler Techniques for Exposing ILP

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.

The way in which compiler recognizes and takes the advantages of ILP depends on two factors:

1) The amount of ILP available in the program

2) The latencies of the functional units in the pipeline.

Pipeline Scheduling and Loop Unrolling

Figure below shows the FP unit latencies. Assume the standard five-stage integer pipeline, so that branches have a delay of one clock cycle. We assume that the functional units are fully pipelined or replicated (as many times as the pipeline depth), so that an operation of any type can be issued on every clock cycle and there are no structural hazards.

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

The following code segment, which adds a scalar to a vector:

```
for (i=999; i>=0; i=i–1)
x[i] = x[i] + s;
```

This loop is parallel by noticing that the body of each iteration is independent.

Showing how the parallelism can be used to improve its performance for a MIPS pipeline with the latencies shown above.

The first step is to translate the above segment to MIPS assembly language. In the following code segment, R1 is 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 address of the last element to operate on.

```
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!=R2
```

Example:

Without any scheduling, the loop will execute as follows, taking nine cycles:

			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

We can schedule the loop to obtain only two stalls and reduce the time to seven cycles:

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

The stalls after ADD.D are for use by the S.D.

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 stalls by creating additional independent instructions within the loop body. The gain from scheduling on the unrolled loop is even larger than on the original loop. This increase arises because unrolling the loop exposes more computation that can be scheduled to minimize the stalls; the code above has no stalls. Scheduling the loop realizes that loads and stores are independent and can be interchanged.

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

- 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 (e.g., name dependences).
- 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.
- Schedule the code, preserving any dependences needed to yield the same result as the original code.

Three different effects limit the gains from loop unrolling:

(1) a decrease in the amount of overhead amortized with each unroll,

(2) code size limitations, and

(3) compiler limitations

1.6 Reducing Branch Costs with Advanced Branch Prediction

Branch directs the processor to some other part of the program. In the processor, executing all outcomes of a branch in parallel is known as prediction. Branch prediction means predicting the outcome of a branch so that those instructions may be executed in parallel with current instructions.

The behavior of the branches can be predicted both statically at compile time and dynamically by the hardware at execution time. Static branch predictors are sometimes used in processors where expectation is that branch behavior is highly predictable at compile time. Static prediction can also be used to assist dynamic predictors.

Static Branch prediction

It is the simplest branch prediction technique because it does not rely on any information about the dynamic history of code executing. To reorder code around the branches, it needs to predict the branch statically when at compile. More accurate scheme predicts branches using profile information collected from earlier runs and modify prediction based on last run.

Dynamic Branch prediction and Branch-Prediction Buffers

The simplest dynamic branch prediction scheme is a branch-prediction 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. This scheme is the simples sort of buffer. It has no tags and is useful to reduce the branch delay.

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. This buffer is effectively a cache where every access is a hit, and the performance of buffer depends on both how often the prediction is for the branch of interest and how accurate the prediction is when it matches.

In simple 1-bit prediction scheme, 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 the weakness, 2 bit prediction schemes are often used. In a 2-bit prediction scheme, a prediction must miss twice before it is changed. Figure below shows the finite state processor for a 2-bit prediction scheme.

A branch prediction buffer can be implemented as a small, special ―cache‖ accessed with the instruction address during the IF pipe stage, or as a pair of bits attached to each block in the instruction cache and fetched with the instruction. If the instruction is decoded as a branch and if the branch is predicted as taken fetching begins from target as soon as the PC is known.

Otherwise sequential fetching and executing continue.

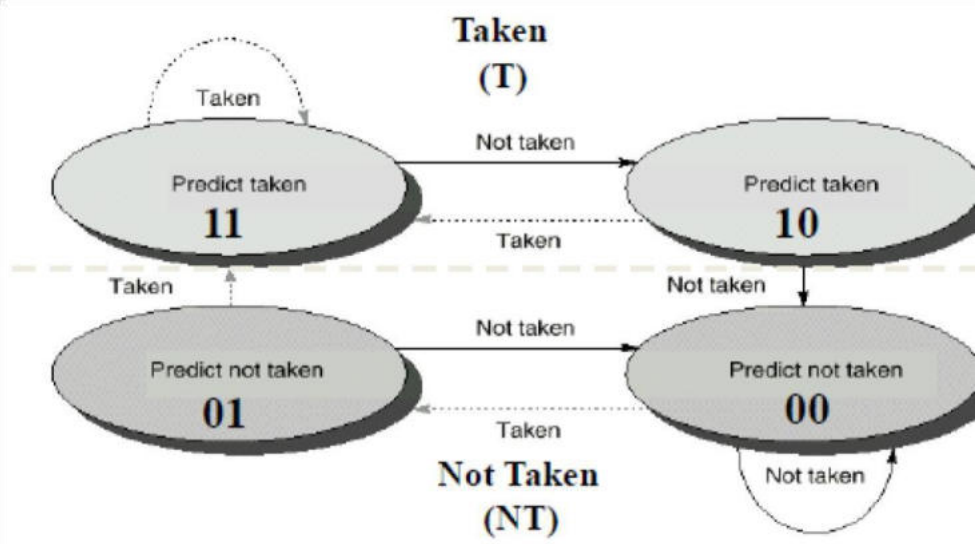


Figure 1.1 The states in a 2 bit prediction scheme

Correlating Branch Predictors

The 2-bit predictor schemes use only the recent behavior of a single branch to predict the future behavior of that branch.

Example: 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:

```
        DADDIU   R3,R1,#-2
        BNEZ     R3,L1          ;branch b1   (aa!=2)
        DADD     R1,R0,R0       ;aa=0
L1:     DADDIU   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, 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*. Existing correlating predictors add information about the behavior of the most recent branches to decide how to predict a given branch.

The number of bits in an (m,n) predictor is

$$2m \times n \times \text{Number of prediction entries selected by the branch address}$$

A 2-bit predictor with no global history is simply a (0,2) predictor.

Tournament 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 (8K–32K bits) 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.

1.7 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.

```
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 fivestage 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 score boarding.

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 floatingpoint unit and the load/store unit is shown below:

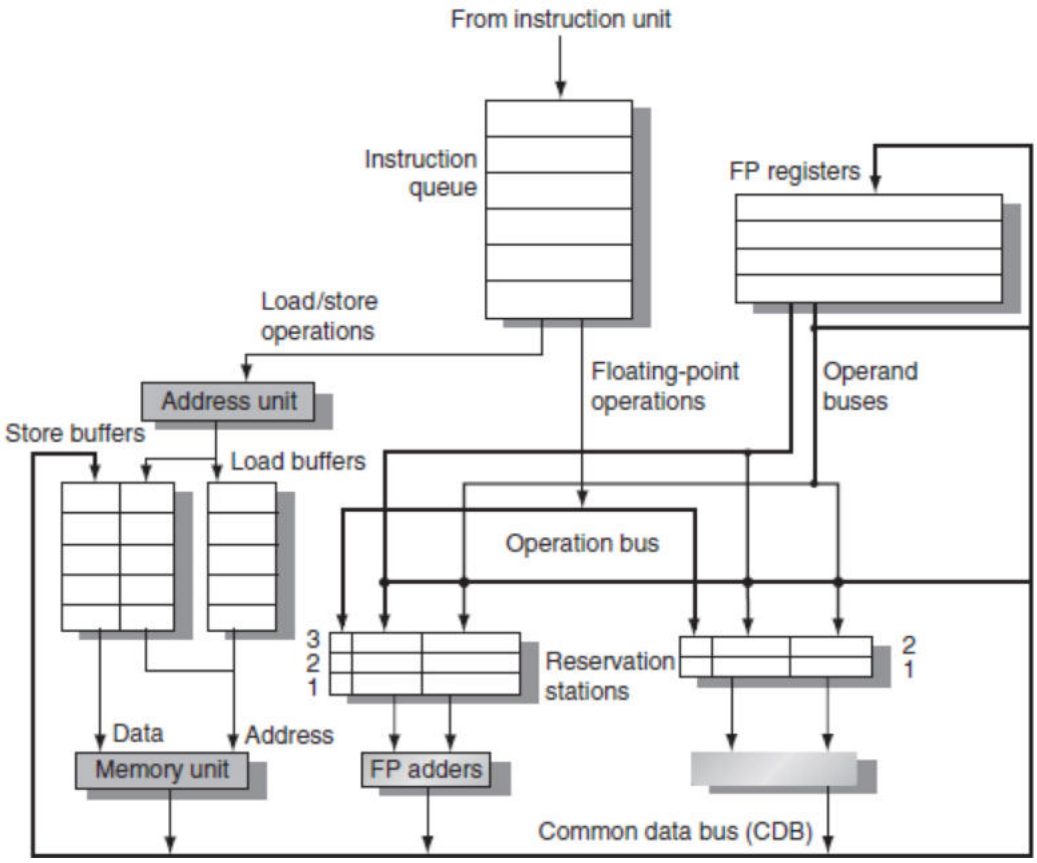


Figure 1.2 The basic structure of a MIPS floating point

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.

There are only three steps in Tomasulo's Approach:

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.

1.8 Hardware-Based Speculation

Branch prediction reduces direct stalls of branches. Instructions can be issued using dynamic branch prediction, but could not be executed until the branch outcome was known. Hardware Speculation extends the ideas of dynamic scheduling.

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 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.

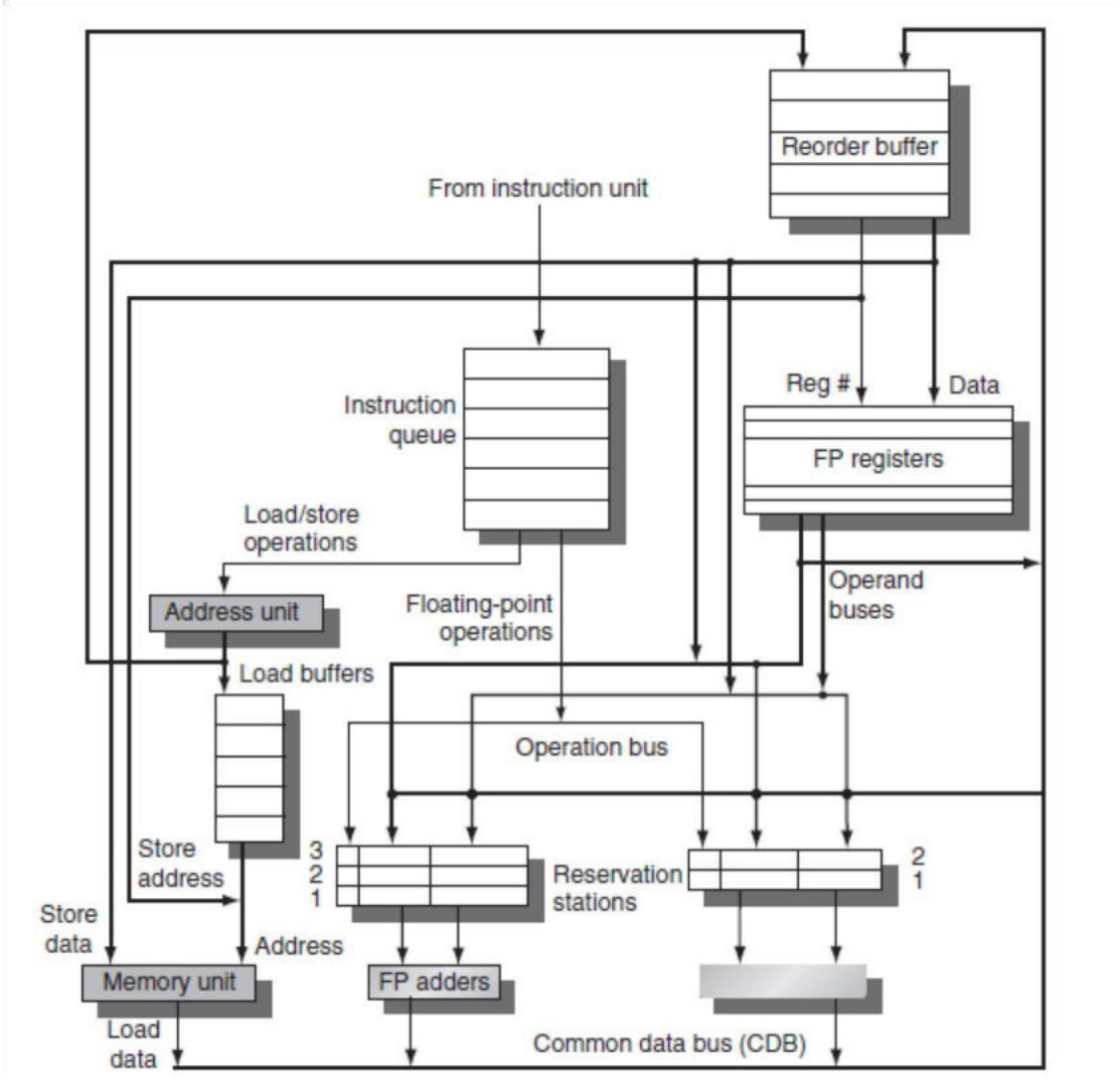


Figure 1.3 The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation

Once an instruction commits, its entry in the ROB is reclaimed and the register or memory destination is updated, eliminating the need for the ROB entry. If the ROB fills, we simply stop issuing instructions until an entry is made free.

Additional set of hardware buffers to hold the results of instructions which have not yet been committed: Reorder buffer (ROB). ROB holds the results of instructions between the time an instruction finishes and the time the instruction is being committed. Each ROB contains four fields –

Instruction type: branch/store/ALU operation

Destination: Register number or memory address where result should be written

Value: value of the instruction

Ready: instruction completed execution

1.9 Exploiting ILP

1.9.1 Exploiting ILP Using Multiple Issue and Static Scheduling

Exploitation of ILP is a effective mechanism for improving the performance of modern super-scalar / VLIW processors.

Single-Issue Processors are the Scalar processors that fetch and issue max one operation in each clock cycle.

Multiple-Issue Processors require fetching more instructions in a cycle (higher bandwidth from the instruction cache)

Multiple-Issue Processors can be:

- Dynamically scheduled (issue a varying number of instructions at each clock cycle).
- Statically scheduled (issue a fixed number of instructions at each clock cycle).

Multiple-issue processors come in three major flavors:

1. Statically scheduled superscalar processors
2. VLIW (very long instruction word) processors 3.
   Dynamically scheduled superscalar processors

The two types of superscalar processors issue varying numbers of instructions per clock and use in-order execution if they are statically scheduled or out-of order execution if they are dynamically scheduled.

VLIW processors issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction. VLIW processors are inherently statically scheduled by the compiler.

The five primary approaches in use for multiple-issue processors and the primary characteristics are shown below:

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Cortex-A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

The Basic VLIW Approach

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).

VLIWs use multiple, independent functional 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.

Let's consider a VLIW processor with 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 80 and 120 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 straightline 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. Trace scheduling selects a se3quence of basic block as a trace and schedules the operations from trace together.

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

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 IA64 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.

1.9.2 Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation

Here, we put all three together, which yields a micro architecture quite similar to those in modern microprocessors. For simplicity, we consider only an issue rate of two instructions per clock, but the concepts are no different from modern processors that issue three or more instructions per clock.

To gain the full advantage of dynamic scheduling we will allow the pipeline to issue any combination of two instructions in a clock, using the scheduling hardware to actually assign operations to the integer and floating-point unit. Because the interaction of the integer and floating-point instructions is crucial, we also extend Tomasulo's scheme to deal with both the integer and floating-point functional units and registers, as well as incorporating speculative execution.

Two different approaches have been used to issue multiple instructions per clock in a dynamically scheduled processor, and both rely on the observation that the key is assigning a reservation station and updating the pipeline control tables. One approach is to run this step in half a clock cycle, so that two instructions can be processed in one clock cycle; this approach cannot be easily extended to handle four instructions per clock, unfortunately.

A second alternative is to build the logic necessary to handle two or more instructions at once, including any possible dependences between the instructions. Modern superscalar processors that issue four or more instructions per clock may include both approaches: They both pipeline and widen the issue logic.

Putting together speculative dynamic scheduling with multiple issue requires overcoming one additional challenge at the back end of the pipeline: we must be able to complete and commit multiple instructions per clock. Like the challenge of issuing multiple instructions the concepts are simple, although the implementation may be challenging in the same manner as the issue and register renaming process.

1.10 Advanced Techniques for Instruction Delivery and Speculation

For a multiple issue processor, predicting branches well is not enough. Deliver a high-bandwidth instruction stream is necessary.

1.10.1 Increasing Instruction Fetch Bandwidth

A multiple-issue processor will require that the average number of instructions fetched every clock cycle be at least as large as the average throughput. Fetching these instructions requires wide enough paths to the instruction cache, but the most difficult aspect is handling branches.

There are two methods for dealing with branches:

1. **Branch-Target Buffers**
2. **Return address predictors**

1. Branch-Target Buffers – It is accessed during IF stage which stores branch, jump addresses, their target addresses and also sometimes the prediction information. 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.* Figure 1.4 shows a branch-target buffer

Because a branch-target buffer predicts 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. If the PC of the fetched instruction matches an address in the prediction buffer, then the corresponding predicted PC is used as the next PC. The hardware for this branch-target buffer is essentially identical to the hardware for a cache.

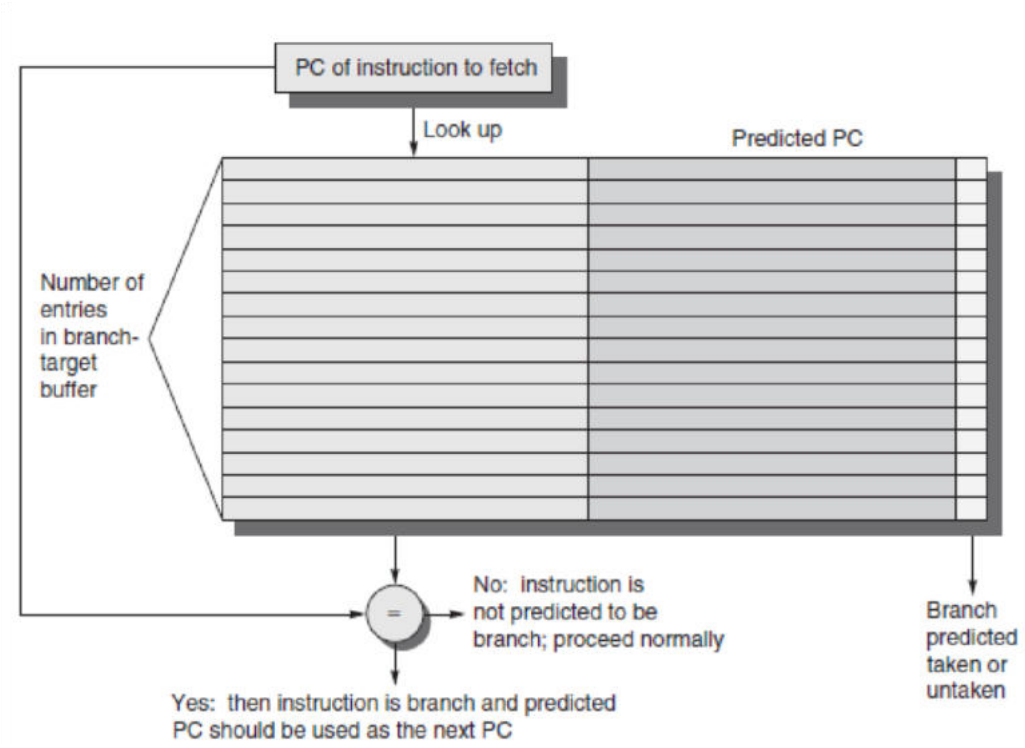


Figure 1.4 A branch-target buffer

If a matching entry is found in the branch-target buffer, fetching begins immediately at the predicted PC. The predictive entry must be matched to this instruction because the predicted PC will be sent out before it is known whether this instruction is even a branch. If the processor 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 worse performance

Figure 1.5 shows the steps when using a branch-target buffer for a simple five-stage pipeline. From this figure we can see that there will be no branch delay if a branch-prediction entry is found in the buffer and the prediction is correct. Otherwise, there will be a penalty of at least two clock cycles. Dealing with the mispredictions and misses is a significant challenge.

To evaluate how well a branch-target buffer works, we first must determine the penalties in all possible cases.

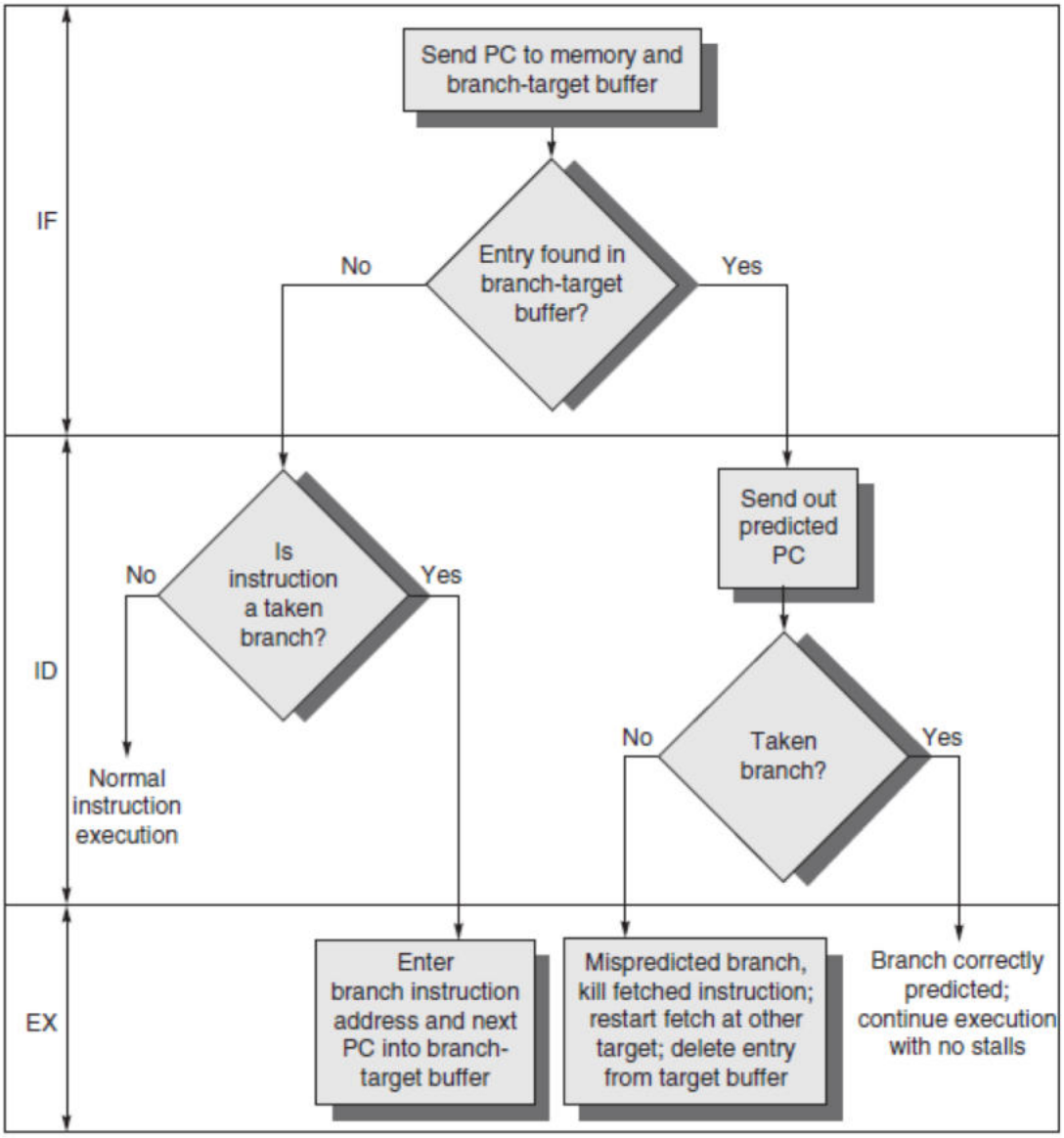


Figure 1.5 Steps involved in handling an instruction with a branch-target buffer

One variation on the branch-target buffer is to store one or more *target instructions* instead of, or in addition to, the predicted *target address*. This variation has two potential advantages.

First, it allows the branch-target buffer access to take longer than the time between successive instruction fetches, possibly allowing a larger branch-target buffer.

Second, buffering the actual target instructions allows us to perform an optimization called *branch folding.*

2. Return Address Predictors – BTB achieves very low accuracy of 60%. To overcome this problem, some designs use a small buffer of return addresses operating as a stack. 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.

Integrated Instruction Fetch Units

To meet the demands of multiple-issue processors, many recent designers have chosen to implement an integrated instruction fetch unit as a separate autonomous unit that feeds instructions to the rest of the pipeline.

Instead, 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 as 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.

3. *Instruction memory access and buffering*—When fetching multiple instructions per cycle a variety of complexities are encountered, including the difficulty that fetching multiple instructions may require accessing multiple cache lines. The instruction fetch unit encapsulates this complexity, using prefetch to try to hide the cost of crossing cache blocks. 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.

1.10.2 Advanced Speculation techniques

One alternative to the use of a reorder buffer (ROB) is the explicit use of a larger physical set of registers combined with register renaming. This approach builds on the concept of renaming used in Tomasulo's algorithm and extends it. With the addition of speculation, register values may also temporarily reside in the ROB.

In the register-renaming approach, an extended set of physical registers is used to hold both the architecturally visible registers as well as temporary values. Thus, the extended registers replace most of the function of the ROB and the reservation stations; only a queue to ensure that instructions complete in order is needed.

During instruction issue, a renaming process maps the names of architectural registers to physical register numbers in the extended register set, allocating a new unused register for the destination. WAW and WAR hazards are avoided by renaming of the destination register, and speculation recovery is handled because a physical register holding an instruction destination does not become the architectural register until the instruction commits. The renaming map is a simple data structure that supplies the physical register number of the register that currently corresponds to the specified architectural register, a function performed by the register status table in Tomasulo's algorithm.

When an instruction commits, the renaming table is permanently updated to indicate that a physical register corresponds to the actual architectural register, thus effectively finalizing the update to the processor state. Although an ROB is not necessary with register renaming, the hardware must still track instructions in a queue-like structure and update the renaming table in strict order

Advantage:

(1) record that the mapping between an architectural register number and physical register number is no longer speculative, and

(2) free up any physical registers being used to hold the ―older‖ value of the architectural register.

Speculation recovery easy because a physical register holding an instruction destination does not become the architectural register until the instruction commits.

Value Prediction

Value Prediction attempts to predict value produced by instruction – E.g., Loads a value that changes infrequently. Value prediction is useful only if it significantly increases ILP – Focus of research has been on loads; So results, no real processor uses value prediction.

Address aliasing prediction is a simple technique that predicts whether two stores or a load and a store refer to the same memory address. If two such references do not refer to the same address,

then they may be safely interchanged. Otherwise, we must wait until the memory addresses accessed by the instructions are known.

Address alias prediction is both more stable and simpler since need not actually predict the address values, only whether such values conflict – Has been used by a few processors.

1.10.3 Limitations of ILP

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. Limitations 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.

3. The 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.

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.

1.11 Multi threading

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. It is the process of creating a new thread of execution within an existing process rather than starting a new process to begin our process.

The main task of multithreading is to Optimize use of computer resources. It is defined as the ability of a program or as process to manage its use by more than one user at a time and to even manage multiple requests by the same user without having to have multiple copies of programming

.

Thread is a stream of instruction within a process. Each thread has its own set of registers, pointers ,stack, and memory. **Approaches to Multithreading**:

1) Fine- grained multithreading
2) Coarse- grained multithreading

Fine- grained Multithreading

- Switch between threads on each inst.
- Multiple threads executed in interleaved manner.
- Usually done in round robin Fashion , skipping any stalled threads.
- CPU must be capable of switching threads on every cycle.

Advantage:

It can hide throughput losses that arise from both short and long stalls ,since inst from other threads can be executed when one thread stalls.

Disadvantage:

Slows down execution of individual threads

- A thread that is ready to execute without stalls will be delayed by inst from other threads.

Coarse- grained Multithreading

- Switch only if current thread has a long latency or costly stall.
- Less like slow down individual thread.

Advantage:

Relieves need to have very fast thread switching .

don't slow down the thread **Disadvantage:**

Limited in ability to overcome throughput losses.

Simultaneous Multithreading (SMT):

SMT is a variation on multithreading that uses resources of a multiple-issue, dynamically scheduled processors to exploit TLP at the same time it exploits ILP i.e., convert thread-level parallelism into more ILP. It tries to eliminate horizontal waste (unused inst slots in a cycle) because it fetches and issues inst from different threads simultaneously. It is basically designed for super scalar processors.

It exploits the following features of modern processors: ▯ multiple functional units

Modern processors typically have more functional units available than aingle thread can utilize

▯ register renaming and dynamic scheduling

Multiple instructions from independent threads can co-exist and co-execute.

Illustration:

The following diagram illustrates the differences in a processor's ability to exploit the resources of a superscalar for the following processor configuration.

a) A superscalar processor with no multithreading support.
b) A superscalar processor with coarse-grain multithreading.
c) A superscalar processor with fine –grain multithreading.
d) A superscalar processor with simultaneous multithreading(SMT).

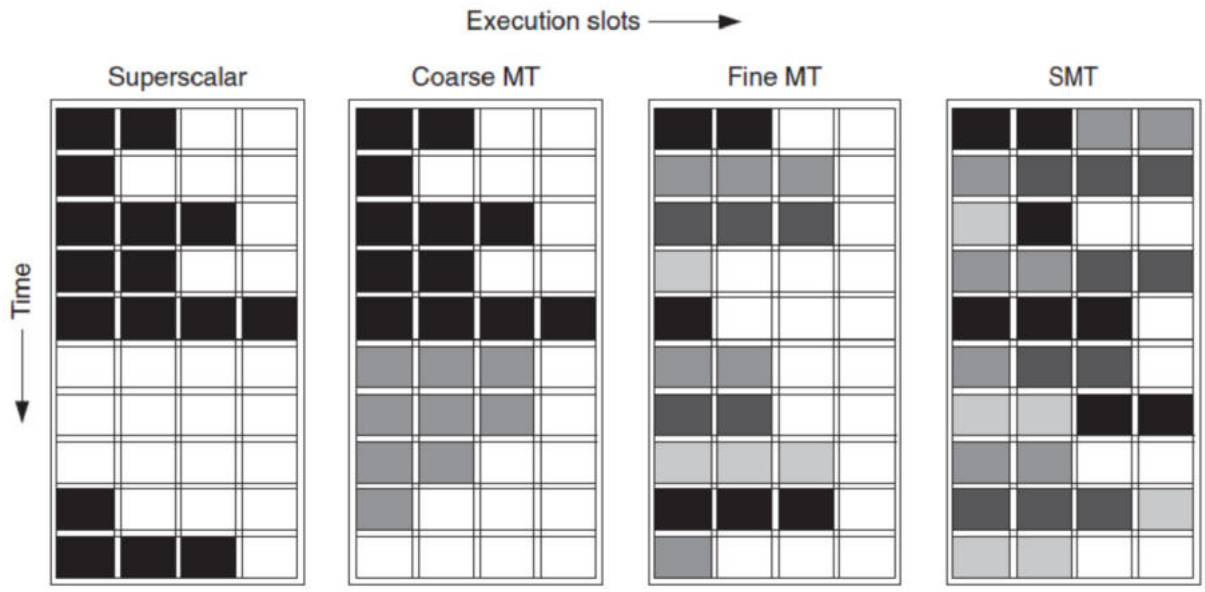


Figure 1.6 Different approaches of superscalar processor

Figure greatly simplifies the real operation of these processors, it does illustrate the potential performance advantages of multithreading in general and SMT in wider issue, dynamically scheduled processors**.** Here Horizontal dimension represents the instruction issue capability in each

clock cycle. Vertical dimension represents a sequence of clock cycles. Empty slots indicate that the corresponding issue slots are unused in that clock cycle.

a. Superscalar processor with no multithreading support

The use of issue slots is limited by a lack of ILP. Stalls such as an instruction cache miss leaver the entire processor idle.

b. Superscalar processor with coarse-grain multithreading:

The long stalls are partially hidden by switching to another thread that uses the resources of the long processor.

- Reduced the number of completely idle cycles.
- But the ILP limitations still lead to idle cycles.

c. Superscalar processor with fine –grain multithreading:

The interleaving of threads eliminates fully empty slots . The ILP limitations still lead to a significant number of time slots within individual clock cycles because only one thread issues instructions in a given clock cycles.

d. Superscalar processor with simultaneous multithreading(SMT):

The thread level parallelism(TLP) and instruction –level parallelism (ILP)
Are exploited simultaneously with multiple threads using the issue slots in a single clock cycle.

The issue slot usage is limited by the following factors

- Imbalance in the resource needs
- Resources availability over multiple threads
- Number of active threads considered
- Finite limitations of buffer
- Ability to fetch enough instruction from multiple threads
- Practical limitations of what instructions combinations can issue from one thread and multiple threads.

Design Challenges:

The design challenges of SMT processor include the following:

- Larger register file needed to 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 TLB conflicts generated by SMT do not degrade performance.

UNIT II MEMORY HIERARCHY DESIGN

Introduction – Optimizations of Cache Performance – Memory Technology and Optimizations – Protection: Virtual Memory and Virtual Machines – Design of Memory Hierarchies – Case Studies

2.1 Memory Hierarchy Introduction

Memory Hierarchy takes advantage of

1. locality
2. cost/performance of memory technologies

A memory hierarchy is organized into several levels in which each level is smaller, faster and more expensive per byte than the next lower level. The levels of the hierarchy usually subset one another

The goal is to provide a memory system with

- Cost almost as low as the cheapest level of memory and
- Speed almost as fast as the fastest level.

Principle of locality

Program access a relatively small portion of the address space at any instant of time is called Principle of locality.

Types:

- Temporal Locality

If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)

- Spatial Locality

If an item is referenced, items whose addresses are close by tend to be referenced soon. (e.g., straight line code, array access)

Figure 2.1 shows a multilevel memory hierarchy, including typical sizes and speeds of access. All data in one level is also found in the level below and all data in that lower level is found in the one below it, and so on until we reach the bottom of the hierarchy. The data contained in a lower level are a superset of the next higher level. This property, called the *inclusion property*, is always required for the lowest level of the hierarchy, which consists of main memory in the case of caches and disk memory in the case of virtual memory

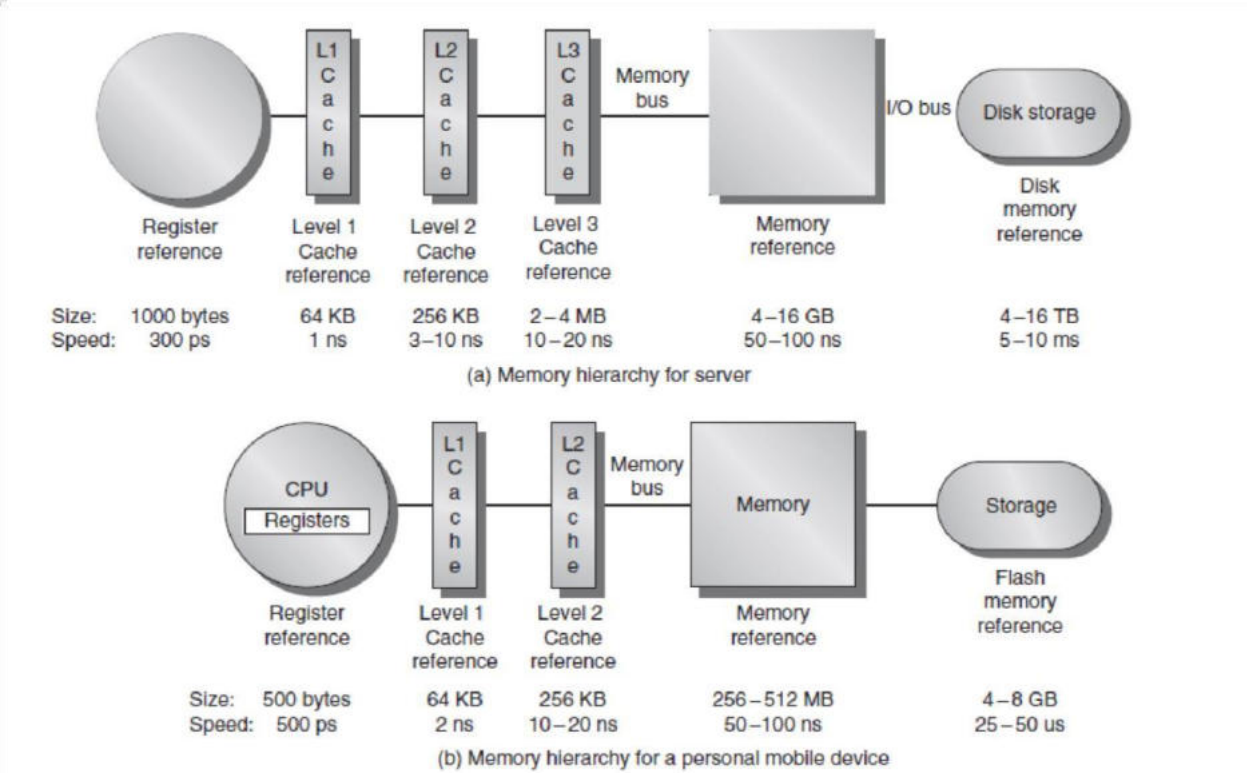


Figure 2.1 The levels in a typical memory hierarchy in a server computer shown on top (a) and in a personal mobile device (PMD) on the bottom (b).

Traditionally, designers of memory hierarchies focused on optimizing average memory access time, which is determined by the cache access time, miss rate, and miss penalty. In highend microprocessors, there may be 10 MB or more of on-chip cache, and a large second- or third-level cache will consume significant power both as leakage when not operating (called *static power*) and as active power, as when performing a read or write (called *dynamic power*).

Basics of Memory Hierarchies

When a word is not found in the cache, the word must be fetched from a lower level in the hierarchy (which may be another cache or the main memory) and placed in the cache before continuing. Multiple words, called a *block* (or *line*), are moved for efficiency reasons, and because they are likely to be needed soon due to spatial locality. Each cache block includes a *tag* to indicate which memory address it corresponds to.

Design decision – where the blocks can be placed in a cache**.** Some of the schemes are listed below.

Set associative: A block can be placed in a restricted set of places in the cache. A set is a group of blocks in the cache. A block is first mapped onto asset, and then block can be placed anywhere within that set. The set is usually obtained by (block address) MOD (Number of sets in a cache)
n-way Set associative: If there are n blocks in a set, the cache is called n-way set associative.
Direct mapped cache: Each block has only one place it can appear in the cache. So a block is placed in same location.

Fully associative: A block can be placed anywhere in the cache.
Write through cache: Write to both the block in the cache and the block in the lower-level memory. It updates item in cache and write through to update main memory.

Write Back: Write only to the block in the cache. It only updates copy in the cache. When a block is about to be replaced, it is copied back to memory.

Write Stall and Write Buffer: When the CPU must wait for writes to complete during Write Through, the CPU is said to Write stall. A common optimization to reduce write stall is a write buffer, which allows the processor to continue as soon as the data are written to the buffer, thereby overlapping processor execution with memory updating.

3 C's of Cache Miss

The three simple categories of cache Miss

- Compulsory – The first access to a block is not in the cache, so the block must be brought into the cache. 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 will occur due to blocks being discarded and later retrieved.
- Conflict – If block placement strategy is set associative or direct mapped, conflict misses will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called collision misses or interference misses.

Some designers prefer measuring *misses per instruction* rather than misses per memory reference (miss rate). These two are related:

$$\frac{Misses}{Instruction} = \frac{Miss\ rate \times Memory\ accesses}{Instruction\ count} = Miss\ rate \times \frac{Memory\ accesses}{Instruction}$$

A better measure is the *average memory access time*:

Average memory access time = Hit time + Miss rate × Miss penalty where *hit time* is the time to hit in the cache and *miss penalty* is the time to replace the block from memory (that is, the cost of a miss). Average memory access time is still an indirect measure of performance; **Techniques to reduce Miss rate**

The five techniques to Reduce miss rate are

- Larger block size
- Larger caches
- Higher associativity
- Way prediction and pseudo associative caches
- Compiler optimizations

2.2 Optimizations of Cache Performance

The average memory access time formula above gives us three metrics for cache optimizations: hit time, miss rate, and miss penalty. Ten advanced cache optimizations can be classified into five categories based on these metrics:

1. *Reducing the hit time*−Small and simple first-level caches and wayprediction. Both techniques also generally decrease power consumption.

2. *Increasing cache bandwidth*−Pipelined caches, multibanked caches, and nonblocking caches. These techniques have varying impacts on power consumption.

3. *Reducing the miss penalty*−Critical word first and merging write buffers. These optimizations have little impact on power.

4. *Reducing the miss rate*−Compiler optimizations. Obviously any improvement at compile time improves power consumption.

5. *Reducing the miss penalty or miss rate via parallelism*− Hardware prefetching and compiler prefetching. These optimizations generally increase power consumption, primarily due to prefetched data that are unused.

First Optimization: Small and Simple First-Level Caches to Reduce Hit Time and Power

The pressure of both a fast clock cycle and power limitations encourages limited size for first-level caches. Similarly, use of lower levels of associativity can reduce both hit time and power, although such trade-offs are more complex than those involving size.

The critical timing path in a cache hit is the three-step process of addressing the tag memory using the index portion of the address, comparing the read tag value to the address, and setting the multiplexor to choose the correct data item if the cache is set associative. Directmapped caches can overlap the tag check with the transmission of the data, effectively reducing hit time. Furthermore, lower levels of associativity will usually reduce power because fewer cache lines must be accessed.

Although the total amount of on-chip cache has increased dramatically with new generations of microprocessors, due to the clock rate impact arising from a larger L1 cache, the size of the L1 caches has recently increased either slightly or not at all. In many recent processors, designers have opted for more associativity rather than larger caches.

One approach to determining the impact on hit time and power consumption in advance of building a chip is to use CAD tools. CACTI is a program to estimate the access time and energy consumption of alternative cache structures on CMOS microprocessors within 10% of more detailed CAD tools.

Second Optimization: Way Prediction to Reduce Hit Time

In *way prediction*, extra bits are kept in the cache to predict the way, or block within the set of the *next* cache access. This prediction means the multiplexor is set early to select the desired block, and only a single tag comparison is performed that clock cycle in parallel with reading the cache data. A miss results in checking the other blocks for matches in the next clock cycle.

Added to each block of a cache are block predictor bits. The bits select which of the blocks to try on the *next* cache access. If the predictor is correct, the cache access latency is the fast hit time. If not, it tries the other block, changes the way predictor, and has a latency of one extra clock cycle.

An extended form of way prediction can also be used to reduce power consumption by using the way prediction bits to decide which cache block to actually access (the way prediction bits are essentially extra address bits); this approach, which might be called *way selection*, saves power when the way prediction is correct but adds significant time on a way misprediction, since the access, not just the tag match and selection, must be repeated. Such an optimization is likely to make sense only in low-power processors.

Third Optimization: Pipelined Cache Access to Increase Cache Bandwidth

This optimization is simply to pipeline cache access so that the effective latency of a first-level cache hit can be multiple clock cycles, giving fast clock cycle time and high bandwidth but slow hits.

For example, the pipeline for the instruction cache access for Intel Pentium processors in the mid-1990s took 1 clock cycle, for the Pentium Pro through Pentium III in the mid-1990s through 2000 it took 2 clocks, and for the Pentium 4, which became available in 2000, and the current Intel Core i7 it takes 4 clocks. This change increases the number of pipeline stages, leading to a greater penalty on mispredicted branches and more clock cycles between issuing the load and using the data, but it does make it easier to incorporate high degrees of associativity

Fourth Optimization: Nonblocking Caches to Increase Cache Bandwidth

For pipelined computers that allow out-of-order execution the processor need not stall on a data cache miss. For example, the processor could continue fetching instructions from the instruction cache while waiting for the data cache to return the missing data.

A *nonblocking cache* or *lockup-free cache* escalates the potential benefits of such a scheme by allowing the data cache to continue to supply cache hits during a miss. This "hit under miss" optimization reduces the effective miss penalty by being helpful during a miss instead of ignoring the requests of the processor.

A subtle and complex option is that the cache may further lower the effective miss penalty if it can overlap multiple misses: a "hit under multiple miss" or "miss under miss" optimization.

The second option is beneficial only if the memory system can service multiple misses; most high-performance processors (such as the Intel Core i7) usually support both, while lower end processors, such as the ARM A8, provide only limited nonblocking support in L2.

The effective miss penalty is not the sum of the misses but the nonoverlapped time that the processor is stalled. The benefit of nonblocking caches is complex, as it depends upon the miss penalty when there are multiple misses, the memory reference pattern, and how many instructions the processor can execute with a miss outstanding.

Deciding how many outstanding misses to support depends on a variety of factors:

- The temporal and spatial locality in the miss stream, which determines whether a miss can initiate a new access to a lower level cache or to memory
- The bandwidth of the responding memory or cache
- To allow more outstanding misses at the lowest level of the cache (where the miss time is the longest) requires supporting at least that many misses at a higher level, since the miss must initiate at the highest level cache ▪ The latency of the memory system.

Fifth Optimization: Multibanked Caches to Increase Cache Bandwidth

Rather than treat the cache as a single monolithic block, we can divide it into independent banks that can support simultaneous accesses. Banks were originally used to improve performance of main memory and are now used inside modern DRAM chips as well as with caches. The Arm Cortex-A8 supports one to four banks in its L2 cache; the Intel Core i7 has four banks in L1 (to support up to 2 memory accesses per clock), and the L2 has eight banks.

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 2.2 shows this interleaving.

Multiple banks also are a way to reduce power consumption both in caches and DRAM.

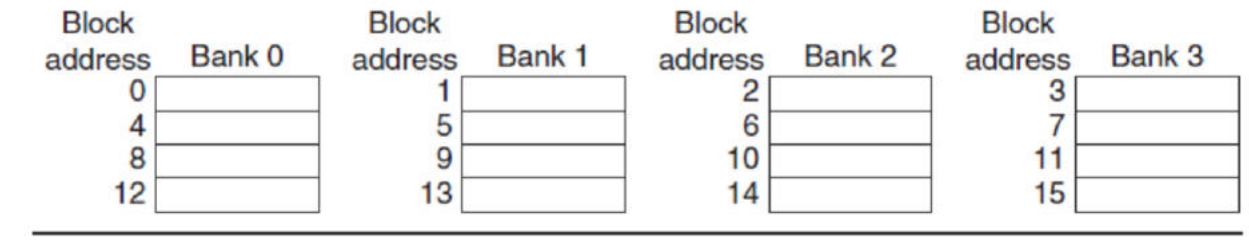


Figure 2.2 Four-way interleaved cache banks using block addressing

Sixth Optimization: Critical Word First and Early Restart to Reduce Miss Penalty

This technique is based on the observation that the processor normally needs just one word of the block at a time.

Two strategies:

Critical word first− Request the missed word first from memory and send it to the processor as soon as it arrives; let the processor continue execution while filling the rest of the words in the block.

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

When there is a second request in critical word first, the effective miss penalty is the nonoverlapped time from the reference until the second piece arrives. The benefits of critical word first and early restart depend on the size of the block and the likelihood of another access to the portion of the block that has not yet been fetched.

Seventh Optimization: Merging Write Buffer to Reduce Miss Penalty

Write-through caches rely on write buffers, as all stores must be sent to the next lower level of the hierarchy. 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 processor's perspective; the processor 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 the new data matches the address of a valid write buffer entry. If so, the new data are combined with that entry. *Write merging* is the name of this optimization. Ex: The Intel Core i7

If the buffer is full and there is no address match, the cache (and processor) 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.

Note that input/output device registers are often mapped into the physical address space. These I/O addresses *cannot* allow write merging because separate I/O registers may not act like an array of words in memory.

Eighth Optimization: Compiler Optimizations to Reduce Miss Rate

Thus far, our techniques have required changing the hardware. This next technique reduces miss rates without any hardware changes.

This magical reduction comes from optimized software - the hardware designer's favorite solution! 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. The optimizations presented below are found in many modern compilers.

Loop Interchange

Some programs have nested loops that access data in memory in nonsequential order. Simply exchanging the nesting of the loops can make the code access the data in the order in which they are stored.

For example, if x is a two-dimensional array of size [5000, 100] allocated so that x[i,j] and x[i,j+1] are adjacent (an order called row major, since the array is laid out by rows), then the two pieces of code below show how the accesses can be optimized:

```
/* 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

Blocked algorithms operate on sub matrices or blocks. the goal is to maximize accesses to the data loaded into the cache before the data re replaced. This optimization tries to reduce misses via improved temporal locality.

```
/* Before */ for (i = 0; i
< N; i = i+1) for (j = 0; j
< N; j = j+1)
{r = 0; for (k = 0; k < N; k =
k + 1) r = r + y[i][k]*z[k][j];
x[i][j] = r;
};
```

A snapshot of the accesses to the three arrays

A dark shade indicates a recent access, a light shade indicates an older access, and white means not yet accessed. The two inner loops read all N-by-N elements of z, read the same N elements in a row of y repeatedly, and write one row of N elements of x.

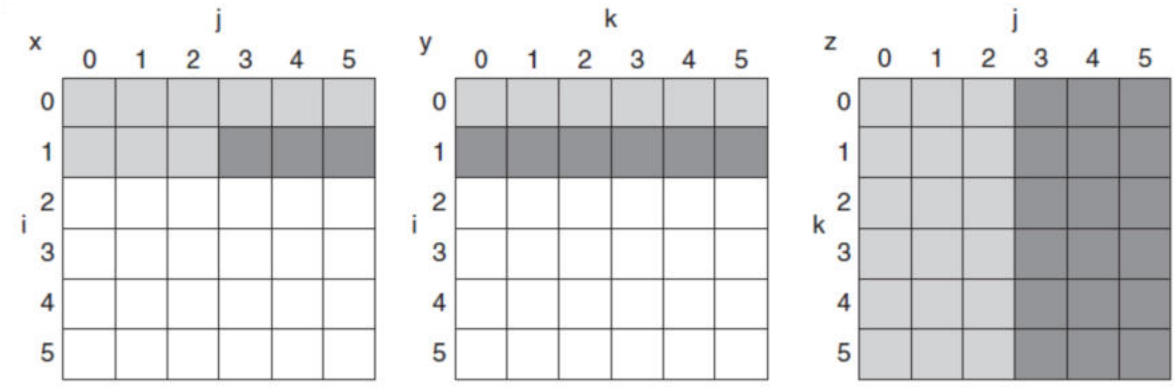


Figure 2.3 A snapshot of the three arrays x, y, and z when *N* = 6 and i = 1

To ensure that the elements being accessed can fit in the cache, the original code is changed to compute on a submatrix of size B by B. Two inner loops now compute in steps of size B rather than the full length of x and z. B is called the *blocking factor*. (Assume x is initialized to zero.)

```
/* After */ for (jj = 0; jj < N;
jj = jj+B)
        for (kk = 0; kk < N; kk = kk+B)                 for (i
= 0; i < N; i = i+1)                    for (j = jj; j <
min(jj+B,N); j = j+1)
                        {       r = 0;
                                for (k = kk; k < min(kk+B,N); k = k + 1)
                                r = r + y[i][k]*z[k][j];
                        x[i][j] = x[i][j] + r;
                                                        };
```

Figure 2.4 illustrates the accesses to the three arrays using blocking. Looking only at capacity misses, the total number of memory words accessed is 2N3/B + N2. This total is an improvement by about a factor of B. Hence, blocking exploits a combination of spatial and temporal locality, since y benefits from spatial locality and z benefits from temporal locality.

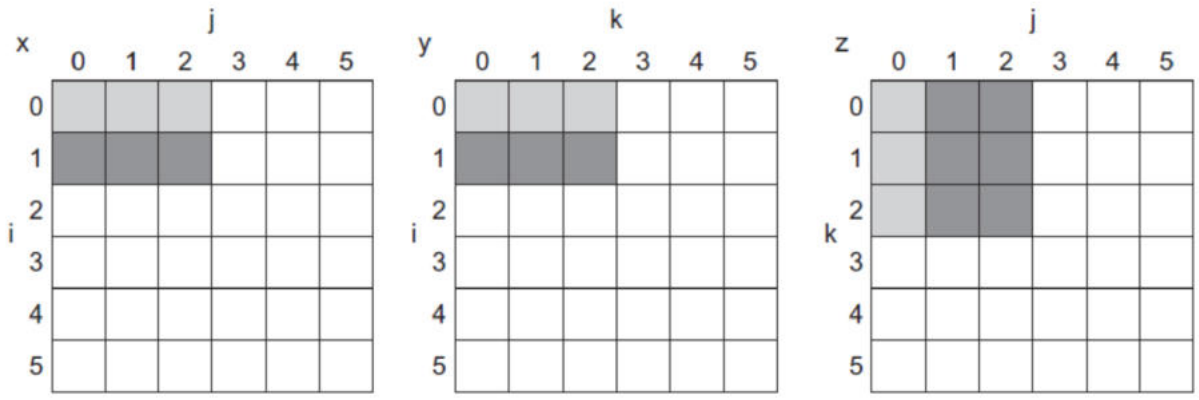


Figure 2.4 The age of accesses to the arrays x, y, and z when *B* = 3

Ninth Optimization: Hardware Prefetching of Instructions and Data to Reduce Miss Penalty or Miss Rate

Nonblocking caches effectively reduce the miss penalty by overlapping execution with memory access. Another approach is to prefetch items before the processor requests them. Both instructions and data can be prefetched, either directly into the caches or into an external buffer that can be more quickly accessed than main memory.

Instruction prefetch is frequently done in hardware outside of the cache. The processor fetches two blocks on a miss

Requested block

- placed in I-cache when it returns **Prefetched block**
- placed in instruction stream buffer (ISB)
- original cache request is canceled ▪ block is read from the stream buffer ▪ next prefetch request is issued.

When prefetched data are not used or useful data are displaced, prefetching will have a very negative impact on power.

Tenth Optimization: Compiler-Controlled Prefetching to Reduce Miss Penalty or Miss Rate

Compiler-controlled prefetching is a prefetching for the compiler to insert prefetch instructions to request the data before they are needed. Register prefetch is to load the value into a register. Cache prefetch is to load data only into the cache.

Faulting vs nonfaulting: the address does or does not cause an exception for virtual address faults and protection violations. The most effective prefetch is "semantically invisible" to a program

- doesn't change the contents of registers and memory and
- cannot cause virtual memory faults

Nonbinding prefetch: nonfaulting cache prefetch

Overlapping execution: CPU proceeds while the prefetcched data are being fetched. The compiler may avoid unnecessary prefetches in hardware. Prefetch instructions incurs instruction overhead.

Summary of reducing cache miss penalty / miss Rate via Paralleleism

The hardware and software prefetching techniques leverage excess memory bandwidth for performance by trying to anticipate the needs of a cache.

2.3 Memory Technology and Optimizations

Main memory is the next level down in the hierarchy. Main memory satisfies the demands of caches and serves as the I/O interface, as it is the destination of input as well as the source for output. Performance measures of main memory emphasize both latency and bandwidth.

To allow memory systems to keep up with the bandwidth demands of modern processors, memory innovations started happening inside the DRAM chips themselves. Latency is quoted using two measures:

Access time – The time between when a read is requested and when the desired word arrives, and

Cycle time **–** The minimum time between unrelated requests to memory.

2.3.1 SRAM Technology

Cache uses SRAM: Static Random Access Memory. SRAMs don't need to refresh, so the access time is very close to the cycle time. SRAM uses six transistors per bit to prevent the information from being disturbed when read.

In earlier times, most desktop and server systems used SRAM chips for their primary, secondary, or tertiary caches; today, all three levels of caches are integrated onto the processor chip. Currently, the largest on-chip, third-level caches are 12 MB, while the memory system for such a processor is likely to have 4 to 16 GB of DRAM. The access times for large, third-level, on-chip caches are typically two to four times that of a second-level cache, which is still three to five times faster than accessing DRAM memory.

2.3.2 DRAM Technology

Figure 2.5 shows the basic DRAM organization. One-half of the address is sent first during the *row access strobe* (RAS). The other half of the address, sent during the *column access strobe* (CAS), follows it. These names come from the internal chip organization, since the memory is organized as a rectangular matrix addressed by rows and columns.

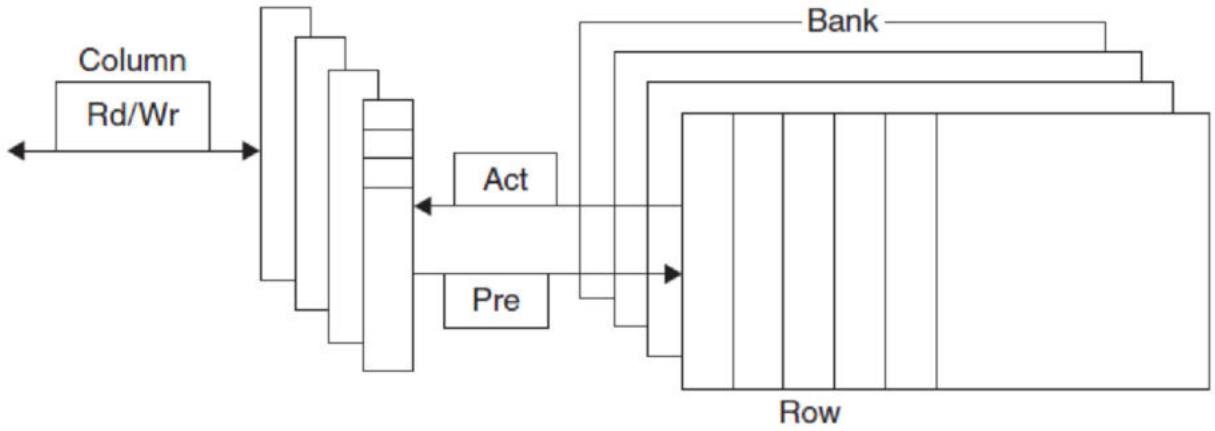


Figure 2.5 Internal organization of a DRAM

DRAM derives from the property signified by its first letter, *D*, for *dynamic*. To pack more bits per chip, DRAMs use only a single transistor to store a bit. Reading that bit destroys the information, so it must be restored. This is one reason why the DRAM cycle time was traditionally longer than the access time; more recently, DRAMs have introduced multiple banks, which allow the rewrite portion of the cycle to be hidden.

In addition, to prevent loss of information when a bit is not read or written, the bit must be "refreshed" periodically. Memory controllers include hardware to refresh the DRAMs periodically. This requirement means that the memory system is occasionally unavailable because it is sending a signal telling every chip to refresh. The time for a refresh is typically a full memory access (RAS and CAS) for each row of the DRAM. Since the memory matrix in a DRAM is conceptually square, the number of steps in a refresh is usually the square root of the DRAM capacity. DRAM designers try to keep time spent refreshing to less than 5% of the total time.

Refresh belies that analogy, since some accesses take much longer than others do. Thus, refresh is another reason for variability of memory latency and hence cache miss penalty.

DIMM

DIMM stands for Dual inline memory module. DRAM chips are commonly sold on small boards called DIMMs. DIMMs typically contain 4 to 16 DRAMs. The new interfaces to improve the data transfer time is

- Slowing down in DRAM capacity growth
- Four times the capacity every three years for more than 20 years.

- New chips only double capacity every two year, since 1998.

DRAM performance is growing at a slower rate

RAS (related to latency): 5% per year

CAS (related to bandwidth): 10% per year

Improving Memory Performance Inside a DRAM Chip

First, DRAMs added timing signals that allow repeated accesses to the row buffer without another row access time. Such a buffer comes naturally, as each array will buffer 1024 to 4096 bits for each access.

The second major change was to add a clock signal to the DRAM interface, so that the repeated transfers would not bear that overhead. It is known as Synchronous DRAM (SDRAM).

Third, to overcome the problem of getting a wide stream of bits from the memory without having to make the memory system too large as memory system density increased, DRAMS were made wider.

The fourth major DRAM innovation to increase bandwidth is to transfer data on both the rising edge and falling edge of the DRAM clock signal, thereby doubling the peak data rate called Double data rate (DDR).

SDRAMs also introduced *banks*, breaking a single SDRAM into 2 to 8 blocks that can operate independently. Creating multiple banks inside a DRAM effectively adds another segment to the address, which now consists of bank number, row address, and column address. When an address is sent that designates a new bank, that bank must be opened, incurring an additional delay. The management of banks and row buffers is completely handled by modern memory control interfaces, so that when subsequent access specifies the same row for an open bank, the access can happen quickly, sending only the column address.

Graphics Data RAMs

GDRAMs or GSDRAMs (Graphics or Graphics Synchronous DRAMs) are a special class of DRAMs based on SDRAM designs but tailored for handling the higher bandwidth demands of graphics processing units. Since Graphics Processor Units require more bandwidth per DRAM chip than CPUs, GDDRs have several important differences:

1. GDDRs have wider interfaces: 32-bits versus 4, 8, or 16 in current designs.
2. GDDRs have a higher maximum clock rate on the data pins. To allow a higher transfer rate without incurring signaling problems, GDRAMS normally connect directly to the GPU and are attached by soldering them to the board, unlike DRAMs, which are normally arranged in an expandable array of DIMMs.

Reducing Power Consumption in SDRAMs

Power consumption in dynamic memory chips consists of both dynamic power used in a read or write and static or standby power; both depend on the operating voltage.

Flash Memory

Flash memory is a type of EEPROM (Electronically Erasable Programmable Read-Only Memory), which is normally read-only but can be erased. The other key property of Flash memory is that it holds it contents without any power. Flash is used as the backup storage in PMDs in the same manner that a disk functions in a laptop or server. The most important differences are:

- Flash memory must be erased before it is overwritten, and it is erased in blocks rather than individual bytes or words
- Flash memory is static
- Flash memory has a limited number of write cycles for any block, typically at least 100,000.
- High-density Flash is cheaper than SDRAM but more expensive than disks ▪ Flash is much slower than SDRAM but much faster than disk

Enhancing Dependability in Memory Systems

Large caches and main memories significantly increase the possibility of errors occurring both during the fabrication process and dynamically, primarily from cosmic rays striking a memory cell. These dynamic errors, which are changes to a cell's contents, not a change in the circuitry, are called *soft errors.*

In addition to fabrication errors that must be fixed at configuration time, *hard errors*, which are permanent changes in the operation of one of more memory cells, can occur in operation.

Dynamic errors can be detected by parity bits and detected and fixed by the use of Error Correcting Codes (ECCs). In very large systems, the possibility of multiple errors as well as complete failure of a single memory chip becomes significant

2.4 Protection: Virtual Memory and Virtual Machines Virtual Machine:

"A fully protected and isolated copy of the underlying physical machine's hardware."

Virtual Machine Monitor (Hypervisor)

*"*A thin layer of software that's between the hardware and the Operating system, virtualizing and managing all hardware resources" **Why Virtual Machine?**

- Multiplex an expensive machine
- Ability to run multiple OSes targeted for different workloads
- Ability to run new OS versions side by side with a stable older version - No problem if new OS breaks compatibility for existing apps; just run them in a VM on old OS **Virtual Memory**
- Break address space up into pages
- Each program accesses a working set of pages **Store pages:**
- In physical memory as space permits
- On disk when no space left in physical memory
- Access pages using virtual address

Protection via Virtual memory

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

Multiprogramming, where several programs running concurrently would share a computer, led to demands for protection. Process is a running program. At any instant, it must be possible to switch from one process to another. This exchange is called a ***process switch*** **or** ***context switch*****.** The operating system and architecture join forces to allow processes to share the hardware yet not interfere with each other. At a minimum, the architecture must do the following:

- Provide User Process and Kernel Process ▪ Readable portion of Processor state:
  - –User supervisor mode bit
  - –Exception enable/disable bit
  - –Memory protection information
- System call to transfer to supervisor mode
  –Return like normal subroutine to user mode ▪

Provide mechanism to limit memory access

A page table is the data structure used by a virtual memory system to store the mapping between virtual addresses and physical addresses. The protection restrictions are included in each page table entry. The protection restrictions might determine whether a user process can read this page, whether a user process can write to this page, and whether code can be executed from this page. Only the OS can update the page table, the paging mechanism provides total access protection.

Paged virtual memory 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. The solution is to 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 requires a second access to translate the address. To reduce address translation time, computers use a cache dedicated to these address translations, called a **translation look-aside buffer**, or simply **translation buffer**. 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 address, protection field, valid bit, and usually a use bit and a dirty bit. Bugs in OS lead to compromising security. Bugs likely due to huge size of OS code .

Protection via Virtual Machines

The goal of Virtualization is Run multiple instances of different OS on the same hardware. Present a transparent view of one or more environments (M-to-N mapping of M "real" resources, N "virtual" resources).

Two components when using virtualization:

– Virtual Machine Monitor (VMM)

– Virtual Machine(s) (VM)

Virtual Machine Monitor-'hypervisor'

The software that supports VMs is called a *virtual machine monitor* (VMM) or *hypervisor*; the VMM is the heart of virtual machine technology. The underlying hardware platform is called the *host*, and its resources are shared among the *guest* VMs. Every Virtual machine has virtual devices that provide same functionality as physical hardware.

Cost of Processor Virtualization

- VM is much smaller than traditional OS
- Isolation portion is only about 10000 lines for a VMM
- Processor bound programs have very little virtualization overhead
- I/O bound jobs have more overhead
- ISA emulation is costly

Benefits of VMs

1. Managing software - VMs provide an abstraction that can run the Complete software stack includes Old Oses like DOS. It also run Current stable OS, next OS release .

2. Managing Hardware – Here Multiple servers avoided. VMs enable hardware sharing. VMMs support migration of a running VM to another machine for balancing load or evacuate from failing hardware.

Requirements of a Virtual Machine Monitor

- Guest software should behave on a VM exactly as if it were running on the native hardware, except for performance-related behavior or limitations of fixed resources shared by multiple VMs.
- Guest software should not be able to change allocation of real system resources directly.

To "virtualize" the processor, the VMM must control access to privileged state, address translation, I/O, exceptions and interrupts even though the guest VM and OS currently running are temporarily using them.

The basic requirements of system virtual machines are:

- At least two processor modes, system and user.
- A privileged subset of instructions that is available only in system mode, resulting in a trap if executed in user mode. All system resources must be controllable only via these instructions.

(Lack of) Instruction Set Architecture Support for Virtual Machines

An architecture that allows the VM to execute directly on the hardware is said to be virtualizable. If VMs are planned for during the design of the ISA, its relatively easy to both reduce the number of instructions that must be executed by a VMM and how long it takes to emulate them.

The VMM must ensure that the guest system only interacts with virtual resources, a conventional guest OS runs as a user mode program on top of the VMM. Then, if a guest OS attempts to access or modify information related to hardware resources via a privileged instruction– for example, reading or writing the page table pointer– it will trap to the VMM. The

VMM can then effect the appropriate changes to corresponding real resources.

Hence, if any instruction that tries to read or write such sensitive information traps when executed in user mode, the VMM can intercept it and support a virtual version of the sensitive information as the guest OS expects.

Impact of Virtual Machines on Virtual Memory and I/O

Another challenge is virtualization of virtual memory, as each guest OS in every VM manages its own set of page tables. To make this work, the VMM separates the notions of *real* and *physical memory* and makes real memory a separate, intermediate level between virtual memory and physical memory. Three abstractions of memory: *virtual memory, physical memory*, and *machine memory*

The guest OS maps virtual memory to real memory via its page tables, and the VMM page tables map the guests real memory to physical memory. The virtual memory architecture is specified either via page tables, or via the TLB structure, as in many RISC architectures.

In each VM, OS creates and manages page tables for its virtual address spaces without modification. But these page tables are not used by the MMU hardware. VMM creates and

manages page tables that map virtual pages directly to machine pages. These tables are loaded into the MMU on a context switch u VMM page tables are the **shadow page tables**.

To virtualize the TLB in many RISC computers, the VMM manages the real TLB and has a copy of the contents of the TLB of each guest VM. To pull this off, any instructions that access the TLB must trap. TLBs with Process ID tags can support a mix of entries from different VMs and the VMM, thereby avoiding flushing of the TLB on a VM switch.

The final portion of the architecture to virtualize is I/O. This is the most difficult part of system virtualization because of the increasing number of I/O devices attached to the computer *and* the increasing diversity of I/O device types. The method for mapping a virtual to physical I/O device depends on the type of device.

2.5 The Design of memory Hierarchies

Protection and Instruction Set Architecture

Protection is a joint effort of architecture and operating systems, but architects had to modify some awkward details of existing instruction set architectures when virtual memory became popular.

Example: 80x86 POPF: Loads flag registers from top of stack in memory

One flag is Interrupt Enable (IE)

In user mode, POPF changes all flags except IE

In system mode, POPF changes all flags

Guest OS runs in user mode inside VM

Problem: Guest expects to change IE

IBM mainframe hardware: 3 steps to improve performance

Reduce cost of processor virtualization

Reduce interrupt overhead

Reduce interrupt cost by steering interrupts to proper VM without going through VMM

Speculative Execution and the Memory System

Speculative and conditional instructions generate exceptions (by generating invalid addresses) that would otherwise not occur, which in turn can overwhelm the benefits of speculation with the exception handling overhead. Such CPUs must be matched with non-blocking caches and only speculate on L1 misses (due to the unbearable penalty of L2).

I/O and Consistency of Cached Data

Cache coherence problem must be addressed when I/O devices also share the same cached data.

UNIT III MULTIPROCESSOR ISSUES

Introduction- Centralized, Symmetric and Distributed Shared Memory Architectures –Cache Coherence Issues – Performance Issues – Synchronization – Models of Memory Consistency – Case Study-Interconnection Networks – Buses, Crossbar and Multi-stage Interconnection

Networks

3.1 Introduction

Multiprocessor Operating System refers to the use of two or more central processing units (CPU) within a single computer system. The importance of multiprocessors was growing throughout the 1990s as designers sought a way to build servers and supercomputers that achieved higher performance than a single microprocessor.

Need of multiprocessor:

- A single CPU can only go so fast, use more than one CPU to improve performance
- Multiple users
- Multiple applications
- Multi-tasking within an application
- Responsiveness and/or throughput
- Share hardware between CPUs

Multiprocessors, which we define as computers consisting of tightly coupled processors whose coordination and usage are typically controlled by a single operating system and that share memory through a shared address space. Such systems exploit thread-level parallelism through two different software models. They are:

1. **Parallel processing** - execution of a tightly coupled set of threads collaborating on a single task

2. **Request level parallelism** - execution of multiple, relatively independent processes that may originate from one or more users.

Request-level parallelism may be exploited by a single application running on multiple processors, such as a database responding to queries, or multiple applications running independently, is called ***multiprogramming***.

Multiprocessors can be used in different ways:

- Uniprossesors (single-instruction, single-data or SISD)
- Within a single system to execute multiple, independent sequences of instructions in multiple contexts (multiple-instruction, multiple-data or MIMD);
- A single sequence of instructions in multiple contexts (single-instruction, multiple-data or SIMD, often used in vector processing);
- Multiple sequences of instructions in a single context (multiple-instruction, single-data or MISD, used for redundancy in fail-safe systems and sometimes applied to describe pipelined processors or hyper threading).

3.1.1 Multiprocessor Architecture: Issues and Approach

Thread-Level parallelism have multiple program counters. It uses MIMD model and it is targeted for tightly-coupled shared-memory multiprocessors. For n processors, need n threads. Amount of computation assigned to each thread is called grain size. Threads can be used for data-level parallelism, but the overheads may outweigh the benefit. The existing shared-memory multiprocessors fall into two classes, depending on the number of processors involved, which in turn dictates a memory organization and interconnect strategy. They are:

1. Centralized shared memory (―Uniform memory access or ―Shared memory processor‖)
2. Physically Distributed Memory multiprocessor (Decentralized memory or memory module with CPU)

1. Centralized shared memory architecture

Centralized shared memory architecture share a single centralized memory, interconnect processors and memory by a bus. It is also known as ―uniform memory access‖ (UMA) or

―Symmetric (shared memory) multiprocessor‖ (SMP).

With large caches, the bus and the single memory, possibly with multiple banks, can satisfy the memory demands of a small number of processors. By replacing a single bus with multiple buses, or even a switch, a centralized shared memory design can be scaled to a few dozen processors. Although scaling beyond that is technically possible, sharing a centralized memory, even organized as multiple banks, becomes less attractive as the number of processors sharing it

increases. Figure 3.1 illustrates the basic structure of a centralized shared memory multiprocessor architecture.

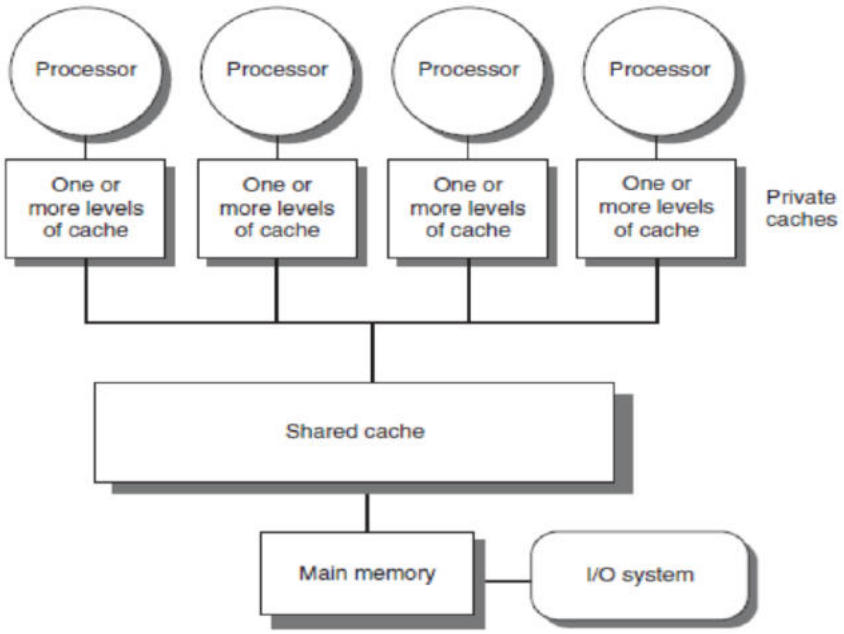


Figure 3.1 Basic structure of a centralized shared-memory multiprocessor based on a multicore chip.

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 often called symmetric shared-memory multiprocessor (SMP), and this style of architecture is sometimes called UMA (Uniform Memory Access).

2. Distributed Shared Memory

The alternative design approach consists of multiprocessors with physically distributed memory, called *distributed shared memory* (DSM). To support larger processor counts, memory must be distributed among the processors rather than centralized; otherwise, the memory system would not be able to support the bandwidth demands of a larger number of processors without incurring excessively long access latency. The introduction of multicore processors has meant that even two-chip multiprocessors use distributed memory. Of course, the larger number of processors raises the need for a high bandwidth interconnection. Both directed networks (i.e., switches) and indirect networks (typically multidimensional meshes) are used. The typical Distributed memory multiprocessor is illustrated in the following diagram 3.2 .

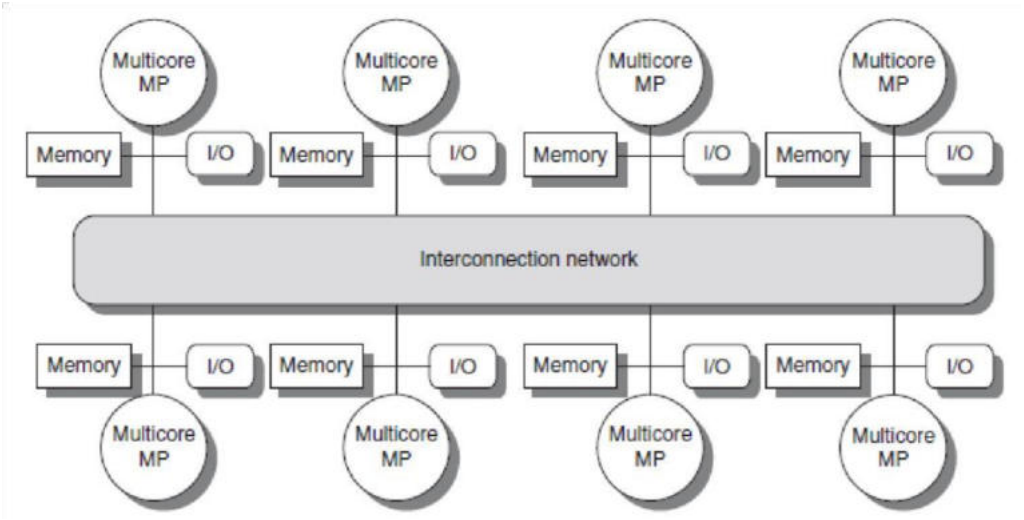


Figure 3.2 The basic architecture of a distributed-memory multiprocessor

Benefits:

Distributing memory among the nodes has two major benefits:

- It is a cost-effective way to scale the memory bandwidth
- It reduces the latency for access to the local memory

Drawbacks:

- Longer communication latency for communicating data between processors ▪ Software model more complex

In both SMP and DSM architectures, communication among threads occurs through a shared address space, meaning that a memory reference can be made by any processor to any memory location, assuming it has the correct access rights. The term *shared memory* associated with both SMP and DSM refers to the fact that the *address space* is shared.

3.1.2 Challenges of Parallel Processing

The application of multiprocessors ranges from running independent tasks to running parallel programs. Two important hurdles which make parallel processing challenging are;

1. Limit Parallelism available in programs

The problem of inadequate application parallelism must be attacked primarily in software with new algorithms that can have better parallel performance.

2. Large latency of remote access in a parallel processor

Reducing the impact of long remote latency can be attacked both by the architecture and by the programmer.

Example for Limited Parallelism:

Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?

Assume that the program operates in only two modes:

1. Parallel with all processors fully used – enhanced mode
2. Serial with only one processor in use

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} + (1 - \text{Fraction}_{\text{enhanced}})}$$

The value of

$\text{Speedup}_{\text{enhanced}}$ = the number of processors

= 100

$\text{Fraction}_{\text{enhanced}}$ = the time spent in parallel mode

$$80 = \frac{1}{\frac{\text{Fraction}_{\text{parallel}}}{100} + (1 - \text{Fraction}_{\text{parallel}})}$$

$$0.8 \times \text{Fraction}_{\text{parallel}} + 80 \times (1 - \text{Fraction}_{\text{parallel}}) = 1$$

$$80 - 79.2 \times \text{Fraction}_{\text{parallel}} = 1$$

$$\text{Fraction}_{\text{parallel}} = \frac{80 - 1}{79.2}$$

$$\text{Fraction}_{\text{parallel}} = 0.9975$$

Thus to achieve a speedup of 80 with 100 processors only 0.25% of original computation can be sequential.

Example for Long latency

Suppose 32 processor MP, 2 GHz, 400ns remote memory all local accesses hit memory hierarchy and base CPI is 0.5. Processors are stalled on a remote request and the processor clock rate is 1 GHz. What is the performance impact if 0.2% instructions involve remote access?

CPI = Base CPI + Remote request rate × Remote request cost Given
base IPC, then

$$\text{CPI} = \frac{1}{\text{Base IPC}} + \text{Remote request rate} \times \text{Remote request cost}$$

Base IPC = 2

Remote request rate = 0.2%

Remote request cost = Remote access cost / Cycle Time

= 400ns / 1ns

= 400 cycles

CPI = ½ + 0.2% × 400

= ½ + 0.2 × 4

= 0.5 + 0.8

= 1.3

Thus the multiprocessor with all local references is 1.3 / 0.5 = 2.6 times faster

3.2 Centralized Shared-Memory Architectures

Symmetric Shared memory Architectures is a small scale multiprocessor where several processors shared a single physical memory connected by a shared bus. It is called so because each processor has the same relationship to one single shared memory.

Symmetric shared-memory machines usually support the caching of both shared and private data.

Private data – are used by a single processor, while **shared data** are used by multiple processors, essentially providing communication among the processors through 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.

Shared data – When shared data are cached, the shared value may be replicated in multiple caches. In addition to the reduction in access latency and required memory bandwidth, this replication also provides a reduction in contention that may exist for shared data items that are being read by multiple processors simultaneously.

3.2.1 Multiprocessor Cache Coherence

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. Caching of shared data, however introduces a new problem is called a cache coherence. That is two processors can have two different values for the same memory location.

Figure 3.3 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.*

Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

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 **coherenc**e, 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 defines the behavior of reads and writes to the same memory location, the memory system is coherent if the following conditions are met:

1. Program order Preserving

- 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.
- The first property simply preserves program order—we expect this property to be true even in uniprocessors.

2. Coherent view of memory

- 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.
- If processors can continuously read the old value we would clearly say that memory was incoherent.

3. Write Serialization

- 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 need for write serialization is more subtle, but equally important. A write by processor P1 may not be seen by a read from processor p2 if the read is made within a very small time after the write has been made.

The memory consistency model defines when a written value must be seen by a following read instruction made by the other processors.

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. 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 restrictions allow the processor to reorder reads, but forces the processor to finish a write in program order.

3.2.2 Basic Schemes for Enforcing Coherence

A program running on multiple processors will normally have copies of the same data in several caches. In a coherent multiprocessor, the caches provide the following key to the performance of shared data:

1. Migration
2. Replication

Migration: 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.

Replication: Coherent caches also provide replication for shared data that are 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.

Cache Coherence Protocols

The protocols to maintain coherence for multiple processors are called cache coherence protocol. Key to implementing a cache coherence protocol is tracking the state of any sharing of a data block. There are two classes of protocols. They are:

1. Directory based
2. Snooping

1. Directory based

Directory based is a one in which the Sharing status of a block of physical memory is kept in just one location called directory.

2. Snooping

Every cache with copy of data also has copy of sharing status of block, but no centralized state is kept. All caches are accessible via some broadcast medium (a bus or switch). All cache controllers monitor or snoop on the medium to determine whether or not they have copy of a block that is requested on bus or switch access.

3.2.3 Protocols for Coherency

Write invalidate protocol – When one processor writes, invalidate all copies of this data that may be in other caches

Write update protocol – When one processor writes, broadcast the value and update any copies that may be in other caches.

Write invalidate protocol

Write invalidate protocol is a method to ensure that a processor has exclusive access to a data item before it writes that item. It invalidates other copies on a write. It allows multiple readers and single writer.

Read miss:

Write-through: memory is always up-to-date Write-back:

snoop in caches to find most recent copy.

Example:

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

Consider a write followed by a read by another processor

When CPU A reads X, it misses in the cache and is forced to fetch a new copy of the data.

When CPU B reads X, it misses in the cache and is forced to fetch a new copy of the data.

When CPU A writes X, ○ Prevent other processor (CPU B) from writing simultaneously.

- Any copy held by the reading processor must be invalidated (CPU B).
  - Since write requires exclusive access.
  - If two processors write at the same time, one wins and obtains exclusive access
  - The processor that completes its write must obtain the new copy of data and the updated value.

Write broadcast or write update protocol

Write update protocol is a protocol used 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.

Read miss:

Write-through: memory is always up-to-date

Write-back: snoop in caches to find most recent copy

Write invalidate versus Broadcast:

Multiple writers to write the same word or word within the block, only one invalidate. Invalidate uses spatial locality: one transaction per block

Basic Implementation Techniques

The technique to implement an invalidate protocol in a small scale multiprocessor is the use of **bus** or other broadcast medium to perform invalidates. To perform an invalidate, the processor simply acquires bus access and broadcasts the address to be invalidated on the bus. All processors continuously snoop on the bus, watching the addresses. The processors check whether the address on the bus is in their cache. If so, the corresponding data in the cache are invalidated. When a write to a block that is shared occurs, the writing processor must acquire bus access to broadcast its invalidation. If two processors attempt to write shared blocks at the same time, their attempts to broadcast an invalidate operation will be serialized when they arbitrate for the bus. The first processor to obtain bus access will cause any other copies of the block it is writing to be

invalidated. If the processors were attempting to write the same block, the serialization enforced by the bus also serializes their writes.

In a **write-through cache**, it is easy to find the recent value of a data item, since all written data are always sent to the memory, from which the most recent value of a data item can always be fetched.

For a **write-back cache**, the problem of finding the most recent data value is harder, since the most recent value of a data item can be in a private cache rather than in the shared cache or memory. Write-back caches can use the same snooping scheme both for cache misses and for writes: Each processor snoops every address placed on the shared 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 (or L3) access to be aborted.

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. To track whether or not a cache block is shared, we can add an extra state bit associated with each cache block, just as we have a valid bit and a dirty bit. By adding a bit indicating whether the block is shared, we can decide whether a write must generate an invalidate. When a write to a block in the shared state occurs, the cache generates an invalidation on the bus and marks the block as *exclusive*.

3.2.5 An Example Protocol

A snooping coherence protocol is usually implemented by incorporating a finite state controller in each core. This controller responds to requests from the processor in the core and from the bus (or other broadcast medium), changing the state of the selected cache block, as well as using the bus to access data or to invalidate it.

Example:

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Write invalidate Coherence Protocol for Write-Back Cache

The following finite-state transition diagram for a single cache block using

- a write invalidation protocol and
- a write-back cache
  Each cache block is in one state:
- Shared: block can be read
- Exclusive: Cache has only copy: its writeable and dirty ▪ Invalid: block contains no data.

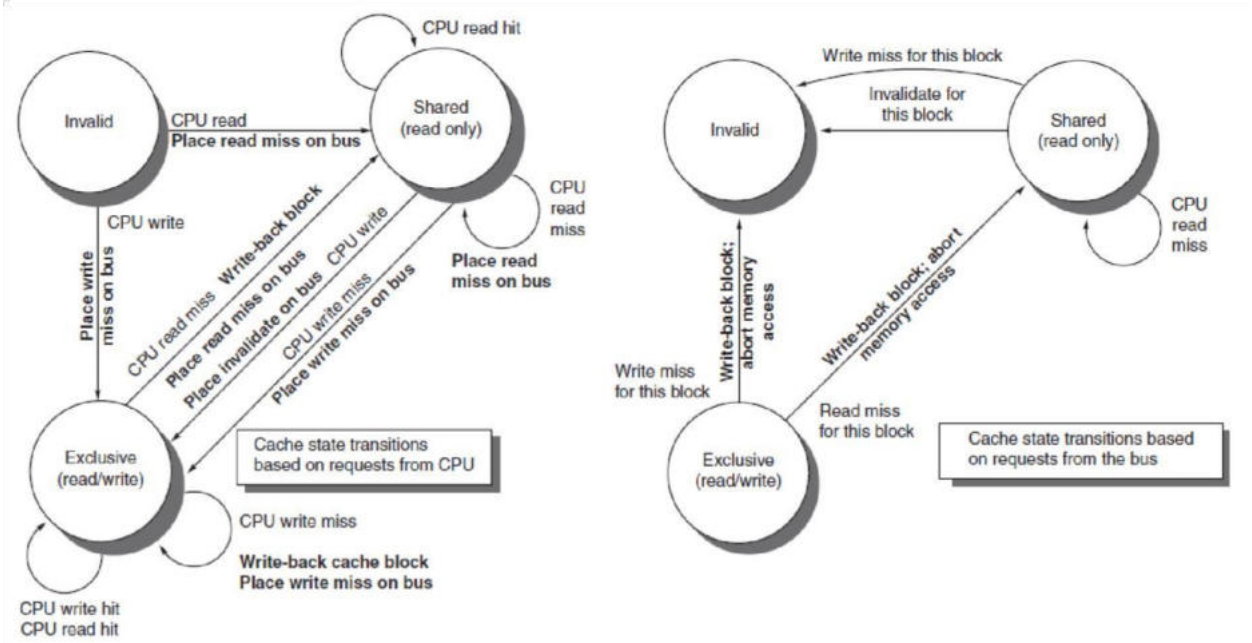


Figure 3.3 Write invalidate, cache coherence protocol for a Write-Back Cache

- The left side of the diagram shows state transitions based on actions of the processor associated
- with this cache;
- The right side shows transitions based on operations on the bus.
- A read miss in the exclusive or shared state and a write miss in the exclusive state occur when the address requested by the processor does not match the address in the local cache block. Such a miss is a standard cache replacement miss.
- An attempt to write a block in the shared state generates an invalidate.
- In actual implementations, these two sets of state diagrams are combined. In practice, there are many subtle variations on invalidate protocols, including the introduction of the exclusive unmodified state, as to whether a processor or memory provides data on a miss.
  - The protocol assumes that operations are *atomic*

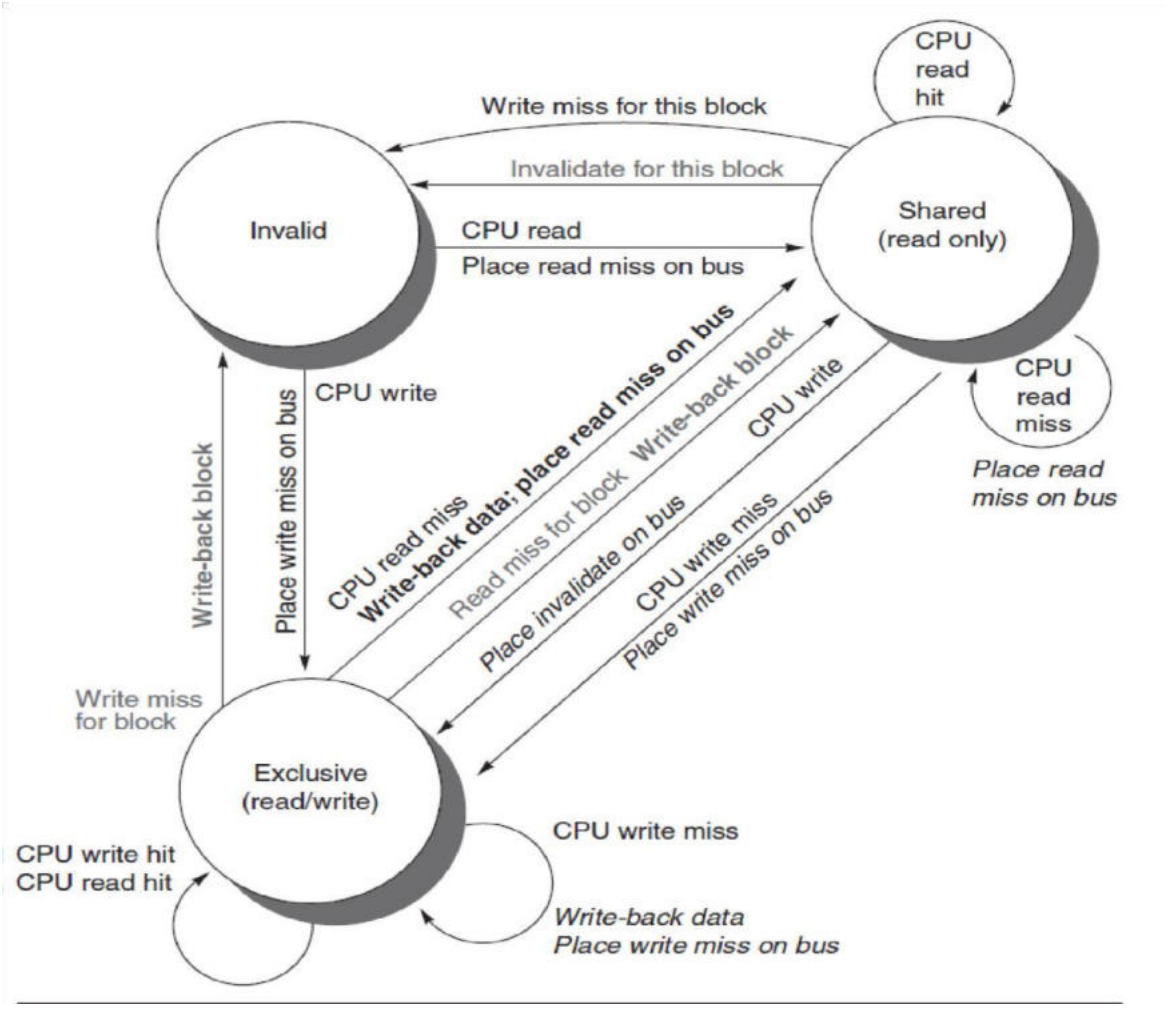


Figure 3.4 Cache coherence state diagram

With multicore processors, the coherence among the processor cores is all implemented on chip, using either a snooping or simple central directory protocol. A multiprocessor built with multiple multicore chips will have a distributed memory architecture and will need an interchip coherency mechanism above and beyond the one within the chip. In most cases, some form of directory scheme is used.

3.3 Distributed Shared – Memory Architectures

Snooping protocol is not so efficient on most DSM machines. Snooping requires a broadcast mechanism, which is easy to do on a bus. Most DSM systems don't have a bus more complex system interconnect so broadcast becomes a much more expensive operation.

Distributed Shared memory architectures is a multiprocessor in which memory is distributed among the nodes and all nodes are interconnected by a network.

Directory Protocol is an alternative coherence protocol to the snooping protocol. A directory keeps state of every block that may be cached. Information in the directory includes which caches (or collections of caches) have copies of the block, whether it is dirty, and so on.

The solution of a single directory used in a multicore is not scalable, even though it avoids broadcast. The directory must be distributed, but the distribution must be done in a way that the coherence protocol knows where to find the directory information for any cached block of memory. The obvious solution is to distribute the directory along with the memory, so that different coherence requests can go to different directories, 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, together with the maintenance of information that says what other nodes may be caching the block, 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.

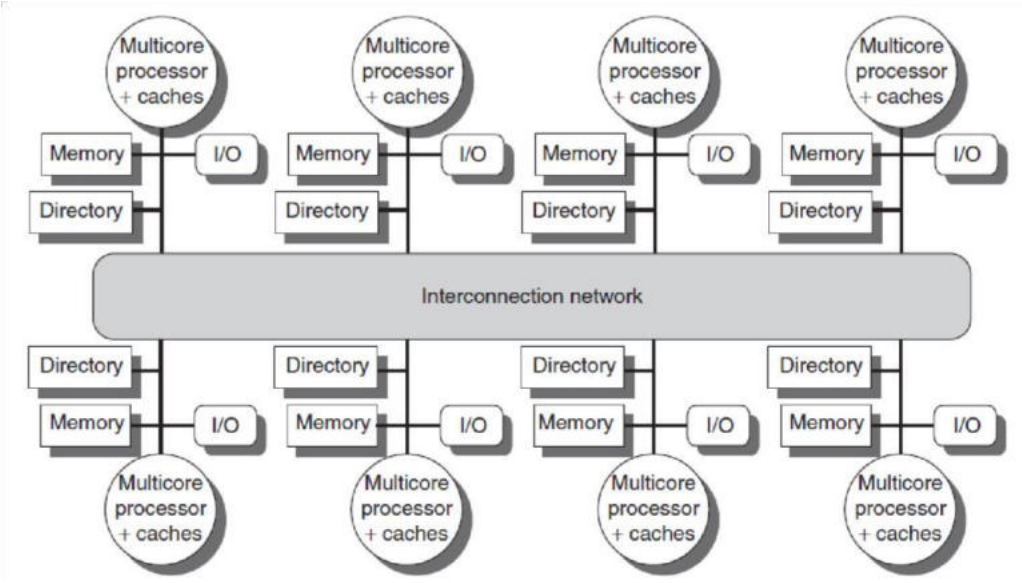


Figure 3.5 Distributed-memory multiprocessor

3.3.1 Directory-Based Cache Coherence Protocols: The Basics

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

- Handling a read miss and
- Handling a write to a shared, clean cache block

To implement these operations, a directory must track the state of each cache block. The states are:

- *Shared*—One or more nodes have the block cached, and the value in memory is up to date (as well as in all the caches).
- *Uncached*—No node has a copy of the cache block.
- *Modified*—Exactly one node 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 potentially shared memory block, we must track which nodes have copies of that block, since those copies 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 chip (which is likely a multicore) 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.

The processes of invalidating and locating an exclusive copy of a data item are different, since they both involve communication between the requesting node and the directory and between the directory and one or more remote nodes. In a snooping protocol, these two steps are combined through the use of a broadcast to all the nodes.

Types of messages:

local node – The local node is the node where a request originates **home node** – is the node where the memory location and the directory entry of an address reside. The local node may also be the home node. **remote node** – is the node that has a copy of a cache block, whether exclusive shared. A remote node may be the same as either local or home node

The table below lists the types of messages, source, destination, contents and function

Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
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.

P = requesting node number, A = requested address, and D = data contents

Data value write-backs occur for two reasons: when a block is replaced in a cache and must be written back to its home memory, and also in reply to fetch or fetch/invalidate messages from the home. Writing back the data value whenever the block becomes shared simplifies the number of states in the protocol, since any dirty block must be exclusive and any shared block is always available in the home memory. **3.3.2 An Example Directory Protocol**

The following figure shows the state diagram for an individual cache block in a directory based system and structure as the transition diagram for an individual cache.

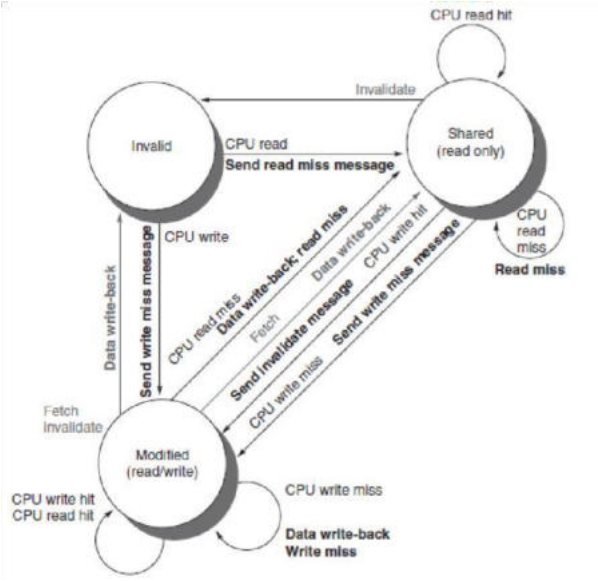


Figure 3.6 State transition diagram in a directory-based system (Individual Cache block)

The states are identical, and the stimulus is almost identical. The write miss operation, which

was broadcast on the bus (or other network) in the snooping scheme, is replaced by the data fetch and invalidates operations that are selectively sent by the directory controller. We assume that an attempt to write a shared block as a miss as in the snooping controller; in practice this transaction can be treated as ownership or upgrade request. This can deliver ownership without requiring the cache block to be fetched.

Directory operation

In a directory-based protocol, the directory implements the other half of the coherence protocol.

A message sent to a directory causes two different types of actions

- Updates of the directory state
- Sending additional messages to satisfy the request The directory receives three different requests:
  - Read miss
  - Write miss
  - Data Write-Back

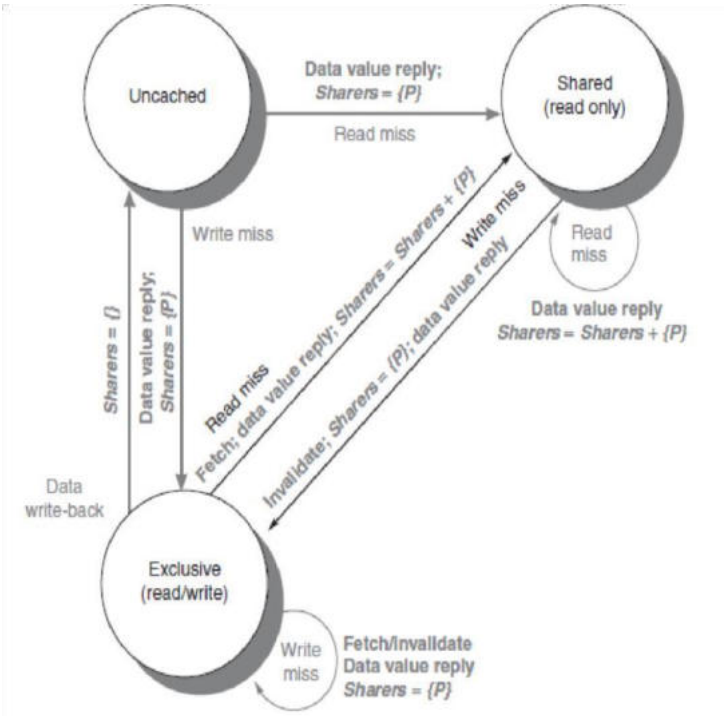


Figure 3.7 State Transition diagram in a Directory based System (A Directory)

Uncached state

When the block is uncached state, the directory can receive only two kinds of messages. ***Read miss*** – The requesting processor is sent the requested data from memory and the requestor is made the only sharing node. The state of the block is made shared.

Write miss – The requesting processor is sent the value and becomes the sharing node. The block is made exclusive to indicate that the only valid copy is cached. Sharers indicate the identity of the owner.

Shared state

When the block is in the shared state the memory value is up-to-date, so the same two requests can occur:

Read miss – Requesting processor is sent back the data from memory & requesting processor is added to the sharing set.

Write miss – Requesting processor is sent the value. All processors in the set sharers are sent invalidate messages & sharers are set to identify of requesting processor. The state of the block is made exclusive.

Exclusive State

When the block is in the exclusive state the current value of the block is held in the cache of the processor identified by the set sharers, so there are three possible directory requests: ***Read miss*** – The owner processor is sent a data fetch message, which causes the state of the block in the owners cache to transition to share and causes the owner to send the data to the dictionary, where it is written to memory and sent back to the requesting processor. The identity of the requesting processor is added to the set sharers, which still contains the identity of the processor that was the owner.

Data write-back – The owner processor is replacing the block and therefore must write it back. This write-back makes the memory copy up to date, the block is now uncached and the sharer set is empty

Write miss – The block has a new owner. A message is sent ot the old owner causing the cache to invalidate the block and send the value to the directory, from which it is sent to the requesting

processor, which becomes the new owner. Sharers are set to the identity of the new owner and the state of the block remains exclusive.

Performance Issues:

When read only is replaced, it does not explicitly notify the directory when a clean block is replaced in the cache. This is fine as the cache simply ignores invalidate messages for blocks that are not currently cached. The only problem is that it will cause the directory to send out a few unnecessary messages.

Deciding the order of access in a distributed memory system is much harder because, without a shared bus, it is impossible to tell which writes came first. It is also not feasible to stall all accesses until a write operation is completed. This can be handled by making all writes to be atomic, but doing so slows down the system greatly.

3.4 Cache Coherence Issues

In a multiprocessor system, data inconsistency may occur among adjacent levels or within the same level of the memory hierarchy. For example, the cache and the main memory may have inconsistent copies of the same object.

As multiple processors operate in parallel, and independently multiple caches may possess different copies of the same memory block, this creates cache coherence problem. Cache coherence schemes help to avoid this problem by maintaining a uniform state for each cached block of data. Cache inconsistencies are caused by data sharing, process migration or I/O.

3.4.1 Inconsistency in Data sharing

The cache inconsistency problem occurs only when multiple private caches are used. In general, three sources of the problem are identified: sharing of writable data, process migration and I/O activity. Figure below illustrates the problem caused by first two sources. Consider a multiprocessor with two processors, each using a private cache, and both sharing the main memory.

Let X be an element of shared data which has been referenced by two processors, P1 and P2. In the beginning, three copies of X are consistent. If the processor P1 writes a new data X1 into the cache, by using **write-through policy**, the same copy will be written immediately into the shared

memory. In this case, inconsistency occurs between cache memory and the main memory. When a **write-back policy** is used, the main memory will be updated when the modified data in the cache is replaced or invalidated.

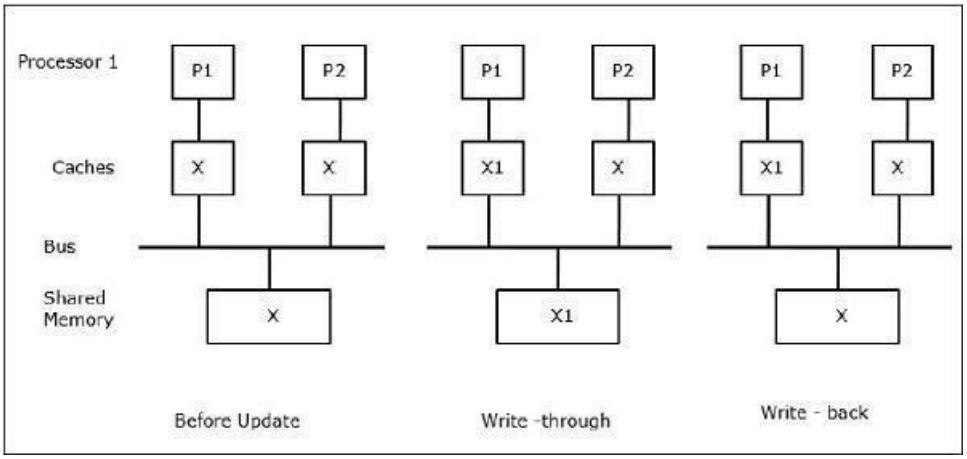


Figure 3.8

In general, there are three sources of inconsistency problem −

1. Sharing of writable data
2. Process migration
3. I/O activity

1. Sharing of writable data

When two processors (P1 and P2) have same data element (X) in their local caches and one process (P1) writes to the data element (X), as the caches are write-through local cache of P1, the main memory is also updated. Now when P2 tries to read data element (X), it does not find X because the data element in the cache of P2 has become outdated**.**

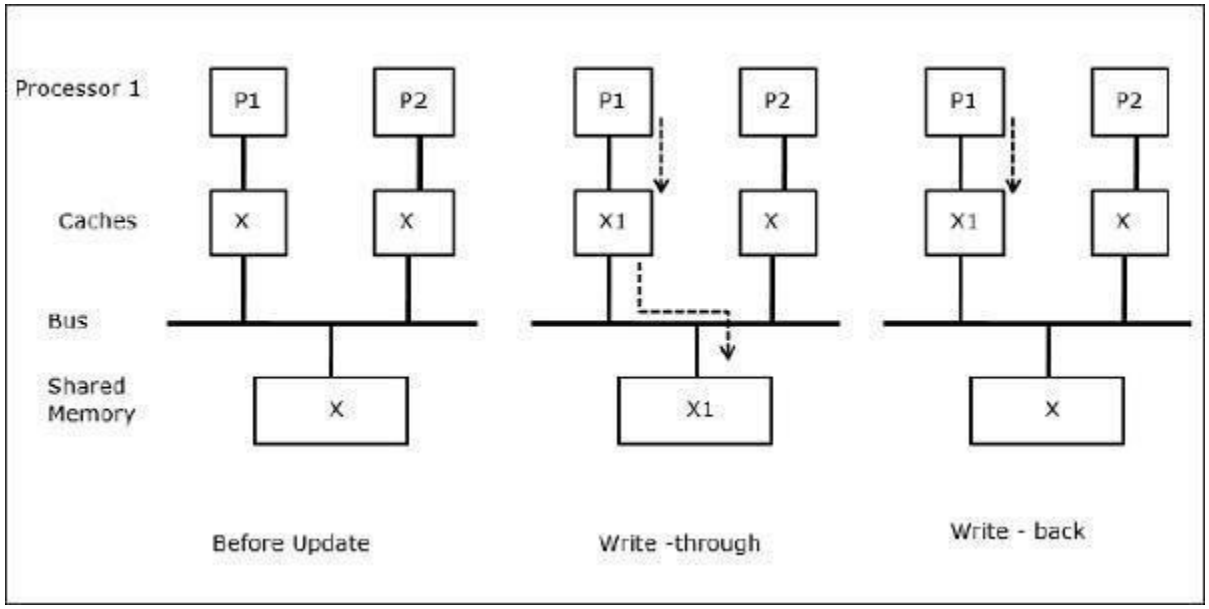


Figure 3.9

2. Process migration

In the first stage, cache of P1 has data element X, whereas P2 does not have anything. A process on P2 first writes on X and then migrates to P1. Now, the process starts reading data element X, but as the processor P1 has outdated data the process cannot read it. So, a process on P1 writes to the data element X and then migrates to P2. After migration, a process on P2 starts reading the data element X but it finds an outdated version of X in the main memory.

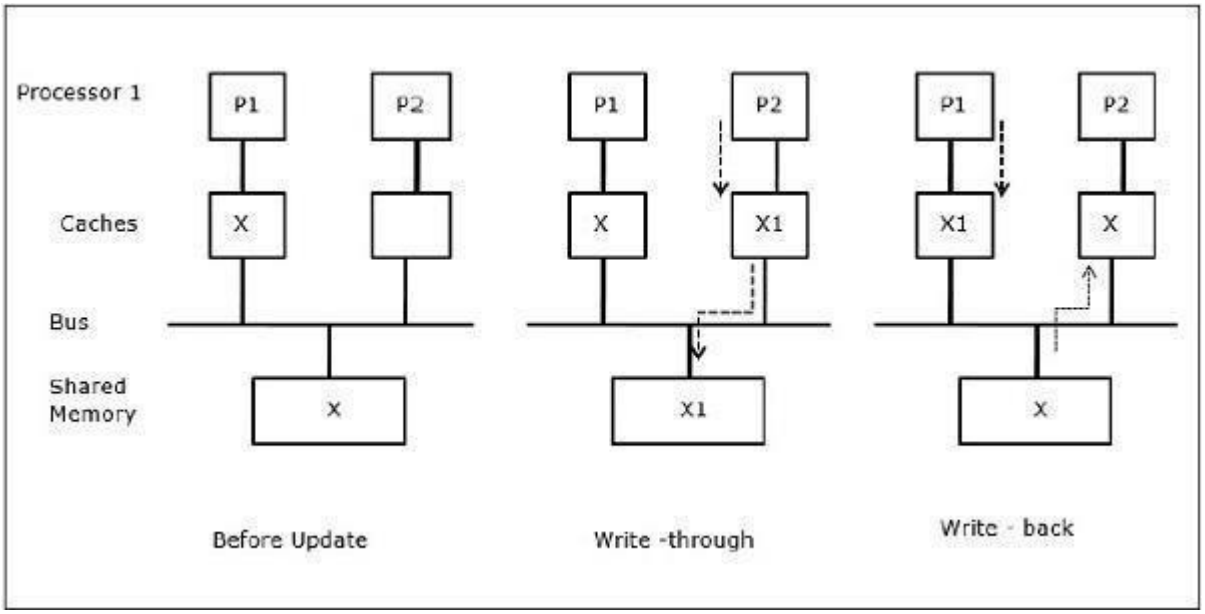


Figure 3.10

3. I/O activity

As illustrated in the figure, an I/O device is added to the bus in a two-processor multiprocessor architecture. In the beginning, both the caches contain the data element X. When the I/O device receives a new element X, it stores the new element directly in the main memory. Now, when either P1 or P2 (assume P1) tries to read element X it gets an outdated copy. So, P1 writes to element X. Now, if I/O device tries to transmit X it gets an outdated copy.

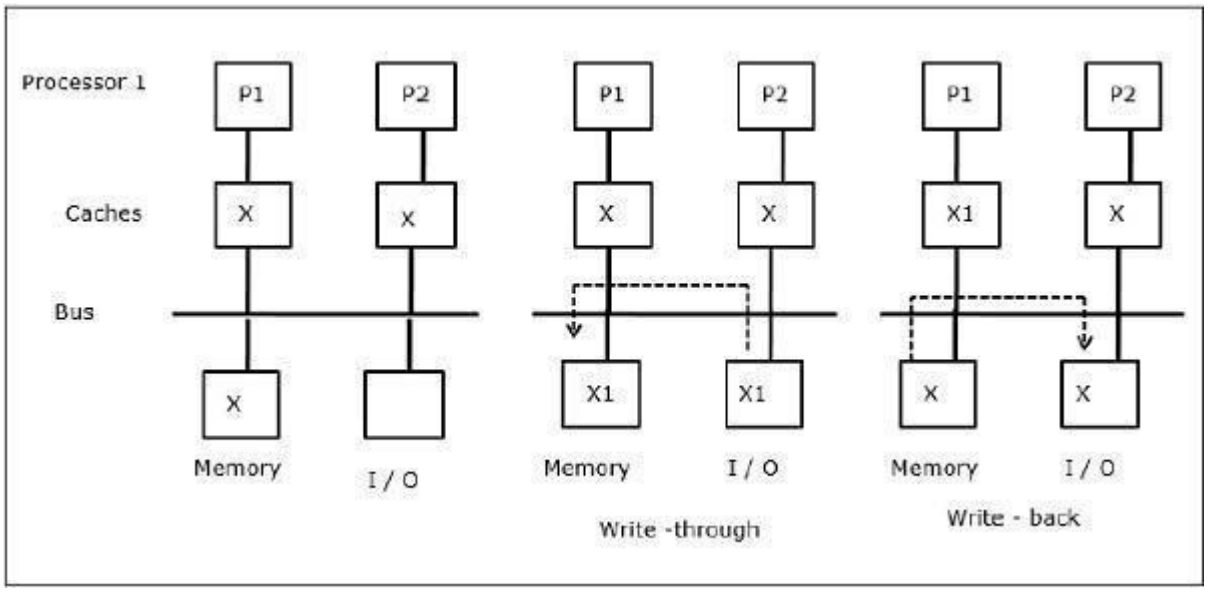


Figure 3.11

3.5 Performance Issues

The overall cache performance is a combination of the behavior of uniprocessor *cache miss traffic* and the *traffic caused by communication*, which results in invalidations and subsequent cache misses.

Factors affecting the two components of miss rate:

1. Changing the processor count,
2. cache size, and
3. block size.

3C's Classification of the uniprocessor miss rate: a)Capacity

b)compulsory, and

c) conflict

The misses that arise from interprocessor communication, which are often called *coherence misses,* can be broken into two separate sources.

- The first source is the so-called *true sharing misses* that arise from the communication of data through the cache coherence mechanism. They directly arise from the sharing of data among processors.
- The second effect, called *false sharing,* arises from the use of an invalidation based coherence algorithm with a single valid bit per cache block.

Example:

Assume that words x1 and x2 are in the same cache block, which is in the shared state in the caches of P1 and P2. Assuming the following sequence of events identify each miss as a true sharing miss or a false sharing miss

Time	P1	P2
1	Write x1	
2		Read x2
3	Write x1	
4		Write x2
5	Read x2	

Here are the classifications by time step:

1. This event is a true sharing miss, since x1 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 x1 in P1, but that value of x1 is not used in P2.
3. This event is a false sharing miss, since the block containing x1 is marked shared due to the read in P2, but P2 did not read x1. The cache block containing x1 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.

3.5.1 Performance measurements of the Commercial workload

The following model are taken for the performance measurements of the commercial workload:

Alpha server 4100

The Alphaserver 4100 has four processors. Each processor has a three level cache hierearchy.

1. L1 consists of a pair of 8 KB direct mapped on-chip caches
    a. One for instruction
    b. One for data

The block size is 32 bytes. The data cache is write through to L2, using write buffer.

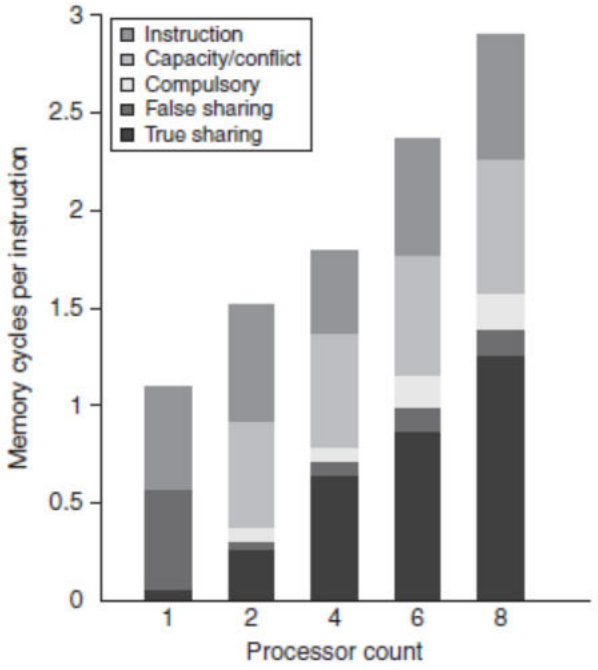
2. L2 is a 96 KB on-chip unified threeway set associative cache with a 32 byte block size using write back.

3. L3 is an off-chip, combined, direct mapped 2 MB cache with 64 byte blocks also using write back

The Latency for access to L2 = 7 cycles, L3 = 21 cycles, Main memory = 80 clock cycles. The execution time breaks down into instruction execution, cache access time, memory system access time, other stalls.

The performance of the DSS and Altavista workloads is reasonable. The performance of the OLTP workload is very poor. The impact of the OLTP benchmark of L3 cache size, processor count, block size are focused because the OLTP workload from the memory system with large numbers of expensive L3 misses.

The effect of increasing the cache size reduces the large number of conflict misses using two-way set-associative caches.



The contribution to memory access cycles increases as processor count increases primarily due to increased true sharing. The compulsory misses slightly increase since each processor must now handle more compulsory misses.

3.5.2 Performance measurements of the Multiprogramming and OS workload

The workload used is two independent copies of the compile phases of the Andrew benchmark, a benchmark that emulates a software development environment. The compile phase consists of a parallel version of the Unix ―make‖ command executed using eight processors. The workload runs for 5.24 seconds on eight processors, creating 203 processes and performing 787 disk requests on three different file systems. The workload is run with 128 MB of memory, and no paging activity takes place.

Three distinct phases:

Compile – substantial compute activity

Install the object files in a binary – dominated by I/O

Remove the object files – dominated by I/O and 2 PEs are active

For the workload measurements, we assume the following memory and I/O systems:

- *Level 1 instruction cache*—32 KB, two-way set associative with a 64-byte block, 1 clock cycle hit time.

- *Level 1 data cache*—32 KB, two-way set associative with a 32-byte block, 1 clock cycle hit time. We vary the L1 data cache to examine its effect on cache behavior.
- *Level 2 cache*—1 MB unified, two-way set associative with a 128-byte block, 10 clock cycle hit time.
- *Main memory*—Single memory on a bus with an access time of 100 clock cycles.
- *Disk system*—Fixed-access latency of 3 ms (less than normal to reduce idle time).

The execution time breaks down for the eight processors using the parameters just listed.

Execution time is broken down into four components:

1. *Idle*—Execution in the kernel mode idle loop
2. *User*—Execution in user code
3. *Synchronization*—Execution or waiting for synchronization variables
4. *Kernel*—Execution in the OS that is neither idle nor in synchronization access

Data Miss rate vs Data cache size

The misses can be broken into three significant classes

- Compulsory misses represent the first access to this block by this processor and are significant in this workload.
- Coherence misses represent misses due to invalidations
- Normal capacity misses include misses caused by interference between OS and user process and between multiple user processes.

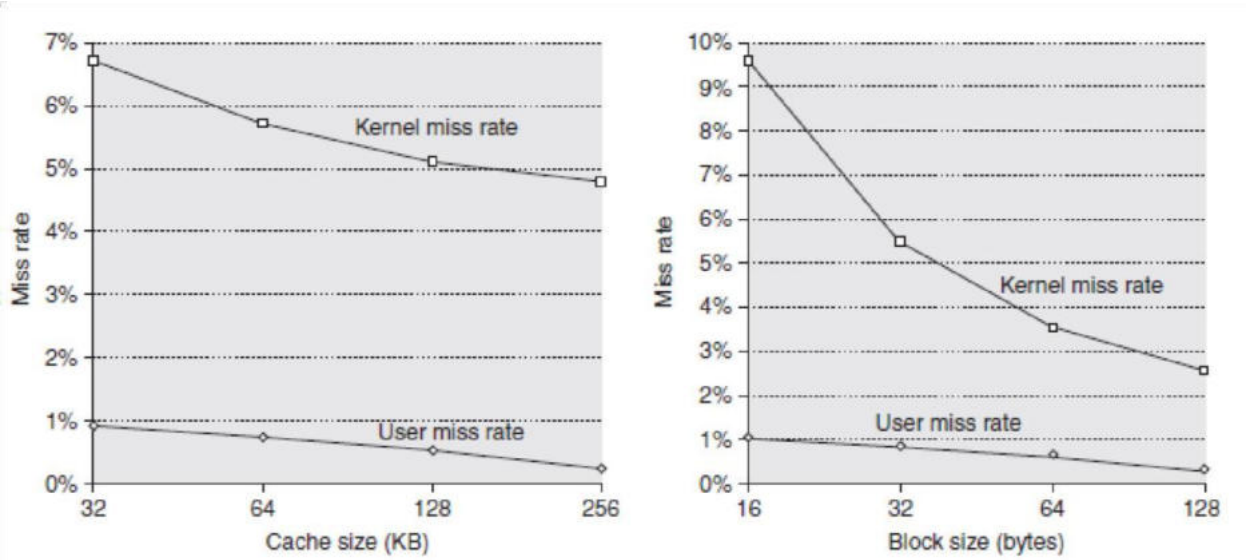
The reasons in which the behavior of the operating system is more complex than the user process are:

- The kernel initializes all pages before allocating to a user
- The kernel shares data and has a nontrivial coherence miss rate.

Miss rate vs Block size for kernel

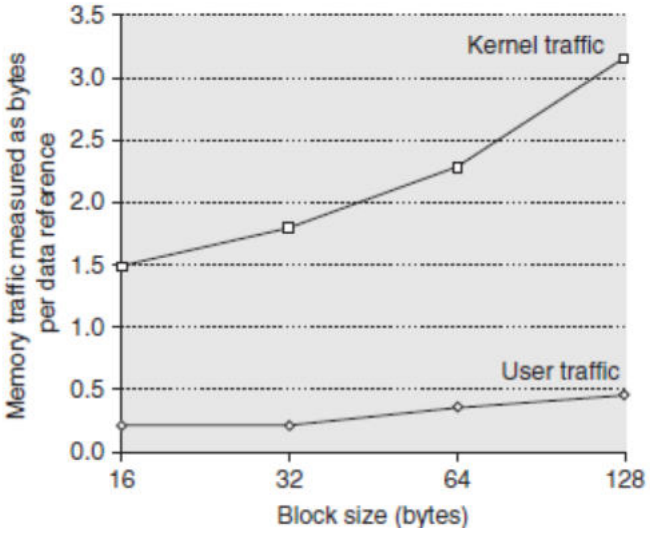
Compulsory and capacity miss can be reduced with larger block sizes Largest improvement is reduction of compulsory miss rate.

Absence of large increases in the coherence miss rate as block size is increased means that false sharing effects are insignificant.



Memory traffic measured as Bytes per Data Reference

The number of bytes needed per data reference grows as block size is increased for both kernel and user components.



The operating system is more demanding user for the memory system for the multiprogrammed workload.

3.6 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.

Synchronization is needed to know when it is safe for different processes to use shared data.

3.6.1 Basic Hardware Primitives

Hardware primitives are the basic building blocks that are used to build user-level synchronization operations. The ability to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to atomically read and modify a memory location.

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 exchan*ge**, which interchanges a value in a register for a value in memory.

0 ⇒ synchronization variable is free
1 ⇒synchronization variable is locked and 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.

A single atomic memory operation introduces some challenges:

- Requires both a memory read and a write in a single, uninterruptible instruction.
- Complicates the implementations of coherence, since the hardware cannot allow any other operations between read and write and it must not deadlock

An alternative is to have a pair of instructions where the second instruction returns a value from which it can be deduced whether the pair of instructions was executed as if the instructions were atomic. The pair of instructions includes the following instructions which are used in sequence:

- a special load called a load linked or load locked
- a special store called a store conditional The store condition fails :
- if the contents of the memory location specified by the load linked are changed before the store conditional to the same address occurs
- if the processor does a context switch between the two instructions

The Load linked returns the initial value and store conditional returns 1 if it was successful or 0 otherwise. The following sequence implement s an atomic exchange on the memory location specified by the contents of R1

```
try:   MOV R3,R4 ; mov exchange value
LL R2,0(R1); load linked
       SC R3,0(R1); store conditional
       BEQZR3,try ; branch store fails
       MOV R4,R2 ; put load value in R4
```

The contents of R4 and the memory location specified by R1 have been atomically exchanged. Any time a processor intervenes and modified the value in memory between the LL and SC instructions, the Sc returns 0in R3, causing the code sequence to try again.

An advantage of the load linked/store conditional mechanism is that it can be used to build other synchronization primitives. For example, here is an atomic fetch-and-increment:

```
try:    LL R2,0(R1) ; load linked
        DADDUIR3,R2,#1 ; increment
        SC R3,0(R1) ; store conditional
        BEQZ R3,try ; branch store fails
```

These instructions are typically implemented by keeping track of the address specified in the LL instruction in a register, often called the *link register.* If an interrupt occurs, or if the cache block matching the address in the link register is invalidated (for example, by another SC), the link register is cleared. The SC instruction simply checks that its address matches that in the link register. If so, the SC succeeds; otherwise, it fails. Since the store conditional will fail after either another attempted store to the load linked address or any exception, care must be taken in choosing what instructions are inserted between the two instructions.

3.6.2 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, 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.

We should modify our spin lock procedure so that it spins by doing reads on a local copy of the lock until it successfully sees that the lock is available. Then it attempts to acquire the lock by doing a swap operation. A processor first reads the lock variable to test its state.

A processor keeps reading and testing until the value of the read indicates that the lock is unlocked. The processor then races against all other processes that were similarly ―spin waiting‖ to see who can lock the variable first. All processes use a swap instruction that reads the old value and stores a 1 into the lock variable.

3.7 Models of Memory Consistency

A **memory consistency model** or memory model specifies how memory behaves with respect to read and write operations from multiple processors. **Cache coherence** ensures that multiple processors see a consistent view of memory. **Consistency** means when a processor must see a value that has been updated by another processor.

Cache coherence ensures that multiple processors see a consistent view of memory. It does not answer the question of *how* consistent the view of memory must be. By ―how consistent‖ we mean, when must a processor see a value that has been updated by another processor? Since processors communicate through shared variables (used both for data values and for synchronization), the question boils down to this: In what order must a processor observe the data writes of another processor? Since the only way to ―observe the writes of another processor‖ is through reads, the question becomes, What properties must be enforced among reads and writes to different locations by different processors?

Consider the following example:

P1: A = 0; P2: B = 0;

....

```
        A = 1;              B = 1;
L1:     if (B == 0) ...     L2: if (A == 0)...
```

Assume that the processes are running on different processors, and that locations A and B are originally cached by both processors with the initial value of 0. The most straightforward model for memory consistency is called ***sequential consistency***.

Sequential consistency requires that the result of any execution be the same as if the memory accesses executed by each processor were kept in order and the accesses among different processors were arbitrarily interleaved.

Sequential consistency eliminates the possibility of some no obvious execution in the previous example because the assignments must be completed before the if statements are initiated.

The simplest way to implement sequential consistency is to require a processor to delay the completion of any memory access until all the invalidations caused by that access are completed. Of course, it is equally effective to delay the next memory access until the previous one is completed.

Remember that memory consistency involves operations among different variables: the two accesses that must be ordered are actually to different memory locations. In our example, we must delay the read of A or B (A == 0 or B == 0) until the previous write has completed (B = 1 or A = 1). Under sequential consistency, we cannot, for example, simply place the write in a write buffer and continue with the read.

3.7.1 The Programmer's View

The challenge is to develop a programming model that is simple to explain and yet allows a high-performance implementation. One such programming model that allows us to have a more efficient implementations to assume that programs are ***synchronized***.

A program is synchronized if all access to shared data are ordered by synchronization operations.

A data reference is ordered by a synchronization operation if, in every possible execution, a write of a variable by one processor and an access (either a read or a write) of that variable by another processor are separated by a pair of synchronization executed before the access by the second processor.

Cases where variables maybe updated without ordering by synchronization are called ***data races*** because the execution outcome depends on the relative speed of the processors, and like races in hardware design, the outcome is unpredictable, which leads to another name for synchronized programs: ***data-race-free***.

It is a broadly accepted observation that most programs are synchronized. This observation is true primarily because if the accesses were unsynchronized, the behavior of the program would likely be unpredictable because the speed of execution would determine which processor won a data race and thus affect the results of the program.

Even with sequential consistency, reasoning about such programs is very difficult. Programmers could attempt to guarantee ordering by constructing their own synchronization mechanisms, but this is extremely tricky, can lead to buggy programs, and may not be supported architecturally, meaning that they may not work in future generations of the multiprocessor. Instead, almost all programmers will choose to use synchronization libraries that are correct and optimized for the multiprocessor and the type of synchronization.

Finally, the use of standard synchronization primitives ensures that even if the architecture implements a more relaxed consistency model than sequential consistency, a synchronized program will behave as if the hardware implemented sequential consistency operations, one executed after the write by the writing processor and one

3.7.2 Relaxed Consistency Models: The Basics

The key idea in relaxed consistency models is to allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering, so that a synchronized program behaves as if the processor were sequentially consistent. There are a variety of relaxed models that are classified according to what read and write orderings they relax.

We specify the orderings by a set of rules of the form X→Y, meaning that operation X must complete before operation Y is done.

Sequential consistency requires maintaining all four possible orderings: R→W, R→R, W→R, and W→W. The relaxed models are defined by which of these four sets of orderings they relax:

1. Relaxing the W→R ordering yields a model known as ***total store ordering*** **or** ***processor consistency*****.** Because this ordering retains ordering among writes, many programs that operate under sequential consistency operate under this model, without additional synchronization
2. Relaxing the W→W ordering yields a model known as ***partial store order***.
3. Relaxing the R→W and R→R orderings yields a variety of models including ***weak ordering***, the PowerPC consistency model, and ***release consistency***, depending on the details of the ordering restrictions and how synchronization operations enforce ordering.

By relaxing these orderings, the processor can possibly obtain significant performance advantages. There are, however, many complexities in describing relaxed consistency models, including the advantages and complexities of relaxing different orders, defining precisely what it means for a write to complete, and deciding when processors can see values that the processor itself has written.

3.8 Case Study-Interconnection Networks–Buses, Crossbar and Multi-stage Interconnection Networks

An interconnection network is used for exchanging data between two processors in a multistage network. There is a hierarchy of media to interconnect computers that varies in cost, performance, and reliability. Network media have another figure of merit, the maximum distance between nodes.

With the rapid advances in technology, it is now possible to build a system consisting of hundred or thousands of processors. The general-purpose parallel/distributed computer systems are divided into two categories: **multiprocessors and multicomputer**. The main difference between them lies in the level at which interactions between the processors occur.

Multiprocessors: A multiprocessor must permit all processors to directly share the main memory. All the processors address a common main memory space. Multiprocessors can be further divided as tightly coupled and loosely coupled. In a ***tightly coupled system***, the main memory is situated at a central location so that the access time from any processor to the memory is the same. In addition to this central memory (also called main memory, shared memory, global memory etc.), each processor might consist of some local memory or cache. In a ***loosely coupled system***, the main memory is partitioned and attached to the processors, although the processors share the same

memory address space. A processor can directly address a remote memory, but the access time is much higher compared to a local memory access.

Multicomputers: In a multicomputer, each processor has its own memory space and sharing between the processors occurs at a higher level as with complete file or data set. A processor cannot directly access another processor‟s local memory. The interaction between the processors relies on message passing between the source and destination processors (nodes). The message passes over a link that directly connects two nodes and might have to pass through several such nodes in a store and forward manner before it reaches its destination. Therefore, each interaction involves a lot of communication overhead, and only those applications that need less interprocessor communication are well suited to multicomputers.

3.8.1 Interconnection Networks (INs)

The performance of a multiprocessor rests primarily on the design of its interconnection network. An interconnection network is a complex connection of switches and links permitting processors in a multiprocessor system to communicate among themselves or with memory modules or I/O devices. It is the path, in which the data must travel in order to access memory in a shared memory computer or to communicate with other processes in a distributed memory environment or to use any I/O devices.

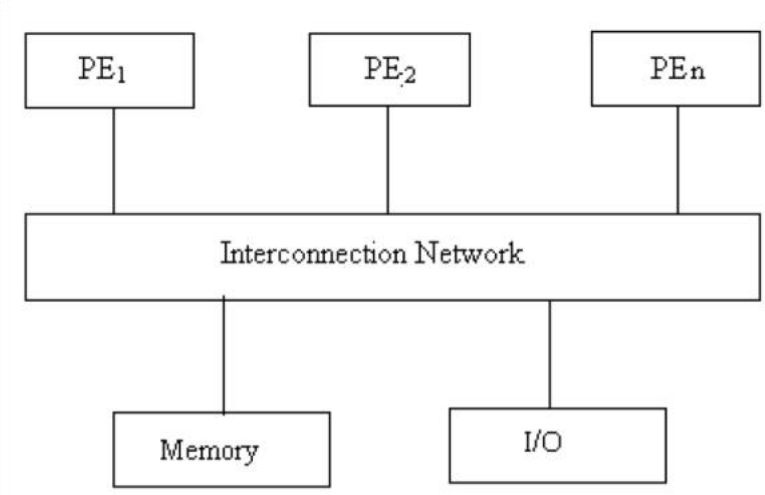


Figure 3.12 An Interconnection Network

3.8.1.1 Design Dimensions of Interconnection Networks

According to Feng (1981), the key design dimensions for interconnection networks are: a) Switching Methodology

b) Operational Mode

c) Control Strategy

d) Network Topology.

a) Switching Methodology

Two major switching methodologies are:

Circuit Switching: In circuit switching an end-to-end physical path is actually established between a source and a destination. This path exists as long as the data transmission is not complete. Circuit switching is suitable for bulk transmission of data.

Packet Switching: In packet switching data is divided into packets and routed through the interconnection network without establishing a physical connection. Packet switching is more efficient for short messages.

b) Operational Modes Operational modes can either be synchronous or asynchronous or a combination of the two.

Synchronous: Synchronous control techniques are characterized by a global clock, which broadcasts clock signals to all devices in a system so that the entire system operates in lock-step fashion.

This mode is useful for either a data manipulating function or for a data instruction broadcast.

Asynchronous: Asynchronous techniques do not utilize a single global clock, rather distribute the control function throughout the system, often utilizing many individual clocks for timing. Asynchronous communication is needed for multi-processing in which connection requests are issued dynamically

c) Control Strategy A typical interconnection network consists of a number of switching elements and interconnection links. Interconnection functions are realized by proper control of switching elements. The control strategy can be of two types:

Centralized Control Strategy: In this strategy, a centralized controller manages all the switching elements.

Distributed Control Strategy: In this, individual switching element manages all control settings.

d) Network Topology A network can be represented by a graph in which nodes indicate switches and edges represent communication links. Topology is the pattern in which the individual switches are interconnected to other elements such as processors, memories and other switching elements. The topologies as shown in Figure 1.3 can be categorized into two groups:

1. Static: In static topology, links between two processors are passive and have dedicated buses, which cannot be reconfigured for direct connection with other processors. Static networks are generally used in message-passing architectures. The following network topologies are commonly used:

- Ring Network
- Star Connected Network
- Completely Connected Network
- Tree Network
- Mesh Network
- Hypercube Network.

2. Dynamic: On the other hand, links in dynamic topology are reconfigured by setting networks active switching elements. Dynamic interconnection networks implement one of the following interconnection techniques:

- Crossbar Network
- Bus based Network
- Multistage Interconnection Network.

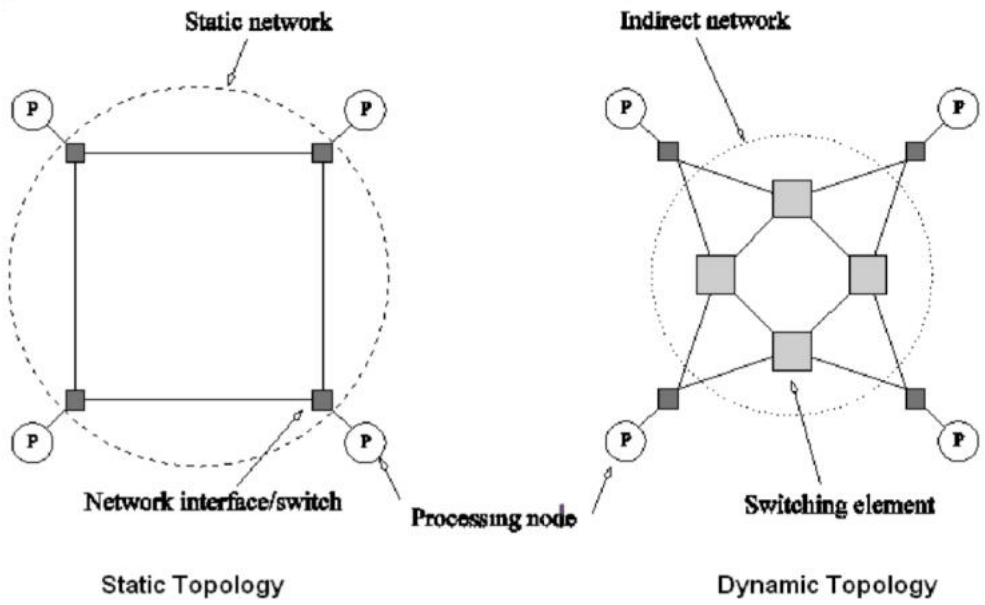


Figure 3.13 Types of Network Topology

Dynamic Topologies

1. Crossbar Networks: The crossbar makes a connection from a given vertical bus to the appropriate horizontal bus and allows traffic to flow along this path. In crossbar network, the other horizontal or vertical buses can be supporting a flow of traffic at the same time. For example, if each horizontal bus needs to talk to a separate vertical bus then they all can move data at the same time. This completely eliminates the single-shared-resource limitation of the system bus. The crossbar is a preferable approach for high performance multiprocessors. Figure

3.14 shows a crossbar network.

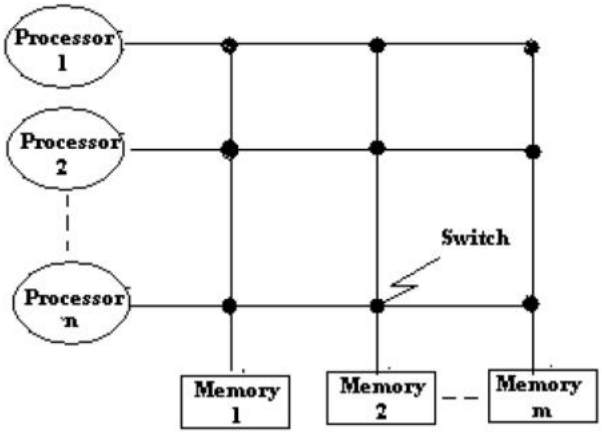


Figure 3.15 A Crossbar Network

2. Bus Network: Bus topology uses a common backbone to connect all the network devices in a network in a linear shape as shown in Figure 3.16. Some of the simplest and earliest parallel machines used bus topology. All processors access a common bus for exchanging data. The distance between any two nodes is O(1) in a bus. The bus also provides a convenient broadcast media. However, the bandwidth of the shared bus is a major bottleneck.

A bus is highly non-scalable architecture, because only one processor can communicate on the bus at a time. A bus network design offers minimum bandwidth. It is highly inefficient and unreliable because of a single bus, the failure of which will make it unusable. Buses are commonly used in shared memory parallel computers to communicate read and write requests to a shared global memory. A single bus organization is simple and inexpensive. Addition of more processors or memory modules increases the bus contention, thus decreases the bus throughput.

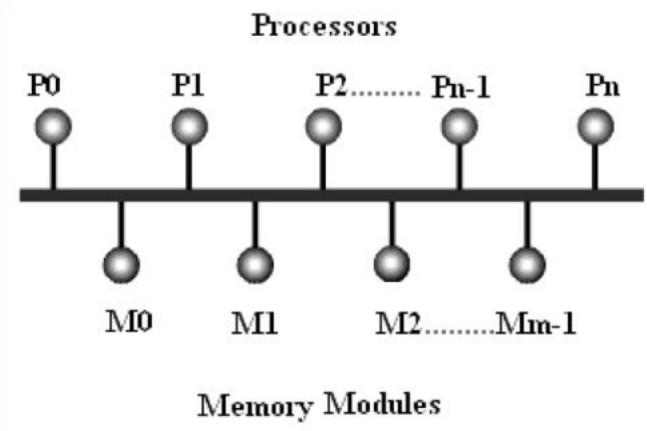


Figure 3.16 A Bus Network

3. Multistage Interconnection Networks: Multistage Interconnection Networks (MINs) consist of more than one stage of small interconnection elements called switching elements and links interconnecting them. MINs are used in multiprocessing systems to provide cost-effective, highbandwidth communication between processors and/or memory modules. A MIN normally connects N inputs to N outputs and is referred as an N×N MIN. The parameter N is called the size of the network.

Figure 3.17 illustrates structure of a multistage interconnection network, which is representative of a general class of networks. This figure shows the connection between p inputs and b outputs via n number of stages. A multistage interconnection network is actually a compromise between

crossbar and shared bus networks (Bhuyan et al., 1989). Table 1.1 describes the properties of various types of multiprocessor interconnection networks.

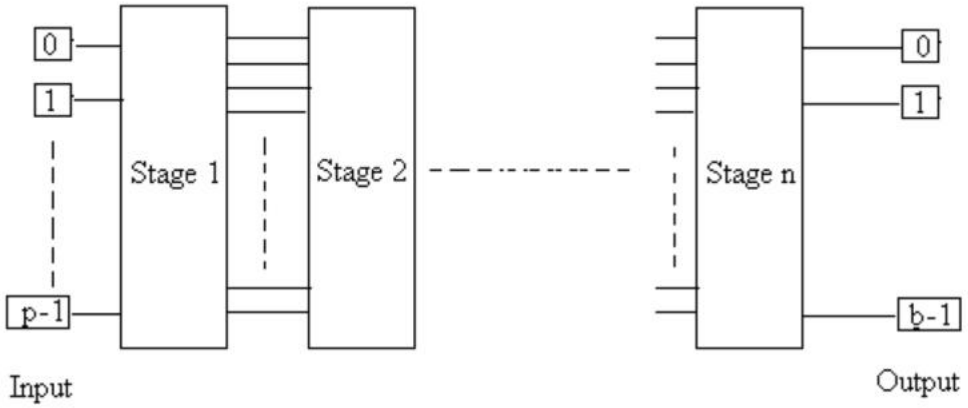


Figure 3.17 A Multistage Interconnection Network (MIN)

Property	Bus	Crossbar	Multistage
Speed	Low	High	High
Cost	Low	High	Moderate
Reliability	Low	High	High
Configurability	High	Low	Moderate
Complexity	Low	High	Moderate

Table 1.1 : Properties of Different Networks

3.8.2 Classification of Multistage Interconnection Networks

Multistage interconnection networks can be classified according to different categories.

The main classification categories are: a)

According to number of paths

b) According to number of switches

c) According to control

d) According to availability of path.

a) Classification according to Number of Paths According to number of paths, MINs can be classified as unique and multi-path networks, as described below:

Unique Path Networks: These networks provide a unique path between every source and destination. The failure of any switching element along the path disconnects some sourcedestination pairs, so adversely affecting the capabilities of existing network. These networks are not reliable for a large multiprocessor system, as they cannot tolerate even a single fault. In case of multiple requests, a source destination connection may be blocked by a previously established connection, thus providing a poor performance.

Multi-path Networks: These networks provide more than one path between a given source and destination. In case of failure of any switching element in the path, the request is routed through some alternative path. Unique path multistage interconnection networks can be made multi-path by adding redundancy in the form of extra switching elements, links, stages, sub networks, by increasing the size of switching elements or by using multiple networks.

Multi-path multistage interconnection networks can be either static or dynamic. For static networks, if a fault is encountered, then data has to backtrack to the source or some fixed point to select an alternate path in the network. The implementation of backtracking is expensive in terms of the hardware. In dynamic networks, if a fault is encountered in a particular stage, a switching element in preceding stage will re-route data through an alternate available path.

b) Classification according to Number of Switches MINs can be classified according to number of switches as regular and irregular networks, as described below:

Regular Networks: Regular multistage interconnection networks have an equal number of switching elements at each stage (Mittal et al., 1995), as a result they may impose equal time delay to all the requests passing through them.

Irregular Networks: Irregular multistage interconnection networks have unequal number of switching elements at each stage (Mittal et al., 1995). For a given source destination pair multiple paths with different path lengths are available.

c) Classification according to Control

Flip Controlled Networks: Flip controlled multistage interconnection networks have a common control signal for switching in various switching elements at a given stage. These networks are less complicated due to lesser number of control signals but have lesser bandwidth.

Distributed Control Networks: Distributed control multistage interconnection networks have a separate control signal for every switching element. These have higher bandwidth due to selection of source destination pair at a given time and are quite complex.

d) According to Availability of Path

Blocking Networks: A network is called a blocking network if simultaneous connections of more than one terminal may result in conflict in use of network communication links. Example of blocking network is Omega network.

Non-blocking Networks: A network is called non-blocking if it is possible to route data from any source to any destination, in presence of other established source-destination routes, provided no two sources have same destination. In other words, a network that can handle all possible connections without blocking is called non- blocking network.

UNIT IV MULTICORE ARCHITECTURES 9

Homogeneous and Heterogeneous Multi-core Architectures – Intel Multicore Architectures –SUN CMP architecture – IBM Cell Architecture. Introduction to Warehouse-scale computers- Architectures- Physical Infrastructure and Costs- Cloud Computing –Case Study- Google Warehouse-Scale Computer.

4.1 Homogeneous and Heterogeneous Multi-core Architectures

4.1.1 Introduction

Multiprocessor systems contain multiple CPUs that are not on the same chip. Multi-core is a design in which single physical processor contains the core logic of more than one processor. These processors are called cores. When these cores are integrated onto a single integrated circuit die then they are known as a Chip multiprocessor or CMP or Dual core , if these cores are built onto multiple dies in a single chip package.

Multi-core processors as the name implies contain two or more distinct cores in the same physical package. The following figure shows multi-core processors have multiple execution cores on a single chip.

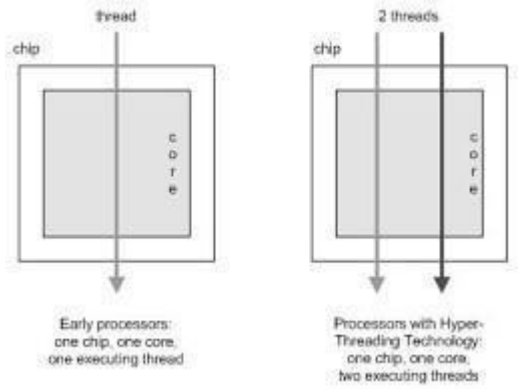


Figure 4.1 Multi-core processors have multiple execution cores on a single chip In this design, each core has its own execution pipeline. And each core has the resources required to run without blocking resources needed by the other software threads. While the example in Figure 2 shows a two-core design, there is no inherent limitation in the number of cores that can be placed on a single chip. Mainframe processors today use more than two cores, so there is precedent for this kind of development.

The multi-core design enables two or more cores to run at somewhat slower speeds and at much lower temperatures. The combined throughput of these cores delivers processing power greater than the maximum available today on single-core processors and at a much lower level of power consumption. In this way Intel increases the capabilities of server platforms as predicted by

Moore's law while the technology no longer pushes the outer limits of physical constraints. A multi-core processor implements multiprocessing in a single physical package. Cores in a multi-core device may be coupled together tightly coupled or loosely coupled. For example, cores may or may not share caches, and they implement message passing or shared memory inter-core communication methods. Common topologies to interconnect cores include: bus, ring, 2-dimensional mesh, and crossbar.

All cores are identical in symmetric multi-core systems and they are not identical in asymmetric multi-core systems.

Why multi-core?

Better performance – By incorporating multiple cores, each core is able to run at a lower frequency, dividing among them the power normally given to a single core. The result is a big performance increase over a single core processor.

Power consumption and Heat generation – Caused from the advance of CPU clock speed. **Save the room of motherboard** – Two single core in one die. By the way we can use this room more efficiently.

Simplicity – We need additional systems to control the several single cores.
Economical efficiency – A dual core is much cheaper than two single cores.

4.1.2 Homogenous Multi-core architecture

A Symmetric or Homogenous multi-core is one that has multiple cores on a single chip, and all those cores are identical. Like AMP, an SMP system has multiple CPUs, but in this case each has exactly the same architecture – i.e. it must be a homogeneous multi-core design. CPUs share memory space (or at least some of it). Normally a single OS is used that runs on all the CPUs, dividing work between them - another significant difference from AMP. Some kind of communication facility between the CPUs is provided this is normally through shared memory,

but accessed through the API of the OS. Typically, SMP is used when an embedded application simply needs more CPU power to manage its workload, in much the way that multicore CPUs are used in desktop computers.

Example :

The Intel core 2 is an example of a symmetric multi-core processor. The core 2 can have either 2 cores on chip (—Core2 Duo‖) or 4 cores on chip (—Core 2 Quad‖). Each core in core 2 chip is symmetrical and can function independent of one another. It requires a mixture of scheduling software and hardware to form tasks out to each core.

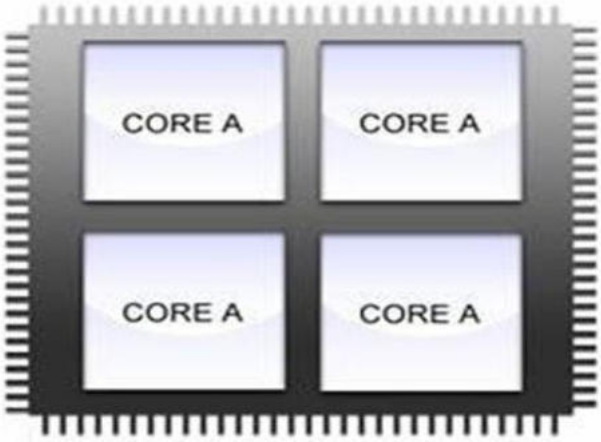


Figure 4.2 Multiple cores of same type are implemented in one CPU

Every Single core has the same architecture and same capabilities. Each core has the same capabilities so it requires that there is an arbitration unit to give each core a specific task. Software that uses techniques like multithreading makes the best use of multi-core processor like the Intel core 2.

4.1.2 Heterogeneous Multi-core architecture

An asymmetric multi-core processor is one that has multiple cores on a single chip, but those cores might be different designs. For instance, there could be 2 general purpose cores and 2 vector cores on a single chip.

Example:

The best example of heterogeneous multi-core is the cell processor which is a heterogeneous multi-core processor comprised of control intensive processor and compute intensive SIMD processor cores, each with its own distinguishing features. In an asymmetric multi-core processor, the chip has multiple cores on board, but the cores might be different designs. Each core will have different capabilities.

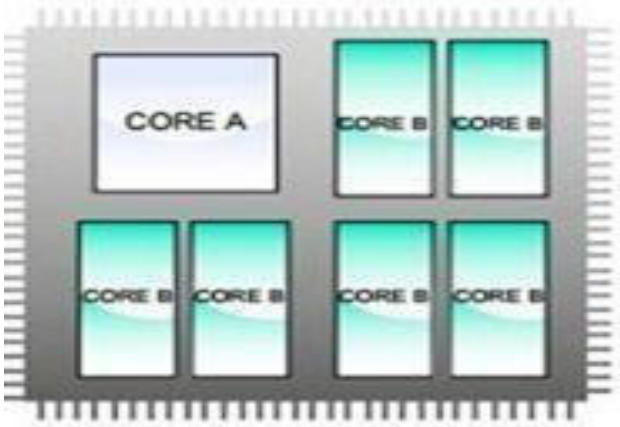


Figure 4.3 Heterogeneous multi-core processor

The figure above shows the structural layout of a configuration of heterogeneous multicore processor. In this, the core A differs from all other common cores (core B) which mean that all the cores within the same processor are not symmetric. Hence heterogeneous processor is also called Asymmetrical multi-core processor.

Heterogeneous system improve performance and efficiency by exposing to programmers architectural features, such as low latency software-controlled memories and inter-processor interconnect, that are either hidden or unavailable in GPPs. The software that executes on these accelerators often bears little resemblance to its CPU counterparts.

Heterogeneous systems integrate computational resources that features instruction sets and functionality that significantly differ from general-purpose processors(GPPs). These systems fall into two categories:

1. an ISA-based tightly-coupled model
2. a device driver based loosely-coupled execution model

Examples of the tightly coupled approach are the SIMD SSE extensions to the X86 ISA[50,56]. Examples of loosely coupled model are systems that use the graphics processing unit (GPU) for general-purpose computing(GPGPU).

A heterogeneous multi-core would consist of the following components in its architecture.

- A highly specialized application specific processor cores with their own unique instruction sets.
- The primary core would be generic processor for running basic programs.
- A graphics core to handle all graphics and display algorithms
- A protocol offload engine – POE which acts as a communication processor core that will handle all communication tasks.
- A math-oriented processor core to handle all the heavy calculations that some of the applications programs may have.

Both approaches will to benefit the industry on the whole. Both are aimed at pursuing a similar goal; increased throughput through parallel hardware, which requires changes to existing software to take advantage of this increased computing power. Many server applications focus primarily on throughput per unit cost and unit power. Desktop users are more interested in single applications or a few applications at the same time. A multicore processor for desktop users will probably be focused on a smaller number of larger, yet powerful, cores with better single-thread performance.

Differences between Homogeneous and heterogeneous multi-core architecture

Homogeneous	**Heterogeneous**
multiple CPUs	multiple CPUs
each of which may be a different architecture [but can be the same]	each of which has the same architecture
each has its own address space	CPUs share memory space [or, at least, some of it]
each may or may not run an OS [and the OSes need not be the same]	normally an OS is used and this is a single instance that runs on all the CPUs, dividing work between them

some kind of communication facility between the CPUs is provided	some kind of communication facility between the CPUs is provided [and this is normally shared memory]

4.2 Intel Multicore Architectures

A multi-core processor is composed of two or more independent cores. It is an integrated circuit which has two or more individual processors. Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor) or onto multiple dies in a single chip package.

Intel multi-core technology provides new levels of energy efficient performance, enabled by advanced parallel processing and 2nd generation high K+ metal gate 32nm technology. Incorporating multiple processor execution cores in a single package delivering full parallel execution of multiple threads, Intel multi-core technology enables higher levels of performance, dividing the power typically required by a higher frequency single core processor with equivalent performance.

Intel® Turbo Boost Technology – With this technology, depending upon the work load the power and frequency can be distributed across many cores or focused on fewer cores through cores through the use of power gating – an internal power management unit that can ―turn off‖ power to entire cores. An Intel® Turbo Boost technology enabled processor can boost performance on the core or cores still active by raising the clock speed.

Intel® Hyper-Threading Technology – can additionally increase efficiency by executing two software threads on a single core to get more done with less power than if separate cores were active. Together these technologies provide a break through experience in performance and responsiveness for notebook and desktop PCs, workstations and servers.

Intel® Core™ Micro architecture – It is the foundation for new Intel® architecture based desktop, mobile and mainstream sever multi-core processors. This state-of-art, multi-core optimized micro architecture delivers a number of new innovative features that have set new standards for energy-sufficient performance.

Intel® Advanced Smart Cache – It is multi-core optimized cache that improves performance and efficiency by increasing the probability that each core of a multi-core processor can access data from a higher performance, more efficient cache subsystem. Intel® Advanced Smart Cache works by sharing the level 2(L2) cache among cores so that data is stored in one place that each core can access. Sharing L2 cache enables each core to dynamically use up to 100 percent of available L2 cache, thus optimizing cache resources.

Intel® Smart memory Access – improves system performance by optimizing available data bandwidth from the memory subsystem and hiding the latency of memory accesses through two techniques: a new capability called memory disambiguation and an instruction pointer-based prefetcher that fetches memory contents before they are requested.

4.3 SUN CMP architecture

Chip multiprocessors also called multi-core processors are now the only way to build high-performance microprocessors for a range 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 instructions stream using conventional superscalar instruction issue techniques.

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 functional 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)

The following diagram 4.4 shows the single core computers and single core chip:

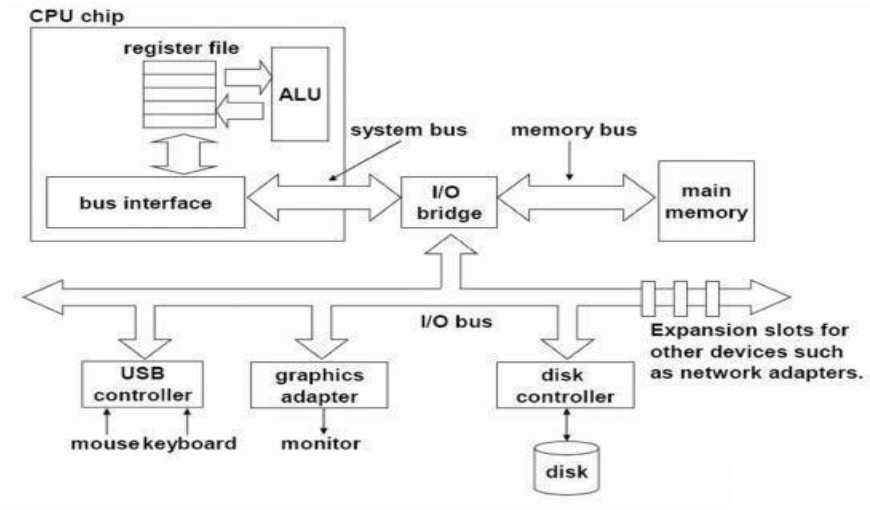


Figure 4.4 Single core computer

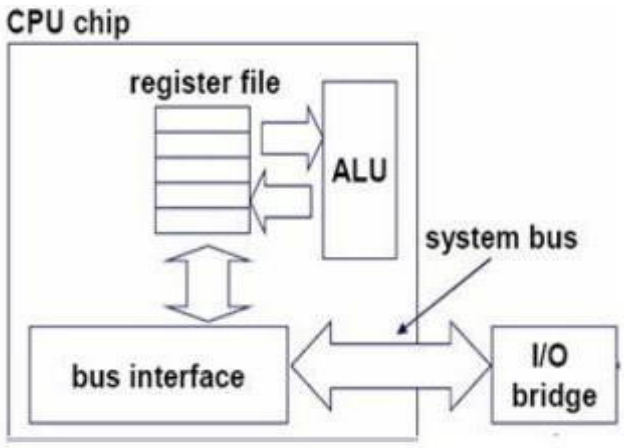


Figure 4.5 Single core CPU chip

CMP stands for Chip Multiprocessing. CMP instantiates multiple processor ―cores‖ on a single die.

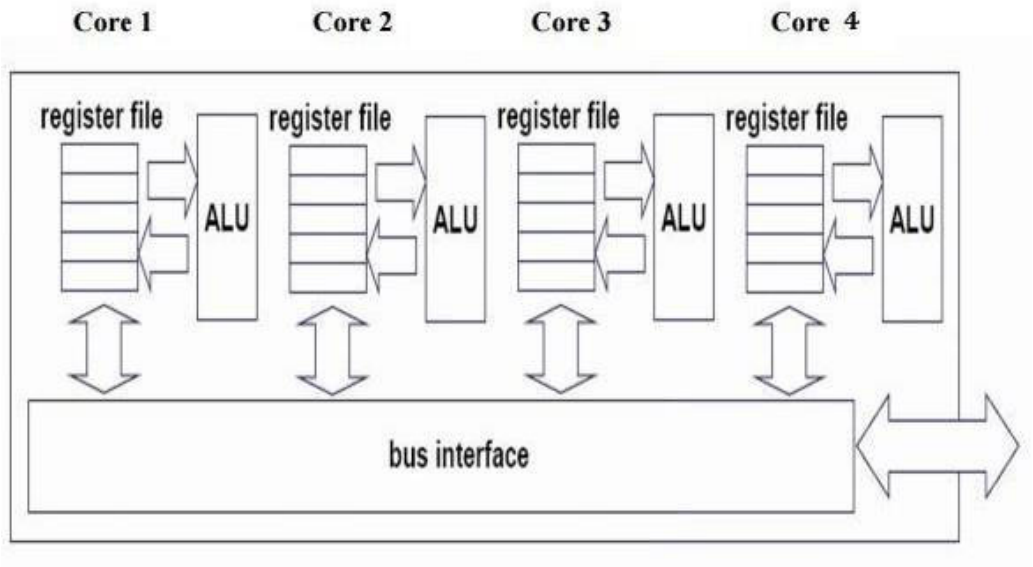


Figure 4.6 Multi-core CPU chip

Chip Multithreading

Chip that provide multiprocessing, called Multi-core Architecture. There are many cores per processor.

HMT: Hardware Multithreading. There are many threads per core.

CMT: Chip Multithreading has support to CMP Technology and HMT Technology. There are many cores with multiple threads per processor

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

Why CMT processor?

For a single thread

- Memory is the bottleneck to improving performance
- Commercial server workloads exhibit poor memory locality
- Only a modest throughput speedup is possible by reducing compute time
- Conventional single- thread processors optimized for ILP have low utilization.

With many threads

- It's possible to find something to execute every cycle

- Significant throughput speedups are possible
- Processor utilization is much higher

The SUN Niagara processor is an example of a CMT approach. Sun Microsystems offers the new UltraSPARC T1 processor, codenamed "Niagara". The processor has 8 parallel cores, which are 4 times multi-threaded, i.e each core can run 4 processes quasi simultaneously. Each core has an Integer pipeline (length 6) with is shared between the 4 threads of the core, to hide memory latency.

Since there is only one floating point unit (FPU) for all cores, the UltraSparc T1 processor is suited for programs with few or none floating point operations, like web servers or databases. Nevertheless the UltraSparc T1 is binary compatible to an UltraSparc IV CPU. Each of the 8 processor cores has its own level 1 instruction and data caches. All cores have access to one common level 2 cache of 3 MB and to the common main memory. Thus the processor behaves like an UMA (uniform memory archtiecture) system.

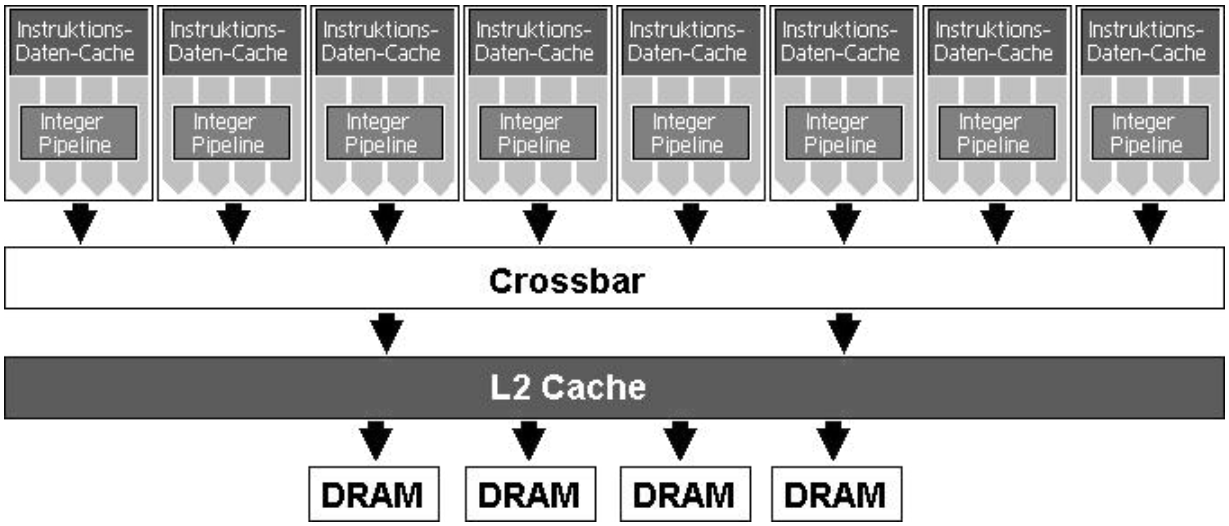


Figure 4.7 SUN Niagara Processor

Advantages:

- CMP has the major benefit of easily being PIN compatible with the previous generation, so a CMP processor could fit into an existing computer and multiply the number of processor available in the pack.

- Low inter-processor communication latency between the cores in a CMP helps make a much wider range of applications feasible candidates for parallel execution multi-chip multiprocessors
- With the exact multiplication of a complete processor core comes another advantage, the validation of the processor increases moderately.

Limitations:

- Larger dies are more expensive to fabricate and to package.
- Also a chip with more transistors dissipates more heat and hence power dissipation is a major concern.
- Limited parallelism

4.4 IBM Cell Architecture

The IBM cell processor is a heterogeneous multi-core processor comprised of intensive processor and compute-intensive SIMD processor cores, each with its own distinguishing features.

IBM in cooperation with fellow industry giants Sony and Toshiba, recently announced a new processor, called the cell

The cell processor is vastly different from conventional processor inside. This tiny chip contains a powerful 64-bit Dual threaded IBM PowerPC core but also eight proprietary Synergistic Processing Elements (SPEs), essentially eight more highly specialized minicomputers on the same die.

4.4.1 Basic Architecture of IBM's Cell

IBM introduced the overall architecture of the Cell processor. The following figure gives an idea of the organization of a cell chip. It shows the different parts and how the cell is communicating with the rest of the workload, through its memory controllers and input/output controllers.

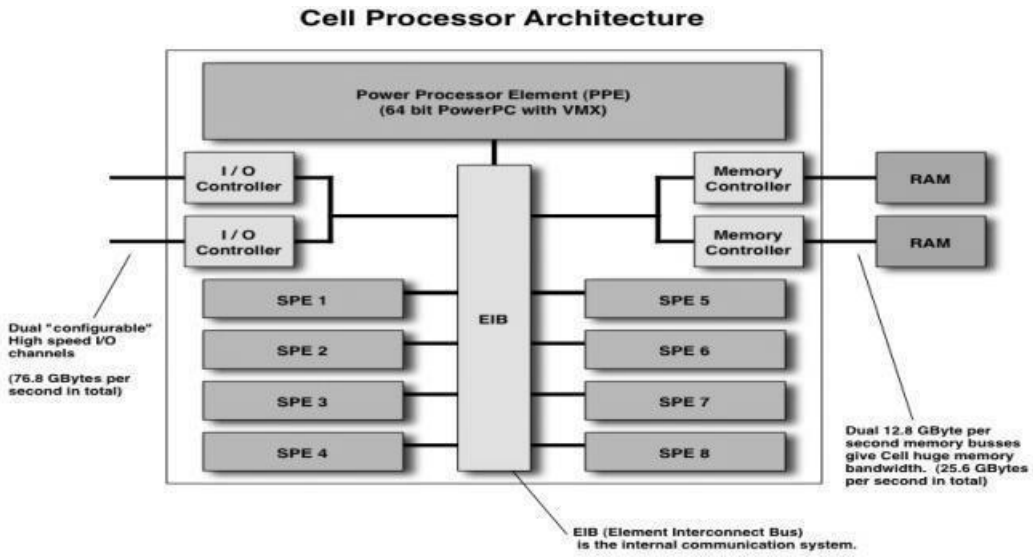


Figure 4.8 Overview of the architecture of a cell chip

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. Any program written for a common PowerPC 970 processor should run on the Cell Broadband Engine. The following figure 4.9 shows a very simplified view of the PPE.

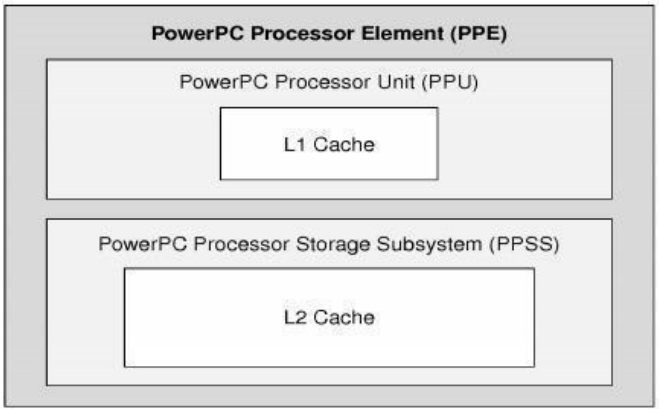


Figure 4.9 PPE block diagram

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).

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.

The PPU is hyper-threaded and supports 2 simultaneous threads. All architectural and special purpose registers are duplicated, except those dealing with system-level resources, such as logical partitions, memory-control and thread-control. On the non-architecural side most resources are shared, except for very small resources, or those offering critical performance improvements with multi-threaded applications.

Power Processor Storage Subsystem (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)

1. The Synergistic Processor Unit (SPU)

This SPU 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.

From the SPU point of view, the LS is nothing but unprotected and untranslated storage. There is no virtual memory on the SPU. Thus, there is no control of a possible stack overflow: IT is the duty of the programmer to be sure memory is used correctly. If some illegal access is made, the SPU simple stops. This event can be detected by the PPU.

2. 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.

4.4.2 Bus design and communication among the Cell

The Element Interconnect Bus

The way all elements are connected together is achieved by the Element Interconnect Bus (EIB). 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.

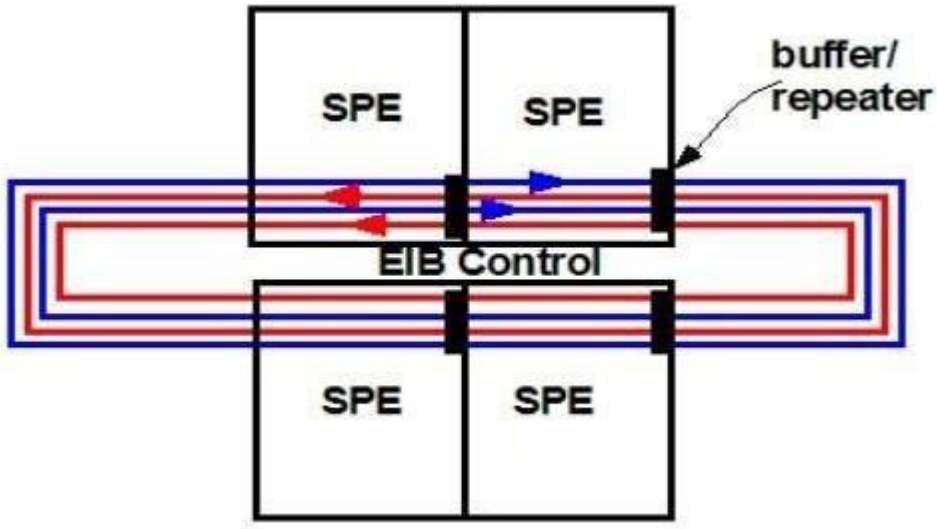


Figure 4.10 Elements Interconnect Bus

Input/output interfaces

A processor is linked to the memory bus, itself being linked to some bridges. The interfaces to the outside world are

1. 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.

2. 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.

Cell performance Characteristics

Cells performance is about an order of magnitude better than GPP for media and other applications that can take advantage if is SIMD capability.

- Performance if its simple PPE is comparable to a traditional GPP performance.
- Its each SPE is able to perform mostly the same as or better than a GPP with SIMD running frequency.
- Key performance advantage comes from its 8 decoupled SPE SIMD engines with dedicated resources including large register files and DMA channels.

Cell can conver a wide range of application space with its capabilities in

- Floating point operations
- Integer operations
- Data streaming / throughput support
- Real time support

4.5 Introduction to Warehouse-scale computers

A cluster is a collection of desktop computers or servers connected together by a local area network to act as a single larger computer. A warehouse-scale computer(WSC) is a cluster comprised of tens of thousands of servers.

The warehouse-scale computer (WSC)1 is the foundation of Internet services many people use every day: search, social networking, online maps, video sharing, online shopping, email services, and so on.

4.5.1 Important design factors for WSC

Cost-performance − Work done per dollar is critical in part because of the scale. Reducing the capital cost of a WSC by 10% could save $15M.

Energy efficiency −Power distribution costs are functionally related to power consumption. Affects power distribution and cooling. Work per joule is critical for both WSCs and servers because of the high cost of building the power and mechanical infrastructure for a warehouse of computers and for the monthly utility bills to power servers.

Dependability via redundancy − The long-running nature of Internet services means that the hardware and software in a WSC must collectively provide at least 99.99% of availability; that is, it must be down less than 1 hour per year. Redundancy is the key to dependability for both WSCs and servers. Multiple WSCs also reduce latency for services that are widely deployed. ***Network I/O*** − Server architects must provide a good network interface to the external world, and WSC architects must also. Networking is needed to keep data consistent between multiple WSCs as well as to interface to the public.

Both interactive and batch processing workloads − While you expect highly interactive workloads for services like search and social networking with millions of users, WSCs, like servers, also run massively parallel batch programs to calculate metadata useful to such services. For example, MapReduce jobs are run to convert the pages returned from crawling the Web into search indices.

There are also characteristics that are *not* shared with server architecture. They are:

Ample computational parallelism is not important

First, batch applications benefit from the large number of independent datasets that require independent processing, such as billions of Web pages from a Web crawl. This processing is *data-level parallelism* applied to data in storage instead of data in memory.

Second, interactive Internet service applications, also known as *software as a service* (*SaaS*), can benefit from millions of independent users of interactive Internet services. Reads and writes are rarely dependent in SaaS, so SaaS rarely needs to synchronize. In *request-level parallelism*, as many independent efforts can proceed in parallel naturally with little need for communication or synchronization;

Operational costs count – Power consumption is a primary, not secondary, constraint when designing system

Scale and its opportunities and problems – Can afford to build customized systems since WSC require volume purchase

4.5.2 Batch processing framework

In addition to the public-facing Internet services such as search, video sharing, and social networking that make them famous, WSCs also run batch applications, such as converting videos into new formats or creating search indexes from Web crawls.

Today, the most popular framework for batch processing in a WSC is Map-Reduce and its open-source twin Hadoop. Map first applies a programmer-supplied function to each logical input record. Map runs on thousands of computers to produce an intermediate result of key-value pairs. Reduce collects the output of those distributed tasks and collapses them using another programmer-defined function.

For example, one MapReduce program calculates the number of occurrences of every English word in a large collection of documents. Below is a simplified version of that program, which shows just the inner loop and assumes just one occurrence of all English words found in a document.

```
map(String key, String value):
// key: document name //
value: document contents for
each word w in value:

EmitIntermediate(w, "1"); // Produce list of all words reduce(String
key, Iterator values):

// key: a word
// values: a list of counts int result = 0; for each v in
values: result += ParseInt(v); // get integer from key-
value pair

Emit(AsString(result));
```

The function EmitIntermediate used in the Map function emits each word in the document and the value one. Then the Reduce function sums all the values per word for each document using ParseInt() to get the number of occurrences per word in all documents. The MapReduce runtime environment schedules map tasks and reduce task to the nodes of a WSC.

MapReduce can be thought of as a generalization of the single-instruction, multiple-data (SIMD) operation except that you pass a function to be applied to the data−that is followed by a function that is used in a reduction of the output from the Map task.

To accommodate variability in performance from thousands of computers, the MapReduce scheduler assigns new tasks based on how quickly nodes complete prior tasks. Obviously, a single slow task can hold up completion of a large MapReduce job. In a WSC, the solution to slow tasks is to provide software mechanisms to cope with such variability that is inherent at this scale. This approach is in sharp contrast to the solution for a server in a conventional datacenter, where

traditionally slow tasks mean hardware is broken and needs to be replaced or that server software needs tuning and rewriting.

Programming frameworks such as MapReduce for batch processing and externally facing SaaS such as search rely upon internal software services for their success. For example, MapReduce relies on the Google File System (GFS) to supply files to any computer, so that MapReduce tasks can be scheduled anywhere.

These internal services often make different decisions than similar software running on single servers. As an example, rather than assuming storage is reliable, such as by using RAID storage servers, these systems often make complete replicas of the data. Replicas can help with read performance as well as with availability; with proper placement, replicas can overcome many other system failures.

WSC hardware and software must cope with variability in load based on user demand and in performance and dependability due to the vagaries of hardware at this scale.

4.6 Computer Architecture of Warehouse-Scale Computers

WSC often use a hierarchy of networks for interconnection. Figure 4.11 shows one example Each 19‖ rack holds 48 1U servers connected to a rack switch. Rack switches are uplinked to switch higher in hierarchy. Thus, the bandwidth leaving the rack is 6 to 24 times smaller−48/8 to 48/2−than the bandwidth within the rack. This ratio is called *oversubscription.*

Goal is to maximize locality of communication relative to the rack.

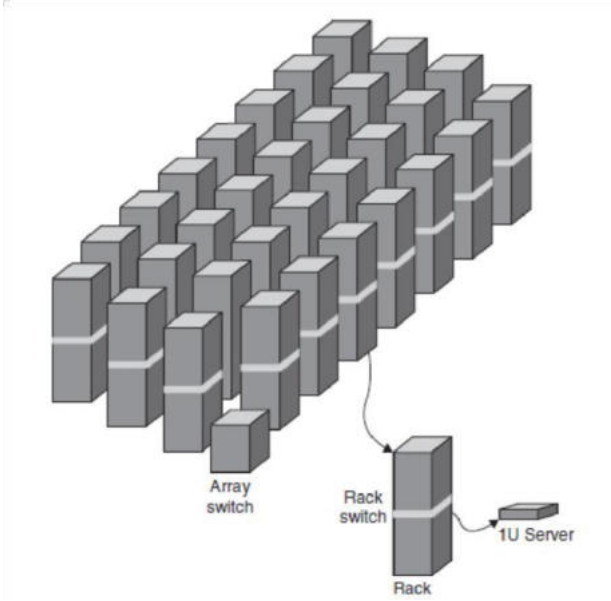


Figure 4.11 Hierarchy of switches in a WSC

Storage

A natural design is to fill a rack with servers, minus whatever space you need for the commodity Ethernet rack switch. This design leaves open the question of where the storage is placed. From a hardware construction perspective, the simplest solution would be to include disks inside the server, and rely on Ethernet connectivity for access to information on the disks of remote servers. The alternative would be to use network attached storage (NAS), perhaps over a storage network like Infiniband. The NAS solution is generally more expensive per terabyte of storage, but it provides many features, including RAID techniques to improve dependability of the storage

Google File System (GFS) uses local disks and maintains at least three replicas to improve dependability by covering not only disk failures, but also power failures to a rack or a cluster of racks by placing the replicas on different clusters.

The term cluster mean the next-sized grouping of computers, in this case about 30 racks. The term *array* means a collection of racks, preserving the original meaning of the word cluster to mean anything from a collection of networked computers within a rack to an entire warehouse full of networked computers.

Array Switch

Array Switch is the switch that connects an array of racks. It is considerably more expensive than the 48-port commodity Ethernet switch. Another reason for the high costs is that these products offer high profit margins for the companies that produce them. For example, network switches are major users of content-addressable memory chips and of field-programmable gate arrays (FPGAs), which help provide these features, but the chips themselves are expensive. While such features may be valuable for Internet settings, they are generally unused inside the datacenter.

WSC Memory Hierarchy

Servers can access DRAM and disks on other servers using a NUMA-style interface.

	Local	Rack	Array
DRAM latency (microseconds)	0.1	100	300
Disk latency (microseconds)	10,000	11,000	12,000
DRAM bandwidth (MB/sec)	20,000	100	10
Disk bandwidth (MB/sec)	200	100	10
DRAM capacity (GB)	16	1040	31,200
Disk capacity (GB)	2000	160,000	4,800,000

- Each server contains 16 GBytes of memory with a 100-nanosecond access time and transfers at 20 GBytes/sec and 2 terabytes of disk that offers a 10-millisecond access time and transfers at 200 MBytes/sec. There are two sockets per board, and they share one 1 Gbit/sec Ethernet port.
- Every pair of racks includes one rack switch and holds 80 2U servers. Networking software plus switch overhead increases the latency to DRAM to 100 microseconds and the disk access latency to 11 milliseconds. Thus, the total storage capacity of a rack is roughly 1 terabyte of DRAM and 160 terabytes of disk storage. The 1 Gbit/sec Ethernet limits the remote bandwidth to DRAM or disk within the rack to 100 MBytes/sec.
- The array switch can handle 30 racks, so storage capacity of an array goes up by a factor of 30: 30 terabytes of DRAM and 4.8 petabytes of disk. The array switch hardware and software increases latency to DRAM within an array to 500 microseconds and disk latency to 12 milliseconds.

The network overhead dramatically increases latency from local DRAM to rack DRAM and array DRAM, but both still have more than 10 times better latency than the local disk. The network collapses the difference in bandwidth between rack DRAM and rack disk and between array DRAM and array disk.

The WSC needs 20 arrays to reach 50,000 servers, so there is one more level of the networking hierarchy. Figure 4.12 shows the conventional Layer 3 routers to connect the arrays together and to the Internet.

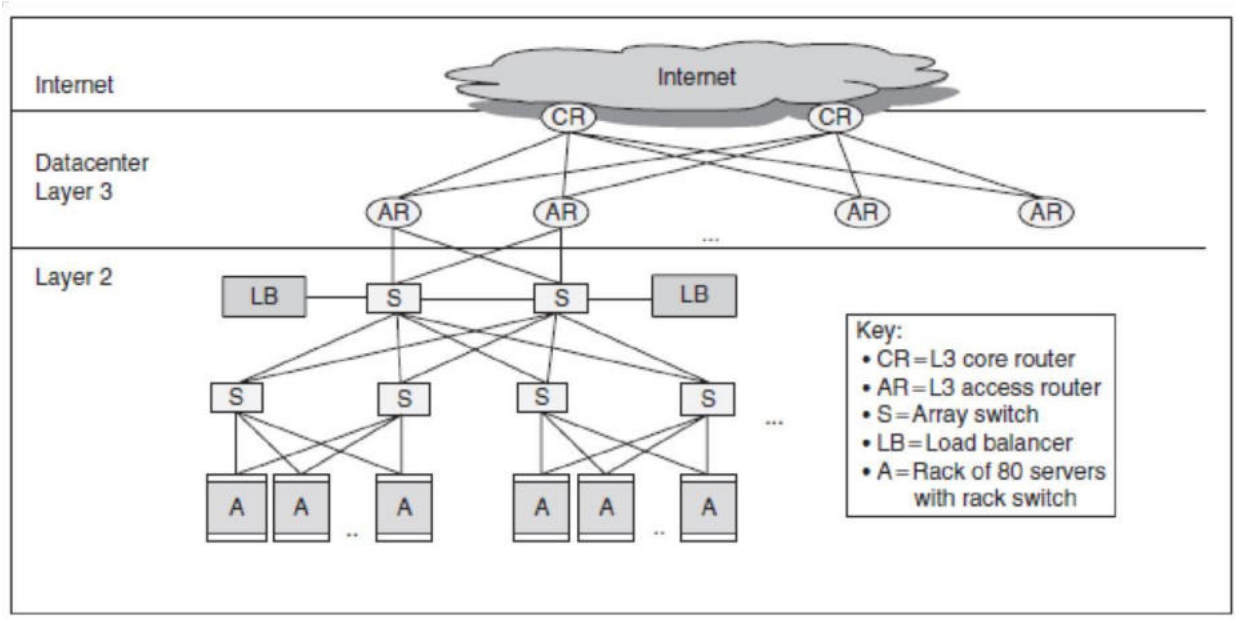


Figure 4.12 The Layer 3 network used to link arrays together and to the Internet

Most applications fit on a single array within a WSC. Those that need more than one array use *sharding* or *partitioning*, meaning that the dataset is split into independent pieces and then distributed to different arrays. Operations on the whole dataset are sent to the servers hosting the pieces, and the results are coalesced by the client computer.

4.7 Physical Infrastructure and Costs of Warehouse-Scale Computers

Infrastructure costs for power distribution and cooling dwarf the construction costs of a WSC, so we concentrate on the former. Figures 4.13 and 4.14 show the power distribution and cooling infrastructure within a WSC.

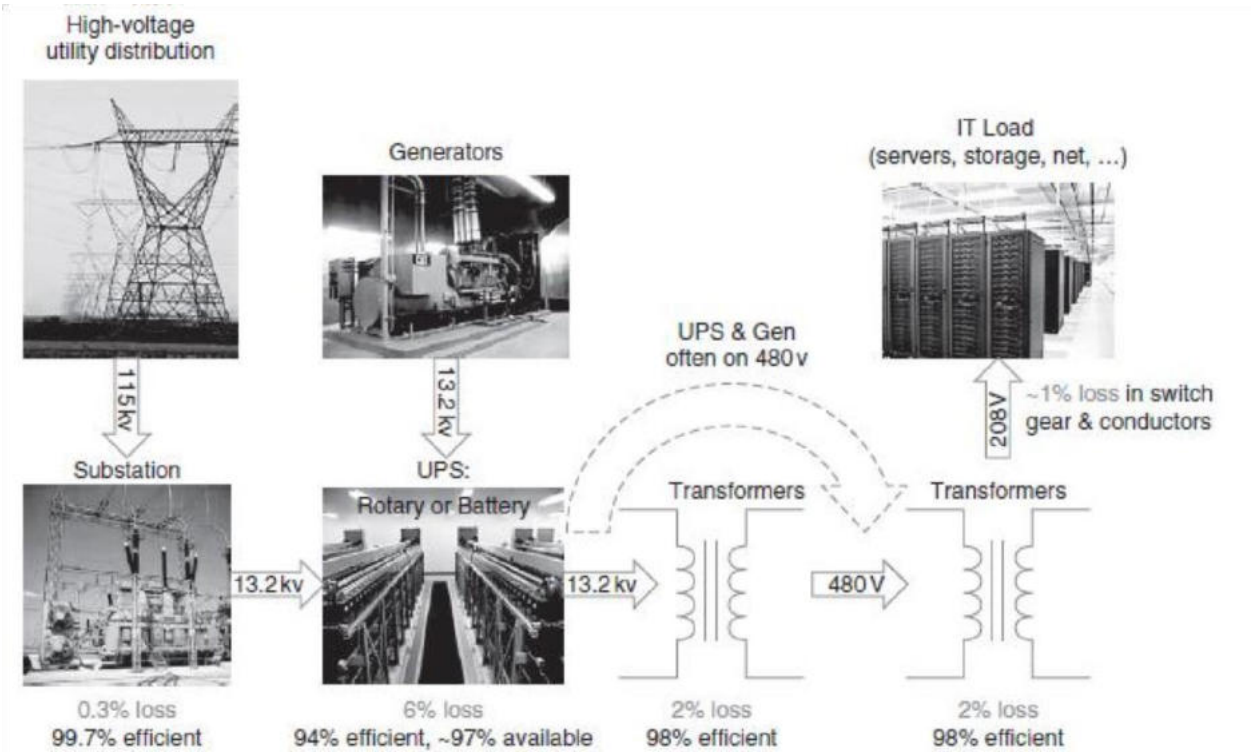


Figure 4.13 Power distribution and where losses occur

- The substation switches from 115,000 volts to medium-voltage lines of 13,200 volts, with an efficiency of 99.7%.
- To prevent the whole WSC from going offline if power is lost, a WSC has an uninterruptible power supply (UPS), just as some servers do. The generators and batteries can take up so much space that they are typically located in a separate room from the IT equipment. The UPS plays three roles: power conditioning (maintain proper voltage levels and other characteristics), holding the electrical load while the generators start and come on line, and holding the electrical load when switching back from the generators to the electrical utility.
- Next in the system is a power distribution unit (PDU) that converts to lowvoltage, internal, three-phase power at 480 volts. The conversion efficiency is 98%. A typical PDU handles 75 to 225 kilowatts of load, or about 10 racks.
- There is yet another down step to two-phase power at 208 volts that servers can use, once again at 98% efficiency.

- The connectors, breakers, and electrical wiring to the server have a collective efficiency of 99%.

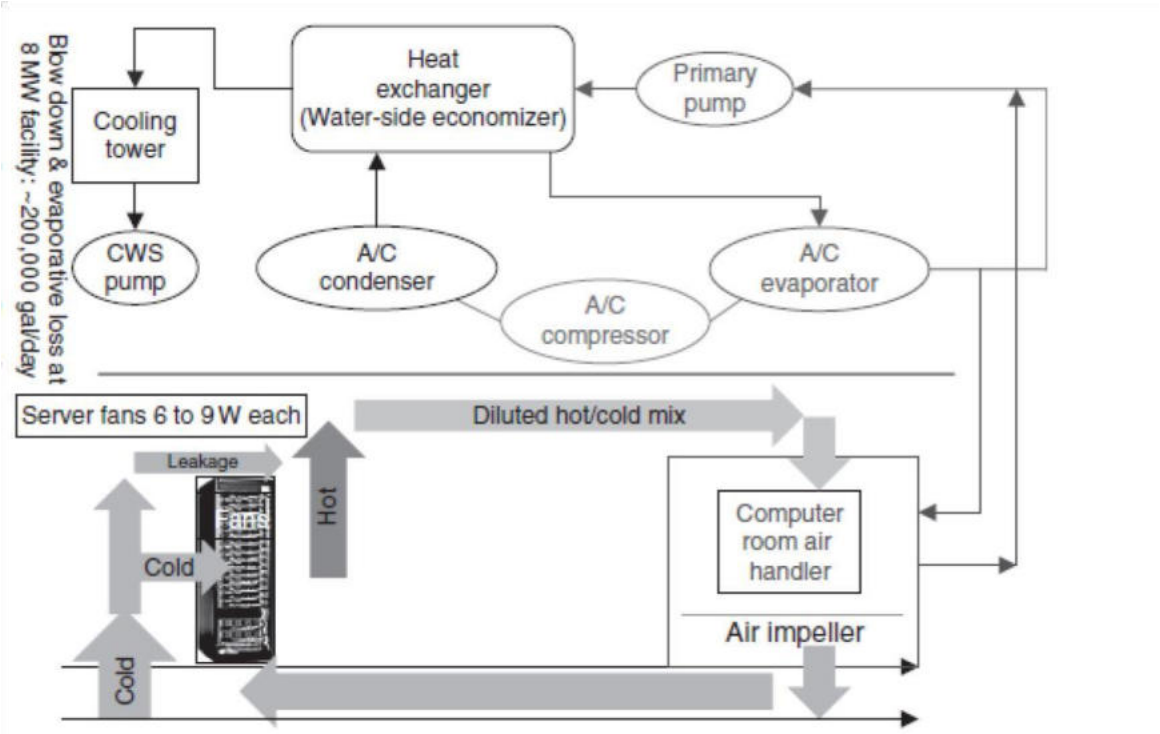


Figure 4.14 Power distribution and where losses occur

The computer room air-conditioning (CRAC) unit (figure 4.14) cools the air in the server room using chilled water, similar to how a refrigerator removes heat by releasing it outside of the refrigerator. As a liquid absorbs heat, it evaporates. Conversely, when a liquid releases heat, it condenses. Air conditioners pump the liquid into coils under low pressure to evaporate and absorb heat, which is then sent to an external condenser where it is released. Thus, in a CRAC unit, fans push warm air past a set of coils filled with cold water and a pump moves the warmed water to the external chillers to be cooled down. The cool air for servers is typically between 64° F and 71° F

Clearly, one of the simplest ways to improve energy efficiency is simply to run the IT equipment at higher temperatures so that the air need not be cooled as much.

In addition to chillers, cooling towers are used in some datacenters to leverage the colder outside air to cool the water before it is sent to the chillers. The temperature that matters is called the *wet-bulb temperature.*

Warm water flows over a large surface in the tower, transferring heat to the outside air via evaporation and thereby cooling the water. This technique is called airside economization. An alternative is use cold water instead of cold air

Airflow is carefully planned for the IT equipment itself, with some designs even using airflow simulators. Efficient designs preserve the temperature of the cool air by reducing the chances of it mixing with hot air.

The relative power costs of cooling equipment to IT equipment in a typical datacenter are as follows:

- Chillers account for 30% to 50% of the IT equipment power.
- CRAC accounts for 10% to 20% of the IT equipment power, due mostly to fans.

The so-called *nameplate power rating* from the server manufacturer is always conservative; it's the maximum power a server can draw.

Measuring Efficiency of a WSC

A widely used, simple metric to evaluate the efficiency of a datacenter or a WSC is called *power utilization effectiveness* (or *PUE*):

PUE = (Total facility power)/(IT equipment power)

Thus, PUE must be greater than or equal to 1, and the bigger the PUE the less efficient the WSC.

Latency (of request, first response, completion...) is the immediately perceivable metric from users' perspective

User productivity = 1 / time of interaction t(interaction)=t(human entry)+t(system response)+t(analysis of response) **Cost of a WSC**

Capital expenditures (CAPEX)

CAPEX is the cost to build a WSC, which includes the building, power and cooling infrastructure, and initial IT equipment (servers and networking equipment).

Operational expenditures (OPEX)

OPEX is the cost to operate a WSC, which includes buying replacement equipment, electricity, and salaries.

The cost advantage of WSCs led large Internet companies to offer computing as a utility where, like electricity, you pay only for what you use. Today, utility computing is better known as cloud computing.

4.8 Cloud Computing

Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Cloud computing can be thought of as providing computing as a utility, where a customer pays for only what they use, just as we do for electricity. Cloud computing relies on increasingly larger WSCs which provides several benefits if properly set up and operated.

- improvements in operational techniques
- economies of scale
- reduces customer risks of over-provisioning or under-provisioning

4.8.1 Improvements in Operational Techniques

WSCs have led to innovation in system software to provide high reliability.

- failover - Automatically restarting an application that fails without requiring administrative intervention.
- firewall - Examines each network packet to determine whether or not it should be forwarded to its destination.
- virtual machine - A software layer that executes applications like a physical machine.
- Protection against denial-of-service attacks

4.8.2 Economies of Scale

WSCs offer economies of scale that cannot be achieved with a data center.

- 5.7 times reduction in storage costs
- 7.1 times reduction in administrative costs
- 7.3 times reduction in networking costs
- volume discount price reductions

- PUE of perhaps 1.2 versus PUE of 2.0 for a data center
- better utilization of WSC by being available to the public

4.8.3 Reducing Customer Risks

WSCs reduce risks of over-provisioning or under-provisioning, particularly for start-up companies.

- Providing too much equipment means overspending.
- Providing too little equipment means demand may not be able to be met, which can give a bad impression to potential new customers.

4.8.4 Amazon Web Services

Amazon offered Amazon Simple Storage Service (Amazon S3) and Amazon Elastic Computer Cloud (Amazon EC2) in 2006.

- Relied on virtual machines.
    - Provides better protection for users.
    - Simplified software distribution within a WSC.
    - The ability to reliably kill a virtual machine made it easier to control resource usage.
    - Being able to limit use of resources simplified providing multiple price points for customers.
    - Improved flexibility in server configuration.
- Relied on open source software o The availability of good-quality software that had no licensing problems or costs. More recently, AWS started offering instances including commercial third-party software at higher prices
- Provided service at very low cost.
- No contract required

In addition to the low cost and a pay-for-use model of utility computing, another strong attractor for cloud computing users is that the cloud computing providers take on the risks of over-provisioning or under-provisioning. Risk avoidance is a godsend for startup companies, as either mistake could be fatal. If too much of the precious investment is spent on servers before the product is ready for heavy use, the company could run out of money. If the service suddenly became

popular, but there weren't enough servers to match the demand, the company could make a very bad impression with the potential new customers it desperately needs to grow. Cloud computing has made the benefits of WSC available to everyone. Cloud computing offers cost associativity with the illusion of infinite scalability at no extra cost to the user: 1000 servers for 1 hour cost no more than 1 server for 1000 hours. It is up to the cloud computing provider to ensure that there are enough servers, storage, and Internet bandwidth available to meet the demand This transfer of risks, cost associativity, and pay-as you-go pricing is a powerful argument for companies of varying sizes to use cloud computing.

Advantages of Cloud Computing ▪

Pay as you go.

- No need to provision for peak loads.
- Time to market.
- Consistent performance and availability.

Potential Drawbacks of Cloud Computing ▪

Privacy and security.

- External dependency for mission critical applications.
- Disaster recovery.

4.9 Case Study- Google Warehouse-Scale Computer

Datacenters are buildings where multiple servers and communication gear are collocated because of their common environmental requirements and physical security needs, and for ease of maintenance. In that sense, a WSC could be considered a type of datacenter. Traditional datacenters, however, typically host a large number of relatively small- or medium-sized applications, each running on a dedicated hardware infrastructure that is de-coupled and protected from other systems in the same facility. Those datacenters host hardware and software for multiple organizational units or even different companies.

WSCs currently power the services offered by companies such as Google, Amazon, Yahoo, and Microsoft's online services division. They differ significantly from traditional datacenters: they

belong to a single organization, use a relatively homogeneous hardware and system software platform, and share a common systems management layer.

Google has shared information about their datacenters up to 2007 (not new, but at this scale everything is a company secret). WSCs are now built with shipping containers (they used to be standard shipping containers, but now they are generally customized).

- Shipping containers are modular — the only external connections are networking, power, and cooling.
- The 2007 version contains 45 40-foot long containers in a 300-foot by 250-foot space, or 75,000sq-ft. To fit in the warehouse, 30 of the containers are stacked too high.
- The data center is probably in Dalles, Oregon, which has a moderate climate near cheap hydroelectric power and an Internet backbone fiber location (Google has an energy subsidiary that can buy and sell energy like any utility).
- The WSC has a PUE of 1.23.
- 85% of the energy overhead goes to cooling losses, 15% goes to power losses.

4.9.1 Containers

Both Google and Microsoft have built WSCs using shipping containers. The idea of building a WSC from containers is to make WSC design modular. Each container is independent, and the only external connections are networking, power and water. The containers in turn supply networking, power, cooling to the servers placed inside them, so the job of WSC is to supply networking, power, and cold water to the containers and to pump the resulting warm water to external cooling towers and chillers.

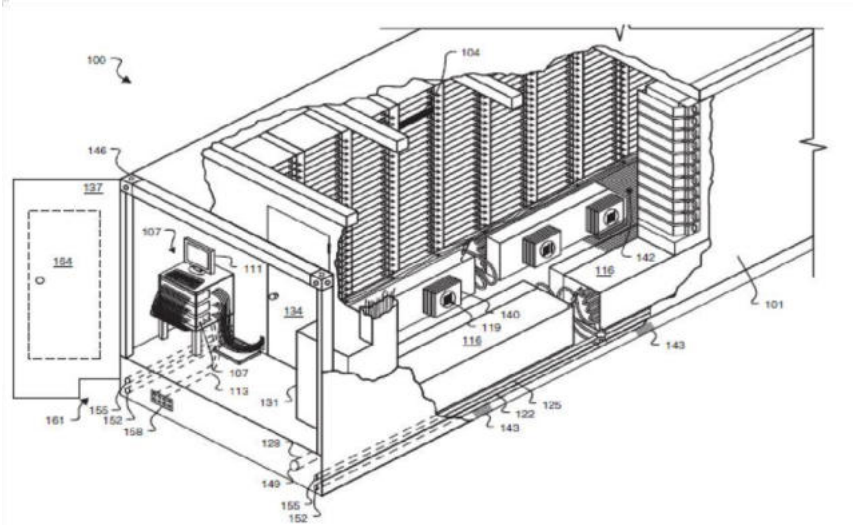


Figure 4.15 Google customizes a standard 1AAA container

The Google WSC that we are looking at contains 45 40-foot-long containers in a 300-foot by 250-foot space, or 75,000 square feet (about 7000 square meters). To fit in the warehouse, 30 of the containers are stacked two high, or 15 pairs of stacked containers.

Although the location was not revealed, it was built at the time that Google developed WSCs in The Dalles, Oregon, which provides a moderate climate and is near cheap hydroelectric power and Internet backbone fiber.

This WSC offers 10 megawatts with a PUE of 1.23 over the prior 12 months. Of that 0.230 of PUE overhead, 85% goes to cooling losses (0.195 PUE) and 15% (0.035) goes to power losses. The system went live in November 2005, and this section describes its state as of 2007. The diagram above shows cutaway drawing of a Google container. A container holds up to 1160 servers, so 45 containers have space for 52,200 servers. The servers are stacked 20 high in racks each, with one row on each side of the container. The rack switches are 48 port, 1Gbit/sec Ethernet switches, which are placed in every other rack.

4.9.2 Cooling and Power in the Google WSC

Figure 4.16 is a cross-section of the container that shows the airflow. The computer racks are attached to the ceiling of the container. The cooling is below a raised floor that blows into the aisle between the racks. Hot air is returned from behind the racks. The restricted space of the container prevents the mixing of hot and cold air, which improves cooling efficiency. Variablespeed fans are

run at the lowest speed needed to cool the rack as opposed to a constant speed. The "cold" air is kept 81°F (27°C), which is balmy compared to the temperatures

in many conventional datacenters. One reason datacenters traditionally run so cold is not for the IT equipment, but so that hot spots within the datacenter don't cause isolated problems. By carefully controlling airflow to prevent hot spots, the container can run at a much higher temperature.

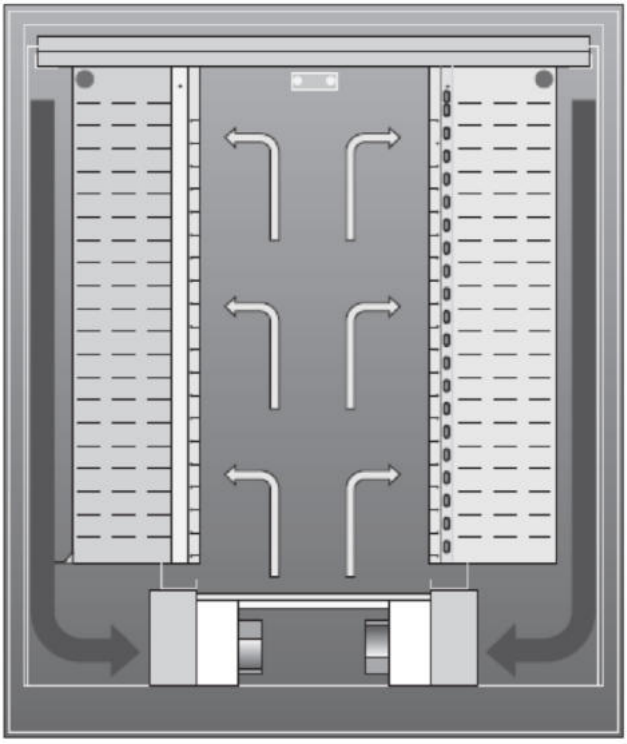


Figure 4.16 Airflow within the container

External chillers have cutouts so that, if the weather is right, only the outdoor cooling towers need cool the water. The chillers are skipped if the temperature of the water leaving the cooling tower is 70°F (21°C) or lower. The cooling towers need heaters to prevent ice from forming.

4.9.3 Servers in a Google WSC

The server in Figure 6.21 has two sockets, each containing a dual-core AMD Opteron processor running at 2.2 GHz. The photo shows eight DIMMS, and these servers are typically deployed with 8 GB of DDR2 DRAM. A novel feature is that the memory bus is downclocked to 533 MHz from the standard 666 MHz since the slower bus has little impact on performance but a significant

impact on power. The baseline design has a single network interface card (NIC) for a 1 Gbit/sec Ethernet link.

Although the photo in Figure 6.21 shows two SATA disk drives, the baseline server has just one. The peak power of the baseline is about 160 watts, and idle power is 85 watts. This baseline node is supplemented to offer a storage (or "diskfull") node.

- First, a second tray containing 10 SATA disks is connected to the server.
- To get one more disk, a second disk is placed into the empty spot on the motherboard, giving the storage node 12 SATA disks.
- Finally, since a storage node could saturate a single 1 Gbit/sec Ethernet link, a second Ethernet NIC was added. Peak power for a storage node is about 300 watts, and it idles at 198 watts

4.9.4 Networking in a Google WSC

The 40,000 servers are divided into three arrays of more than 10,000 servers each. (Arrays are called *clusters* in Google terminology.) The 48-port rack switch uses 40 ports to connect to servers, leaving 8 for uplinks to the array switches.

Array switches are configured to support up to 480 1 Gbit/sec Ethernet links and a few 10 Gbit/sec ports. The 1 Gigabit ports are used to connect to the rack switches, as each rack switch has a single link to each of the array switches. The WSC uses two datacenter routers for dependability, so a single datacenter router failure does not take out the whole WSC.

The number of uplink ports used per rack switch varies from a minimum of 2 to a maximum of 8. In the dual-port case, rack switches operate at an oversubscription rate of 20:1. That is, there is 20 times the network bandwidth inside the switch as there was exiting the switch.

4.9.5 Monitoring and Repair in a Google WSC

Extensive automatic monitoring is done to track the health of the servers. Diagnostics run continually. Automated solutions solve many problems (server freeze→reboot, etc.). Repair technicians use the automated tests for quick repair (e.g., they don't diagnose if the automatic test says a part is bad). Goal: less than 1% in the repair queue at any time.

UNIT V VECTOR, SIMD AND GPU ARCHITECTURES

Introduction-Vector Architecture – SIMD Extensions for Multimedia – Graphics Processing Units – Case Studies – GPGPU Computing – Detecting and Enhancing Loop Level Parallelism-Case Studies.

5.1 Introduction

It was suggested that a key aspect of vector architecture is the single-instructionmultipledata (SIMD) execution model. SIMD support results from the type of data supported by the instruction set, and how instructions operate on that data.

- SIMD architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processors
- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially
- Three variations of SIMD:
  - vector architectures,
  - multimedia SIMD instruction set extensions, and
  - graphics processing units (GPUs).

These *vector architectures* are easier to understand and to compile to than other SIMD variations, but they were considered too expensive for microprocessors until recently.

The second SIMD variation borrows the SIMD name to mean basically simultaneous parallel data operations and is found in most instruction set architectures today that support multimedia applications. For x86 architectures, the SIMD instruction extensions started with the MMX (Multimedia Extensions) in 1996, which were followed by several SSE (Streaming SIMD Extensions) versions in the next decade, and they continue to this day with AVX (Advanced Vector Extensions).

The third variation on SIMD comes from the GPU community, offering higher potential performance than is found in traditional multicore computers today. This environment has a system processor and system memory in addition to the GPU and its graphics memory. In fact, to recognize those distinctions, the GPU community refers to this type of architecture as *heterogeneous.*

For x86 computers, we expect to see two additional cores per chip every two years and the SIMD width to double every four years. Given these assumptions, over the next decade the potential speedup from SIMD parallelism is twice that of MIMD parallelism.

5.2 Vector Architecture

- The most efficient way to execute a vectorizable application is a vector processor.
- Vector architectures grab sets of data elements scattered about memory, place them into large, sequential register files, operate on data in those register files, and then disperse the results back into memory.
- A single instruction operates on vectors of data, which results in dozens of register– register operations on independent data elements.
- These large register files act as compiler-controlled buffers, both to hide memory latency and to leverage memory bandwidth.

5.2.1 VMIPS

This processor, is loosely based on the Cray-1. Figure 5.1 shows the basic structure of Vector architecture. The primary components of the instruction set architecture of VMIPS are the following:

Vector registers—Each vector register is a fixed-length bank holding a single vector. VMIPS has eight vector registers, and each vector register holds 64 elements, each 64 bits wide. The vector register file needs to provide enough ports to feed all the vector functional units. These ports will allow a high degree of overlap among vector operations to different vector registers. The read and write ports, which total at least 16 read ports and 8 write ports, are connected to the functional unit inputs or outputs by a pair of crossbar switches.

Vector functional units—Each unit is fully pipelined, and it can start a new operation on every clock cycle. A control unit is needed to detect hazards, both structural hazards for functional units

and data hazards on register accesses. VMIPS has five functional units (FP add/sub, FP mul, FP div, FP integer, FP logical). But the main focus is on the floating point functional units.

Vector load/store unit—The vector memory unit loads or stores a vector to or from memory. It is a fully pipelined unit to load / store a vector. IT may have multiple LSUs. The words can be moved between the vector registers and memory with a bandwidth of one word per clock cycle, after an initial latency.

A set of scalar registers—Scalar registers can also provide data as input to the vector functional units, as well as compute addresses to pass to the vector load/store unit. These are the normal 32 general-purpose registers and 32 floating-point registers of MIPS. One input of the vector functional units latches scalar values as they are read out of the scalar register file.

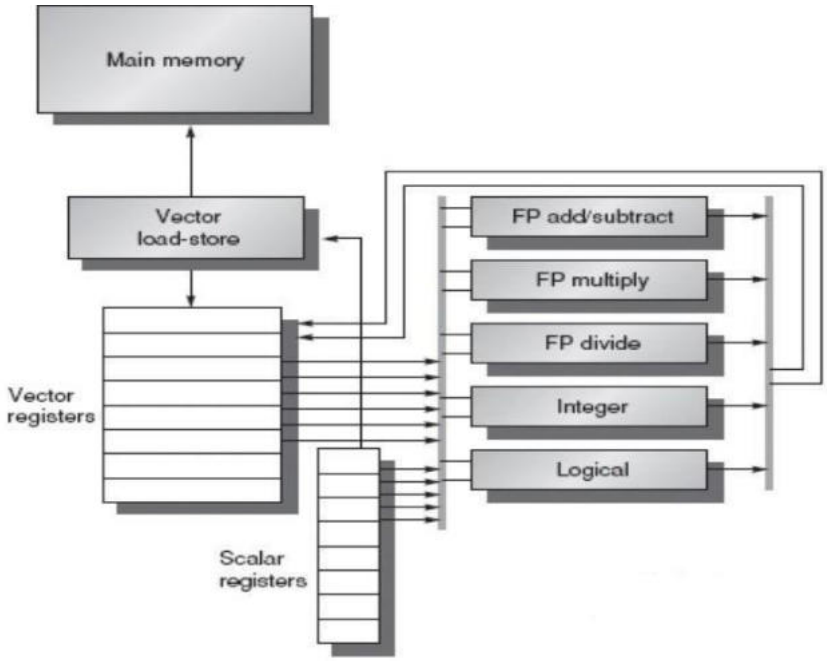


Figure 5.1 The basic structure of a vector architecture, VMIPS

In VMIPS, vector operations use the same names as scalar MIPS instructions, but with the letters ―VV‖ appended. Thus, ADDVV.D is an addition of two double-precision vectors. The vector instructions take as their input either a pair of vector registers (ADDVV.D) or a vector register and a scalar register, designated by appending ―VS‖ (ADDVS.D).

The operation ADDVS.D will add the contents of a scalar register to each element in a vector register. The vector functional unit gets a copy of the scalar value at issue time. The names LV and SV denote vector load and vector store, and they load or store an entire vector of double-precision

data. One operand is the vector register to be loaded or stored; the other operand, which is a MIPS general-purpose register, is the starting address of the vector in memory.

ADDVV.D: add two vectors

ADDVS.D: add vector to a scalar

LV/SV: vector load and vector store from address

Example: DAXPY (double precision a*X+Y)
Here is the MIPS code.

```
        L.D     F0,a        ; load scalar a
        DADDIU  R4,Rx,#512  ; last address to load
Loop:   L.D     F2,0(Rx)    ; load X[i]
        MUL.D   F2,F2,F0    ; a x X[i]
        L.D     F4,0(Ry)    ; load Y[i]
        ADD.D   F4,F2,F2    ; a x X[i] + Y[i]
        S.D     F4,9(Ry)    ; store into Y[i]
        DADDIU  Rx,Rx,#8    ; increment index to X
        DADDIU  Ry,Ry,#8    ; increment index to Y
        SUBBU   R20,R4,Rx   ; compute bound
        BNEZ    R20,Loop    ; check if done
```

Requires almost 600 MIPS ops

Here is the VMIPS code for DAXPY

```
L.D     F0,a        ; load scalar a
LV      V1,Rx       ; load vector X
MULVS.D V2,V1,F0    ; vector-scalar multiply
LV      V3,Ry       ; load vector Y
ADDVV   V4,V2,V3    ; add
SV      Ry,V4       ; store the result
```

Requires 6 instructions

This reduction occurs because the vector operations work on 64 elements and the overhead instructions that constitute nearly half the loop on MIPS are not present in the VMIPS code. When the compiler produces vector instructions for such a sequence and the resulting code spends much of its time running in vector mode, the code is said to be **vectorized or vectorizable**. Loops can

be vectorized when they do not have dependences between iterations of a loop, which are called **loop-carried dependences**

5.2.2 Vector Execution Time

Execution time depends on three factors:

- Length of operand vectors
- Structural hazards
- Data dependencies

VMIPS functional units consume one element per clock cycle

- Execution time is approximately the vector length

convoy – which is the set of vector instructions that could potentially execute together. The instructions in a convoy *must not* contain any structural hazards; if such hazards were present, the instructions would need to be serialized and initiated in different convoys. To keep the analysis simple, we assume that a convoy of instructions must complete execution before any other instructions (scalar or vector) can begin execution .

Chaining – allows a vector operation to start as soon as the individual elements of its vector source operand become available: Sequences with read-after-write dependency hazards can be in the same convey via *chaining.* The results from the first functional unit in the chain are

―forwarded‖ to the second functional unit.

Chime – To turn convoys into execution time we need a timing metric to estimate the time for a convoy. It is called a *chime*, which is simply the unit of time taken to execute one convoy. Thus, a vector sequence that consists of m convoys executes in m chimes; for a vector length of n, for VMIPS this is approximately $m \times n$ clock cycles.

Example:

```
LV        V1,Rx      ;load vector X
MULVS.D   V2,V1,F0   ;vector-scalar multiply
LV        V3,Ry      ;load vector Y
ADDVV.D   V4,V2,V3   ;add two vectors
SV        Ry,V4      ;store the sum
```

Convoys:

1	LV	MULVS.D
2	LV	ADDVV.D
3	SV	

3 chimes, 2 FP ops per result, cycles per FLOP = 1.5 For

64 element vectors, requires 64 x 3 = 192 clock cycles

Optimizations:

Multiple Lanes: > 1 element per clock cycle

Vector Length Registers: Non-64 wide vectors

Vector Mask Registers: IF statements in vector code

Memory Banks: Memory system optimizations to support vector processors

Stride**:** Multiple dimensional matrices
Scatter-Gather: Sparse matrices

Programming Vector Architectures: Program structures affecting performance

Multiple Lanes: Beyond One Element per Clock Cycle

A single vector instruction can include scores of independent operations yet be encoded in the same number of bits as a conventional scalar instruction. The parallel semantics of a vector instruction allow an implementation to execute these elemental operations using a deeply pipelined functional unit, as in the VMIPS implementation.

Figure 5.2 illustrates how to improve vector performance by using parallel pipelines to execute a vector add instruction. The vector processor (a) on the left has a single add pipeline and can complete one addition per cycle. The vector processor (b) on the right has four add pipelines and can complete four additions per cycle. The VMIPS instruction set has the property that all vector arithmetic instructions only allow element *N* of one vector register to take part in operations with element *N* from other vector registers.

It simplifies the construction of a highly parallel vector unit, which can be structured as multiple parallel *lanes*. Figure 5.3 shows the structure of a four-lane vector unit. Thus, going to four lanes from one lane reduces the number of clocks for a chime from 64 to 16. For multiple lanes to be advantageous, both the applications and the architecture must support long vectors; Each lane contains one portion of the vector register file and one execution pipeline from each vector functional unit. Each vector functional unit executes vector instructions at the rate of one element group per cycle using multiple pipelines, one per lane.

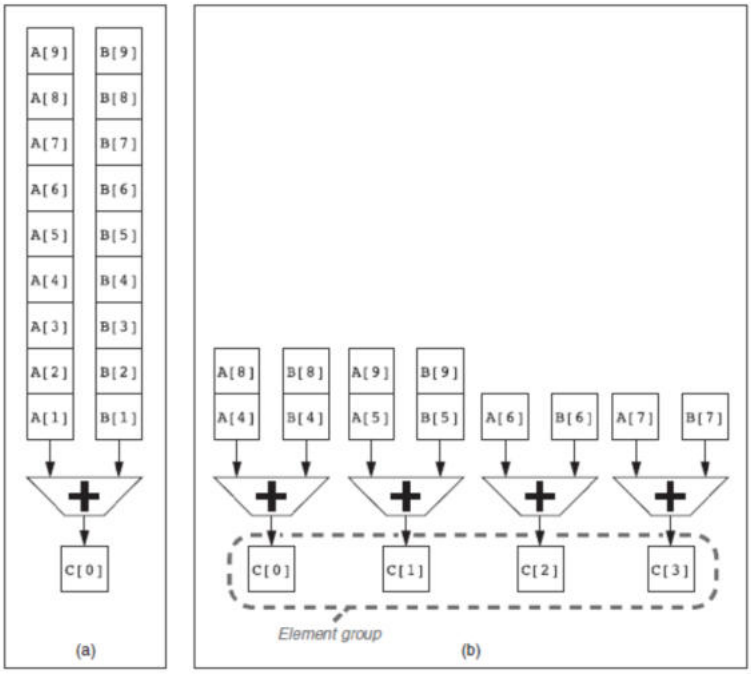


Figure 5.2 Using multiple functional units to improve the performance of a single vector add instruction, C =A + B.

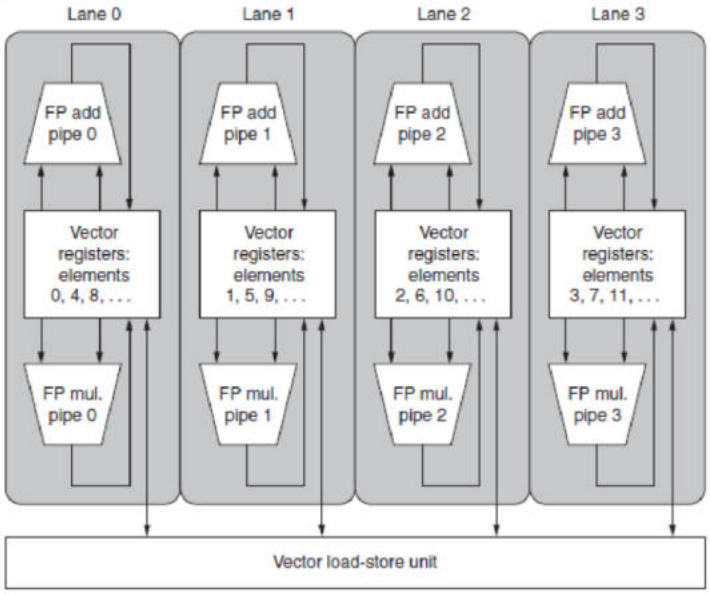


Figure 5.3 Structure of a vector unit containing four lanes

The first lane holds the first element (element 0) for all vector registers, and so the first element in any vector instruction will have its source and destination operands located in the first lane. This

allocation allows the arithmetic pipeline local to the lane to complete the operation without communicating with other lanes. Accessing main memory also requires only intralane wiring.

Adding multiple lanes is a popular technique to improve vector performance as it requires little increase in control complexity and does not require changes to existing machine code. It also allows designers to trade off die area, clock rate, voltage, and energy without sacrificing peak performance.

Vector-Length Registers: Handling Loops Not Equal to 64

A vector register processor has a natural vector length determined by the number of elements in each vector register. This length, which is 64 for VMIPS, is unlikely to match the real vector length in a program. Moreover, in a real program the length of a particular vector operation is often *unknown* at compile time. In fact, a single piece of code may require different vector lengths. For example, consider this code:

for (i=0; i <n; i=i+1) Y[i] = a * X[i] + Y[i]; The size of all the vector operations depends on n, which may not even be known until run time! The solution to these problems is to create a *vector-length register* (VLR). The VLR controls the length of any vector operation, including a vector load or store. The value in the VLR, however, cannot be greater than the length of the vector registers. This solves our problem

as long as the real length
is less than or equal to
the *maximum vector*
length (MVL). The MVL
determines the number
of data elements in a
vector of an architecture.
This parameter means
the length of vector
registers can grow in
later computer
generations without
changing the instruction
set;

What if n > Max. Vector Length (MVL)?

To tackle this problem where the vector is longer than the maximum length, a technique called *strip mining* is used. Strip mining is the generation of code such that each vector operation is done for a size less than or equal to the MVL.

Vector Mask Registers: Handling IF Statements in Vector Loops

The presence of conditionals (IF statements) inside loops and the use of sparse matrices are two main reasons for lower levels of vectorization. Programs that contain IF statements in loops cannot be run in vector mode using the previous techniques because the IF statements introduce control dependences into a loop.

Consider the following loop written in C:

for (i = 0; i < 64; i=i+1) if (X[i] != 0) X[i]
= X[i] – Y[i];

This loop cannot normally be vectorized because of the conditional execution of the body; however, if the inner loop could be run for the iterations for which X[i] ≠ 0, then the subtraction could be vectorized.

The common extension for this capability is *vector-mask control.* Mask registers essentially provide conditional execution of each element operation in a vector instruction. The vector-mask control uses a Boolean vector to control the execution of a vector instruction, just as conditionally executed instructions use a Boolean condition to determine whether to execute a scalar instruction.

Similarly, vector instructions executed with a vector mask still take the same execution time, even for the elements where the mask is zero.

Memory Banks: Supplying Bandwidth for Vector Load/Store Units

Memory system must be designed to support high bandwidth for vector loads and stores. Spreading accesses across multiple independent memory banks usually delivers the desired rate.

Most vector processors use memory banks mainly for three reasons:

- Control bank addresses independently
- Load or store non sequential words
- Support multiple vector processors sharing the same memory

In combination, these features lead to a large number of independent memory banks, as the following example shows**:**

Consider 32 processors, each generating 4 loads and 2 stores/cycle then, Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns. How many memory banks needed?

32x6=192 accesses, 15/2.167≈7 processor cycles, 192 × 7 = 1344 memory banks

Stride: Handling Multidimensional Arrays in Vector Architectures

Suppose adjacent elements not sequential in memory (e.g. matrix multiplication).

```
for (i = 0; i < 100; i=i+1) for
(j = 0; j < 100; j=j+1) {
A[i][j] = 0.0;

for (k = 0; k < 100; k=k+1) A[i][j] =
A[i][j] + B[i][k] * D[k][j];
```

}

We could vectorize the multiplication of each row of B with each column of D and stripmine the inner loop with k as the index variable.

Consider how to address adjacent elements in B and adjacent elements in D. When an array is allocated memory, it is linearized and must be laid out in either row-major (as in C) or column-major (as in Fortran) order. This linearization means that either the elements in the row or the elements in the column are not adjacent in memory.

For example, the C code above allocates in row-major order, so the elements of D that are accessed by iterations in the inner loop are separated by the row size times 8 (the number of bytes per entry) for a total of 800 bytes.

This distance separating elements to be gathered into a single register is called the ***stride.***
Once a vector is loaded into a vector register, it acts as if it had logically adjacent elements.

Thus, a vector processor can handle strides greater than one, called ***non-unit strides***, using only vector load and vector store operations with stride capability. This ability to access nonsequential memory locations and to reshape them into a dense structure is one of the major advantages of a vector processor.

Supporting strides greater than one complicates the memory system. **Bank conflict** (stall) occurs when the same bank is hit faster than bank busy time.

$$\frac{Number\ of\ Banks}{Least\ Common\ multiple(Stride, Number\ of\ Banks)} < Bank\ Busy\ Time$$

Gather-Scatter: Handling Sparse Matrices in Vector Architectures

The primary mechanism for supporting sparse matrices is *gather-scatter operations* using index vectors. The goal of such operations is to support moving between a compressed representation (i.e., zeros are not included) and normal representation (i.e., the zeros are included) of a sparse matrix.

A *gather* operation takes an *index vector* and fetches the vector whose elements are at the addresses given by adding a base address to the offsets given in the index vector. The result is a dense vector in a vector register. After these elements are operated on in dense form, the sparse vector can be stored in expanded form by a *scatter* store, using the same index vector. Hardware

support for such operations is called *gather-scatter* and it appears on nearly all modern vector processors.

For example, Ra, Rc, Rk and Rm the starting addresses of vectors Use index vector:

```
LV      Vk, Rk          ;load K
LVI     Va, (Ra+Vk)     ;load A[K[]]
LV      Vm, Rm          ;load M
LVI     Vc, (Rc+Vm)     ;load C[M[]]
ADDVV.D Va, Va, Vc      ;add them
SVI     (Ra+Vk), Va     ;store A[K[]]
```

This technique allows code with sparse matrices to run in vector mode.

Programming Vector Architectures

An advantage of vector architectures is that compilers can tell programmers at compile time whether a section of code will vectorize or not, often giving hints as to why it did not vectorize the code.

Today, the main factor that affects the success with which a program runs in vector mode is the structure of the program itself. Several studies of the performance of applications on vector processors show a wide variation in the level of compiler vectorization.

5.3 SIMD Extensions for Multimedia

SIMD Multimedia Extensions started with the simple observation that many media applications operate on narrower data types than the 32-bit processors were optimized for. When processors began to include graphics instructions, architects realized that operations not necessarily need to be 32-bit instructions

- Graphics for instance often operates on several 8-bit operations (one each for red, green, blue, transparency)
  - so while a datum might be 32 bits in length, it really codified 4 pieces of data, each of which could be operated on simultaneously within the adder
- Additionally, sounds are typically stored as segments of 8 or 16 bit data

Table below summarizes typical multimedia SIMD instructions. Like vector instructions, a SIMD instruction specifies the same operation on vectors of data. Unlike vector machines with large register files such as the VMIPS vector register, which can hold as many as sixty-four 64bit elements in each of 8 vector registers, SIMD instructions tend to specify fewer operands and hence use much smaller register files.

Instruction category	Operands
Unsigned add/subtract	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Maximum/minimum	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Average	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Shift right/left	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Floating point	Sixteen 16-bit, eight 32-bit, four 64-bit, or two 128-bit

SIMD extensions have three major omissions:

- Multimedia SIMD extensions fix the number of data operands in the opcode, which has led to the addition of hundreds of instructions in the MMX, SSE, and AVX extensions of the x86 architecture. Vector architectures have a vector length register that specifies the number of operands for the current operation. Moreover, vector architectures have an implicit maximum vector length in the architecture, which combined with the vector length register avoids the use of many opcodes.
- Multimedia SIMD does not offer the more sophisticated addressing modes of vector architectures, namely strided accesses and gather-scatter accesses.
- Multimedia SIMD usually does not offer the mask registers to support conditional execution of elements as in vector architectures.

These omissions make it harder for the compiler to generate SIMD code and increase the difficulty of programming in SIMD assembly language.

For the x86 architecture, the MMX instructions added in 1996 repurposed the 64-bit floating-point registers, so the basic instructions could perform eight 8-bit operations or four 16bit operations simultaneously.

The Streaming SIMD Extensions (SSE) separate registers that were 128 bits wide, so now instructions could simultaneously perform sixteen 8-bit operations, eight 16-bit operations, or four 32-bit operations. It also performed parallel single-precision floating-point arithmetic. Since SSE had separate registers, it needed separate data transfer instructions. With each generation, they also added ad hoc instructions whose aim is to accelerate specific multimedia functions perceived to be important.

The Advanced Vector Extensions (AVX), doubles the width of the registers again to 256 bits and thereby offers instructions that double the number of operations on all narrower data types. AVX includes preparations to extend the width to 512 bits and 1024 bits in future generations of the architecture.

Another advantage of short, fixed-length ―vectors‖ of SIMD is that it is easy to introduce instructions that can help with new media standards, such as instructions that perform permutations or instructions that consume either fewer or more operands than vectors can produce.

5.3.1 Programming Multimedia SIMD Architectures

Given the ad hoc nature of the SIMD multimedia extensions, the easiest way to use these instructions has been through libraries or by writing in assembly language.

By borrowing techniques from vectorizing compilers, compilers are starting to produce SIMD instructions automatically. For example, advanced compilers today can generate SIMD floating-point instructions to deliver much higher performance for scientific codes. However, programmers must be sure to align all the data in memory to the width of the SIMD unit on which the code is run to prevent the compiler from generating scalar instructions for otherwise vectorizable code.

5.3.2 The Roofline Visual Performance Model

One visual, intuitive way to compare potential floating-point performance of variations of SIMD architectures is the Roofline model.

It ties together floating-point performance, memory performance, and arithmetic intensity in a two-dimensional graph. *Arithmetic intensity* is the ratio of floating-point operations per byte of memory accessed. It can be calculated by taking the total number of floating-point operations for

a program divided by the total number of data bytes transferred to main memory during program execution.

Figure 5.4 shows the relative arithmetic intensity of several example kernels.

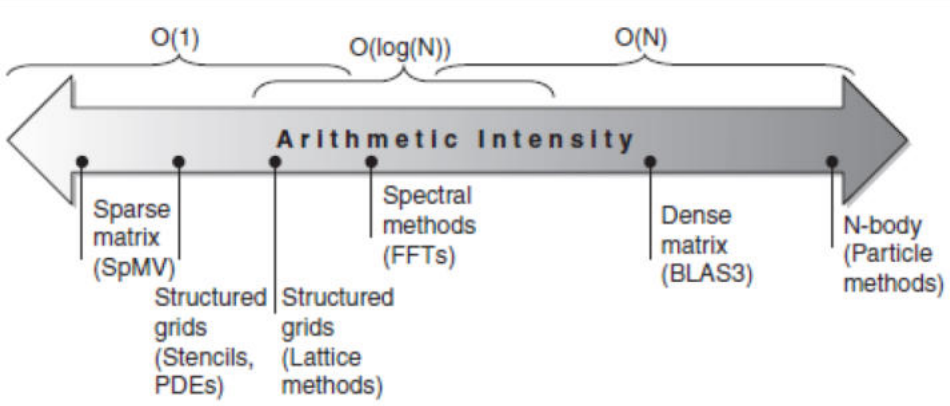


Figure 5.4 Arithmetic intensity, specified as the number of floating-point operations to run the program divided by the number of bytes accessed in main memory

Peak floating-point performance can be found using the hardware specifications. Figure 5.5 shows the Roofline model for the NEC SX-9 vector processor on the left and the Intel Core i7 920 multicore computer on the right. The vertical Y-axis is achievable floating-point performance from 2 to 256 GFLOP/sec. The horizontal X-axis is arithmetic intensity, varying from 1/8th FLOP/DRAM byte accessed to 16 FLOP/ DRAM byte accessed in both graphs. Note that the graph is a log–log scale, and that Rooflines are done just once for a computer.

Since the X-axis is FLOP/byte and the Y-axis is FLOP/sec, bytes/sec is just a diagonal line at a 45-degree angle in this figure.

Attainable GFLOPs/sec = Min(Peak Memory BW × Arithmetic Intensity, Peak Floating-Point Perf.)
The horizontal and diagonal lines give this simple model its name and indicate its value.

The ―Roofline‖ sets an upper bound on performance of a kernel depending on its arithmetic intensity.

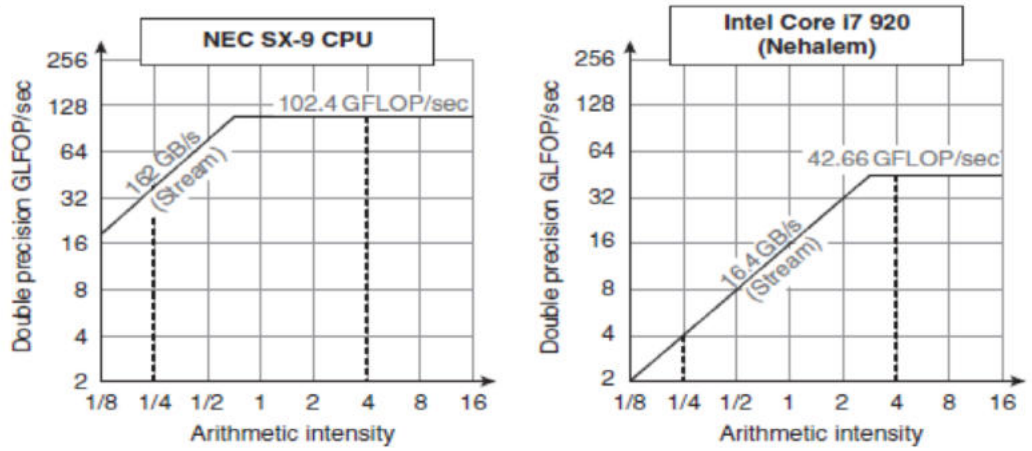


Figure 5.5 Roofline model

Note that the ―ridge point,‖ where the diagonal and horizontal roofs meet, offers an interesting insight into the computer. If it is far to the right, then only kernels with very high arithmetic intensity can achieve the maximum performance of that computer. If it is far to the left, then almost any kernel can potentially hit the maximum performance.

5.4 Graphics Processing Units

A specialized circuit designed to rapidly manipulate and alter memory. The challenge for the GPU programmer is not simply getting good performance on the GPU, but also in coordinating the scheduling of computation on the system processor and the GPU and the transfer of data between system memory and GPU memory.

GPUs have virtually every type of parallelism that can be captured by the programming environment: multithreading, MIMD, SIMD, and even instruction-level.

NVDIA Develop a C-like programming language for GPU called Compute Unified Device Architecture (CUDA). A similar programming language is OpenCL, which several companies are developing to offer a vendor-independent language for multiple platforms. And Unifying all forms of GPU parallelism as *CUDA thread.*

NVIDIA classifies the CUDA programming model as Single Instruction, Multiple Thread (*SIMT*). Threads that are blocked together and executed in groups of 32 threads, called a *Thread Block*. The hardware that executes a whole block of threads is a *multithreaded SIMD Processor.* A thread is associated with each data element. *CUDA threads*, with thousands of which being utilized to

various styles of parallelism: multithreading, SIMD, MIMD, ILP. Threads are organized into blocks.

Thread Blocks: groups of up to 512 elements

Multithreaded SIMD Processor: hardware that executes a whole thread block (32 elements executed per thread at a time)

5.4.1 NVIDIA GPU Computational Structures

A ***Grid*** is the code that runs on a GPU that consists of a set of *Thread Blocks*. Blocks are organized into a grid. The GPU code that works on the whole 8192 element multiply is called a ***Grid* (or vectorized loop).** To break it down into more manageable sizes, a Grid is composed of ***Thread Blocks***. Blocks are executed independently and in any order. Different blocks cannot communicate directly but can *coordinate* using atomic memory operations in Global Memory. ***The Grid and Thread Block*** are programming abstractions implemented in GPU hardware that help programmers organize their CUDA code. (The Thread Block is analogous to a stripminded vector loop with a vector length of 32.)

A Thread Block is assigned to a processor that executes that code, which we call a *multithreaded SIMD Processor*, by the *Thread Block Scheduler*. The Thread Block Scheduler has some similarities to a control processor in a vector architecture. It determines the number of thread blocks needed for the loop and keeps allocating them to different multithreaded SIMD Processors until the loop is completed.

Figure 5.6 shows a simplified block diagram of a multithreaded SIMD Processor. It is similar to a Vector Processor, but it has many parallel functional units instead of a few that are deeply pipelined, as does a Vector Processor. Each multithreaded SIMD Processor is assigned 512 elements of the vectors to work on. SIMD Processors are full processors with separate PCs and are programmed using threads.

The GPU hardware then contains a collection of multithreaded SIMD Processors that execute a Grid of Thread Blocks (bodies of vectorized loop); that is, a GPU is a multiprocessor composed of multithreaded SIMD Processors. The first four implementations of the Fermi architecture have 7, 11, 14, or 15 multithreaded SIMD Processors; future versions may have just 2 or 4. To provide transparent scalability across models of GPUs with differing number of multithreaded SIMD

Processors, the Thread Block Scheduler assigns Thread Blocks (bodies of a vectorized loop) to multithreaded SIMD Processors.

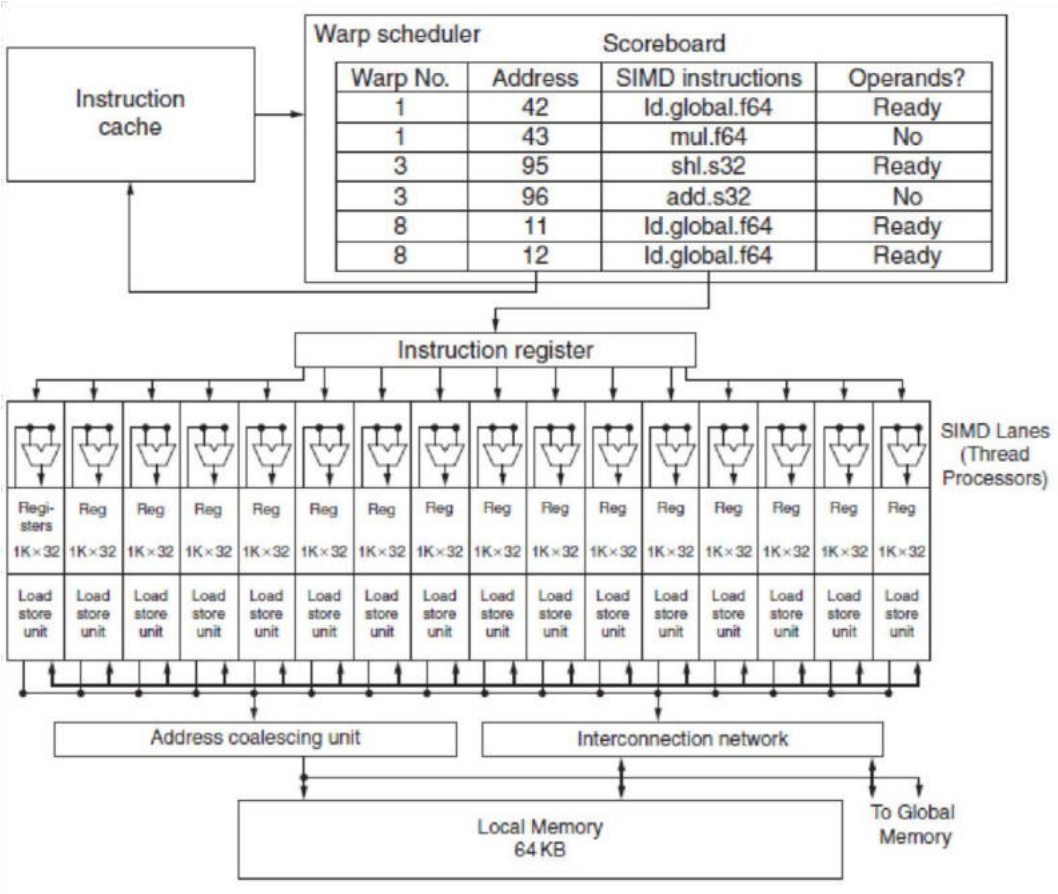


Figure 5.6 Simplified block diagram of a Multi threaded SIMD Processor

Dropping down one more level of detail, the machine object that the hardware creates, manages, schedules, and executes is a ***thread of SIMD instructions***. It is a traditional thread that contains exclusively SIMD instructions. The ***SIMD Thread Scheduler*** includes a scoreboard that lets it know which threads of SIMD instructions are ready to run, and then it sends them off to a dispatch unit to be run on the multithreaded SIMD Processor. GPU hardware has two levels of hardware schedulers:

(1) ***Thread Block Scheduler***

(2) ***SIMD Thread Scheduler within a SIMD Processor***

The Thread Block Scheduler that assigns Thread Block to multithreaded SIMD Processors, which ensures that thread blocks are assigned to the processors whose local memories have the corresponding data. The SIMD Thread Scheduler within a SIMD Processor, which schedules when threads of SIMD instructions should run.

The SIMD instructions of these threads are 32 wide, so each thread of SIMD instructions in this example would compute 32 of the elements of the computation. In this example, Thread Blocks would contain 512/32 = 16 SIMD threads.

Figure 5.7 shows the SIMD Thread Scheduler picking threads of SIMD instructions in a different order over time. The assumption of GPU architects is that GPU applications have so many threads of SIMD instructions that multithreading can both hide the latency to DRAM and increase utilization of multithreaded SIMD Processors.

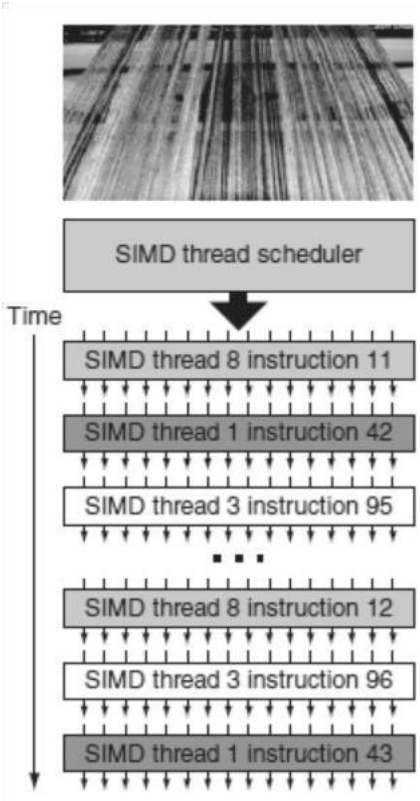


Figure 5.7 Scheduling of threads of SIMD instructions

5.4.2 NVIDA GPU Instruction Set Architecture

PTX (*Parallel Thread Execution*) provides a stable instruction set for compilers as well as compatibility across generations of GPUs. The hardware instruction set is hidden from the

programmer. PTX instructions describe the operations on a single CUDA thread,and usually map one-to-one with hardware instructions, but one PTX can expand to many machine instructions, and vice versa. The format of a PTX instruction is

opcode.type d, a, b, c;

where d is the destination operand; a, b, and c are source operands; and the operation type is one of the following:

Type	.type Specifier
Untyped bits 8, 16, 32, and 64 bits	.b8, .b16, .b32, .b64
Unsigned integer 8, 16, 32, and 64 bits	.u8, .u16, .u32, .u64
Signed integer 8, 16, 32, and 64 bits	.s8, .s16, .s32, .s64
Floating Point 16, 32, and 64 bits	.f16, .f32, .f64

Source operands are 32-bit or 64-bit registers or a constant value. Destinations are registers, except for store instructions.

5.4.3 Conditional Branching in GPUs

GPU branch hardware uses internal masks, a branch synchronization stack, and instruction markers to manage when a branch diverges into multiple execution paths and when the paths converge.

At the GPU hardware instruction level, control flow includes branch, jump, jump indexed, call, call indexed, return, exit, and special instructions that manage the branch synchronization stack. GPU hardware provides each SIMD thread with its own stack; a stack entry contains an identifier token, a target instruction address, and a target thread-active mask.

The PTX assembler typically optimizes a simple outer-level IF/THEN/ELSE statement coded with PTX branch instructions to just predicated GPU instructions, without any GPU branch instructions.

The PTX assembler identifies loop branches and generates GPU branch instructions that branch to the top of the loop, along with special stack instructions to handle individual lanes breaking out of the loop and converging the SIMD Lanes when all lanes have completed the loop. GPU indexed jump and indexed call instructions push entries on the stack so that when all lanes complete the switch statement or function call the SIMD thread converges.

The PTX branch instruction then depends on that predicate. If the PTX assembler generates predicated instructions with no GPU branch instructions, it uses a per-lane predicate register to enable or disable each SIMD Lane for each instruction. The SIMD instructions in the threads inside the THEN part of the IF statement broadcast operations to all the SIMD Lanes. Those lanes with the predicate set to one perform the operation and store the result, and the other SIMD Lanes don't perform an operation or store a result.

For the ELSE statement, the instructions use the complement of the predicate (relative to the THEN statement), so the SIMD Lanes that were idle now perform the operation and store the result while their formerly active siblings don't. At the end of the ELSE statement, the instructions are unpredicated so the original computation can proceed. Thus, for equal length paths, an IF-THEN-ELSE operates at 50% efficiency.

Also uses Branch synchronization stack so that the entries consist of masks for each SIMD lane i.e. which threads commit their results.

5.4.4 NVIDIA GPU Memory Structures

Figure 5.8 shows the memory structures of an NVIDIA GPU. Each SIMD Lane in a multithreaded SIMD Processor is given a private section of off-chip DRAM called *Private*

Memory. It is used for the stack frame, for spilling registers, and for private variables that don't fit in the registers. SIMD Lanes do not share Private Memories. Recent GPUs cache this Private Memory in the L1 and L2 caches to aid register spilling and to speed up function calls.

Each multithreaded SIMD processor also has local memory that is *on-chip.* Shared by SIMD lanes / threads *within a block only. The off-chip memory shared by SIMD processors is GPU Memory. Host can read and write GPU memory.*

The system processor, called the host, can read or write GPU Memory. Local Memory is unavailable to the host. GPUs traditionally use smaller streaming caches and rely on extensive multithreading of threads of SIMD instructions to hide the long latency to DRAM, since their working sets can be hundreds of megabytes. Caches also save energy, since on-chip cache accesses take much less energy than accesses to multiple, external DRAM chips. The GPU memory controller will also hold requests and send ones to the same open page together to improve memory bandwidth

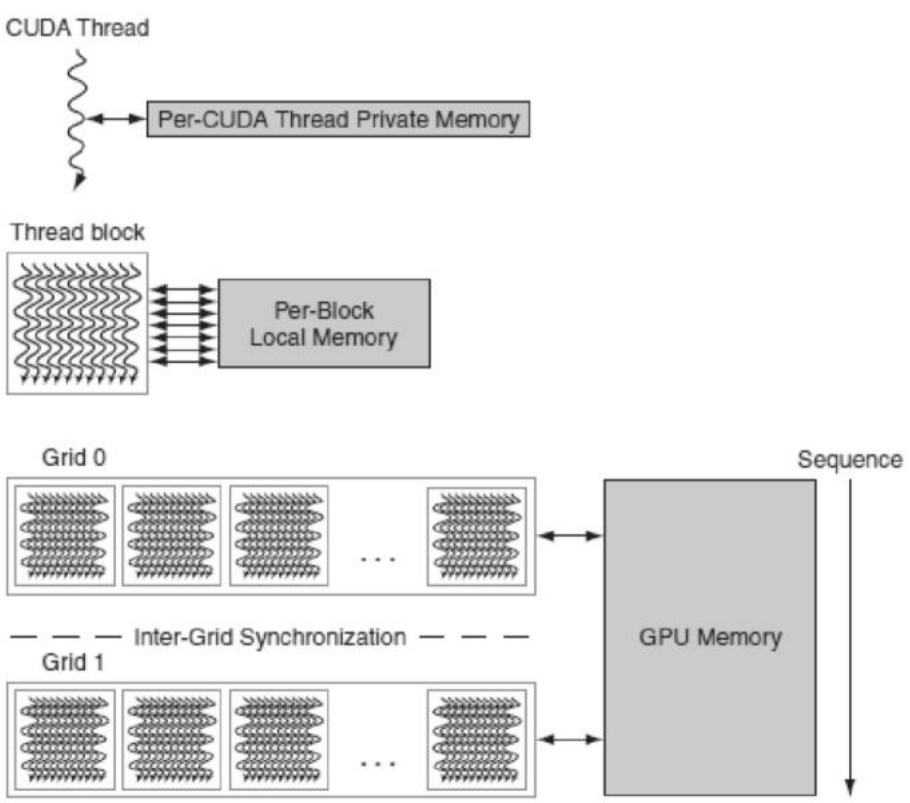


Figure 5.8 GPU Memory structures

5.4.5 Innovations in the Fermi GPU Architecture

Figure 5.9 shows the Dual Scheduler issuing instructions and Figure 5.10 shows the block diagram of the multithreaded SIMD Processor of a Fermi GPU.

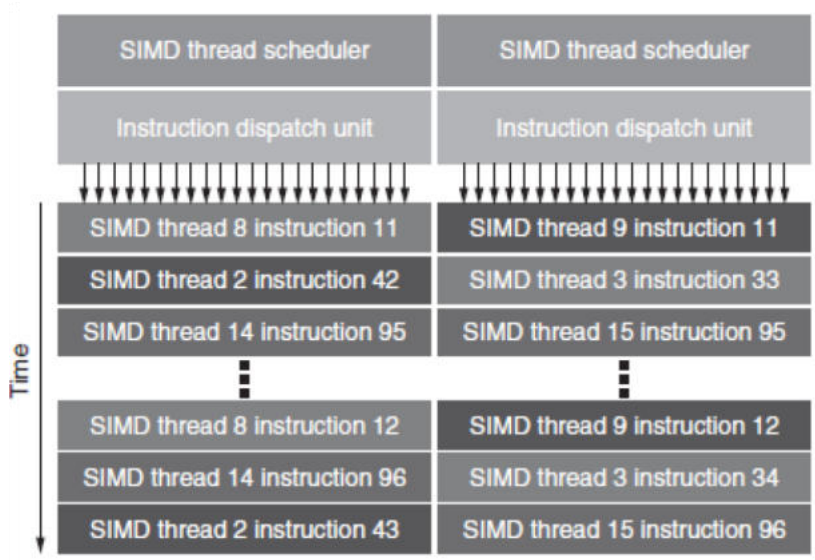


Figure 5.9 Block Diagram of Fermi's Dual SIMD Thread Scheduler

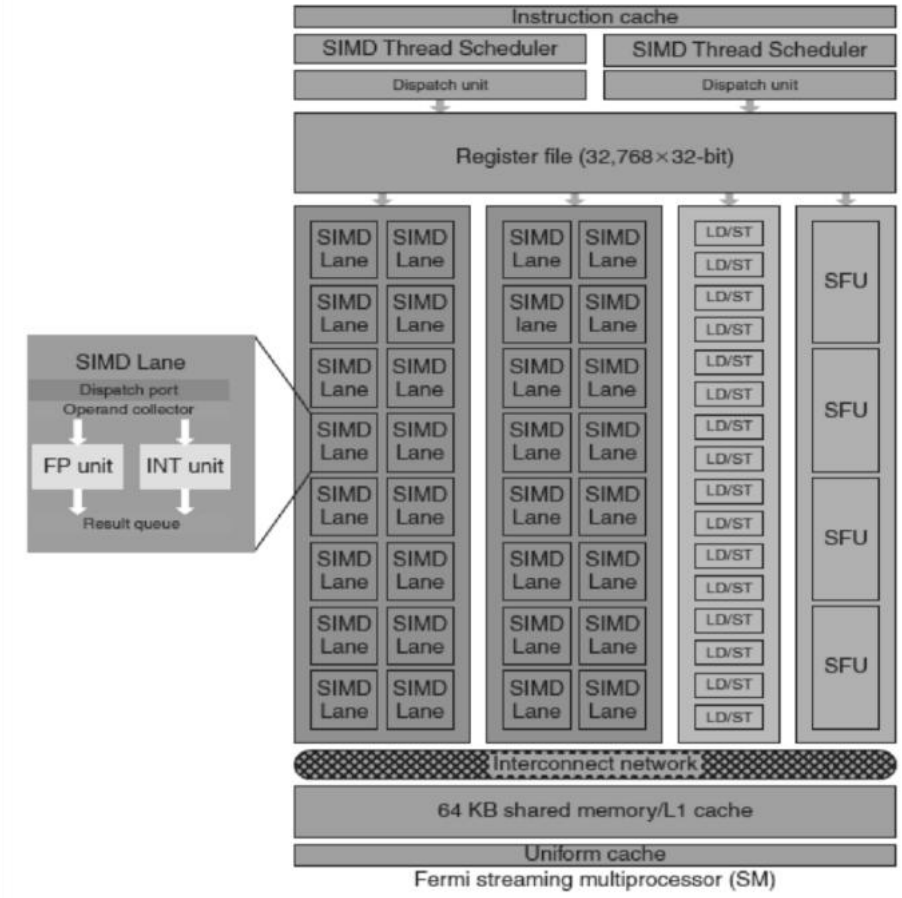


Figure 5.10 Block diagram of the multithreaded SIMD Processor of a Fermi GPU Fermi introduces several innovations to bring GPUs much closer to mainstream system processors. They are:

Fast Double-Precision Floating-Point Arithmetic - The peak double-precision performance grew from 78 GFLOP/sec in the predecessor GPU to 515 GFLOP/sec when using multiply-add instructions.

Caches for GPU Memory - Fermi includes both an L1 Data Cache and L1 Instruction Cache for each multithreaded SIMD Processor and a single 768 KB L2 cache shared by all multithreaded SIMD Processors in the GPU.

64-Bit Addressing and a Unified Address Space for All GPU Memories - provide the pointers needed for C and C++.

Error Correcting Codes to detect and correct errors in memory and registers.

Faster Context Switching - Fermi can switch in less than 25 microseconds, about 10× faster than its predecessor can.

Faster Atomic Instructions – Fermi improves performance of Atomic instructions by 5 to 20×, to a few microseconds.

5.5 GPGPU Computing

General-purpose computing on graphics processing units (**GPGPU**, rarely **GPGP**) is the use of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU)

- GPUs are high performance many-core processors capable of very high computation and data throughput.
- Once specially designed for computer graphics and difficult to program, todays GPUs are general purpose parallel processors with support for accessible programming interfaces and industry standard languages such as C.
- GPGPU is the usage of a GPU to perform general-purpose scientific computation
- The model for GPU computing is to use a CPU and GPU(s) together in a heterogeneous co-processing computing architecture
- The serial part of the program is left to the CPU, the parallel one is deployed on the GPU

GPGPU technology is simple – it will increase the speed of many types of tasks consumers do everyday by using the GPU and CPU in tandem for ―general purpose‖ computations. When this technology full matures, consumers will see noticeable performance increases when they convert audio, video files, play graphics-intensive games and in other daily tasks.

The first GPGPU was initially created in 1978 (a programmable raster display system). Before 2006, there were only a handful of other systems that incorporated GPGPU technology. In November 2006, AMD's Website stated they started the GPGPU revolution with the first iteration of their GPGPU technology that has now evolved into ATI stream.

CPU vs GPU CPU:

- A few out of order cores with huge caches
- Sequential Computation

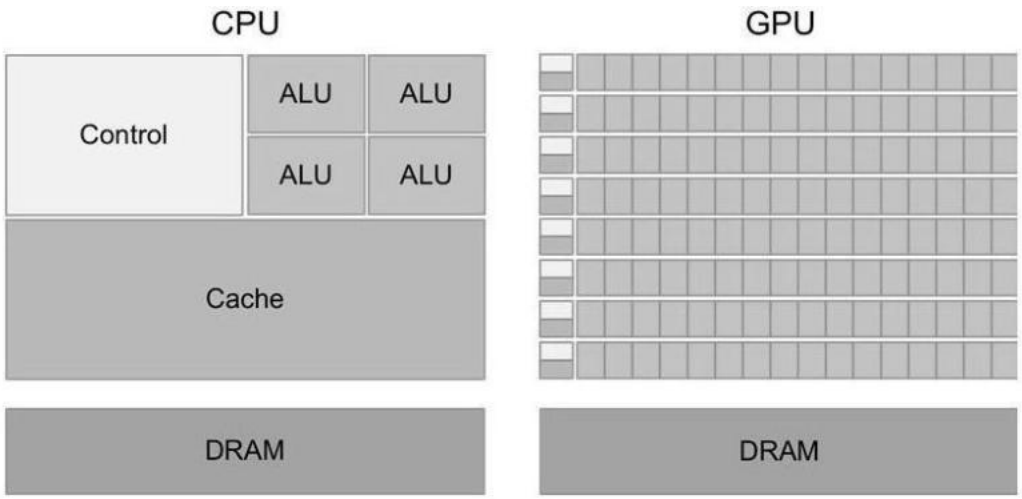


Figure 5.11 Design differences between GPU and CPU

GPU:

- Many in-order cores
- Massively parallel computation

GPUs are designed to solve problems that can be formulated as data-parallel computations – the same instructions are executed in parallel on many data elements with a high ratio between arithmetic operations and memory accesses. This is similar with the SIMD approach of the parallel computers taxonomy. Because the same instructions are executed on each of the data element there is no need for complicated flow control circuits and the memory latency can be hidden by arithmetic computations instead of using data cache.

The CUDA parallel programming model has three main key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization. These abstractions are exposed to the programmer as language extensions. They provide fine grain data parallelism and thread parallelism together with task parallelism that can be considered coarse grain parallelism.

The CUDA parallel programming model requires programmers to partition the problem to be solved into coarse tasks that can be independently executed in parallel by blocks of threads and each task is further divided into finer pieces of code that can be executed cooperatively in parallel by the threads within the block. This model allows threads to cooperate when solving each task, and also enables automatic scalability. Each block of threads can be scheduled for execution on any of the available processor cores, concurrently or sequentially. This allows a CUDA program to be executed on any number of processor cores.

Differences:

	GPU / GPGPU	**CPU**
Memory Latency	Context Switching automatically done in HW (Zero cycle Latency) GPU can switch to different thread until data returns from memory	Context Switching automatically done in SW (by OS) Stalls if there is an L1/L2 cache miss. Creates bubbles during program execution on cache misses

Thread management	Thread scheduler and dispatch unit implemented in HW Solving problems with massively parallel data such as — Sliding window‖ operations like FIRs, convolutions and other signal processing algorithms are more efficient on the GPU.	Thread overhead and work-item creation done in SW. Maximum threads is limited compared to GPU
Parallelism	HW designed to facilitate parallel operations on all shaders	CPU/SW and multicore CPU communication overhead exists. CPU thread overhead is heavier than equivalent GPU thread overhead

5.6 Detecting and Enhancing Loop Level Parallelism

Loop-level parallelism is normally analyzed at the source level. Instruction Level Parallelism is done once instructions have been generated by the compiler.

Loop Carried Dependence focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations.

Example:

for (i=999; i>=0; i=i-1) x[i] = x[i]

+ s;

- the two uses of x[i] are dependent
- But this dependence is within a single iteration and is not loop carried

Finding loop-level parallelism involves recognizing structures such as loops, array references, and induction variable computations, the compiler can do this analysis more easily at or near the source level, as opposed to the machine-code level.

Consider a loop like this one: ***for (i=0; i<100; i=i+1) { A[i+1] = A[i] + C[i]; /* S1 */***

B[i+1] = B[i] + A[i+1]; /* S2 */

}

Assume that A, B, and C are distinct, nonoverlapping arrays. There are two different dependences:

1. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1], which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].

2. S2 uses the value A[i+1] computed by S1 in the same iteration

These two dependences are different and have different effects.

- A[i+1] is dependent on earlier iteration A[i] (i.e) Loop Dependence and is loop carried
- B[i+1] is dependent on earlier iteration B[i] (i.e) Loop Dependence
- B[i+1] is dependent on A[i+1] is within an iteration so not loop carried.

It is also possible to have a loop-carried dependence that does not prevent parallelism, as the next example shows.

Consider a loop like this one: ***for (i=0; i<100; i=i+1) {***

A[i] = A[i] + B[i]; /* S1 */

B[i+1] = C[i] + D[i]; /* S2 */

}

Statement S1 uses the value assigned in the previous iteration by S2, so there is a loopcarried dependence between S2 and S1. However, this can be made parallel, because the dependence is not circular — neither statement depends on itself, and only S1 depends on S2, not the other way around.

A loop is parallel if it can be written without a cycle in the dependencies: the absence of a cycle means that the dependences give a partial ordering on the statements. We do have to make this loop conform to the parallel ordering to expose the parallelism. We make two observations:

1. There is no dependence from S1 to S2. If there were, then this would be a cycle in the dependencies, and the loop would not be parallel. Since this other dependency is absent, we can exchange the order of the statements, which won't affect the execution of S2.

2. On the first iteration of the loop, S1 depends on the value of B[0], which was computed prior to the loop.

With these two observations, we can replace the loop above with the following code:

```
A[0] = A[0] + B[0];    for (i=0; i<99; i=i+1) {          B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
    }
    B[100] = C[99] + D[99];
```

Often loop-carried dependences are in the form of a ***recurrence***. A recurrence occurs when a variable is defined based on the value of that variable in an earlier iteration, often the one immediately preceding, as in the following code fragment:

```
for (i=1;i<100;i=i+1) {
Y[i] = Y[i-1] + Y[i];
}
```

Detecting a recurrence can be important for two reasons: Some architectures (especially vector computers) have special support for executing recurrences, and, in an ILP context, it may still be possible to exploit a fair amount of parallelism.

5.6.1 Finding Dependences

Finding the dependences in a program is important both to determine which loops might contain parallelism and to eliminate name dependences. And also important for good scheduling for code.

In simplest terms, a one-dimensional array index is affine if it can be written in the form $a \times i + b$, where a and b are constants and i is the loop index variable. The index of a multidimensional array is affine if the index in each dimension is affine.

To determine whether there is a dependence between two references, to the same array in a loop is equivalent to determining whether two affine functions can have the same value of different indices between the bounds of the loop.

Example:

Say we have stored to an array element with index value a × i + b and loaded from an array element with the index value c × i + d where i is the for-loop index that runs from m to n. Dependence exists if two conditions hold:

1. There are two iteration indexes, j and k, that are both within the limits of the for loop. E.g., m ≤ j ≤ n, m ≤ k ≤ n.
2. The loop stores into an array element indexed by a × j + b and later fetches from the same array element when indexed by c × k + d. In other words, a × j + b = c × k + d

In general, we can't determine this at compile time. We might not know a, b, c, and d (they could be values in other arrays, for instance)

However, luckily, most programs contain simple indexes where a, b, c, and d are all constants. We can actually come up with a test at compile time for dependence.

One test is the greatest common divisor (GCD) test. It is based on the following observation:

If a loop carried dependence exists, then GCD(c,a) must divide (d - b). (Recall: an integer, x, divides an integer, y, if we get an integer quotient when we do the division y/x and there is no remainder.)

Use the GCD test to determine whether dependencies exist in the following loop:

```
    for (i=0; i<100; i=i+1) {
X[2*i+3] = X[2*i] * 5.0;
    }
```

Given the values $a = 2$, $b = 3$, $c = 2$, and $d = 0$, then GCD(a,c) = 2, and $d - b = -3$. Since 2 does not divide −3, no dependence is possible.

We can unroll the loop to see that there isn't a dependence:

i==0, x[3] = x[0] * 5 i==1,

x[5] = x[2] * 5 i==2, x[7]

= x[4] * 5 … i==99,

x[201] = x[198] * 5;

The GCD test is sufficient to guarantee that no dependence exists, but there are cases where the GCD test succeeds but there aren't any dependences. The GCD test doesn't look at array bounds, for instance.

The following loop has multiple types of dependencies. Find all the true dependencies, output dependencies, and antidependences, and eliminate the output dependences and antidependences by renaming.

- A ***—true dependency‖*** is one that cannot be changed by renaming, although it may or may not limit the ability to parallelize (if it is loop carried, for instance).
- An ***—antidependency‖*** occurs when one instruction writes to a register or memory location that another instruction reads. The ordering is important for antidependences.
- An ***—output dependency‖*** occurs when two instructions write to the same register or memory location. Again, the ordering must be preserved to ensure that the value that is finally written is the one that happens last.

```
for (i=0; i<100; i=i+1) { Y[i] =
X[i] / c; /* S1 */
    X[i] = X[i] + c; /* S2 */
    Z[i] = Y[i] + c; /* S3 */
    Y[i] = c - Y[i]; /* S4 */
    }
```

Answer:

1. There are true dependencies from S1 to S3 and from S1 to S4 because of Y[i]. These aren't loop carried, so they do not prevent the loop from being parallelized. They will, however, force S3 and S4 to wait for S1 to complete.
2. There is an antidependence from S1 to S2, based on X[i]
3. There is an antidependence from S4 to S4 for Y[i]
4. There is an output dependence from S1 to S4, based on Y[i]

5.6.2 Eliminating Dependent Computations

Take a look at a common recurrence, the dot product:

for (i=9999; i>=0; i=i-1) { sum = sum + x[i] * y[i]; } This loop isn't parallel because there is a loop-carried dependence on the variable sum. We can transform it to a set of loops, to remove the dependence. One loop will be completely parallel, and the other loop will be partially parallel. Here is the completely parallel loop:

for (i=9999; i>=0; i=i-1) { sum[i] = x[i] * y[i]; }

The sum variable is now a vector (array) of its own, but each partial sum is independent. But, now we need to reduce this sum into a scalar value again:

for (i=9999; i>=0; i=i-1) { finalsum = finalsum + sum[i]; }

This is just a reduction. But not completely parallel, it can be partially parallelized, and still run with multiple processors. E.g., with 10 processors, each with their own processor number, p:

for (i=999; i>=0; i=i-1) {
finalsum[p] = finalsum[p] * sum[i+1000*p]; }

The one caveat to all of this: the transformation above relies on associativity of addition (i.e., the order of the addition operation doesn't matter).

This is true for numbers with unlimited range and precision, but not necessarily true on computers! In fact, a reduction done on a GPU with floating point values might be slightly different than the result done on a CPU in serial. This can make debugging particularly hard.