TOPIC1 PIPELINING

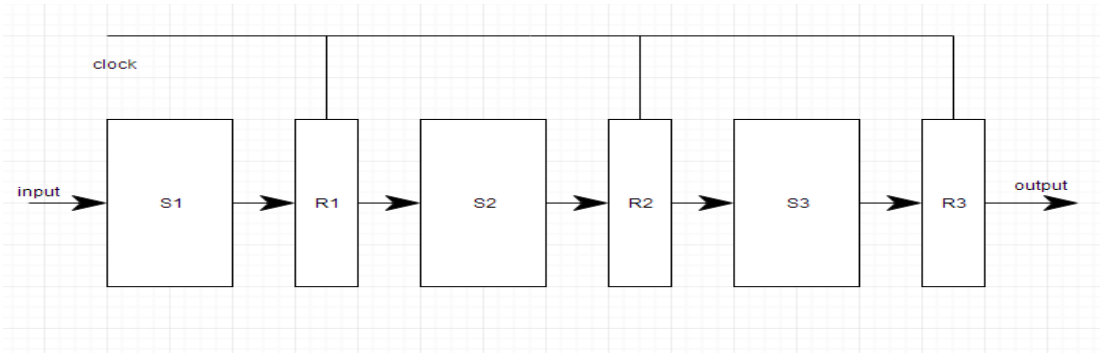
What is Pipelining?

Pipelining is the process of accumulating instruction from the processor through a pipeline. It allows storing and executing instructions in an orderly process. It is also known as **pipeline processing**.

Pipelining is a technique where multiple instructions are overlapped during execution. Pipeline is divided into stages and these stages are connected with one another to form a pipe like structure. Instructions enter from one end and exit from another end.

Pipelining increases the overall instruction throughput.

In pipeline system, each segment consists of an input register followed by a combinational circuit. The register is used to hold data and combinational circuit performs operations on it. The output of combinational circuit is applied to the input register of the next segment.



Pipeline system is like the modern day assembly line setup in factories. For example in a car manufacturing industry, huge assembly lines are setup and at each point, there are robotic arms to perform a certain task, and then the car moves on ahead to the next arm.

Types of Pipeline

It is divided into 2 categories:

1. Arithmetic Pipeline
2. Instruction Pipeline

Arithmetic Pipeline

Arithmetic pipelines are usually found in most of the computers. They are used for floating point operations, multiplication of fixed point numbers etc. For example: The input to the Floating Point Adder pipeline is:

```
X = A*2^a

Y = B*2^b
```

Here A and B are mantissas (significant digit of floating point numbers), while **a** and **b** are exponents.

The floating point addition and subtraction is done in 4 parts:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract mantissas
4. Produce the result.

Registers are used for storing the intermediate results between the above operations.

Instruction Pipeline

In this a stream of instructions can be executed by
overlapping *fetch*, *decode* and *execute* phases of an instruction cycle. This type of technique is used to increase the throughput of the computer system.

An instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipeline. Thus we can execute multiple instructions simultaneously. The pipeline will be more efficient if the instruction cycle is divided into segments of equal duration.

Pipeline Conflicts

There are some factors that cause the pipeline to deviate its normal performance. Some of these factors are given below

---

1. Timing Variations

All stages cannot take same amount of time. This problem generally occurs in instruction processing where different instructions have different operand requirements and thus different processing time.

2. Data Hazards
When several instructions are in partial execution, and if they reference same data then the problem arises. We must ensure that next instruction does not attempt to access data before the current instruction, because this will lead to incorrect results.

---

3. Branching

In order to fetch and execute the next instruction, we must know what that instruction is. If the present instruction is a conditional branch, and its result will lead us to the next instruction, then the next instruction may not be known until the current one is processed.

---

4. Interrupts

Interrupts set unwanted instruction into the instruction stream. Interrupts effect the execution of instruction.

5. Data Dependency
It arises when an instruction depends upon the result of a previous instruction but this result is not yet available.

Advantages of Pipelining

1. The cycle time of the processor is reduced.
2. It increases the throughput of the system
3. It makes the system reliable.

Disadvantages of Pipelining

1. The design of pipelined processor is complex and costly to manufacture.
2. The instruction latency is more.

TOPIC2 PIPELINE PERFORMANCE

PIPELINE PERFORMANCE •

Speedup from Pipelining = (Average Instruction Time Un-pipeli ned)/(Average Instruction Time Pipelined) = (CPI Un-pipelined)/(CPI Pipelined) x (Clock cycle Time Un-Pipelined)/(Clock Cycle Time Pipelined)

Where CPI Pipelined = 1 + Pipeline stall clo ck cycles p er instruction.

Now – Assuming Equal Cycle Time: Speedup = CPI Un-Pipelined / (1 + Pipeline stall cycles per Instruction)

Speedup = Pipeline Depth / 1 + Pipeline stall cycles per instruction

TOPIC3 Pipeline Hazards

Pipeline Hazards

There are situations, called **hazards**, that prevent the next instruction in the instruction stream from being executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining.

There are three classes of hazards:

A.**Structural Hazards.** They arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
B.**Data Hazards.** They arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline
C. **Control Hazards.**They arise from the pipelining of branches and other instructions that change the PC.

Hazards in pipelines can make it necessary to stall the pipeline. The processor can stall on different events:

- A ***cache miss***. A cache miss stalls all the instructions on pipeline both before and after the instruction causing the miss.
- A ***hazard in pipeline.*** Eliminating a hazard often requires that some instructions in the pipeline to be allowed to proceed while others are delayed. When the instruction is stalled, all the instructions issued *later* than the stalled instruction are also stalled. Instructions issued *earlier* than the stalled instruction must continue, since otherwise the hazard will never clear.

A hazard causes pipeline bubbles to be inserted.The following table shows how the stalls are actually implemented. As a result, no new instructions are fetched during clock cycle 4, no instruction will finish during clock cycle 8.

In case of structural hazards:

Clock cycle number

Instr	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	EX	MEM	WB				
Instr i+2			IF	ID	EX	MEM	WB			
Stall				bubble	bubble	bubble	bubble	bubble		
Instr i+3					IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB

To simplify the picture it is also commonly shown like this:

Clock cycle number

Instr	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	EX	MEM	WB				
Instr i+2			IF	ID	EX	MEM	WB			
Instr i+3				stall	IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB

In case of data hazards:

Clock cycle number

Instr	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	bubble	EX	MEM	WB			
Instr i+2			IF	bubble	ID	EX	MEM	WB		

Instr i+3	bubble	IF	ID	EX	MEM	WB	
Instr i+4			IF	ID	EX	MEM	WB

which appears the same with stalls:

	Clock cycle number									
Instr	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	stall	EX	MEM	WB			
Instr i+2			IF	stall	ID	EX	MEM	WB		
Instr i+3				stall	IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB

Performance of Pipelines with Stalls

A stall causes the pipeline performance to degrade the ideal performance.

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} = \frac{\text{CPI}_{\text{unpipelined}} * \text{Clock Cycle Time}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}} * \text{Clock Cycle Time}_{\text{pipelined}}}$$

The ideal CPI on a pipelined machine is almost always 1. Hence, the pipelined CPI is

$$\text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction} = 1 + \text{Pipeline stall clock cycles per instruction}$$

If we ignore the cycle time overhead of pipelining and assume the stages are all perfectly balanced, then the cycle time of the two machines are equal and

$$\text{Speedup} = \frac{\text{CPI}_{\text{unpipelined}}}{1 + \text{Pipeline stall cycles per instruction}}$$

If all instructions take the same number of cycles, which must also equal the number of pipeline stages (the depth of the pipeline) then unpipelined CPI is equal to the depth of the pipeline, leading to

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

If there are no pipeline stalls, this leads to the intuitive result that pipelining can improve performance by the depth of pipeline.

Structural Hazards

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

Common instances of structural hazards arise when

- Some functional unit is not fully pipelined. Then a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle
- Some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute.

Example1:
a machine may have only one register-file write port, but in some cases the pipeline might want to perform two writes in a clock cycle.

Example2:
a machine has shared a single-memory pipeline for data and instructions. As a result, when an instruction contains a data-memory reference(load), it will conflict with the instruction reference for a later instruction (instr 3):

	Clock cycle number							
Instr	1	2	3	4	5	6	7	8
Load	IF	ID	EX	MEM	WB			
Instr 1		IF	ID	EX	MEM	WB		
Instr 2			IF	ID	EX	MEM	WB	
Instr 3				IF	ID	EX	MEM	WB

To resolve this, we stall the pipeline for one clock cycle when a data-memory access occurs. The effect of the stall is actually to occupy the resources for that instruction slot. The following table shows how the stalls are actually implemented.

	Clock cycle number								
Instr	1	2	3	4	5	6	7	8	9
Load	IF	ID	EX	**MEM**	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Stall				bubble	bubble	bubble	bubble	bubble	
Instr 3					**IF**	ID	EX	MEM	WB

Instruction 1 assumed not to be data-memory reference (load or store), otherwise Instruction 3 cannot start execution for the same reason as above.

To simplify the picture it is also commonly shown like this:

	Clock cycle number								
Instr	1	2	3	4	5	6	7	8	9

Load	IF	ID	EX	MEM	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Instr 3				stall	IF	ID	EX	MEM	WB

Introducing stalls degrades performance as we saw before. Why, then, would the designer allow structural hazards? There are two reasons:

- To reduce cost. For example, machines that support both an instruction and a cache access every cycle (to prevent the structural hazard of the above example) require at least twice as much total memory.
- To reduce the latency of the unit. The shorter latency comes from the lack of pipeline registers that introduce overhead.

Data Hazards

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. This introduces data and control hazards. **Data hazards** occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on the unpipelined machine.

Consider the pipelined execution of these instructions:

	1	2	3	4	5	6	7	8	9
ADD **R1**, R2, R3	IF	ID	EX	MEM	**WB**				
SUB R4, R5, **R1**		IF	**ID$_{sub}$**	EX	MEM	WB			
AND R6, **R1,** R7			IF	**ID$_{and}$**	EX	MEM	WB		
OR R8, **R1**, R9				IF	**ID$_{or}$**	EX	MEM	WB	
XOR R10,**R1**,R11					IF	**ID$_{xor}$**	EX	MEM	WB

All the instructions after the ADD use the result of the ADD instruction (in R1). The ADD instruction writes the value of R1 in the WB stage (shown black), and the **SUB** instruction reads the value during ID stage (**ID$_{sub}$**). This problem is called ***a data hazard***. Unless precautions are taken to prevent it, the SUB instruction will read the wrong value and try to use it.

The **AND** instruction is also affected by this data hazard. The write of R1 does not complete until the end of cycle 5 (shown black). Thus, the AND instruction that reads the registers during cycle 4 (**ID$_{and}$**) will receive the wrong result.

The **OR** instruction can be made to operate without incurring a hazard by a simple implementation technique. ***The technique*** is to perform register file reads in the second half of the cycle, and writes in the first half. Because both **WB** for ADD and **ID$_{or}$** for OR are performed in one cycle 5, the write to register file by ADD will perform in the first half of the cycle, and the read of registers by OR will perform in the second half of the cycle.

The **XOR** instruction operates properly, because its register read occur in cycle 6 after the register write by ADD.

The next page discusses forwarding, a technique to eliminate the stalls for the hazard involving the SUB and AND instructions.

TOPIC5 Parallel processing

Parallel processing is an integral part of everyday life. The concept is so inbuilt in our existence that we benefit from it without realizing.

Hardware Architecture of Parallel Computer

The core element of parallel processing is CPUs. The essential computing process is the execution of sequence of instruction on asset of data. The term stream is used here to denote a sequence of items as executed by single processor or multiprocessor. Based on a number of

instruction and data streams can be processed simultaneously, Flyn's classified the parallel computer system into following categories [4, 7, 10, 14]:

- Single Instruction Single Data (SISD)
- Single Instruction Multiple Data (SIMD)
- Multiple Instruction Single Data (MISD)
- Multiple Instruction Multiple Data (MIMD)

Single Instruction Single Data (SISD)

The single processing element executes instructions sequentially on a single data stream. The operations are thus ordered in time and may be easily traced from start to finish. Modern adaptations of this uniprocessor use some form of pipelining technique to improve performance and, as demonstrated by the Cray supercomputers, minimise the length of the component interconnections to reduce signal propagation times

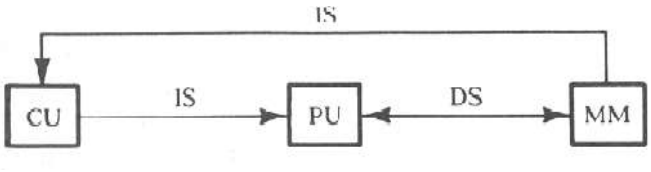


Figure 1.2 SISD Processor

Single Instruction Multiple Data (SIMD)

Machines apply a single instruction to a group of data items simultaneously. A master instruction is thus acting over a vector of

related operands. A number of processors, therefore, obey the same instruction in the same cycle and may be said to be executing in strict lock-step. Facilities exist to exclude particular processors from participating in a given instruction cycle.

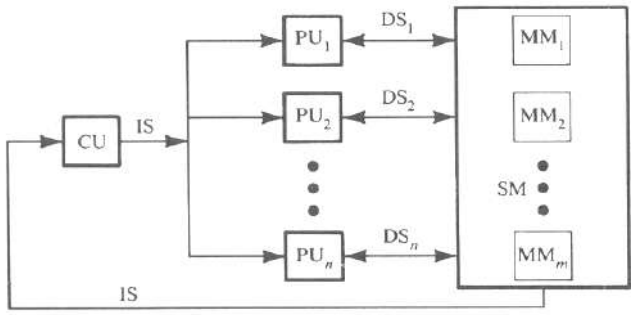


Figure 1.3 SIMD Processor

Multiple Instruction Single Data (MISD)

In this system there are n processor units, each receiving distinct instructions operating over the same data stream. The result of one processor becomes the input of the next processor. One the closest architecture to this concept is a pipelined computer. This structure has received much less attention and has no real implementation.

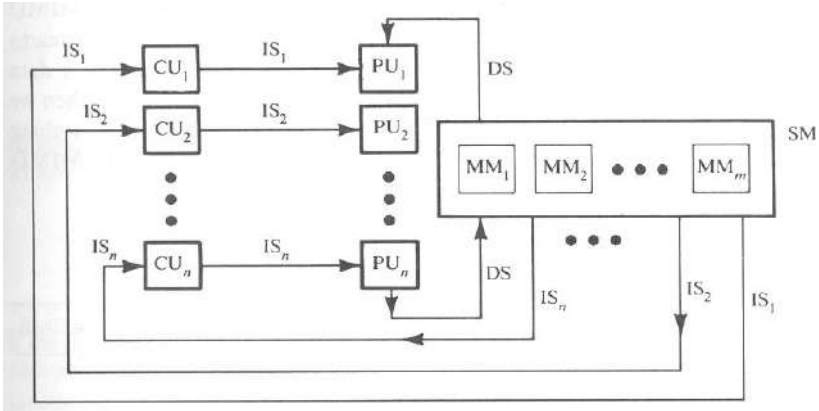


Figure 1.4 MISD Processor

Multiple Instruction Multiple Data (MIMD)

MIMD systems provide a separate set of instructions for each processor. This allows the processors to work on different parts of a problem asynchronously and independently. Such systems may consist of a number of interconnected, dedicated processor and memory nodes, or interconnected "stand-alone" workstations. The processors within the MIMD classification autonomously obey their own instruction sequence and apply these instructions to their own data. By providing these processors with the ability to communicate with each other, they may interact and therefore, co-operate in the solution of a single problem.

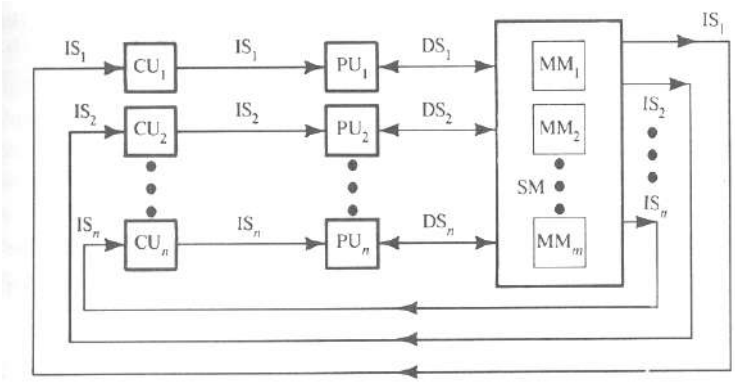


Figure 1.5 MIMD Processor

Memory Architecture of Parallel Computer

The primary memory architectures are:

- Shared memory
- Distributed memory
- Hybrid Distributed-Shared memory

Shared memory

In shared memory architecture multiple processors operate independently but share the same memory resources. Only one processor can access the shared memory location at a time.

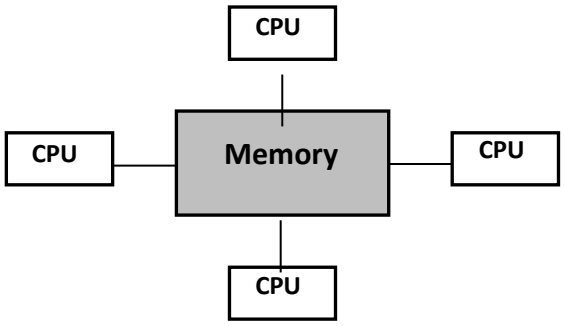


Figure 1.6 Shared Memory Architecture

Changes in a memory location effected by one processor are visible to all other processors. Synchronization is achieved by controlling tasks reading from and writing to the shared memory. Shared memory machines can be divided into two main classes based upon memory access times: **UMA** and **NUMA**.

Non-Uniform Memory Access (NUMA)

Non-Uniform Memory Access (NUMA) is designed to take the best attributes of MPP and SMP systems. NUMA based machines can be extremely cost effective and scalable while preserving the semantics of a shared memory Symmetric Multiprocessor: the NUMA architecture enables users to preserve their investments in SMP applications. NUMA is a means of implementing a

distributed, shared memory system that can make processor/memory interaction appear transparent to application software [2, 11, 19].

In a NUMA machine, physical memory is distributed amongst the computing nodes so that a small portion of the total machine memory is local to each. Since this is shared memory architecture, the remaining machine memory may be remotely accessed across an interconnection network. Local accesses have roughly the same latency as accesses in a SMP system, while remote data accesses can be orders of magnitude slower; hence the term "Non-Uniform Memory Access".

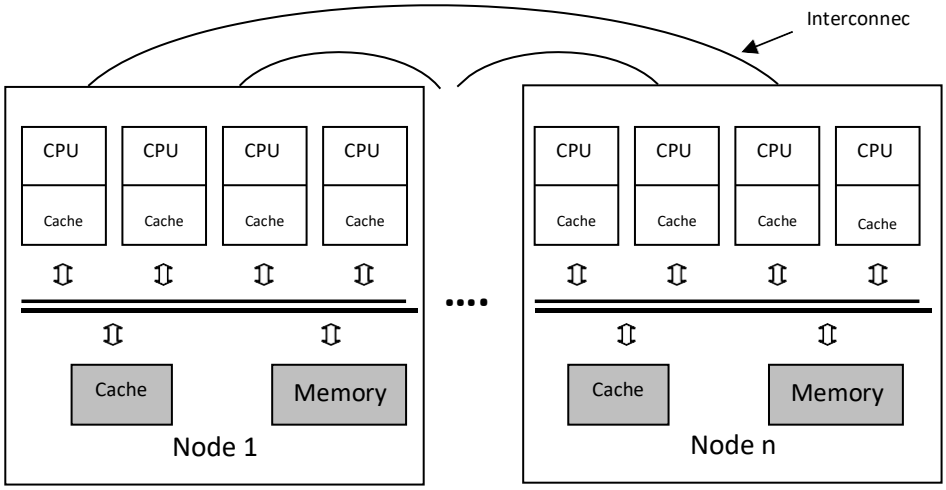


Figure 1.7 NUMA architecture

NUMA system is composed of nodes that are linked together via a fast interconnect, with each node containing small number of processors, its own bus, and its own physical memory. In most NUMA architectures each node has not only its own local memory, but also a local cache for storing local copies of recently accessed data that natively resides on other nodes (Figure 1.7). At the

hardware level, NUMA introduces the concept of “local memory” (which is memory that physically resides on that node) and “remote memory” (which is memory that physically resides on other nodes). To reduce the latency of accesses to both local and remotely held data, caches are used. However, this introduces the problem of keeping cached copies of data coherent when one node in the system modifies a data item. This task may fall to the memory system hardware: “Cache Coherent”-NUMA (cc-NUMA).

Uniform Memory Access (UMA)

Uniform Memory Access (UMA) is a shared memory architecture used in parallel computers. All the processors in the UMA model share the physical memory uniformly.

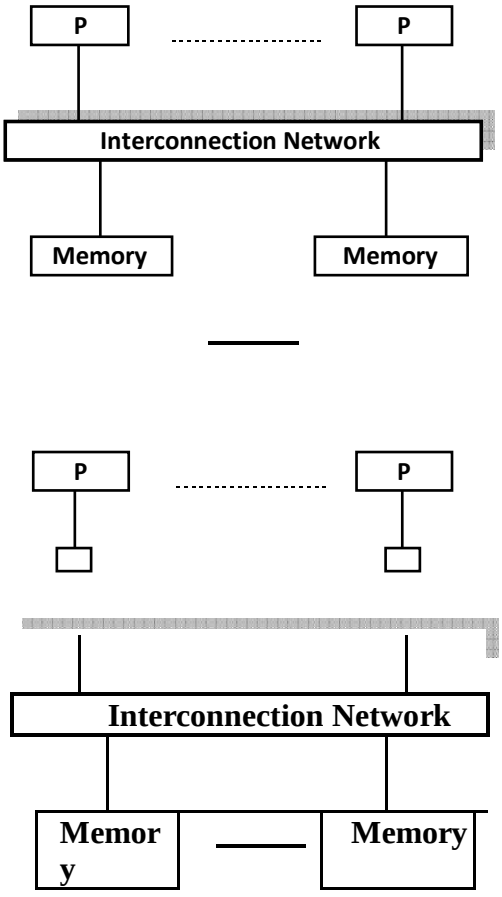


Figure 1.8 UMA architecture

In UMA architecture, access time to a memory location is independent of which processor makes the request or which memory chip contains the transferred data [2, 14, 15]. In the UMA architecture, each processor may use a private cache. Peripherals are also shared in some fashion; The UMA model is suitable for general purpose and time sharing applications by multiple users. It can be used to speed up the execution of a single large program in time critical applications.

Cache only Memory Architecture (COMA)

It is Just like as NUMA but distributed memory is converted to caches. There is no memory hierarchy at each processor node. All the caches form a global address space. In NUMA, each address in the global address space is typically assigned a fixed home node. When processors access some data, a copy is made in their local cache, but space remains allocated in the home node. Instead, with COMA, there is no home [2, 11, 14]. An access from a remote node may cause that data to migrate.

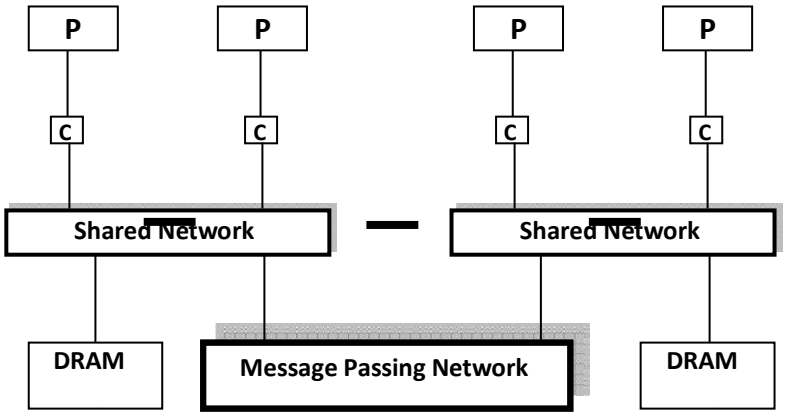


Figure 1.9 COMA architecture

Compared to NUMA, this reduces the number of redundant copies and may allow more efficient use of the memory resources. On the other hand, it raises problems of how to find a particular data and what to do if a local memory fills up. Hardware memory coherence mechanisms are typically used to implement the migration. When all copies are kept consistent in COMA model then it is **called Cache- Coherent COMA.**

TOPIC6 Cache Coherence

What is a Cache Coherence Problem?

Cache coherence is a concern raised in a multi-core system distributed L1 and L2 caches. Each core has its own L1 and L2 caches and they need to always be in-sync with each other to have the most up-to-date version of the data.

The Cache Coherence Problem is the challenge of keeping multiple local caches synchronized when one of the processors updates its local copy of data which is shared among multiple caches.

Imagine a scenario where multiple copies of same data exists in different caches simultaneously, and if the processors are allowed to update their own copies freely, an inconsistent view of memory can result.

For example, imagine a dual-core processor where each core brought a block of memory into its private cache, and then one core writes a value to a specific location. When the second core attempts to read that value from its cache, it will not have the most recent version unless its cache entry is invalidated and a cache miss occurs. This cache miss forces the second core's cache entry to be updated. To trigger the cache invalidation we need cache coherence policies. If there is no cache coherence policy in-place, the wrong data would be read and invalid results would be produced, possibly crashing the program.

Cache Write Policies

There two main cache write policies.

- ***Write back*** : Write operations are usually made only to the cache. Main memory is only updated when the corresponding cache line is flushed from the cache.
- ***Write through*** : All write operations are made to main memory as well as to the cache, ensuring that main memory is always valid.

From the above description it is clear that ***Write back policy*** results in inconsistency. If two caches contain the same line, and the line is updated in one cache, the other cache will unknowingly have an invalid value. Subsequently read to that invalid line produce invalid results.

But if we think deeper even the ***Write through policy*** also has consistency issues. Even though memory is updated inconsistency can occur unless other cache monitor the memory traffic or receive some direct notification of the update.

In order to solve these issue we introduce **Cache Coherency Protocols**. *Objective of any cache coherency protocol is to load the recently used local variables into the appropriate caches and keep them through numerous reads and writes, while using the protocol to maintain consistency of shared variables that might be in multiple caches at the same time.*

First let's see what are the common ways of writing into a Cache before we jump into the solutions to the Cache Coherence Problem.

Write Through (WT) Protocol

There are two fundamental implementations of the WT protocol.

- **Write through with update protocol**
- **Write through with invalidation of copies**

Write through with update protocol

When a processor writes a new value into its cache, the new value is also written into the memory module that holds the cache block being changed. Some copies of this block may exist in other caches, these copies must be updated to reflect the change caused by the write operation.

We update the other cache copies by doing a broadcast with the updated data to all processor modules in the system. Each processor module receives the broadcast data, it updates the contents of the affected cache block if this block is present in its cache.

Write through with invalidation of copies

When a processor writes a new value into its cache, this value is written into the memory and all other copies in other caches are invalidated. This also done by broadcasting the invalidation request through the system. All caches receives this invalidation request and the cache which contains the updated data flushes its cache line.

Write Back (WB) Protocol

In the WB protocol, multiple copies of a cache block may exist if different processors have loaded (read) the block into their caches.

In this approach if some processor wants to change this block, it must first become the exclusive owner of the block.

When the ownership is granted to this processor by the memory module that is the home location of the block. All other copies, including the one in the memory module, are invalidated.

Now the owner of the block may change the contents of the memory.

When another processor wishes to read this block, the data are sent to this processor by the current owner. The data are also sent to the home memory module, which requires ownership and updates the block to contain the latest value.