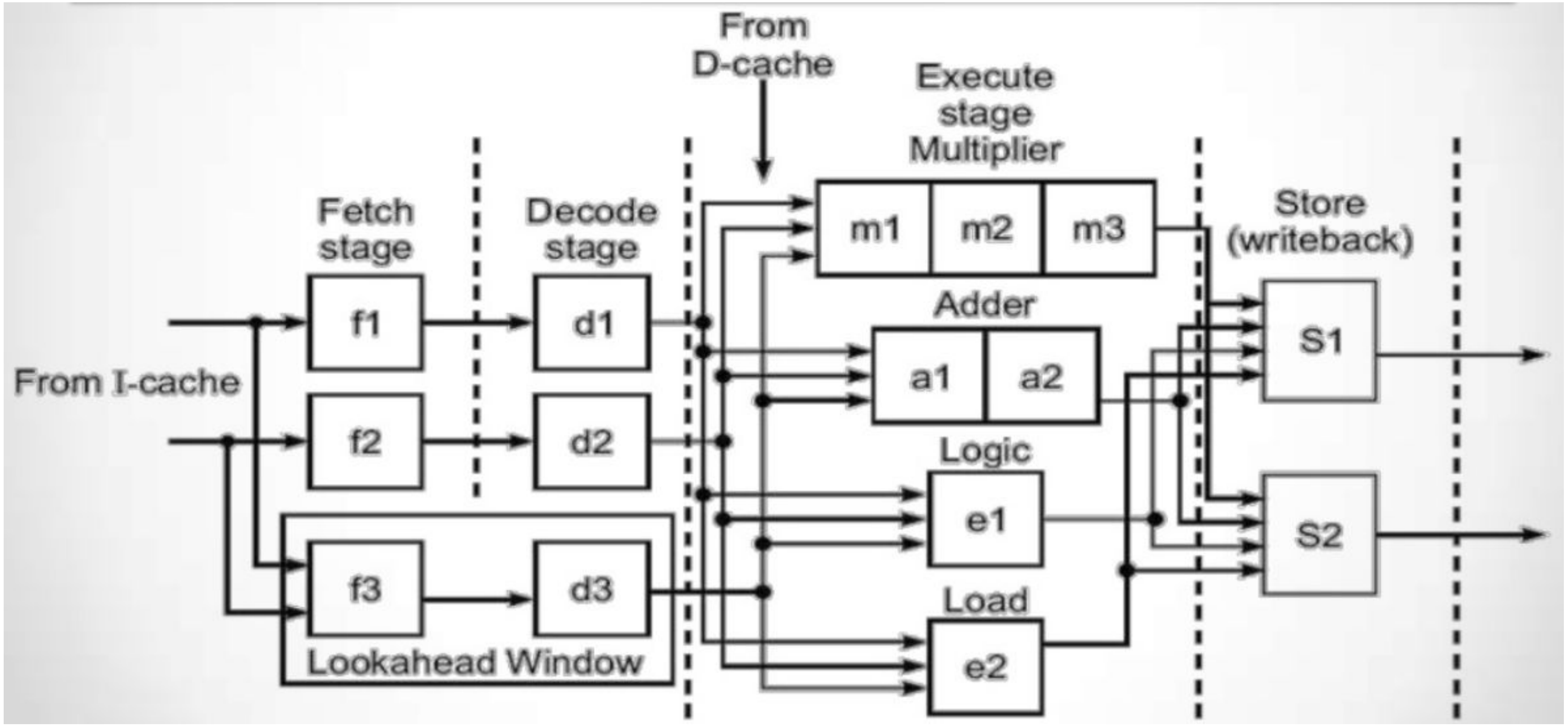
Superscalar Pipeline Structure

DANY JOY
21
S7 CSE

Superscalar Pipeline Structure

- In an m-issue superscalar processor, the instruction decoding and execution resources are increased to form effectively m pipelines operating concurrently. At some pipeline stages, the functional units may be shared by multiple pipelines. This resource shared multiple pipeline structure is illustrated in next page.
- The process can issue two instructions per cycle if there is no resource conflict and no data dependence problem.There are essentially two pipelines in the design.Both having four processing stages labelled fetch,decode,execute and store respectively.

A dual-pipeline Superscalar Processor



- Each pipeline essentially has its own fetch unit,and store unit.The two instruction streams flowing the two pipelines are retrieved from a single source stream (the I cache).
- The fan out from a single instruction stream is subject to resource constraints and a data dependence relationship among the successive instructions.
- For simplicity,we assume that each pipeline stage requires one cycle,expect the execute stage which may require a variable number of cycles.
- Four functional units,multiplier,adder,logic unit,and load unit , are available for use in the execute stage.

- The multiplier itself has 3 pipeline stages,the adder has 2 stages, and the others each have only one stage.
- The 2 store units (s1,s2) can be dynamically used by the 2 pipelines,depending on availability at a particular cycle.
- There is a lookahead window with its own fetch and decoding logic.this window is used for instruction lookahead in case out of order instruction issue is desired to achieve better pipeline throughput.
- It requires complex logic to schedule multiple pipelines simultaneously,especially when the instructions are retrieved from same source.The aim is to avoid pipeline stalling and minimize pipeline idle time.

Superscalar Pipeline Stalling

BY,

ELDHO SHAJU

S7 CSE 22

ABOUT

- A superscalar processo**r** is a CPU that implements a form of parallelism called instruction-level parallelism within a single processor. While a superscalar CPU is typically also pipelined, superscalar and pipelining execution are considered different performance enhancement techniques.

Superscalar Processor

- Superscalar processors are designed to exploit more instruction-level parallelism in user programs.
- Only independent instructions can be executed in parallel without causing a wait state.
- The amount of instruction-level parallelism varies widely depending on the type of code being executed.

Pipelining in Superscalar Processor

- In order to fully utilise a superscalar processor of degree m , m instructions must be executable in parallel. This situation may not be true in all clock cycles. In that case, some of the pipelines may be stalling in a wait state

- In a superscalar processor, the simple operation latency should require only one cycle, as in the base scalar processor.

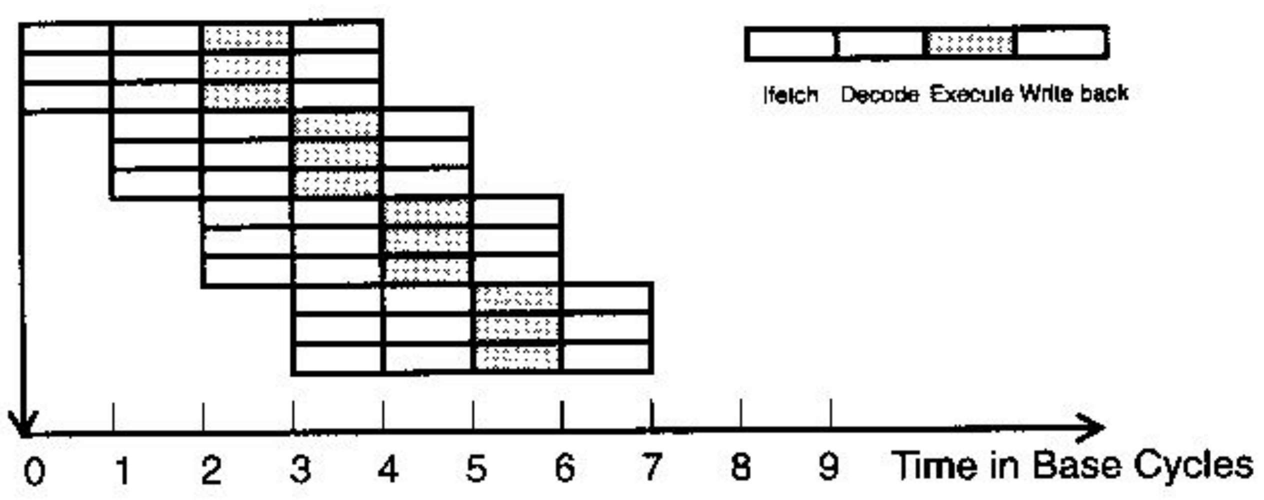


Figure 4.11 A superscalar processor of degree m = 3.

THANK YOU!

SuperScalar Pipeline Scheduling

Elizabath Saba
S7,CSE(23)

- Instruction issue and completion policies are critical to superscalar processor performance.
- Based on the way a dispatch unit of superscalar processor issue the instructions,superscalar architectures can be classified into
    - In-Order issue processor
    - Out-Of -Order issue processor

In-Order Issue processor

- Issue instructions in the order in which they appear in the program .
- Consider the below figure in which superscalar processor of degree 2 is shown

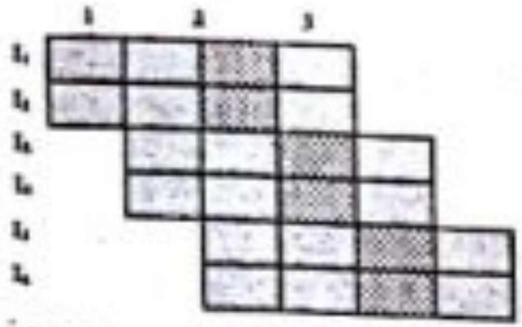
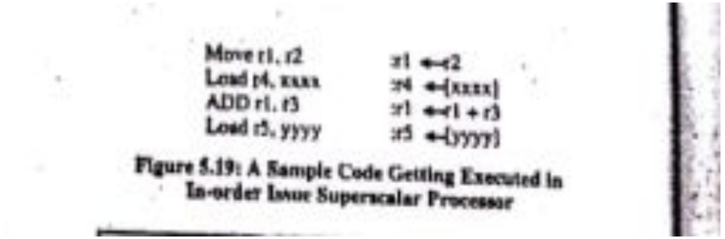


Figure 5.18: Superscalar Processor of Degree 2 with 4-Staged Pipeline

- The pattern in which piece of code get executed is



```
Move r1, r2          :r1 ←r2
Load r4, xxxx        :r4 ←(xxxx)
ADD r1, r3           :r1 ←r1 + r3
Load r5, yyyy        :r5 ←(yyyy)
```

Figure 5.19: A Sample Code Getting Executed in In-order Issue Superscalar Processor

- The first two instructions are fetched simultaneously and decoded and scheduled for execution in parallel.
- Since add instruction has a raw dependency with the move instruction,it is stalled for one cycle.

Out of order issue processor

- In out-of-order issue processor it would have not stalled the last load instruction.
- It provides much better performance than in order issue processor.
- It requires much more complex hardware to implement.

Superscalar Pipeline Design In-order and Out-of-order Issue

Submitted by,

Elizabeth Eldho
S7 CSE, 24

Superscalar Processor

A **superscalar processor** is a CPU that implements a form of parallelism called instruction-level parallelism within a single processor. In contrast to a scalar processor that can execute at most one single instruction per clock cycle, a superscalar processor can execute more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to different execution units on the processor.

It therefore allows for more throughput than would otherwise be possible at a given clock rate. Each execution unit is not a separate processor , but an execution resource within a single CPU such as an arithmetic logic unit.

In-order instruction execution

- instructions are fetched, executed & committed in compiler-generated order .If one instruction stalls, all instructions behind it stall
- instructions are statically scheduled by the hardware
- scheduled in compiler-generated order
- how many of the next n instructions can be issued, where n is the superscalar issue width
- superscalars can have structural & data hazards within the n instructions
- advantage of in-order instruction scheduling: simple implementation

  1.faster clock cycle

  2.fewer transistors

  3.faster design/development/debug time

Out-order instruction execution

- instructions are fetched in compiler-generated order
- instruction completion may be in-order (today) or out-of-order (older computers)
- in between they may be executed in some other order
- instructions are dynamically scheduled by the hardware
- superscalars can have structural & data hazards within the n instructions
- advantages: higher performance
    - better at hiding latencies, less processor stalling
    - higher utilization of functional units

In-order instruction issue: Alpha 21164

2 styles of static instruction scheduling

- dispatch buffer & instruction slotting (Alpha 21164)
- shift register model (UltraSPARC-1)

Instruction slotting:

- can issue up to 4 instructions
  - completely empty the instruction buffer before filling it again
  - compiler can pad with nop s so a conflicting instruction is issued with the following instructions, not alone

21164 Instruction Unit Pipeline

Fetch & Issue:

- S0: instruction fetch, branch prediction bits read
- S1: opcode decode, target address calculation, if predict taken, redirect the fetch
- S2: instruction slotting: decide which of the next 4 instructions can be, issued intra-cycle structural hazard check, intra-cycle data hazard check
- S3: instruction dispatch inter-cycle load-use hazard check, register read

UltraSparc 1:Shift register model

- can issue up to 4 instructions per cycle
- shift in new instructions after every group of instructions is issued

Superscalars

Hardware impact:

- more & pipelined functional units
- multi-ported registers for multiple register access
- more buses from the register file to the additional functional units
- multiple decoders
- more hazard detection logic
- more bypass logic
- wider instruction fetch
- multi-banked L1 data cache

Else the processor has structural hazards (due to an unbalanced design) and stalling.There are restrictions on instruction types that can be issued together to reduce the amount of hardware. Static (compiler) scheduling helps.

Superscalar Performance

BY:

ELIZEBETH SHIJU

Definition

- Superscalar processing is the ability to initiate multiple instructions during the same clock cycle.
- A typical Superscalar processor fetches and decodes the incoming instruction stream several instructions at a time.
- Superscalar architecture exploit the potential of ILP(Instruction Level Parallelism).

To compare the relative performance of a superscalar processor with that of a scalar base machine

- Estimate the ideal execution of N independent instructions through pipeline.
- The time required by the scalar base machine is,

$$T(1,1)=k+N-1 \text{ (base cycles)}$$

- The ideal execution time required by an m-issue superscalar machine is

  $$T(m,1)=k+(N-m)/m \text{ (base cycles)}$$

  Where k is the time required to execute the first m instructions through m pipelines

  The second term is the the time required to execute the remaining N-m instructions,m per cycle,through m pipelines.

- The ideal speedup of the superscalar machine over the base machine is

$$S(m,1)=T(1,1)/T(m,1)$$

$$=(N+k-1)/(N/m+k-1)$$

$$=(m(N+k-1))/(N+m(k-1))$$

As N->infinity,the speedup limit S(m,1)->m,as expected.

Latency-hiding techniques

Febi Justin-Roll No:26

Introduction

- In distributed shared memory machines, access to remote memory is likely to be slow compared to the ever-increasing speeds of processors.
- Thus, any scalable architecture must rely on techniques to reduce/hide/tolerate remote-memory-access latencies.
- There are four methods,

    1. Use of prefetching techniques
    2. Use of coherent cacheing techniques
    3. Relaxing the memory consistency requirements
    4. Using multiple-contexts to hide latency

Prefetching Techniques

- Prefetching is either software-controlled or hardware-controlled.
- In software-controlled prefetching explicit "prefetch" instructions are issued for data that is "known" to be remote.
- Hardware-controlled prefetching is done through the use of long cache lines to capitalize on spatial locality or through the use of instruction lookahead.
- Long cache lines introduce the problem of "false sharing", whereas instruction lookahead is limited by branches and/or branch-prediction accuracy.

Coherent Caches

- There are examples of maintaining coherent caches using snoopy
  caches for bus-based systems and using directory-based caches for
  general distributed memory machines.
- A major problem with the design of coherent caches is scalability.

Relaxed Memory Consistency Models

- Under the sequential consistency model, there is a partial ordering of all reads and writes by all processors, which obeys the program ordering.
- Thus, the result of any execution appears as some interleaving of the operations of the individual processors (as if executed on some multithreaded sequential machine).
- Most of the references to sequential consistency often attribute stronger conditions in that they require a total ordering of all reads and writes (a property that is better named "dynamic atomicity").

- Under the Weak consistency model, synchronization operators are not allowed to "perform" until all preceeding load/store operations have completed.
- Similarily, no load/store operations are allowed to "perform" until all preceeding synchronization operations have completed.
- In addition, (only) synchronization operators are sequentially consistent.

- Under processor consistency, writes issued by the same processor are never seen out of order (by any other processor in the system), but writes by different processors may be observed in different orders by different processors.
- Read operations following a write operation may bypass it (i.e. be observed by other processors as happening before the write).

Shared Virtual Memory : The Architecture Environment

The Architecture Environment

- Single address space multi-processors must use shared virtual memory.
- **Stanford Dash Experiment** provides the model for such an architectural environment

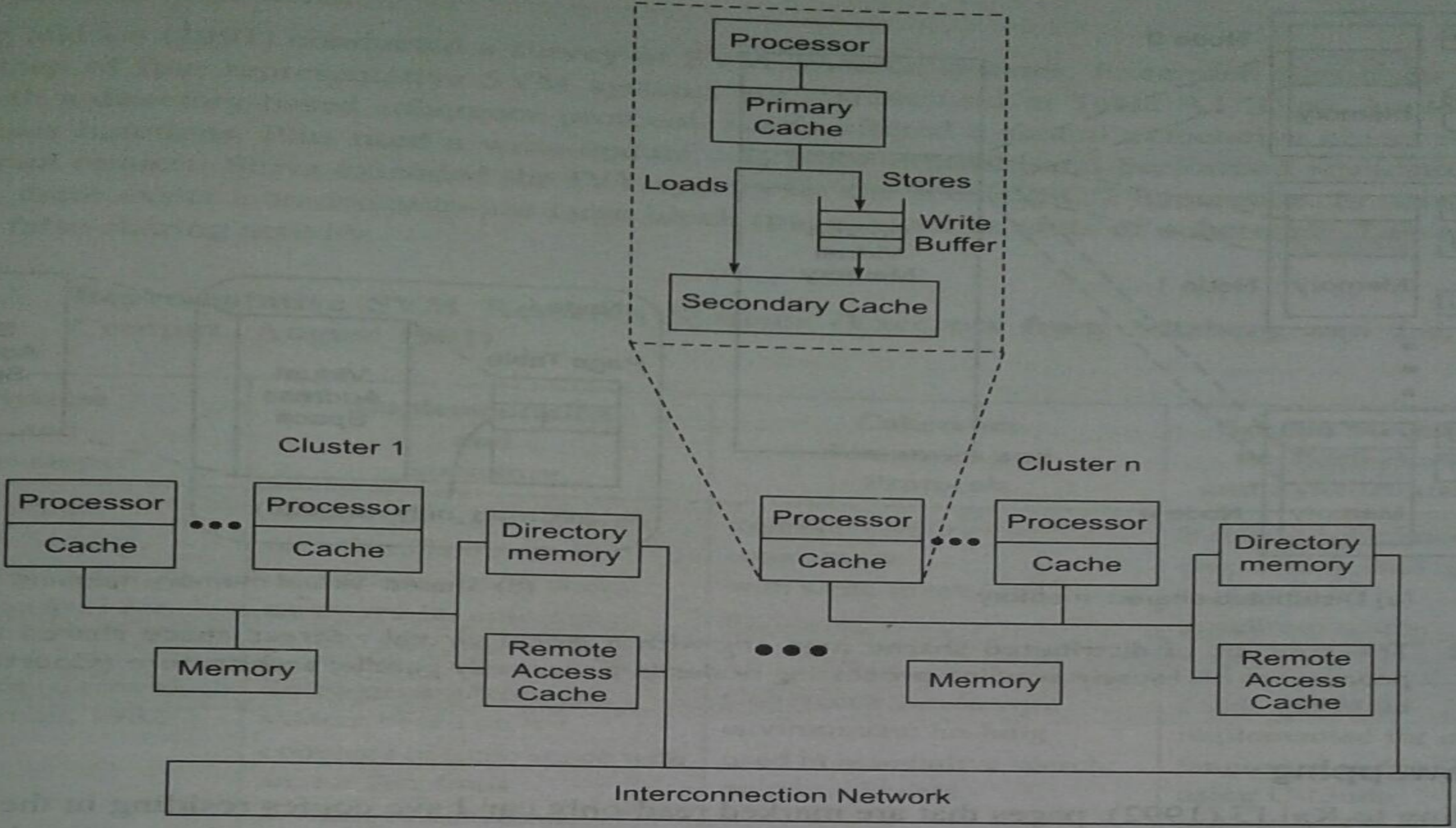


Fig. 9.1 A scalable coherent cache multiprocessor with distributed shared memory modeled after the Stanford Dash (Courtesy of Anoop Gupta et al, *Proc. 1991 Ann. Int. Symp. Computer Arch.*)

- The Dash architecture (in previous slide) has a large scale, cache-coherent,NUMA multiprocessor system.
- It consist of multiprocessor clusters connected through a scalable,low latency interconnection network.
- Physical m/y is distributed among the processing nodes in various clusters.
- Distributed m/y forms a global address space.
- Cache-coherence is maintained using an invalidating, distributed directory based protocol.
- For each m/y block, directory keep tracks of remote node caching it.
- When write occurs : Point-to-point message is send to invalidate remote copies.
- Ack messages were used to inform the originating nodes about completion of invalidation.

- Two levels of local cache were used.
- Loads and writes were separated using write buffers.
- Main m/y share all processing nodes in same cluster.
- For prefetching,directory based coherence protocol, directory m/y and remote access are used for each cluster.
- Remote access cache are shared by all processors in same cluster.

Shared Virtual Memory

George Scaria
S7, CSE

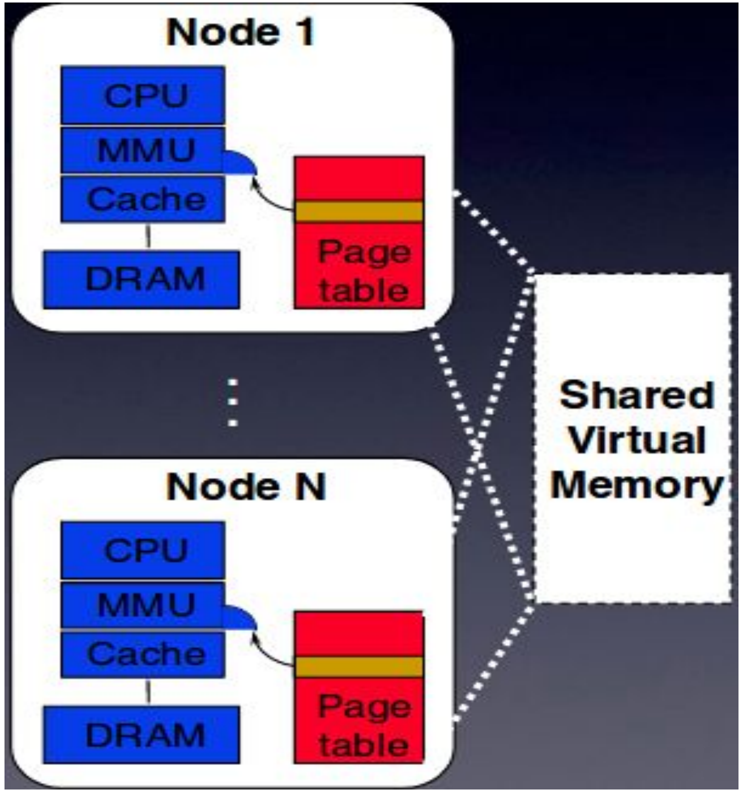
Shared Virtual memory

- Pool of “shared pages”: if not local, page is not mapped
- Page table entry access bits

Virtual page #	Physical page #	Valid	Access

- H/w detects read access to invalid page
    - read faults
- H/w detects writes to mapped memory with no write access
    - write faults

Shared Virtual memory



Shared Virtual memory

- Virtual memory makes it easy for processes to share memory as all memory accesses are decoded using page tables.

- For processes to share the same virtual memory, the same physical pages are referenced by many processes. The page tables for each process contain the Page Table Entries that have the same physical PFN.

Shared Virtual memory

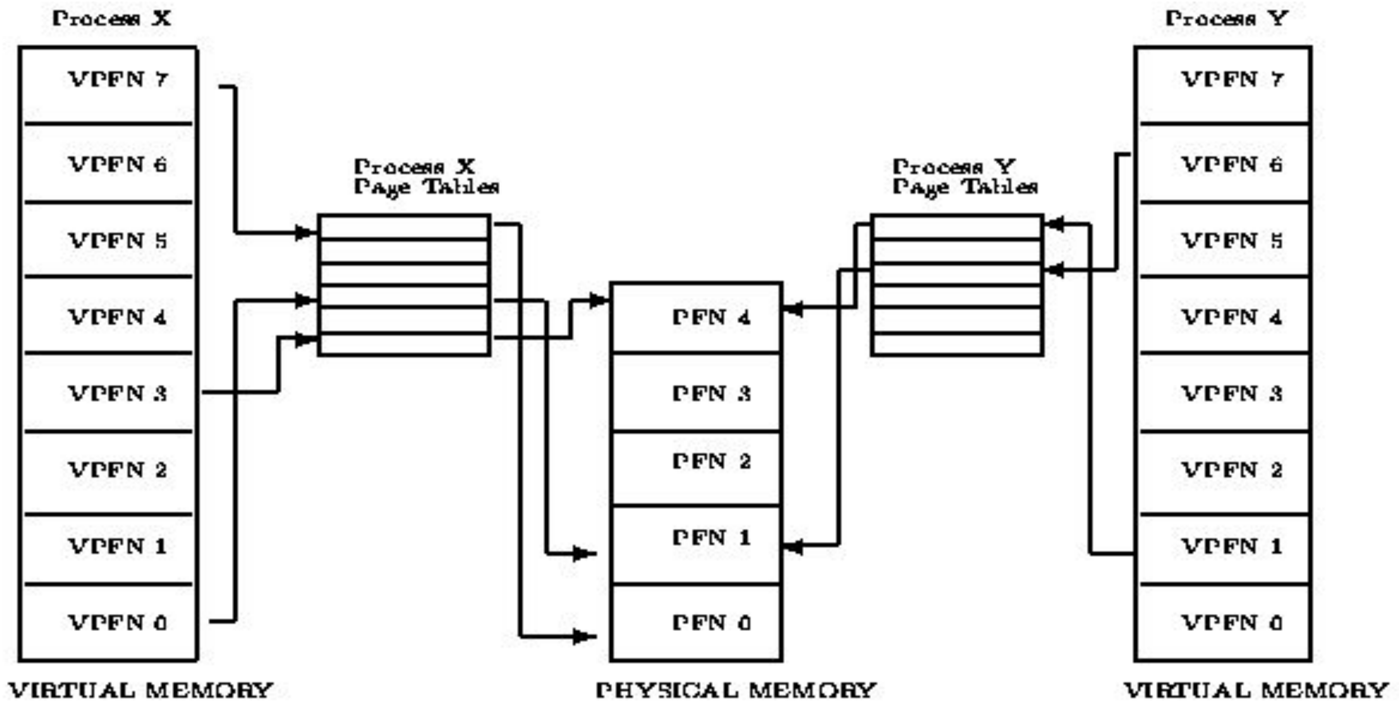


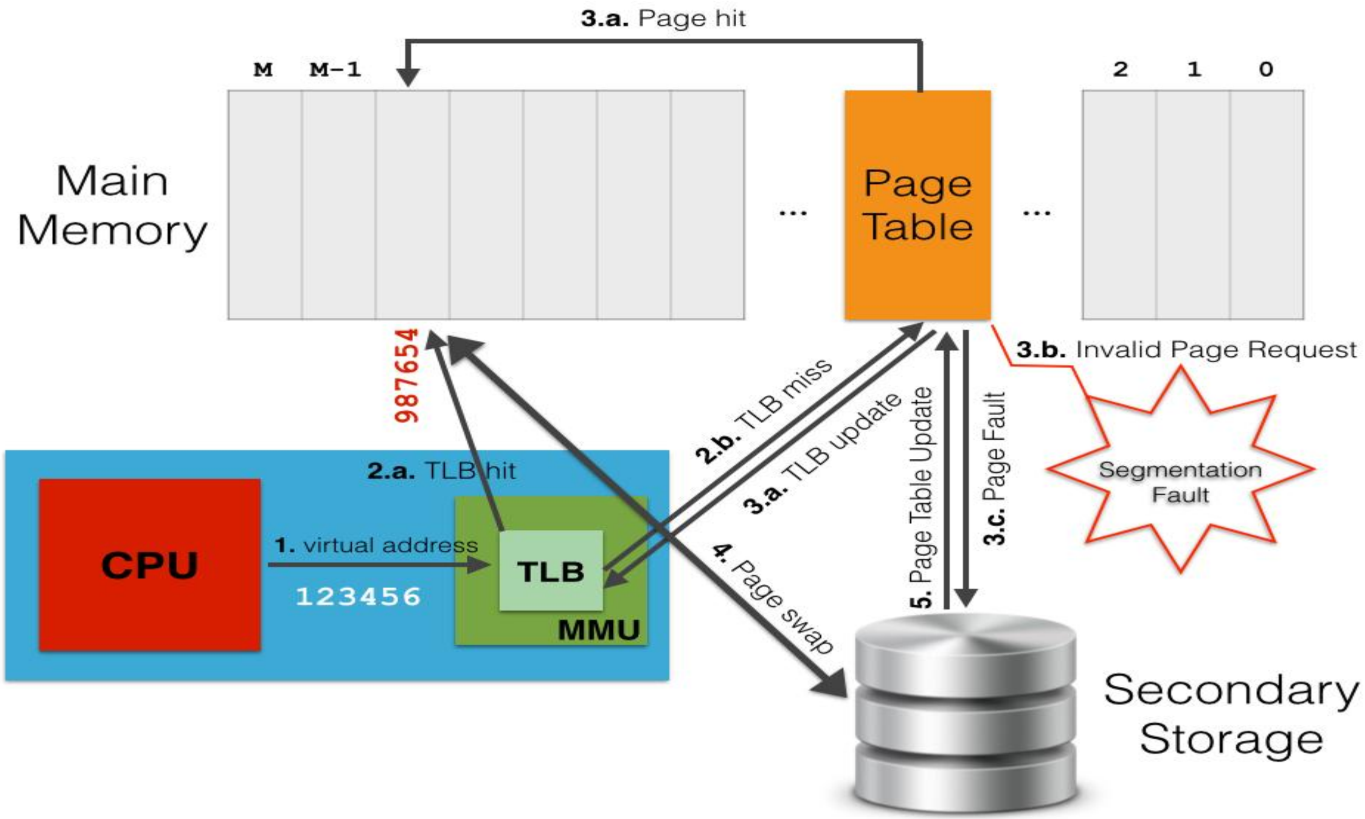
Fig : Sharing physical memory between processes

Shared Virtual Memory - Page Swapping

Submitted by
Gopika Chandrakumar
29 s7,CSE

- ❑ If a process needs to bring a virtual page into physical memory and there are no free physical pages available, the operating system must make room for this page by discarding another page from physical memory.

- ❑ If the page to be discarded from physical memory came from an image or data file and has not been written to then the page does not need to be saved. Instead it can be discarded and if the process needs that page again it can be brought back into memory from the image or data file.


3.a. Page hit
M M−1
2 1 0
Main Memory
...
Page Table
...
987654
3.b. Invalid Page Request
2.b. TLB miss
3.a. TLB update
5. Page Table Update
3.c. Page Fault
Segmentation Fault
2.a. TLB hit
CPU
1. virtual address
123456
TLB
MMU
4. Page swap
Secondary Storage

- However, if the page has been modified, the operating system must preserve the contents of that page so that it can be accessed at a later time. This type of page is known as a *dirty* page and when it is removed from memory it is saved in a special sort of file called the swap file. Accesses to the swap file are very long relative to the speed of the processor and physical memory and the operating system must juggle the need to write pages to disk with the need to retain them in memory to be used again.

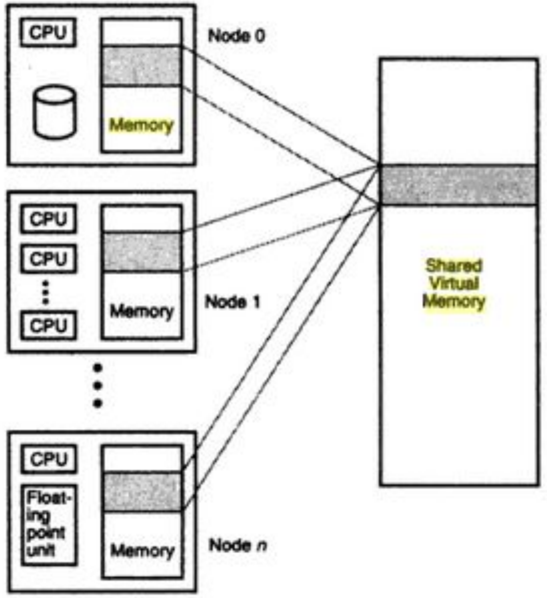
- If the algorithm used to decide which pages to discard or swap (the *swap algorithm* is not efficient then a condition known as *thrashing* occurs. In this case, pages are constantly being written to disk and then being read back and the operating system is too busy to allow much real work to be performed. If, for example, physical page frame number 1 in Figure 3.1 is being regularly accessed then it is not a good candidate for swapping to hard disk. The set of pages that a process is currently using is called the *working set*. An efficient swap scheme would make sure that all processes have their working set in physical memory.

- Linux uses a Least Recently Used (LRU) page aging technique to fairly choose pages which might be removed from the system. This scheme involves every page in the system having an age which changes as the page is accessed. The more that a page is accessed, the younger it is; the less that it is accessed the older and more stale it becomes. Old pages are good candidates for swapping.
-

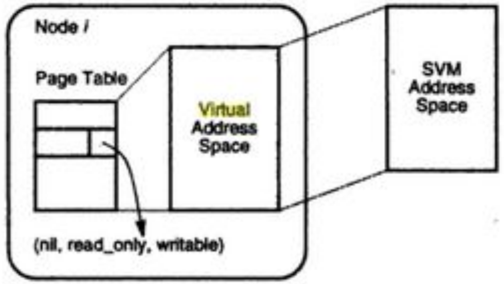
Shared Virtual Memory

SUBMITTED BY,
GOPIKA G NAIR
S7 CSE,NO:30

- Shared Virtual Memory was first developed in a Ph.D thesis by Li at Yale University.
- The idea is to implement coherent shared memory on a network of processors without physically shared memory.
- The system uses virtual addresses instead of physical address for memory references.
- Each virtual address space can be large as a single node can provide and is shared by all the nodes in the system.



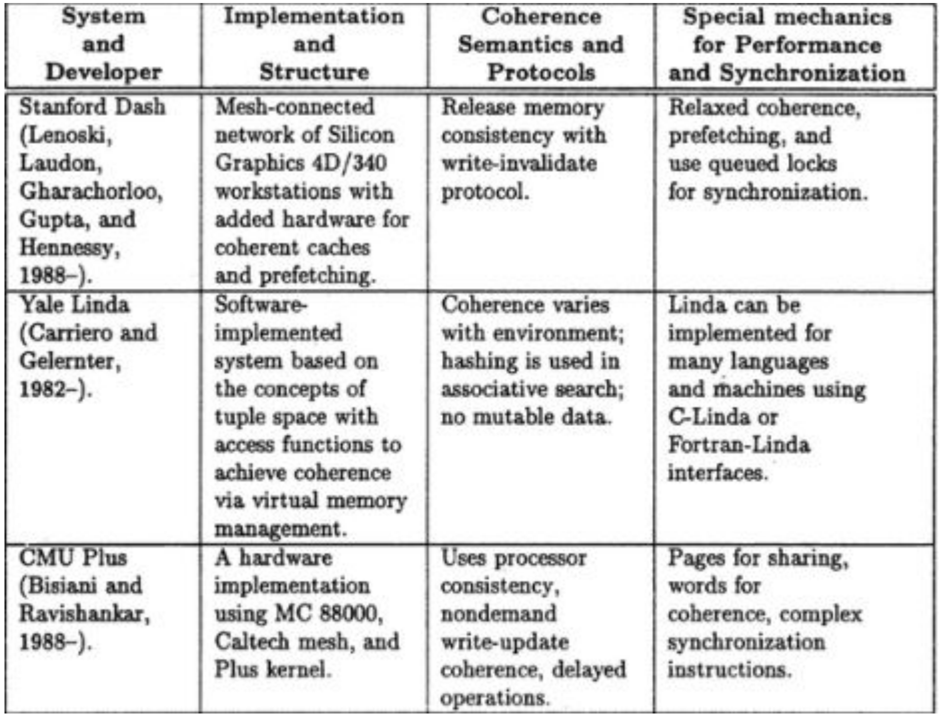
(a) Distributed shared memory



(b) Shared virtual memory mapping

- The SVM address space is organized in pages which can be accessed by any node in the system.
- A memory-mapping manager on each node views it local memory as a large cache of pages for its associated processor.
- SVM system uses page replacement policies to find an available page frame,swapping its content to the sending node.
- The memory coherence problem is solved in SVM through fault handlers and their servers.
- To client programs,this mechanism is completely transparent.

EXAMPLES OF SVM SYSTEMS



System and Developer	Implementation and Structure	Coherence Semantics and Protocols	Special mechanics for Performance and Synchronization
Stanford Dash (Lenoski, Laudon, Gharachorloo, Gupta, and Hennessy, 1988–).	Mesh-connected network of Silicon Graphics 4D/340 workstations with added hardware for coherent caches and prefetching.	Release memory consistency with write-invalidate protocol.	Relaxed coherence, prefetching, and use queued locks for synchronization.
Yale Linda (Carriero and Gelernter, 1982–).	Software-implemented system based on the concepts of tuple space with access functions to achieve coherence via virtual memory management.	Coherence varies with environment; hashing is used in associative search; no mutable data.	Linda can be implemented for many languages and machines using C-Linda or Fortran-Linda interfaces.
CMU Plus (Bisiani and Ravishankar, 1988–).	A hardware implementation using MC 88000, Caltech mesh, and Plus kernel.	Uses processor consistency, nondemand write-update coherence, delayed operations.	Pages for sharing, words for coherence, complex synchronization instructions.

PREFETCHING TECHNIQUES

Submitted by,
Greeshma M Benny
S7, CSE,31

Prefetching

- In **computer architecture**, instruction **prefetch** is a technique used in central processor units to speed up the execution of a program by reducing wait states.
- **Prefetching** occurs when a processor requests an instruction or data block from main memory before it is actually needed.

Prefetching Approaches

- Software-based
  - Explicit “fetch” instructions Additional instructions executed
  - Additional instructions executed
- Hardware-based
  - Special hardware
  - Unnecessary prefetchings (w/o compile time information)

Software Data Prefetching

- fetch instruction
  - Non-blocking memory operation
  - Cannot cause exceptions (e.g. page faults)
- Modest hardware complexity
- Challenge -- prefetch scheduling
  - Placement of fetch inst relative to the matching load or store inst
  - Hand-coded by programmer or automated by compiler

Loop-based Prefetching

- Loops of large array calculations
    - Common in scientific codes
    - Poor cache utilization
    - Predictable array referencing patterns
- fetch instructions can be placed inside loop bodies s.t. current iteration prefetches data for a future iteration

Limitation of Software-based Prefetching

- Normally restricted to loops with array accesses
- Hard for general applications with irregular access patterns
- Processor execution overhead
- Significant code expansion
- Performed statically

Sequential Prefetching

- Take advantage of spatial locality
- One block lookahead (OBL) approach
    - Initiate a prefetch for block b+1 when block b is accessed
    - Prefetch-on-miss
        - Whenever an access for block b results in a cache miss
    - Tagged prefetch
        - Associates a tag bit with every memory block
        - When a block is demand-fetched or a prefetched block is referenced for the first time.

Software vs. Hardware Prefetching

- Software
  - Compile-time analysis, schedule fetch instructions within user program
- Hardware
  - Run-time analysis w/o any compiler or user support
- Integration
  - e.g. compiler calculates degree of prefetching (K) for a particular reference stream and pass it on to the prefetch hardware.

Benefits of Prefetching

Irene Abraham
32
s7,cse

PREFETCHING

Prefetching uses knowledge about the expected misses in a program to move the corresponding data close to the processor before it is actually needed.prefetching can be classified based on whether it is binding or nonbinding, and whether it is controlled be hardware or software.

Prefetching is the loading of a resource before it is required to decrease the time waiting for that resource. Examples include instruction prefetching where a CPU caches data and instruction blocks before they are executed, or a web browser requesting copies of commonly accessed web pages. Prefetching functions often make use of a cache.

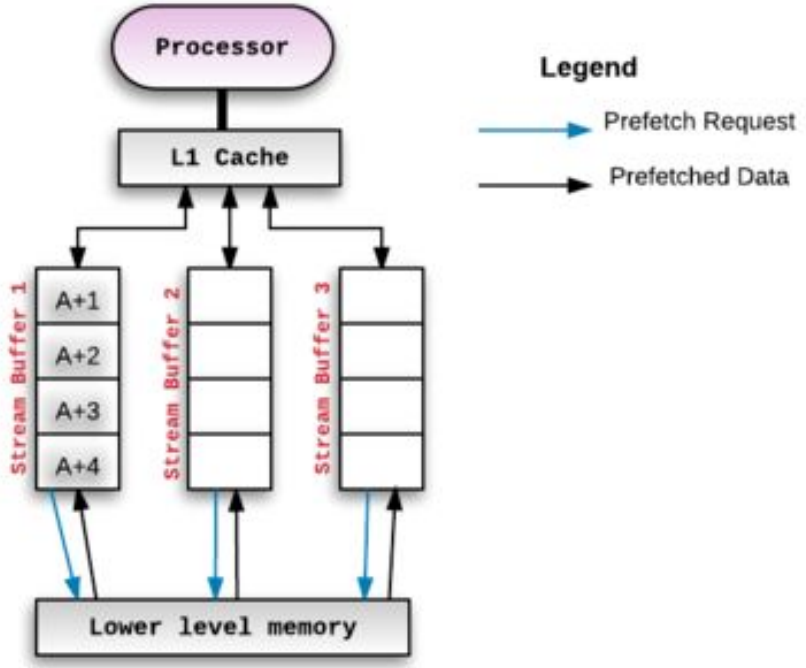
Benefits of Prefetching

The benefits of prefetching come from different sources.

- Most obvious benefit occur when a prefetch id issued early enough in the code so that the line is already in the cache by the time it is referenced
- Prefetching can even improve performance even when the above one is not possible. Example, when the address of a data structure cannot be determined until immediately before it is referenced.
- If multiple prefetches are issued back to back to fetch the data structure,the latency of all but the first prefetched reference can be hidden due to the pipelining of the memory access.

- Prefetching offers another benefits in multiprocessors that use an ownership based cache coherence protocol
- If a cache block line is to be modified,prefetching it directly with ownership can significantly reduce the write latencies and the ensuing network traffic for obtaining ownership.
- Network traffic is reduced in read-modify-write instructions, since prefetching with ownership avoids first fetching a read shared copy.
- Prefetching allows applications and hardware to maximize performance and minimize wait times by preloading resources that users will need before they request them.

Prefetching in cache



Prefetching

Benchmark Results

Prefetching

Prefetching in computer science is a technique for speeding up fetch operations by beginning a fetch operation whose result is expected to be needed soon. Usually this is before it is known to be needed, so there is a risk of wasting time by prefetching data that will not be used. The technique can be applied in several circumstances:

- Cache prefetching, a speedup technique used by computer processors where instructions or data are fetched before they are needed
- Prefetch input queue (PIQ), in computer architecture, pre-loading machine code from memory
- Link prefetching, a web mechanism for prefetching links
- The Prefetcher technology in modern releases of Microsoft Windows
- Prefetch buffer, a feature of DDR SDRAM memory
- Swap prefetch, in computer operating systems, anticipatory paging

Benchmarking

- **Benchmarking** is comparing ones business processes and performance metrics to industry bests and best practices from other companies.
- In project management benchmarking can also support the selection, planning and delivery of projects.
- Dimensions typically measured are quality, time and cost.
- In the process of best practice benchmarking, management identifies the best firms in their industry, or in another industry where similar processes exist, and compares the results and processes of those studied to one's own results and processes.
- Benchmarking is used to measure performance using a specific indicator (cost per unit of measure, productivity per unit of measure, cycle time of x per unit of measure or defects per unit of measure) resulting in a metric of performance that is then compared to others.

Benchmark Results

- For each benchmark, the two bars correspond to the cases with no prefetching (**N**) and with selective prefetching (**S**).
- In each bar, the bottom section is the amount of time spent executing instructions (including instruction overhead of prefetching), and the section above that is the memory stall time.
- For the prefetching cases, there is also a third component-stall time due to memory overheads caused by prefetching.
- Specifically, the stall time corresponds to two situations:
    - when the processor attempts to issue a prefetch but the prefetch issue buffer is already full,
    - when the processor attempts to execute a load or store when the cache tags are already busy with a prefetch fill.

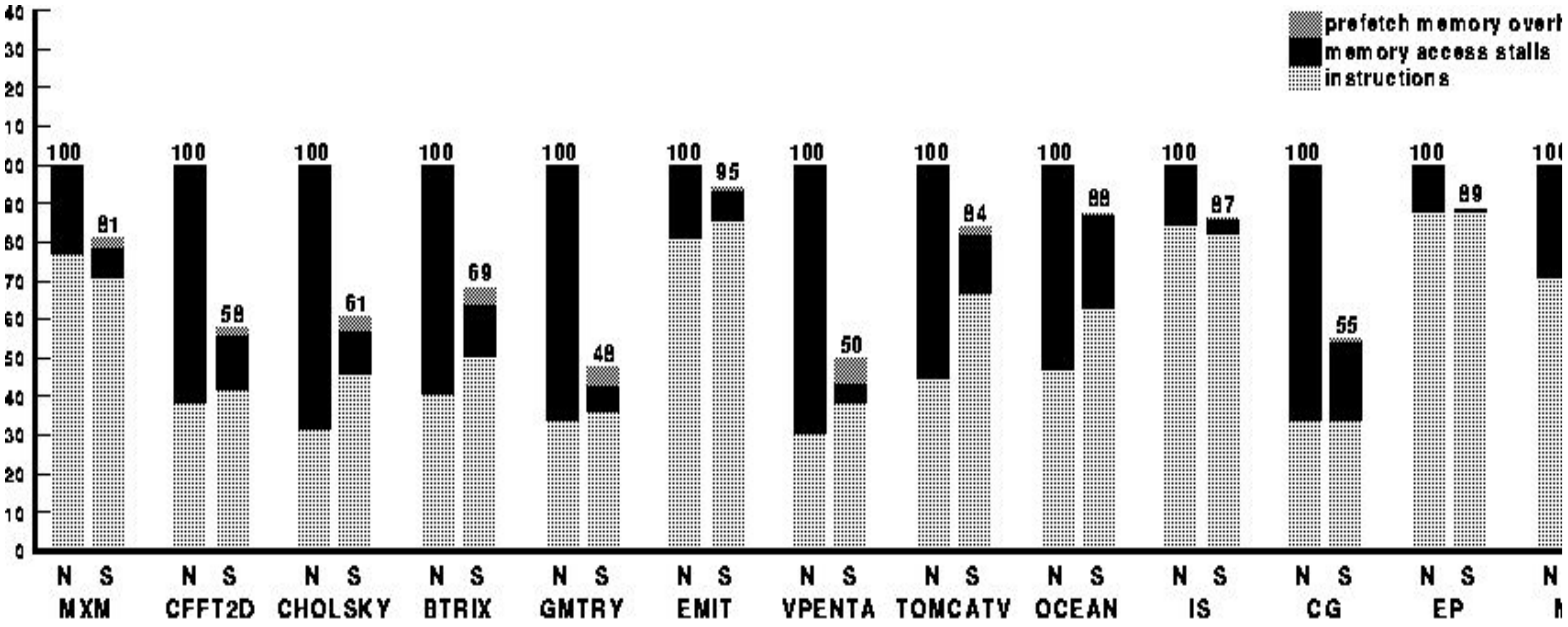


Figure 3: Overall performance of the selective prefetching algorithm (N = no prefetching, and S = selective prefetching).

- Having established the benefits of prefetching, we now focus on the costs. Figure shows that the instruction overhead of prefetching causes less than a 15%increase in instruction count in over half of the benchmarks. In fact, in two of those cases (MXM and IS) the number of instructions actually decreased due to savings through loop unrolling.
- In other cases (CHOLSKY, BTRIX, VPENTA, TOMCATV, OCEAN), the number of instructions increased by 25%to 50%.
- Finally, the stalls due to prefetching memory overhead are typically small-never more than 15%of original execution time. In each case, we observe that the overheads of prefetching are low enough compared to the gains that the net improvement remains large.

- As shown in Figure, the speedup in overall performance ranges from 5%to 100%, with 6 of the 13 benchmarks improving by over 45%. The memory stall time is significantly reduced in all the cases.
- The miss penalty is reduced because even if a prefetched line is replaced from the primary cache before it can be referenced, it is still likely to be present in the secondary cache.
- Also, the miss latency may be partially hidden if the miss occurs while the prefetch access is still in progress. Overall, 50%to 90%of the original memory stall cycles are eliminated.

Distributed Coherent Caches - Dash Experience and Benefits of Caching

By
Jishnu Prakasan

Distributed shared memory is an architectural approach that allows multiprocessors to support a single shared address space that is implemented with physically distributed memories. Hardware-supported distributed shared memory is becoming the dominant approach for building multiprocessors with moderate to large numbers of processors. Cache coherence allows such architectures to use caching to take advantage of locality in applications without changing the programmer's model of memory.

DASH is a scalable shared-memory multiprocessor whose architecture consists of powerful processing nodes, each with a portion of the shared-memory, connected to a scalable interconnection network. A key feature of DASH is its distributed direction-based cache coherence protocol. Unlike traditional snoopy coherence protocols, the DASH protocol does not rely on broadcast; instead it uses point-to-point messages sent between the processors and memories to keep caches consistent. Furthermore, the DASH system does not contain any single serialization or control point. While these features provide the basis for scalability, they also force a reevaluation of many fundamental issues involved in the design of a protocol. These include the issues of correctness, performance, and protocol complexity.

Caching

Caching has long been recognized as a powerful performance enhancement technique in many areas of computer design. Most modern computer systems include a hardware cache between the processor and main memory, and many operating systems include a software cache between the file system routines and the disk hardware. In a distributed file system, where the file systems of several client machines are separated from the server backing store by a communications network, it is desirable to have a cache of recently used file blocks at the client, to avoid some of the communications overhead. In this configuration, special care must be taken to maintain consistency between the client caches, as some disk blocks may be in use by more than one client. For this reason, most current distributed file systems do not provide a cache at the client machine. Those systems that do place restrictions on the types of file blocks that may be shared, or require extra communication to confirm that a cached block is still valid each time the block is to be used. The Caching Ring is a combination of an intelligent network interface and an efficient network protocol that allows caching of all types of file blocks at the client machines. Blocks held in a client cache are guaranteed to be valid copies. We measure the style of use and performance improvement of caching in an existing file system, and develop the protocol and interface architecture of the Caching Ring. Using simulation, we study the performance of the Caching Ring and compare it to similar schemes using conventional network hardware.

DASH

HTTP Adaptive Streaming (HAS) is gradually being adopted by Over The Top (OTT) content providers. In HAS, a wide range of video bitrates of the same video content are made available over the internet so that clients' players pick the video bitrate that best fit their bandwidth. Yet, this affects the performance of some major components of the video delivery chain, namely CDNs or transparent caches since several versions of the same content compete to be cached. In this context we investigate the benefits of a Cache Friendly HAS system (CF-DASH), which aims to improve the caching efficiency in mobile networks and to sustain the quality of experience of mobile clients. Firstly, we motivate our work by presenting a set of observations we made on large number of clients requesting HAS contents. Secondly we introduce the CF-Dash system and our testbed implementation. Finally, we evaluate CF-dash based on trace-driven simulations and testbed experiments. Our validation results are promising. Simulations on real HAS traffic show that we achieve a significant gain in hit-ratio that ranges from 15% up to 50%

Benefits of Caching

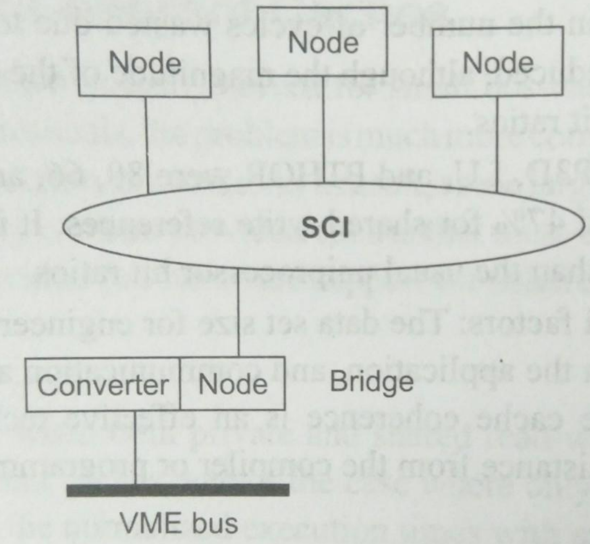
The advantages of cache memory are as follows −

- Cache memory is faster than main memory.
- It consumes less access time as compared to main memory.
- It stores the program that can be executed within a short period of time.
- It stores data for temporary use.

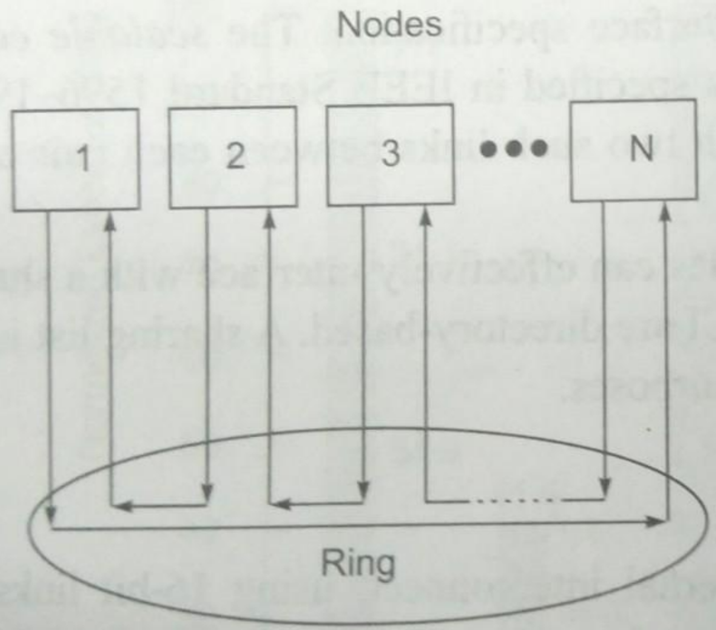
SCI Interconnect Models

Jose D Maliakkal
S7 cse
36

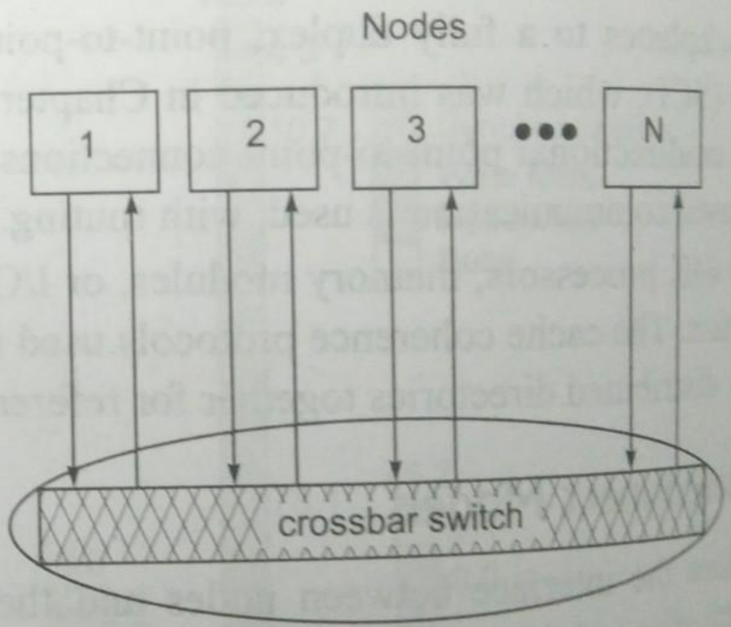
- SCI defines the interface between nodes and the external interconnect, using a 16 bit links with a bandwidth of upto 1 gigabyte per link.
- As a result backplane buses have been replaced by unidirectional point to point links.
- Each SCI node can be a processor with attached memory and I/O devices.
- The SCI interconnector can be a ring structure or a crossbar switch.
- Each node has an input link and an output link which are connected from or to the SCI ring or crossbar.



(a) Typical SCI configuration with bridge to other bus



(b) A ring for point-to-point transactions

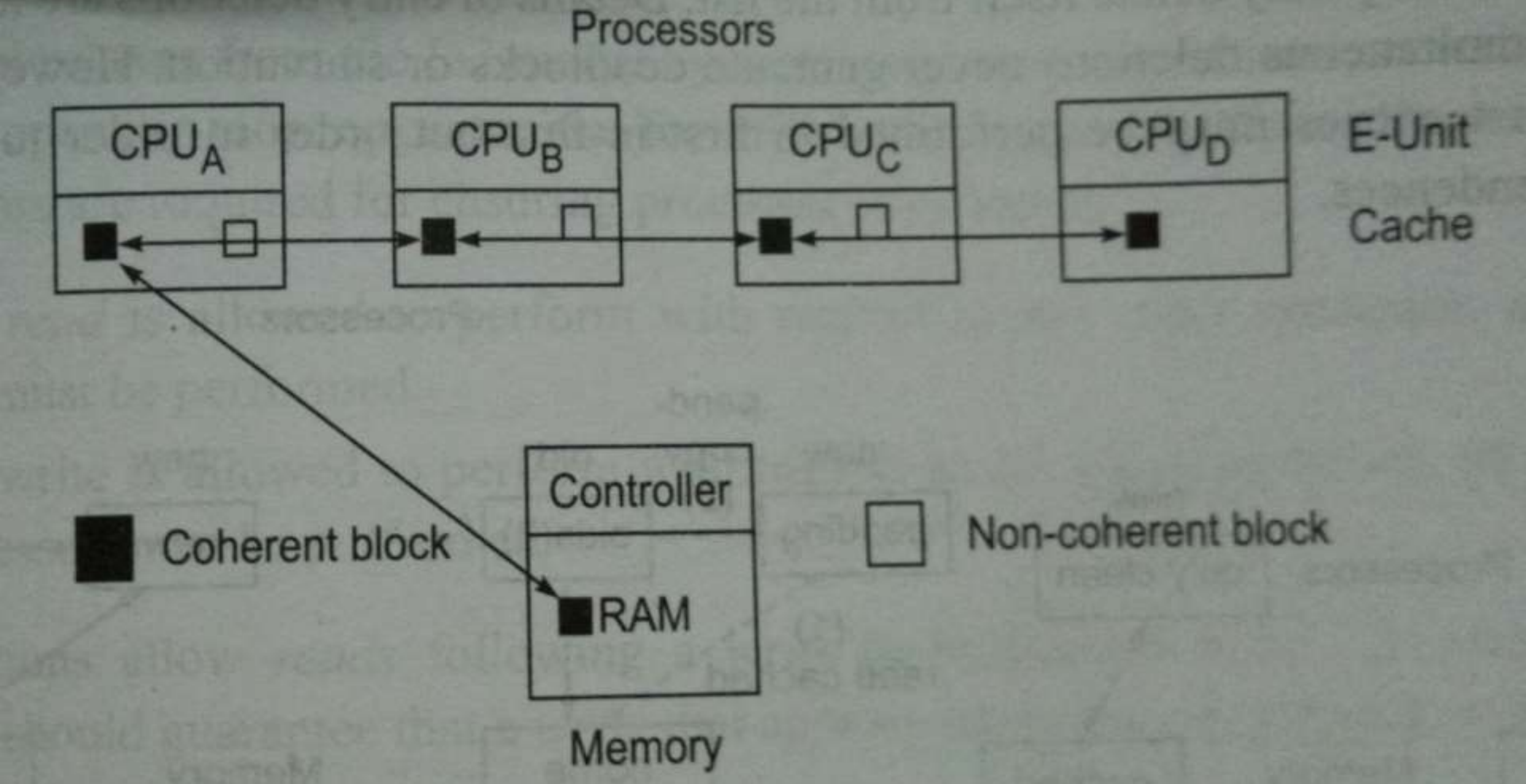


(b) A crossbar multiprocessor

Standard 1596–1992

SHARING LIST STRUCTURES

- Sharing lists are used in SCI to build chain directories for cache coherence use. The length of the sharing lists is efficiently unbounded.
- Sharing list are dynamically created, pruned and destroyed.
- Each coherently cached block is entered onto a list of processors sharing the block.



SCI cache coherence protocol with distributed directory

- Processors have the option of bypassing the coherence protocol for locally cashed data. Cache block of 64 bytes are assumed.
- By distributing the directories among the sharing processors, SCI avoids scaling limitations imposed by using a central directory.
- Communication among sharing processors are supported by heavily shared memory controllers.
- Other blocks may be locally cached and are not invisible to the coherence protocol.

- For every block address, the memory and cache entries have additional tag bits which are used to identify the first processor(head) in the sharing list and to link the previous and following nodes.
- Doubled linked list are maintained between processors in the sharing list, with forward ang backward pointer(double arrow in each link in the figure).
- Non coherent copies may also be made coherent by page-level control.
- However, such higher level software coherent protocol are beyond the scope of the SCI standard.

Effect Of Release Consistency

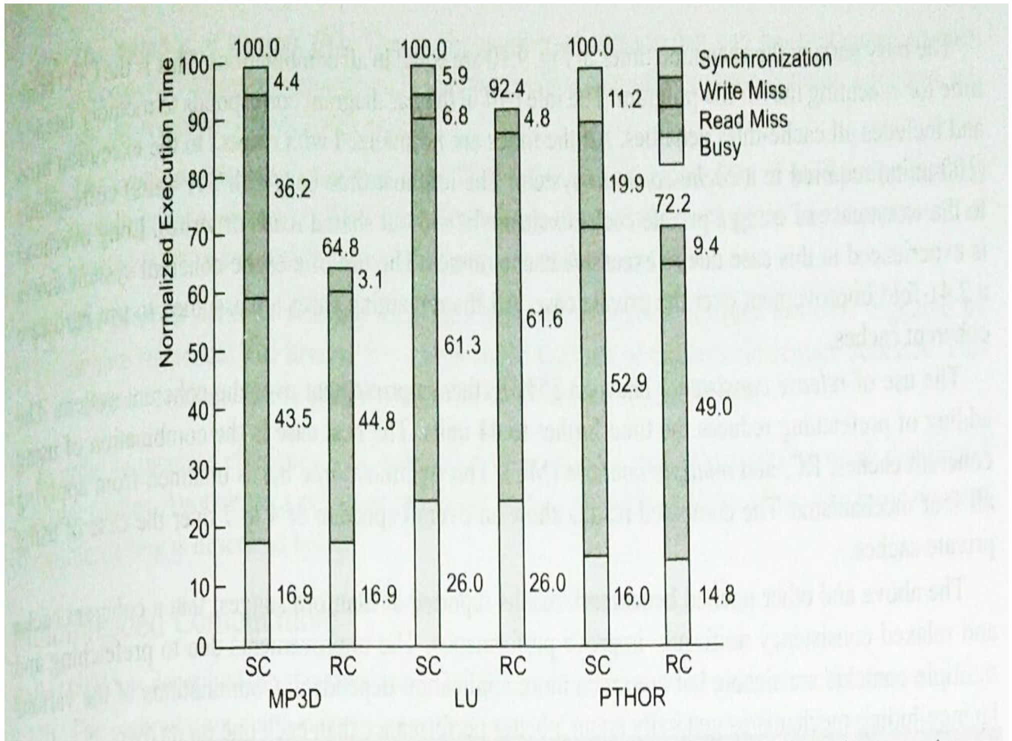


Fig. 9.9 Effect of relaxing the shared-memory model from sequential consistency (SC) to release consistency (RC) (Courtesy of Gupta et al, *Proc. Int. Symp. Comput. Archit.*, Toronto, Canada, May 1991)

- RC removes all idle time due to write-miss latency (from fig).
- The gains are large in MP3D and PTHOR since the write-miss time constitutes a large portions of the executions time under SC(35 and 20% respectively), while the gain is small in LU due to the relatively small write-miss time under SC(7%).

Effect Of Combining Mechanisms - 1

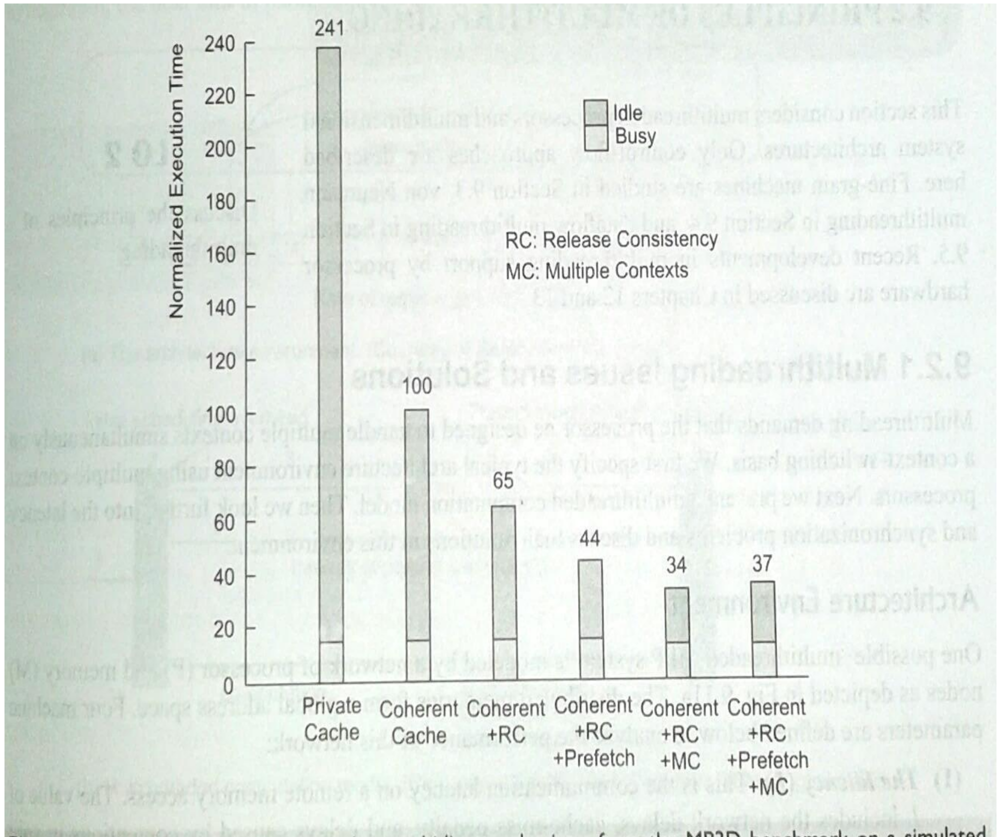


Fig. 9.10 Effect of combining various latency-hiding mechanisms from the MP3D benchmark on a simulated Dash multiprocessor (Courtesy of Gupta, 1992)

- Busy parts of the execution times are equal in al combinations.
- The idle part in the bar diagram corresponds to memory latency and includes all cache-miss penalties.
- All the times are normalized with respect to the execution time(100 units) required in cache-coherent system.
- Left most time bar – the worst case of using a private cache exclusively without shared reads or writes.

Effect Of Combining Mechanisms - 2

- Long overhead is experienced in this case due to excessive cache misses.
- All the remaining cases are assumed to use hardware coherent caches.
- The use of release consistency shows 35% further improvement over the coherent systems.
- The adding of prefetching reduces the time further to 44 units.
- The best case is the combination of using coherent caches, RC, and multiple contexts(MC).

Effect Of Combining Mechanisms - 3

- The rightmost time bar is obtained from applying all 4 mechanisms.
- The combined results show an overall speedup of 4 to 7 over the case of using private caches.
- Coherent cache and relaxed consistency uniformly improve performance.
- The improvements due to prefetching and multiple contexts are sizable but are much more applicant-dependent.
- Combinations of the various latency-hiding mechanisms generally attain a better performance than each one on its own.