Parallel Processing Shared Memory with Snoopy Cache

[1]Kuntam Babu Rao, [2]Dr. N Satyanarayana

[1]Vice-Principal & HOD of Computer Science and Engineering Department, Prasad Engineering College, Jangaon, Warangal District, Telangana – 506167,
[2]Principal, Nagole Institute of Technology and Science, Kuntloor(V), Hayathanagar(M), RR District, Telangana
Email: [1]kbrswe@gmail.com, [2]nits.principal@gmail.com

Abstract— Implementation of Parallel Processing with snoopy cache is an idea to enhance the performance of computing in real-time. When we consider uniprocessor the execution will take sequential. In uniprocessor we may not face problems. When we come to multi-core processors in Parallel Processing the program is divided into slices and each slice may be independent or dependent. If it is independent then no problem, but dependent case is different. There may be variables which may be used anywhere in program. Such variables or data is passed to a shared memory. That memory should be tracked or notified to the CPUs in execution process. There are some problems in normal process. We do not know how big the program and when it uses variables from shared memory. Some variables may be changed because of the instruction. That type of variables should be notified to other computing cores in time. This paper deals with how we overcome the problem. There are advantages in using multi-cores and there are disadvantages. They are expensive and should be handled carefully.

Index Terms— Parallel Processing, uniprocessor, CPUs, multi-core processors, variables.

I. INTRODUCTION

A. Parallel Processing

Parallel processing is the simultaneous use of more than one CPU or processor core to execute a program or multiple computational threads. Ideally, parallel processing makes programs run faster because there are more engines (CPUs or cores) running it. In practice, it is often difficult to divide a program in such a way that separate CPUs or cores can execute different portions without interfering with each other. Most computers have just one CPU, but some models have several, and multi-core processor chips are becoming the norm. There are even computers with thousands of CPUs.

Parallel processing is the ability of the brain to do many things (aka, processes) at once. For example, when a person sees an object, they don't see just one thing, but rather many different aspects that together help the person identify the object as a whole. For example, you may see the colors red, black, and silver. These colors alone may not mean too much, but if you also see shapes such as rectangles, circles, and curved shapes, your brain may perceive all the elements simultaneously, put them together and identify it as a car. Note that motion and depth of the object can also be perceived. These cues processed in the brain tell the person that the red car is headed straight at them so they jump out of the way. Without parallel processing, the brain would have to process each aspect of the car separately in progression. By the time the person identified the car, it would be too late.

A single program can be executed with single CPU sequentially. But time utilization is more. This type is suitable only for small amount of program. But we have large amount of program where it is executed with multi-core processors. Parallel processing is also called parallel computing. In the quest of cheaper computing alternatives parallel processing provides a viable option. The term parallel processing is used to represent a large class of techniques which are used to provide simultaneous data processing tasks for the purpose of increasing the computational speed of a computer system. Faster execution time, so higher throughput. For achieving Parallel Processing, more hardware and power is required.

Programs should run faster and pipeline is used for it. Example if you take a single program which is static. In that program we can have loops so that sequential execution will have top to bottom and bottom to top in dynamic execution with is shown in Fig. 1. Dynamic Execution path is known as "Thread of Execution". Fig. 1 single program execution with loops.

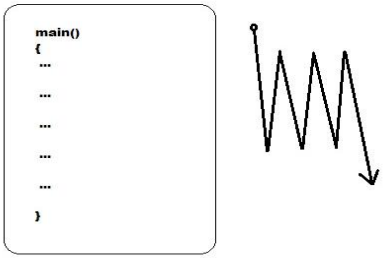


Fig. 1 Single Program.

Path length = number of instructions along path

We have been building machines to execute one thread (quickly).Fig. 2 Execution of one thread.

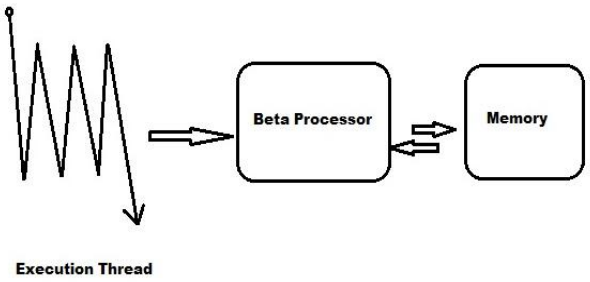


Fig. 2 Execution of one thread.

Time = (Path length X Clocks-per-instruction) / Clocks-per-second

B. Pipeline

A pipeline is a set of data processing elements connected in series, where the output of one element is the input of the next one. The elements of a pipeline are often executed in parallel or in time-sliced fashion; in that case, some amount of buffer storage is often inserted between elements.

An instruction pipeline is a technique used in the design of computers to increase their instruction throughput (the number of instructions that can be executed in a unit of time). The basic instruction cycle is broken up into a series called a pipeline. Rather than processing each instruction sequentially (one at a time, finishing one instruction before starting the next), each instruction is split up into a sequence of steps so different steps can be executed concurrently (by different circuitry) and in parallel (at the same time). Fig. 3 example for pipeline stage.

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Fig.3 Basic five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back). In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline.

Refers to the utilization of multiple CPUs in a single computer system. This is also called parallel processing. Parallel processing is not suitable for home PC or laptops. Mostly supercomputers are used for this. Single program can be executed dynamically. As per pipeline system, Single program can divided into small portions and each portion is executed by one CPU from multiple core CPUs.

C. Cache

A CPU cache is a cache used by the central processing unit (CPU) of a computer to reduce the average time to access memory. The cache is a smaller, faster memory which stores copies of the data from frequently used main memory locations. Most CPUs have different independent caches, including instruction and data caches, where the data cache is usually organized as a hierarchy of more cache levels (L1, L2 etc.).

II. LITERATURE SURVEY

1) Parallel Processing with Tri-Master and N-Slaves

Implementation of Parallel Processing with Tri – Master and N - Slaves is an idea to enhance the performance of computing in real-time. Usual schematics of parallel algorithm take more space and cost. But this model is a solution for that. In this, two master processors are used to organize the I/O operations & another one processor is a universal control for the system. Slaves are controlled by the sub-master processors & sub-master processors are controlled by universal master processor. This system consists of three operating systems. Main OS is executed on Master processor, Sub – Main OS is executed on Sub – master processors & Slave OS is executed on Slave processors. Main – Master can handle Memory Management and Sub – Master processors can handle Process Management. The System consists of only one universal main master processor and two sub– master processors. This system can be modified into different structures by incrementing or decrementing slaves in the system. Order of system performance can be calculated using slave processors. This is a study about processors as Master and Slaves but there is information related about memory. There is information about shared memory but not inner depth. There will be lot of parameters in a program which should be shared among all cores.

2) Task spreading and shrinking on multiprocessor systems and networks of workstations

In this paper, we see how computational model can be used for the problems of processor allocation and task mapping. The intended applications for this model include the dynamic mapping problems of shrinking or spreading an existing mapping when the available pool of processors changes during execution of the problem. The concept of problem edge class and other features of this model are developed to realistically and efficiently support task partitioning and merging for static and dynamic mapping. Algorithms form the basis for shrinking and spreading, and yields realistic results for a variety of problems mapped onto real systems. When we go for chain of instructions executed by multi-cores with

shared memory there may be a problem of unknowing updated variable.

III. ANALYSIS

A. EXISTING SYSTEM

Since the mid-1990s, U.S. businesses have sought parallel processing, a computing technique used mainly for scientific computing, for its potential to increase computational performance. Parallel processing speeds the execution of a program by dividing the program into multiple segments that can be executed simultaneously, each on a separate processor. As businesses anticipated database growth into tens and even hundreds of terabytes (one trillion bytes), they seriously considered adopting parallel processing to reduce the time needed to analyze these exponentially large amounts of data. Although many businesses had computers with multiple central processing units (CPUs), use of parallel computing had been limited because it required that programmers learn parallel-processing techniques for writing software. Companies that had invested large amounts of resources in business applications and systems were wary of having to rewrite their existing serial code into parallel code, because this meant finding and hiring programmers from the limited pool of those skilled in parallel programming. Programmers should also keep knowledge related to Parallelism. In Parallel computing the program which is written sequentially is executed in pieces. Programmer should have a knowledge whether all instruction variables are available or updated, if updated keep the copy. There is a possibility to execute more than one instruction per clock because of pipeline.

B. Two places to find parallelism.

1) Instruction Level (ILP)

Fetch and issue groups of independent instructions within a thread of execution. Instruction-level parallelism (ILP) is a measure of how many of the operations in a computer program can be performed simultaneously. The potential overlap among instructions is called instruction level parallelism. There are two approaches to instruction level parallelism:

- Hardware
- Software

Hardware level works upon dynamic parallelism whereas; the software level works on static parallelism. The Pentium processor works on the dynamic sequence of parallel execution but the Itanium processor works on the static level parallelism.

Micro-architectural techniques that are used to exploit ILP include:

a) Instruction Pipeline

Instruction pipelining where the execution of multiple instructions can be partially overlapped. Consider the following example in Fig. 4. This is an example for smarter coding. The execution of a program is changed in the parallel code.

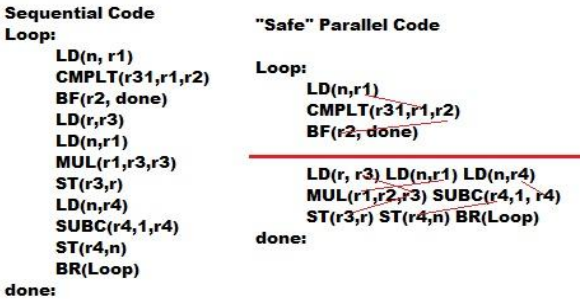


Fig. 4 Smarter Code.

b) Superscalar Parallelism

A superscalar CPU architecture (Fig. 5) implements a form of parallelism called instruction-level parallelism within a single processor. It therefore allows faster CPU throughput than would otherwise be possible at a given clock rate. A superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor. Each functional unit is not a separate CPU core but an execution resource within a single CPU such as an arithmetic logic unit, a bit shifter, or a multiplier. Fig. 6 gives simple superscalar pipeline. A single-core superscalar processor is classified as an SISD processor (Single Instructions, Single Data), while a multi-core superscalar processor is classified as an MIMD processor (Multiple Instructions, Multiple Data). While a superscalar CPU is typically also pipelined, pipelining and superscalar architecture are considered different performance enhancement techniques. The superscalar technique is traditionally associated with several identifying characteristics (within a given CPU core):

- Instructions are issued from a sequential instruction stream
- CPU hardware dynamically checks for data dependencies between instructions at run time (versus software checking at compile time)
- The CPU processes multiple instructions per clock cycle

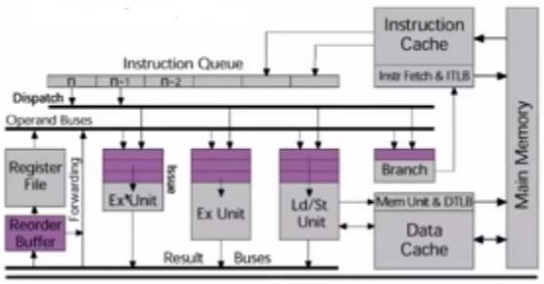


Fig. 5 Superscalar Parallelism

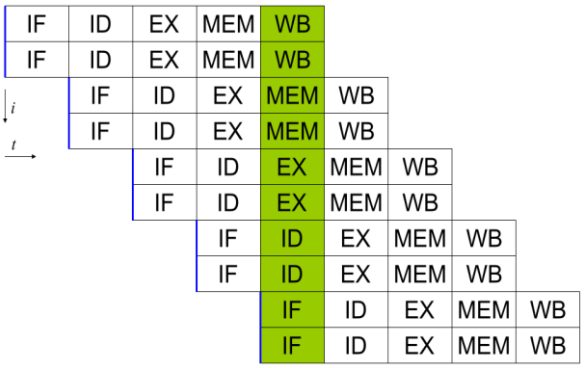


Fig. 6 Simple superscalar pipeline. By fetching and dispatching two instructions at a time, a maximum of two instructions per cycle can be completed. (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back, i = Instruction number, t = Clock cycle [i.e., time])

c) SIMD Processing

Single instruction, multiple data (SIMD), is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Thus, such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment. SIMD is particularly applicable to common tasks like adjusting the contrast in a digital image or adjusting the volume of digital audio. Most modern CPU designs include SIMD instructions in order to improve the performance of multimedia use. One instruction and multiple functional units. This is a vector method. There will be more one vector element. Each functional unit is attached to vector element. Each datapath has its own local data (register file). All datapaths execute the same instructions as shown in Fig. 7.

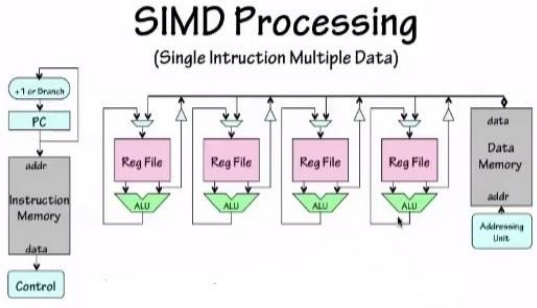


Fig. 7 for SIMD Processing.

SIMD coprocessing unit contain SIMD datapath added to a traditional CPU core, register-only operands. Core CPU handles many memory traffic. Partitionable datapaths for variable-sized "packed operands". Fig. 8 shows SIMD coprocessing unit.

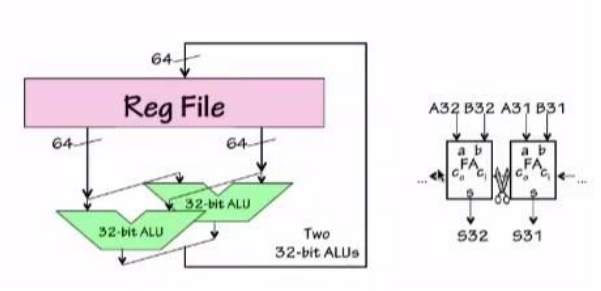


Fig. 8 SIMD Coprocessing Unit.

We can go for more SIMD coprocessing units as shown in Fig. 9. This type of technique can be used for graphics, signal processing, multimedia apps and soon.

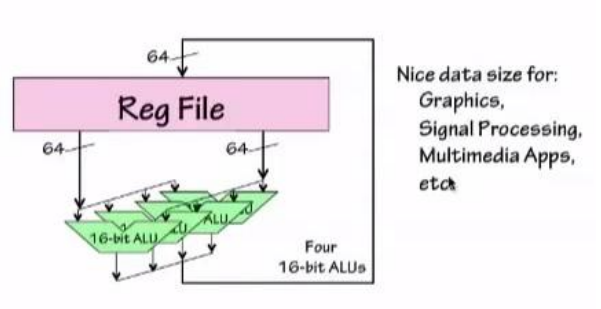


Fig. 9 SIMD 64 bit Coprocessing Unit.

(1) VLIW

Very long instruction word or VLIW (Fig. 10) refers to a processor architecture designed to take advantage of instruction level parallelism (ILP). Whereas conventional processors mostly only allow programs that specify instructions to be executed one after another, a VLIW processor allows programs that can explicitly specify instructions to be executed at the same time (i.e. in parallel). This type of processor architecture is intended to allow higher performance without the inherent complexity of some other approaches. A single-wide instruction controls multiple heterogeneous datapaths. Exposes parallelism to compiler software and hardware.

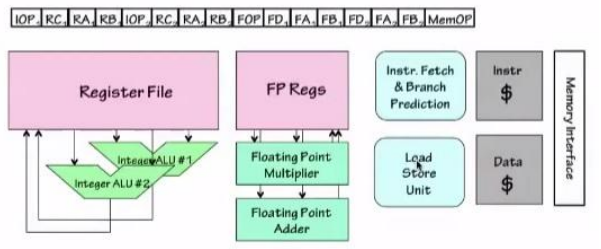


Fig. 10 VLIW

2) Thread Level Pipeline

Another strategy of achieving performance is to execute multiple programs or threads in parallel. This area of research is known as parallel computing. In Flynn's taxonomy, this strategy is known as Multiple Instructions-Multiple Data or MIMD as shown in Fig. 11

MIMD. All processors share a common memory leverages existing CPU designs.

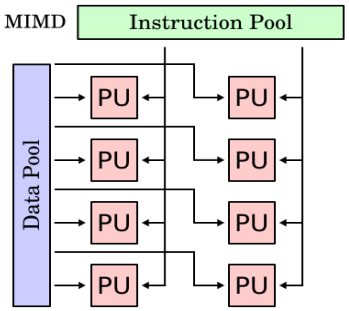


Fig. 11 MIMD

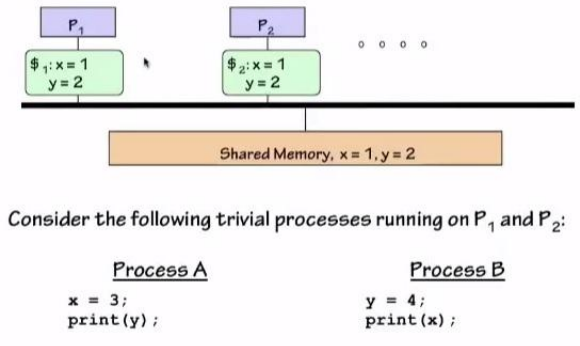
C. PROPOSED SYSTEM

MIMD (multiple instructions, multiple data) is a technique employed to achieve parallelism. Machines using MIMD have a number of processors that function asynchronously and independently. At any time, different processors may be executing different instructions on different pieces of data. MIMD architectures may be used in a number of application areas such as computer-aided design/computer-aided manufacturing, simulation, modeling, and as communication MIMD machines can be of either shared memory or distributed memory categories. These classifications are based on how MIMD processors access memory. Shared memory machines may be of the bus-based, extended, or hierarchical type. Distributed memory machines may have hypercube or mesh interconnection schemes. Easy to map "processes (thread)" to "processors". Each processor is communicated through shared memory. Memory is a bottle neck problem. Do they really share memory?

IV. RESULT

A. How does it work?

Consider the following example, process A work on P1 and process B work on P2 as shown in Fig. 12.



Fig, 12 Example.

Where we have shared memory for x and y. Process A change x to 3 and print y and process B change y to 4 and print x. Possible outcomes of above example are given in Fig. 13.

Fig. 13 Outcomes

The values of x and y are changed as shown in Fig. 14.

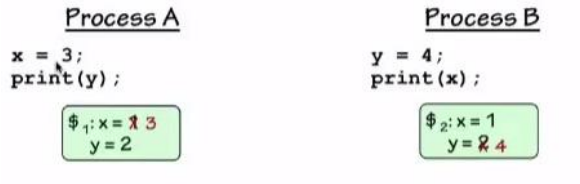


Fig. 14 Changes in Outcome.

Possible execution sequence is given as shown in Fig. 15.

Possible Execution Sequence:

SEQUENCE	A prints	B prints
x=3; print(y); y=4; print(x);	2	1
x=3; y=4; print(y); print(x);	2	1
x=3; y=4; print(x); print(y);	2	1
y=4; x=3; print(x); print(y);	2	1
y=4; x=3; print(y); print(x);	2	1
y=4; print(x); x=3; print(y);	2	1

Fig. 15 Possible Execution Sequence.

When we execute processes on uniprocessor. Then the possible outcome may be as shown in Fig. 16. Possible outcome when we ran Process A and Process B on a single timed-shared memory.

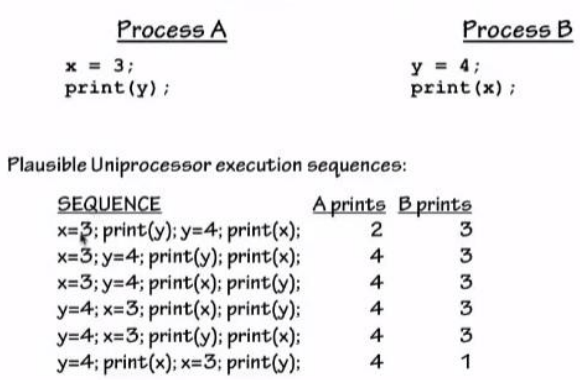


Fig. 16 Outcome on Uniprocessor

This output depends on the order of execution of processes. The semantic rule or constraint is that the program should run parallel. When we want to run a program in parallelism, it should give exact outputs for the program. Result of executing N parallel programs should correspond to some interleaved execution on a single processor as shown in Fig. 17.

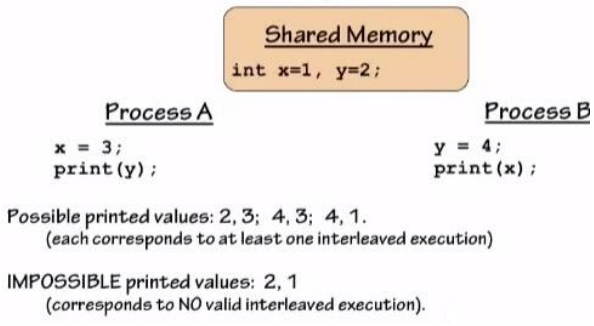


Fig.17. Possible values.

V. SOLUTION

The problem is going to arise because of cache incoherence. The problem is because of stale in cache values. The problem is not that memory has stale values, but that other caches may have. The change made by process A should be known to process B about value x. Where there is a change in program that should be notified to shared memory as shown in Fig. 18.

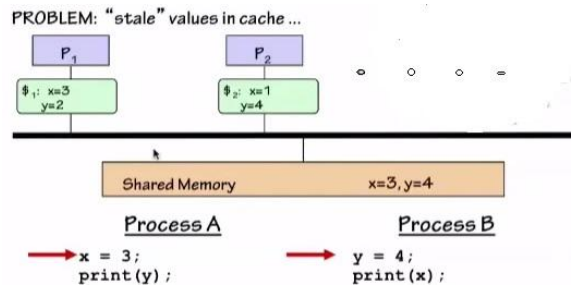


Fig. 18 Shared Memory

Solution to it is developing a snoopy cache as shown in Fig. 19.

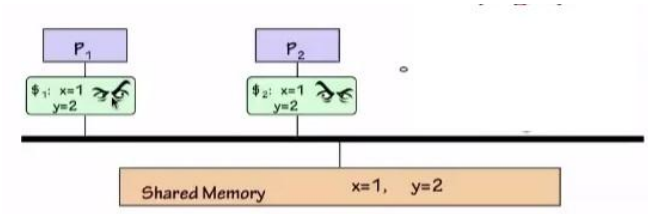


Fig. 19. Snoopy Cache

Sophisticated cache which does as normal cache doing. Snoopy cache watches bus that any memory variable is changed or not and if anything happen it keeps a copy of it. Whenever there is a change to one variable, then other process variable will also change as shown in Fig. 20. A snoopy cache solves the problem in shared memory.

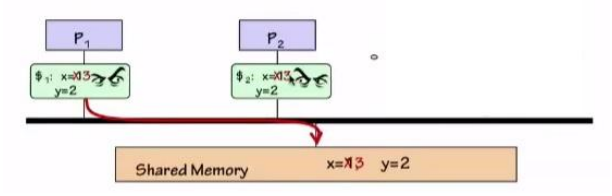


Fig. 20. Snoopy Cache with shared memory.

P1 writes 3 into x, write through cache causes bus transaction. P2, snooping, sees transaction on bus. INVALIDATES or UPDATES its cache x value.

A. Snoopy Cache Design

Two-bit state in Cache line encodes one of M, E, S, I states ("MESI" Cache) as shown in Fig. 21.

Invalid: Cache line unused.

Shared Access: Read-only, valid, not dirty, shared with other read-only copies elsewhere, most invalidate other copies before writing.

Exclusive: Exclusive copy, not dirty, on writes becomes modified.

Modified: Exclusive access, read-write, valid, dirty, must be written back to memory eventually, meanwhile can be written or read by local processor.

Current State	Read Hit	Read Miss Snoop Hit	Read Miss Snoop Miss	Write Hit	Write Miss	Snoopy for Read	Snoopy for Write
Modified	Modified	Invalid (Wr-Back)	Invalid (Wr-Back)	Modified	Invalid (Wr-Back)	Shared (Push)	Invalid (Push)
Exclusive	Exclusive	Invalid	Invalid	Modified	Invalid	Shared	Invalid
Shared	Shared	Invalid	Invalid	Modified (Invalidate)	Invalid	Shared	Invalid
Invalid	X	Shared (Fill)	Exclusive (Fill)	X	Modified (Fill-Inv)	X	X

Fig. 21. Snoopy Cache Design.

Small amount of processors we can have sequential comeback whereas large amount of processors it is difficult.

B. Implementation

A memory barrier, also known as a membar, memory fence or fence instruction, is a type of barrier instruction which causes a central processing unit (CPU) or compiler to enforce an ordering constraint on memory operations issued before and after the barrier instruction. This typically means that operations issued prior to the barrier are guaranteed to be performed before operations after the barrier. However, to be clear, it does not mean any operations WILL have completed by the time the barrier completes; only the ORDERING of the completion of operations (when they do complete) is guaranteed.

Memory barriers are necessary because most modern CPUs employ performance optimizations that can result in out-of-order execution. This reordering of memory operations (loads and stores) normally goes unnoticed within a single thread of execution, but can cause unpredictable behaviour in concurrent programs and device drivers unless carefully controlled. The exact nature of an ordering constraint is hardware dependent and defined by the architecture's memory ordering model. Some architectures provide multiple barriers for enforcing different ordering constraints.

In Memory Barrier, when you change, I will know it. Similar to human brain or how each person is updated.

VI. CONCLUSION

Parallel Processing is a technique used in supercomputers and above. In real time we cannot assume that the program slices are independent. There is a chance of having dependent slices. They have to use variable data which are updated and put in share memory. When there are loops almost the program for each loop will have top and bottoms and reside in cache. Whenever there is an updating related to a variable that should be notified or keep a copy of it. This will be done by snoopy cache. Parallel Processing is not a normal PC or laptop; it is an expensive device and should give accurate results. The Software Professionals should write programs by understanding the Parallel Processing system architecture properly for better future.

VII. FUTURE WORK

There is lot of prospects for future CPU architectures. They should consider about pipeline, superscalar, SIMD, VLIW, MIMD, single-thread limits and brain work.

REFERENCES

[1] "Parallel Processing with Tri-Master and N-Slaves" by G Kalyanaraman

[2] "Snoopy and Directory Based cache coherence Protocols: A Critical Analysis" Samacher Al-Hothalli, Safeeullah Soomro, Khurran Tanvir, Ruchi Tuli.

[3] J.C. Jacob and S.-Y. Lee, "Task Spreading and Shrinking on Multiprocessor Systems and Networks of Workstations" IEEE Transactions on Parallel and Distributed Systems, pp.1082- 1101, vol. 10. No. 10, October 1999.

[4] J.C. Jacob and S.-Y. Lee, "Task Spreading and Shrinking with Various Edge Classes", Proceedings of the 25th International Conference on Parallel Processing, vol. III, pp174-181, August 1996.