Research Paper on Cache Memory

Author's Details:
[1] **Pooja Agarwal** [2] **Prerna Mangla** [3]**Preeti Kumari** [1]Department of Computer Science & Engg. Dronacharya College of Engg. Gurgaon, Haryana, India [2]Department of Computer Science & Engg. Dronacharya College of Engg. Gurgaon, Haryana, India [3]Department of Computer Science & Engg. Dronacharya College of Engg. Gurgaon, Haryana, India

Abstract:
Cache memory *is a memory which is used by the central processing unit in a computer to reduce the burden on the main access memory.* ***Cache memory*** *stores all the recent instructions and this is the only reason it is nearest to CPU. It lies in the path between processor and memory and this is the only reason why it use less time to access memory than main memory. Ere cache uses 100ns speed main memory uses 700 ns speed. The subject of using the data is displayed by a factor that is named* ***hit ratio****. In this paper, a new process for increasing the hit ratio is introduced which is referred to as the* ***programmable cache memory****. This method is a grouping of both the physical and logical approach, i.e., in adding together to use a high speed SRAM, a programmable logical circuit is added to the cache manager in order to increase the hit ratio. In the paper we have thrown light on cache working that the very basic functioning of the processor. We have also reached the topic discussing the cache memories in uniprocessors. Besides we have discussed the directory protocols used in cache where main protocol is snoopy protocol. And at last we have gone through cache consistencies that is how it works in tough times. And at last we have concluded with some important points.*

Keywords:*Hit Ratio, Locality Of Reference, Programmable Cache Memory, Snoopy Protocols, Cache Coherence, Cache Consistency*

INTRODUCTION:-

In some applications the size of memory segments which must be processed, is huge and each catalogue in a division is used only once, so none of the cache methods will be helpful, because the storage of index which will not be used, has no improvement. In fact the main disadvantage of all the preceding cache methods is that they cannot provide data which have not been used earlier than by the program at runtime. This subject is referred to as the ***cold start misses***. To cover this problem, some prefetching methods have been projected. We propose a programmable cache which is able to provide the data which have not been used before, while the program uses several dissimilar data segments with sizes larger than the cache size. The proposed method runs a programmable prefetching procedure while the CPU is running other commands. The programmable cache memory is made of three major parts: Data registers, control and timer registers, and cache driver program.

DATA REGISTERS

These contain the addresses and information, which are used by the CPU. This part of our proposed cache is a small completely associative cache memory. **Control registers** and timers are used to conclude when and from where the valid data must be read or written to the memory. The control registers and timers are automatic by the cache driver program in order to provide the valid data at the required time in data registers. The cache driver program investigate a proposed program before runtime and determines the approximate times of its memory operations at the runtime according to the CPU cycle, and the position and order of the commands which is used in the projected program and then programs the control registers and timers. There are situations in which the program contains interrupts or conditional branches, so the driver cannot determine the accurate time of R/W operations. In these cases, the projected cache method acts as a fully associative cache memory while in other situation it acts as mention before. The following figure shows the block diagram of the proposed cache memory. In this paper a occupied map of the programmable cache memory will be given. Then, the structure of its driver program will be articulated. Finally, we look into the performance of the proposed method in a matrix processing application , when it is compare to the fully associative and other prefetching methods.

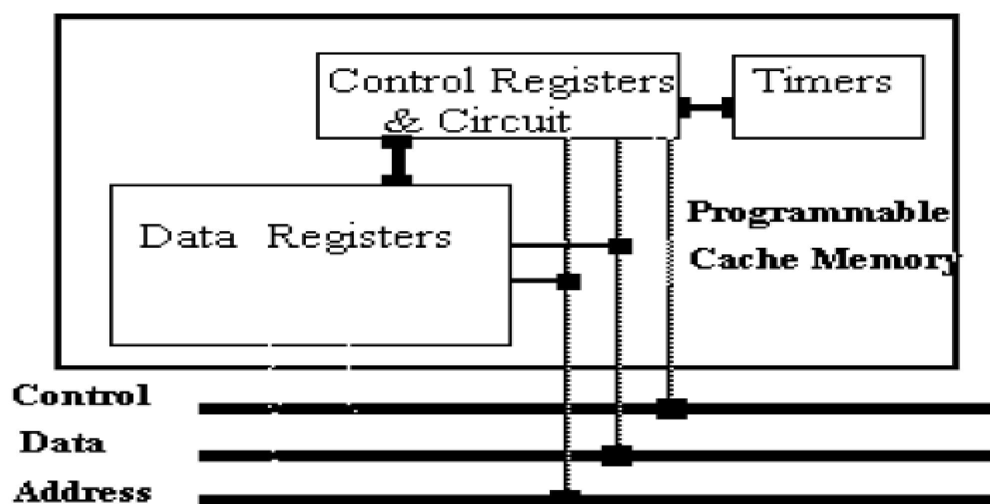

Fig 1.1 The block diagram of the programmable cache memory

The maximum show of shared memory systems is very sensitive to both bus bandwidth and memory access time. Since cache memories considerably reduce both bus traffic and average access time, they are an essential element of this class of multiprocessor. Figure 1.2 shows a multiprocessor system with a secret cache memory for each workstation. When using a secret cache memory for each workstation, it is necessary to make sure that all official copies of a given cache line are the same. (A cache line is the part of data transport between cache and main memory.) This requirement is called the ***multicache consistency* or *cache coherence* problem.**

P = processor
C = cache memory
X = bus transceiver
M = main memory

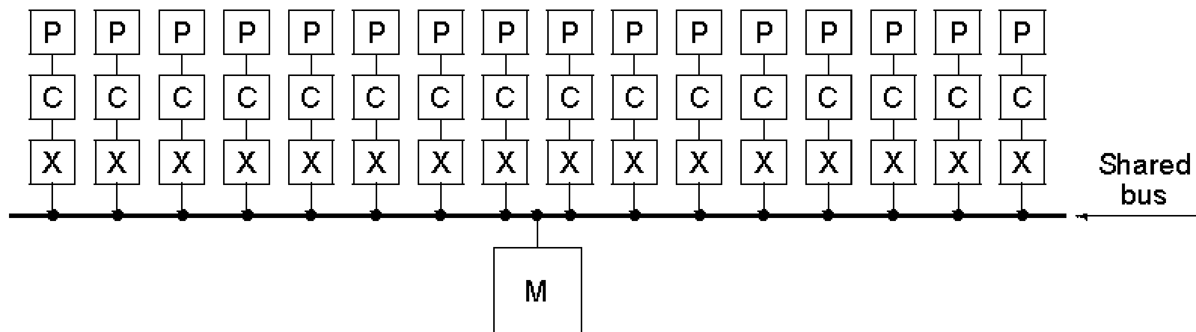

Fig 1.2: Shared memory multiprocessor with caches

Many solution to the cache uniformity problem have been planned, including software and hardware consistency algorithms, and combination of both. Software solutions require the compiler or operating system to identify all shared region of memory and to topic appropriate commands to the caches to guarantee constancy. This is typically done at the side of using lock variables to put in force jointly exclusive access to shared region and having each workstation glow its cache prior to releasing the shared exclusion lock. The large amount of memory transfer generated by the common cache flushes makes software solutions unrealistic for large numbers of processors. Furthermore, the requirement that shared regions be identified by software is difficult to achieve and can add notably to the density of the compiler and operating system. The most promise hardware solutions to the cache constancy problem require that all processors share a common main memory bus (or the consistent comparable). Each cache monitors all bus activity to classify reference to its lines by other caches in the system. This monitoring is termed as *snooping* on the bus or *bus watching*. The advantage of nosy caches is that regularity is managed by the hardware in a decentralized trend, avoiding the transfer jam of a central directory.

CACHE MEMORIES

It is one of the most effectual solutions to the bandwidth difficulty of multis is to relate a cache memory with each CPU. A cache is a buffer memory used to momentarily hold copies of portions of main memory that are presently in use. A cache memory appreciably reduces the main memory traffic for each workstation, since mainly memory references are handled in the cache.

BASIC CACHE MEMORY ARCHITECTURE

The simplest cache memory preparation is called a *direct mapped* cache. The basic unit of data in a cache is called a *row* (also sometimes called a *block*). All position in a cache are the same size, and this size is firm by the exacting cache hardware plan. In current technology, the row size is always either the basic word size of
the engine or the product of the word size and a small essential power of two. For example, most current processors have a 32 bit (4 byte) word size. For these processors, cache row sizes of 4, 8, 16, 32, or 64 bytes would be common. Associated with each line of data is an attend to tag and some control in sequence. The mixture of a data row and its associated address tag and control in sequence is called a cache entry. The cache exposed in figure has eight entries. In sensible cache designs, the amount of entries is generally a power of two in the range 64 to 8192. The process of this cache begins when an attend to is received from the CPU. The address is separated into a line number and a page number, with the maximum order bits forming the line number. In the example shown, only the three lowest bits would be used to form the row number, since there are only eight lines to choose from. The line number is used as an address into the cache memory to choice the

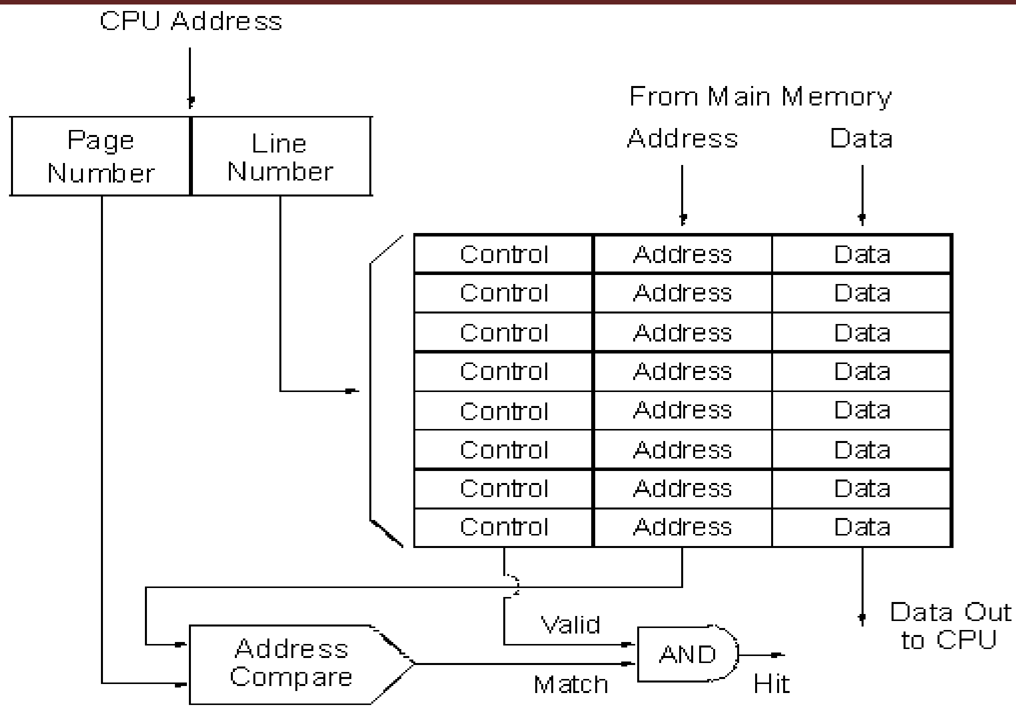

Figure 2.1: Direct mapped cache

FIG 2.1 DIRECT MAPPED CACHE

Appropriate row of data along with its address tag and control information. The address tag from the cache is compared with the page number from the CPU address to see if the row stored in the cache is from the required page. It is also essential to check a bit in the organize information for the line to see if it contains valid data. The figures in a line may be invalid for several reasons: the line has not been used since the method was initialized, the line was invalidate by the operating system after a situation switch, or the line was invalidated as branch of a cache consistency protocol. If the addresses match and the line is suitable, the position is said to be a *hit*. Otherwise, the position is classified as a *miss*. If the CPU was performing a read process and a hit occurred, the data from the cache is used, avoiding the bus travel and delay that would occur if the data had to be obtained from main memory. If the CPU was performing a write action and a hit occurred, bus usage is dependent on the cache design. The two common approaches to handling write operations are *write through* (also called *store through*) and *write back* (also called *copy back*, *store back*, or *write to*). In a write during cache, when a write operation modifies a line in the cache, the new data is also directly transmitted to main memory. In a write back cache, write operations affect just the cache, and main memory is updated later when the line is separated from the cache. This typically occurs when the line must be replaced by a new row from a different main memory address. When a miss occurs, the required data must be read from or written to main memory using the method bus. The suitable cache row must also be loaded, along with its corresponding address tag. If a write back cache is being used, it is essential to decide whether bringing a new row into the cache will swap a row that is valid and has been customized since it was loaded from main memory. Such a row is said to be *dirty*. Dirty position are identified by keeping a bit in the control in sequence associated with the line that is set when the row is written to and cleared when a new row is loaded from main memory. This bit is called a *dirty bit*. The logic used to control the transport of lines between the cache and main memo y is not shown in aspect in Figure 2.1. The propose shown in Figure 2.1 is called a *direct mapped* cache, since each line in chief memory has only a single place in the cache into which it may be placed. A disadvantage of this plan is that if two or more commonly referenced locations in main memory chart to the same location in the cache, only one of them can still be in the cache at any given time. To defeat this limitation, a plan called a *set associative* cache may be used. In a two way set associative cache, the whole memory array and its associated address comparator logic is simulated twice. When an address is obtained from the CPU, both halves are checked concurrently for a possible hit. The benefit of this scheme is that each line in main memory now has two likely cache locations instead of one. The disadvantages are that two sets of address contrast logic are needed and supplementary logic is needed to determine which half to load a new row into when a miss occurs. In commercially available machines, the degree of set associatively has always been a power of two ranging from individual (direct mapped) to sixteen. A cache which allows a row from main memory to be placed in any position in the cache is called a *fully associative* cache. Although this plan completely eliminates the difficulty of having multiple memory lines map to the same cache location, it requires an address comparator for every line in the cache. This makes it impractical to construct large fully associative caches, although advances in VLSI technology may ultimately permit their construction. Almost all modern mainframe computers, and many smaller machines, use

cache memories to develop performance. Cache memories develop performance because they have much shorter contact times than
main memories, usually by a factor of four to ten. Two factors put in to their speed. Since cache memories are a lot smaller than main memory, it is practical to use a very fast memory technology such as ECL (emitter coupled logic) RAM. Cost and heat debauchery limitations usually force the use of a slower expertise such as MOS dynamic RAM for main memory. Cache memories also can have closer objective and rational proximity to the workstation since they are smaller and are normally accessed by only a single workstation, while main memory must be reachable to all processors in a multi. The use of cache memories are so invasive in today's computer systems it is complex to dream processors without them. Cache memories, along with virtual memories and workstation registers form a continuum of memory hierarchies that rely on the principle of region of reference. Most applications exhibit sequential and spatial localities among information and data. Spatial region implies that memory locations that are spatially (address-wise) near the currently referenced address will likely be referenced. Temporal locality implies that the currently referenced address will expected be referenced in the near future (time-wise). Memory hierarchies are planned to keep most likely referenced items in the fastest procedure.
This grades in an effective reduction in access time.

CACHE MEMORIES IN UNIPROCESSORS.

A cache memory can be viewed as a amount of cache blocks (or cache lines), organized into a number of sets, each set containing one or more cache shape. Each cache line contains several bytes of data and a tag to identify the (virtual) memory address to which the data belongs. In direct mapped caches each set consists of a solo cache line, while in a k-way set associative cache, each set contains k cache appearance. The virtual address encountered by the workstation is divided into a byte-offset (to locate the byte within a cache line), a set index which locates a set (with k cache lines), and a Tag which will be compared with the Tag standards of the k cache lines in the set: a tag match in any cache line indicates a "hit". On a miss, the cache memory is laden with the missing data from main memory. Higher set associatively will reduce the quantity of cache misses caused by a program, but, direct mapped caches are more common because they are less compound and can use faster clock signals.

READING VERSUS WRITING.

On a read access, it is potential to read the cache line at the same time that the tag is read and compared. On a hit, the necessary data is readily available, and on a miss, the read data can be excess awaiting cache line replacement. However, on a write hit, it is essential to first read the cache line, modify only the effected bytes before writing the row back to cache. It is possible to pipeline writes to cover the two steps required for a write. It is also essential to address how the main memory is notified of the changes made to cache line. In *write-through*, all writes to a cache row will also be written to main memory. Often buffers are used to procedure writes to main memory, to avoid jamming the processor on a write. In *write-back* (or *copy back*) the main memory is reorganized only when a cache line is replaced. Write back practice is more common since it eliminates some writes to main memory and reduces memory bus traffic. However, write back can result in incompatible data in main memory (see Cache Coherency below). Multi-level caches may use write-through from L-1 (closest to CPU) to L-2 (second level cache), and write-back from L-2 to chief memory.

IMPROVING PERFORMANCE.

Numerous methods have been projected and implemented to enhance the performance of cache memories. Two factors pressure the performance of cache memories: *miss rate* and *miss penalty*. Miss rate refers to the ratio of memory references that cause cache misses to the total number of recollection references. Miss penalty refers to the amount of time needed to handle a cache miss. Strategies for dropping miss rates include: victim caches, higher set-associatively, big cache lines and prefetching of cache lines. Victim caches are effective as they contain a small fully associative cache that hold freshly displaced (i.e. victims) cache lines so that future references to the victims can be content by the victim cache. Strategies for reducing miss punishment include non-blocking caches and multi-level caches. Non-blocking caches allow access to cache lines while one or more cache misses are until. These and other variation have found their performance in modern uniprocessors such as Pentium II, PowerPC, DEC Alpha.

CACHE COHERENCY.

The inconsistency that exists between main memory and write-back caches in a uniprocessor organization does not cause any problems. But techniques are needed to ensure that stable data is available to all processors in a multiprocessor method. Cache coherency can be maintained besides by hardware techniques or software techniques. We will first set up hardware solutions
.

SNOOPY PROTOCOLS

Shared memory multiprocessors (also known as Symmetric Multiprocessors, SMPs) with a little number processing nodes attached to a common bus are gradually replacing high-end workstations in both systematic and commercial arenas. In such systems, the common (shared) memory is likewise accessible to all processors. In addition to the shared memory, each workstation contains a local cache memory (or multi-level caches). As noted above, write-back cache can guide to inconsistencies

between the data in main memory and that in a local cache. Since all cache memories (or the controller hardware) are attached to a common bus, the cache memories can *snoop* on the bus for maintaining logical data.

In such protocols, each cache line is related with a state, and the cache director will modify the states to track changes to cache appearance made either locally or remotely. A hit on a read implies that the cache line is steady with that in main memory. A read miss leads to a demand for the data. This demand can be satisfied by either the main memory (if no other cache has a copy of the data), or by a different cache which has a (possibly newer) copy of the data. Initially, when only one cache has a duplicate, the collection line is set to *Exclusive* state. However, when other caches demand for a read duplicate, the state of the cache line (in all processors) is set to *Shared*. Consider what happens when a workstation attempts to write to a (local) cache line. On a hit, if the state of the limited cache line is *Exclusive* (or *Modified*), the write can continue without any delay, and situation is hanged to *modified*. If the local state is *Shared*, then an dissolution signal must be broadcast on the regular bus, so that all other caches will set their cache lines to *Invalid* situation. Following the invalidation, the write can be finished in local cache, changing the situation to *customized*.

On a write-miss demand is placed on the common bus. If no other cache contains a duplicate, the data comes from memory, the write can be finished by the workstation and the cache line is set to *customized*. If other caches have the requested data in *Shared* situation, the copies are invalidated, the write can full with a single *Modified* copy. If a different workstation has a *Modified* copy, the data is written back to chief memory and the workstation invalidates its copy. The write can now be finished, leading to a *Modified* line at the requesting workstation. Such snoopy protocols are sometimes called MESI, standing for the names of states connected with cache lines: Modified, Exclusive, Shared or Invalid. Instead of invalidating shared copies on a write, it may be potential to provide updated copies. The trade-off between withdrawal and Update centres' on the misses caused by invalidations as compared to the improved bus travel due to updates.

DIRECTORY PROTOCOLS

Snoopy protocols rely on the capability to listen and broadcast invalidations on a general bus. However, the common bus also seats a limit on the number of processing nodes in a SMP system. Large scale multiprocessor and scattered systems must use more multipart interconnection mechanisms, such as many buses, N-dimensional grids, Crossbar switch and multistage interconnection networks (see Network Topologies and Network Architecture). New techniques are required to assure that invalidation messages are received by all caches with copies of the collective data. This is usually achieved by keeping a directory with main memory units at each site. There exists one directory entry related to each cache block, and the entrance keeps track of shared copies, or the identification of the workstation that contains modified data. On a read miss, a workstation requests the memory unit for data. The demand may go to a remote memory unit depending on the address. If the data is not customized, a copy is send to the requesting cache, and the directory entry is customized to reflect the existence of a collective copy. If a modified copy exists at a diverse cache, the new data is written back to the memory, a copy of the data is provided to their questing cache, and the index is marked as shared. In order to switch writes, it is essential to maintain state in sequence with each cache block at local caches, somewhat parallel to the Snoopy protocols. On a write hit, the write can ensue immediately if the status of the cache line is customized Otherwise (the state is shared), Invalidation significance is communicated to the memory unit, which in turn sends invalidation signals to all caches with joint copies. Only after the conclusion of this procedure can the processor proceed with a write. The directory is marked to reflect the existence of a modified copy. A write miss is handled as a combination of read-miss and write-hit. Notice that in the advance outlined here, the directory related with each memory unit is accountable for tracking the shared copies and for sending invalidation signals. We can consider distribute the work as follows. On the first demand for data, the memory unit provisions the requested data, and marks the directory with the requesting processor identification. Future read requests will be forward to the processor which has the copy. The processors with copies of the data are thus concurrent, and track all shared copies. On a write request, an cancellation signal is sent along the concurrent list to all collective copies. The memory unit can also send the classification of the writer so that invalidations can be approved directly by the processors with shared copies. Scalable Coherence Interface (SCI) standard uses a all the more linked list of shared copies. This permits a processor to remove itself starting the linked list when it no longer contains a copy of the shared cache line. Numerous variations have been proposed and implemented to improve the concert of the directory based protocols. Hybrid techniques that combine Snoopy protocols with address list based protocols have also been investigated in Stanford DASH system. Such systems can be viewed as networks of clusters, where each cluster relies on bus snooping and use directories crosswise clusters (see Cluster Computing).

SOFTWARE BASED COHERENCY TECHNIQUES.

Using large cache blocks can decrease certain types of overheads in maintaining consistency as well as reduce the overall cache miss rates. However, larger cache blocks will increase the opportunity of false-sharing. False sharing refers to the state of affairs when 2 or more processors which do not really share any specific memory address, however they appear to split a cache line, since the variables (or addresses) accessed by the different processors fall to the same cache line. Compile time analysis can detect and get rid of unnecessary invalidations in some false sharing cases. Software can also help in improving the performance of hardware based coherency techniques described above. It is possible to sense when a processor no longer accesses a cache line (or variable), and "self invalidation" can be used to get rid of unnecessary invalidation signals. Migration of processes or threads on or after one node to another can lead to poor cache performances since the migration can cause "false" sharing: the original node everywhere the thread resided may falsely assume that cache lines are shared with the new node to where the thread migrated. Some software techniques to selectively invalidate cache lines when clothes migrate have been proposed. Software aided prefetching of cache lines is repeatedly used to reduce cache misses. In shared memory systems, prefetching may actually increase misses, unless it is potential to predict if a prefetched cache line will be invalidated before its use.

CACHE OPERATION

The successful process of a cache memory depends on the locality of memory references. Over short periods of time, the memory references of a program will be distributed nonconsistently over its address space, and the portions of the address space which are referenced most commonly tend to remain the same over long periods of time. Several factors contribute to this locality: most directions are executed sequentially programs spend much of their time in loops, and related data items are frequently stored near each other. Locality can be considered by two properties. The first, reprocess or *temporal locality*, refers to the fact that a substantial portion of locations referenced in the near future will have been referenced in the recent past. The second, prefetch or *spatial locality*, refers to the fact that a considerable fraction of locations referenced in the near future will be to locations near recent past references. Caches exploit chronological locality by saving recently referenced data so it can be rapidly accessed for future reuse. They can take advantage of spatial locality by prefetching in sequence lines consisting of the stuffing of several contiguous memory locations. Several of the cache design parameters will have a noteworthy effect on system performance. The choice of line size is significant. Small lines have several **advantages**:

- They necessitate less time to transmit between main memory and cache.
- They are less likely to contain superfluous information.
- They require fewer memory cycles to access if the main memory size is narrow.

On the other hand, large lines also have **advantages**:

- They necessitate fewer address tag bits in the cache.
- They reduce the number of fetch operations if all the information in the line is actually essential prefetch).
- Acceptable performance is achievable with a lower degree of set associativity.
- Since the unit of relocate between the cache and main memory is one line, a line size of less than the bus width could not use the full bus width. Thus it absolutely does not make sense to have a line size smaller than the bus width.

CACHE CONSISTENCY

A problem with cache recollections in multiprocessor systems is that modifications to data in one cache are not necessarily reflected in all caches, so it may be possible for a processor to location data that is not current. Such data is called *stale data*, and this problem is termed as the *cache consistency* or *cache coherence* problem. The benchmark software solution to the cache consistency problem is to place all shared writable data in non-cacheable storage and to flush a processor's cache each time the processor performs a context switch. Since collective writable data is non-cacheable, it cannot become inconsistent in any cache. Unshared data could potentially become inconsistent if a development migrates from one processor to another; however, the cache flush on context switch prevents this situation from occurring. Although this scheme does offer consistency, it does so at a very high cost to performance. The classical hardware solution to the cache consistency problem is to transmit all writes. Each cache sends the address of the customized line to all other caches. The other caches invalidate the modified line if they have it. Although this scheme is simple to put into practice, it is not practical unless the number of processors is very small. As the number of processors is increased, the cache traffic resulting from the broadcasts rapidly becomes exorbitant. An alternative approach is to use a federal directory that records the location or locations of each line in the system. Although it is better than the broadcast scheme, since it avoids interfering with the cache accesses of other processors, address list access conflicts can become a restricted access. The most practical solutions to the cache consistency problem in a system with a large number of processors employ variations on the directory scheme in which the directory information is distributed among the caches. These schemes make it possible to construct systems in which the simply limit on the maximum

Number of processors is that compulsory by the total bus and memory bandwidth. They are called "snooping cache" schemes, since each cache must monitor addresses on the system bus, inspection each reference for a possible cache hit. They have also been referred to as "two-bit directory" schemes, since each line in the cache usually has two bits associated with it to specify one of four states for the data in the line. Describes the use of a cache memory to reduce bus traffic and presents a description of the *write-once* cache policy, a easy snooping cache scheme. The write-once scheme takes advantage of the broadcast ability of the shared bus between the local caches and the global main memory to energetically classify cached data as local or shared, thus ensuring cache consistency without broadcasting every write operation or using a global directory. Goodman defines the four cache line states as follows: 1) **Invalid**, here is no information in the line; 2) **Valid**, there is data in the line which has been read beginning main memory and has not been modified (this is the state which always results after a read miss has been serviced); 3) **Reserved**, the data in the line has been in the vicinity modified exactly once since it has been brought into the cache and the change has been written through to main memory; and 4) **Dirty**, the data in the line has been locally customized more than once since it was brought into the cache and the latest change has not been transmitted to main memory. Since this is a snooping cache scheme, each cache must check the system bus and check all bus references for hits. If a hit occurs on a bus write operation, the appropriate line in the cache is marked invalid. If a hit occurs on a read operation, no action is taken unless the state of the line is reserved or dirty, in which case its state is changed to valid. If the line was dirty, the cache must restrain the read operation on main memory and supply the data itself. This data is transmitted to both the cache manufacture the request and main memory. The arrangement of the protocol ensures that no more than one copy of a particular line can be dirty at any one time. The need for access to the cache address tags by both the limited processor and the system bus makes these tags a potential bottleneck. To ease this problem, two identical copies of the tag memory can be kept, one for the local processor and one for the system bus. Since the

tags are read much more often than they are written, this allows the processor and bus to access them concurrently in most cases. An alternative would be to use dual ported memory for the tags, although currently available dual ported recollections are either too expensive, too slow, or both to make this approach very attractive. Goodman used simulation to investigate the performance of the write-once system. In terms of bus traffic, it was found to execute about as well as write back and it was superior to write through. The states are named **Invalid**, **Exclusive- Unmodified**, **Shared-Unmodified**, and **Exclusive-Modified**, matching respectively to **Invalid**, **Reserved**, **Valid**, and **Dirty**. The scheme is nearly the same to the write-once scheme, except that when a line is loaded following a read miss, its state is set to Exclusive-Unmodified if the line was obtain from main memory, and it is put to Shared-Unmodified if the line was obtained from another cache, while in the write-once scheme the state would be set to Valid (Shared-Unmodified) in spite of of where the data is obtained notes that the change reduce unnecessary bus traffic when a line is written after it is read. An estimated analysis was used to approximation the performance of this scheme, and it appears to perform well as long as the fraction of data that is shared between processors is small. describes two supplementary versions of snooping cache schemes. The first, called the **RB** scheme (for "read broadcast"), has only three states, called **Invalid**, **Read**, and **Local**. The read and local states are alike to the valid and dirty states, respectively, in the write-once scheme of, while there is no state matching to the reserved state (a "dirty" state is assumed immediately after the first write). The second, called the **RWB** scheme (in all probability for "read write broadcast"), adds a fourth state called **First** which corresponds to the engaged state in write-once. A feature of RWB not present in write-once is that when a cache detects that a line read from main memory by an additional processor will hit on an invalid line, the data is encumbered into the invalid line on the grounds that it might be used, while the invalid line will surely not be useful. The advantages of this are controversial, since loading the line will tie up cache cycles that might be used by the processor on that cache, and the chance of the line being used may be low is concerned primarily with formal exactness proofs of these schemes and does not consider the performance implications of realistic implementations of them.

PERFORMANCE OF CACHE CONSISTENCY MECHANISMS

Even though the snooping cache approaches appear to be similar to broadcasting writes, their presentation is much better. Since the caches record the shared or restricted status of each line, it is only necessary to broadcast writes to shared lines on the bus; bus activity for exclusive lines is avoided. Thus, the cache bandwidth trouble is much less strict than for the broadcast writes scheme. The protocols for enforcing cache consistency with snooping caches can be separated into two major classes. Both use the snooping hardware to vigorously identify shared writable lines, but they vary in the way in which write operations to shared lines are handled. In the first class of protocols, when a processor writes to a shared line, the address of the line is broadcast on the bus to all other caches, which then nullify the line. Two examples are the Illinois protocol and the Berkeley Ownership Protocol . Protocols in this class are termed as *write-invalidate* protocols. In the second class of protocols, when a processor writes to a shared line, the written data is broadcast on the bus to all other caches, which then bring up to date their copies of the line. Cache invalidations are by no means performed by the cache consistency protocol. Two examples are the protocol in DEC's Firefly multiprocessor workplace and that in the Xerox Dragon multiprocessor. Protocols in this group are called *write-broadcast* protocols. Each of these two classes of protocol has convinced advantages and disadvantages, depending on the pattern of references to the shared data. For a shared data line that tends to be read and written several times in succession by a single processor before a different processor references the same line, the write-invalidate protocols execute better than the write-broadcast protocols. The write-invalidate protocols use the bus to nullify the other copies of a shared line each time a new processor makes its first reference to that shared line, and then no further bus accesses are necessary until a different processor accesses that line. Invalidation can be performed in a single bus cycle, since only the address of the customized line must be transmitted. The write-broadcast protocols, on the other hand, must use the bus for every write operation to the shared data, even when a solitary processor writes to the data several times consecutively. Furthermore, multiple bus cycles may be needed intended for the write, since both an address and data must be transmitted. For a shared data line that tends to be read much more than it is written, with writes occurring from arbitrary processors, the write-broadcast protocols tend to perform enhanced than the write-invalidate protocols.

SUMMARY AND CONCLUSIONS

Cache memories are a significant component of modern high performance computer systems, especially multiprocessor systems. When cache memories are used in a multiprocessor system, it is essential to prevent data from being modified in numerous caches in an inconsistent manner. Efficient means for ensuring cache consistency require a shared bus, so that each one cache can monitor the memory references of the other caches. Multiple buses can be used to obtain higher entirety bandwidth, but they introduce tricky cache consistency and bus arbitration problems. A modified multiple bus structural design that avoids these problems of this thesis. The single shared bus multiprocessor has been for the mainly part commercially successful multiprocessor system design up to this time. Electrical loading trouble and incomplete bandwidth of the shared bus have been the most limiting factors in these systems. This critique presents design for logical buses that will allow snooping cache protocols to be used lacking the electrical loading problems that result from attaching all processors to a single bus. A new bus bandwidth model was residential that considers the effects of electrical loading of the bus as a function of the number of processors. Using this model, most advantageous bus configurations can be determined. Trace driven simulations show that the presentation estimates obtained from the bus model developed have the same opinion closely with the performance that can be expected when running a realistic multiprogramming workload. The model was also tried with a parallel workload to investigate the effects of violating the liberty assumption. It was found that violation of the independence assumption produced large errors in the mean service time estimate, but the bus consumption estimate was still reasonably accurate (within 6% for the workload used). a new system association was

proposed to allow systems with more than one reminiscence bus to be constructed. This architecture is fundamentally a crossbar network with a cache memory at each cross point. A two-level cache organization is suitable for this architecture. A small cache may be placed close to each processor, if at all possible on the CPU chip, to minimize the effective memory access time. A outsized cache built from slower, less expensive memory is then placed at each cross point to minimize the bus traffic. Cache memories play a important role in improving the performance of today's computer systems. frequent techniques for the use of caches and for maintaining coherent data have been proposed and implemented in business SMP systems. The use of cache memories in a distributed dispensation system, however, must be carefully understood to benefit from them. In a related research on memory consistency models, performance improvement are achieved by require data consistency only at synchronization points (see Memory Consistency).

FUTURE SCOPE:-

It would be useful to find a method for influential the optimal interconnection topology as a function of the number of processors and the delay characteristics of the transceiver technology. To provide input data for future imitation studies, a much larger collection of practical programs could be used, both for multiprogramming and parallel algorithm workloads. Also, performance could be investigated in a mixed surroundings in which both multiprogramming and parallel workloads coexist. a simple protocol to enforce cache consistency with direct mapped caches was presented. Further studies could examine protocols suitable for use with set associative caches. Also, optional protocols could be investigated. An alternative approach would be to use a copy back policy for the on-chip caches, which would decrease the traffic on the processor buses but would appreciably complicate the actions required when a snoop hit occurs. Performance studies of these alternative implementations could be performed. Another interesting topic for future work would be to investigate the most efficient ways of incorporating implicit memory support into the cross point cache architecture. Tradeoffs between virtual and physical address caches and could be investigated, along with tradeoffs involving transformation look a side buffer placement. In summary, this work may be extended by considering alternative bus implementations, more miscellaneous workloads, and additional cache implementations and policies, and by investigating the implications of supporting virtual memory. Selected papers on software base cache coherency in distributed systems. The investigate in cache memories very active and the reader can find more recent techniques for improving cache performance in yearly conference proceedings, such as the International conference on Computer Architecture, High Performance Computer Architecture, Symposium on Architectural sustain for Programming Languages and Operating Systems.

REFERENCES

1. RUSSELL R. ATKINSON AND EDWARD M. MCCREIGHT. "The Dragon Processor". *Proceedings Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Palo Alto, California, IEEE Computer Society Press, October 5–8, 1987, pages 65–69.

2. JAMES K. ARCHIBALD. "A Cache Coherence Approach For Large Multiprocessor Systems". *1988 International Conference on Supercomputing*, St. Malo, France, ACM Press, July 4–8, 1988, pages 337–345.

3. C. GORDON BELL. "Multis: A New Class of Multiprocessor Computers". *Science*, volume 228, number 4698, April 26, 1985, pages 462–467.

4. PAUL L. BORRILL. "MicroStandards Special Feature: A Comparison of 32-Bit Buses". *IEEE Micro*, IEEE Computer Society, volume 5, number 6, December 1985, pages 71–79.

5. REED BOWLBY. "The DIP may take its final bows". *IEEE Spectrum*, volume 22, number 6, June 1985, pages 37–42.

6. KHALED A. EL-AYAT AND RAKESH K. AGARWAL. "The Intel 80386 — Architecture and Implementation". *IEEE Micro*, IEEE Computer Society, volume 5, number 6, December 1985, pages 4–22.

7. JAMES R. GOODMAN. "Using Cache Memory to Reduce Processor–Memory Traffic". *The 10th Annual International Symposium on Computer Architecture Conference Proceedings*, Stockholm, Sweden, IEEE Computer Society Press, June 13–17, 1983, pages 124–131
8. HUMOUD B. HUMOUD. *A Study in Memory Interference Models*. Ph.D. dissertation, The University of Michigan Computing Research Laboratory, Ann Arbor, Michigan, April 1985.

9. TOM´AS LANG, MATEO VALERO, AND IGNACIO ALEGRE. "Bandwidth of Crossbar and Multiple-Bus Connections for Multiprocessors". *IEEE Transactions on Computers*, volume C-31, number 12, December 1982, pages 1227–1234.

10. ALAN JAY SMITH. "Cache Memories". *Computing Surveys*, Association for Computing Machinery, volume 14, number 3, September 1982, pages 473–530.