ISSN 2319 – 7757 EISSN 2319 –7765

Indian Journal of Engineering

E-cache memory becoming a boon towards memory management system

Ankita Goel, Dimple Ahuja, Aaisha Khanam, Renu Yadav, Jyoti Yadav

Dronacharya College of engineering, Gurgaon, Haryana-06, India

Received 06 August; accepted 21 September; published online 15 October; printed 28 October 2013

ABSTRACT

The word "Cache memory" means it is random access memory (RAM) that a computer microprocessor can access more quickly than it can access regular RAM. The cache is a smaller, faster memory which stores copies of the data from frequently used main memory locations. As the microprocessor processes data, it looks first in the cache memory and if it finds the data there (from a previous reading of data), it does not have to do the more time-consuming reading of data from larger memory. Cache memory is sometimes described in levels of closeness and accessibility to the microprocessor. Here, in this thesis we will first define the various cache memory issues and how they affects the memory performance in multiprocessors with various "cache-write-policies" and finally brief their solutions and conclusions along with cache coherency with their protocols.

Keywords: Cache memories; cache coherency; protocols; write-back policies; Snoopy Protocol; Virtual Address Cache

To Cite This Article
Ankita Goel, Dimple Ahuja, Aaisha Khanam, Renu Yadav, Jyoti Yadav. E-cache memory becoming a boon towards memory management system, *Indian Journal of Engineering*, 2013, 5(13), 38-43

1. INTRODUCTION

When the processor needs to read from or write to a location in main memory, it first checks whether a copy of that data is in the cache. If so, the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to main memory. "Cache memories are small fast memories used to temporarily hold the contents of portions of main memory that are (believed to be) likely to be used. The basic concepts of using cache memories to improve processor performance have been well studied and understood. Today, caches have become an integral part of all processors. However, as the performance gap between processor and main memory continues to widen, increasingly optimized implementations of caches are needed" (Jih-Kwon Peir Windsor). Most modern desktop and server CPUs have at least three independent caches: an instruction cache to speed up executable instruction fetch, a data cache to speed up data fetch and store, and a translation look aside buffer (TLB) used to speed up virtual-to-physical address translation for both executable instructions and data. The data cache is usually organized as a hierarchy of more cache levels (L1, L2, etc.), (Figure 1).

2. FUNDAMENTALS OF CACHE MEMORY

The maximum performance of shared memory systems is extremely sensitive to both bus bandwidth and memory access time. Cache memories are an essential component of the class of multiprocessor since they significantly reduce both bus traffic and average access time. Multiprocessor system with a private cache memory is shown for each processor in Figure 2. It is necessary to ensure that all valid copies of a given cache line are the same, when using a private cache memory for each processor. (The unit of data transfer between cache and main memory is what; we call it as a cache line). This requirement is called the '*multi-cache consistency*' or '*cache coherence*' problem.

3. CACHE COHERENCY

Proposed multiprocessor designs often include a private cache for each processor in the system, which gives rise to the cache coherence problem. The inconsistency that exists between main memory and write-back caches in a uniprocessor system does not cause any problems. A mechanism must exist if multiple caches are allowed to have copies simultaneously of a given memory location, so as to ensure that when the contents of that memory location are modified then all copies remain consistent i.e.; consistent data is available to all processors in a multiprocessor system. In some systems, to prevent the existence of multiple copies i.e.; cache coherency, a software and hardware approach is taken by marking shared blocks as not to be cached, and by restricting or prohibiting task migration. To maintain consistency, an alternate approach is to allow all blocks to be cached by all processors and to rely on a cache coherence protocol (between the cache controllers and, in some cases, memory controllers). Many such protocols have been described or proposed as follows.

3.1. Some of the hardware solutions are

3.1.1. Snoopy Protocols

"Shared memory multiprocessors (also known as Symmetric Multiprocessors, SMPs) with a small number processing nodes connected to a common bus are gradually replacing high-end workstations in both scientific and commercial arenas. In such

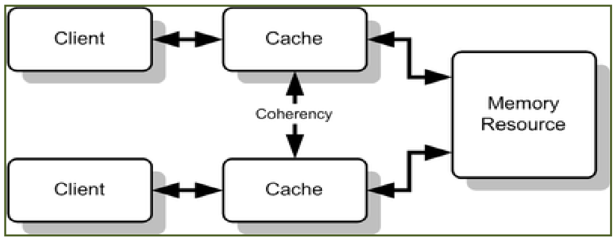

Figure 1

Description of cache with the CPU

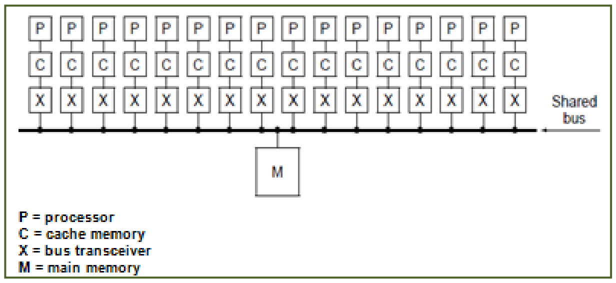

Figure 2

Shared memory multiprocessor with caches [Donald Charles]

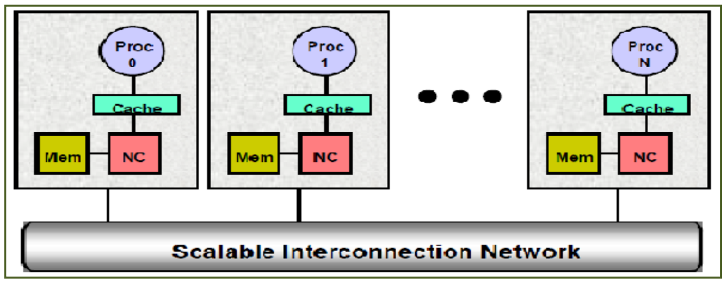

Figure 3

Distributed shared-memory architecture with directory-based cache coherence. Each node maintains a *directory* which tracks the sharing information of every cache line in that node's

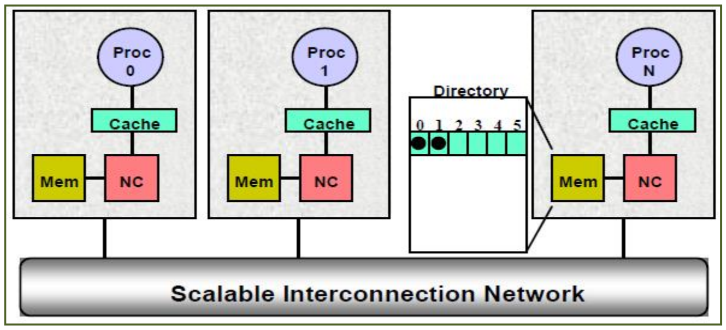

Figure 4

A distributed shared-memory architecture with directory-based cache coherence. Each node maintains a *directory* which tracks the sharing information of every cache line in that node's

systems, the common (shared) memory is equally accessible to all processors. In addition to the shared memory, each processor contains a local cache memory (or multi-level caches). As noted down, write-back cache can lead to inconsistencies between the data in main memory and that in a local cache. Since all cache memories (or the controller hardware) are connected to a common bus, the cache memories can *snoop* on the bus for maintaining coherent data. In such protocols, each cache line is associated with a state, and the cache controller will modify the states to track changes to cache lines made either locally or remotely. A hit on a read implies that the cache line is consistent with that in main memory. A read miss leads to a request for the data. This request can be satisfied by either the main memory (if no other cache has a copy of the data), or by another cache which has a (possibly newer) copy of the data. Initially, when only one cache has a copy, the cache line is set to *Exclusive* state. However, when other caches request for a read copy, the state of the cache line (in all processors) is set to *Shared*.

Consider what happens when a processor attempts to write to a (local) cache line. On a hit, if the state of the local cache line is *Exclusive* (or *Modified*), the write can proceed without any delay and state is changed to *Modified*. If the local state is *Shared*, then an invalidation signal must be broadcast on the common bus, so that all other caches will set their cache lines to *Invalid* state. Following the invalidation, the write can be completed in local cache, changing the state to *Modified*.

On a write-miss request is placed on the common bus. If no other cache contains a copy, the data comes from memory, the write can be completed by the processor and the cache line is set to *Modified*. If a different processor has a *Modified* copy, the data is written back to main memory and the processor invalidates its copy. The write can now be completed, leading to a *Modified* line at the requesting processor. Such snoopy protocols are sometimes called MESI, standing for the names of states associated with cache lines: Modified, Exclusive, Shared or Invalid" [Krishna M. Kavi].

3.1.2. Directory Protocols

There are two main classes of cache coherence protocols, *snoopy protocols* and *directory- based protocols*. Snoopy protocols require the use of a broadcast medium in the machine and hence apply only to small-scale bus-based multiprocessors. In these broadcast systems each cache "snoops" on the bus and watches for transactions which affect it. Any time a cache sees a write on the bus it invalidates that line out of its cache if it is present. Any time a cache sees a read request on the bus it checks to see if it has the most recent copy of the data, and if so, responds to the bus request. These snoopy bus-based systems are easy to build, but unfortunately as the number of processors on the bus increases, the single shared bus becomes a bandwidth bottleneck and the snoopy protocol's reliance on a broadcast mechanism becomes a severe scalability limitation.

To address these problems, architects have adopted the distributed shared memory (DSM) architecture. In a DSM multiprocessor each node contains the processor and its caches, a portion of the machine's physically distributed main memory, and a node controller which manages communication within and between nodes (see Figure 3). Rather than being connected by a single shared bus, the nodes are connected by a scalable interconnection network. The DSM architecture allows multiprocessors to scale to thousands of nodes, but the lack of a broadcast medium creates a problem for the cache coherence protocol. Snoopy protocols are no longer appropriate, so instead designers must use a directory-based cache coherence protocol. The *directory* is simply an auxiliary data structure that tracks the caching state of each cache line in the system. For each cache line in the system, the directory needs to track which caches, if any, have read-only copies of the line, or which cache has the latest copy of the line if the line is held exclusively. A directory-based cache-coherent machine works by consulting the directory on each cache miss and taking the appropriate action based on the type of request and the current state of the directory. Figure 4 shows a directory-based DSM machine. Just as main memory is physically distributed throughout the machine to improve aggregate memory bandwidth, so the directory is distributed to eliminate the bottleneck that would be caused by a single monolithic directory. If each node's main memory is divided into cache-line-sized blocks, then the directory can be thought of as extra bits of state for each block of main memory. Any time a processor wants to read cache line *L*, it must send a request to the node that has the directory for line *L*. This node is called the *home* node for *L*. The home node receives the request, consults the directory, and takes the appropriate action. On a cache read miss, for example, if the directory shows that the line is currently uncached or is cached read-only (the line is said to be *clean*) then the home node marks the requesting node as a sharer in the directory and replies to the requester with the copy of line *L* in main memory. If, however, the directory shows that a third node has the data modified in its cache (the line is *dirty*), the home node forwards the request to the *remote* third node and that node is responsible for retrieving the line from its cache and responding with the data. The remote node must also send a message back to the home indicating the success of the transaction. Even the simplified examples above will give the savvy reader an inkling for the complexities of implementing a full cache coherence protocol in a machine with distributed memories and distributed directories. Because the only serialization point is the directory itself, races and transient cases can happen at other points in the system, and the

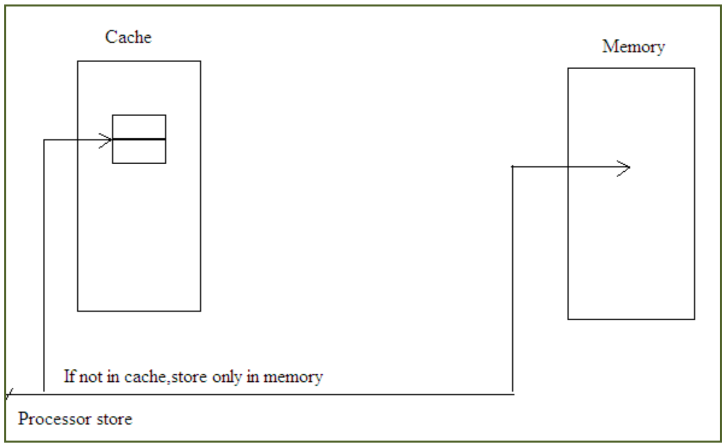

Figure 5
Write Policies

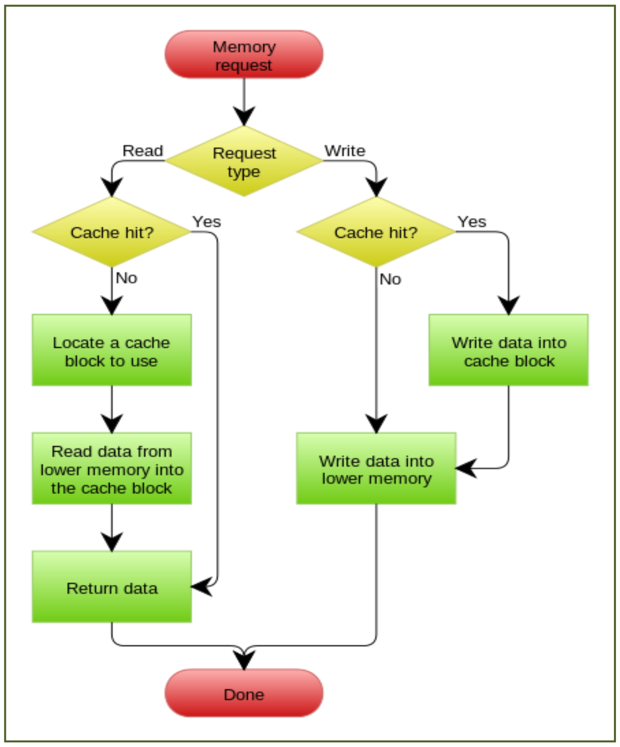
Memory request
Request type
Read
Write
Cache hit?
Yes
No
Cache hit?
Yes
No
Locate a cache block to use
Write data into cache block
Read data from lower memory into the cache block
Write data into lower memory
Return data
Done

Figure 6
Flowchart showing the working of write- through cache

cache coherence protocol is left to deal with the complexity. There are two major components to every directory-based cache coherence protocol:

- the directory organization
- the set of message types and message actions

The directory organization refers to the data structures used to store the directory information and directly affects the number of bits used to store the sharing information for each cache line. The memory required for the directory is a concern because it is "extra" memory that is not required by non-directory-based machines. The ratio of the directory memory to the total amount of memory is called the *directory memory overhead*. The designer would like to keep the directory memory overhead as low as possible and would like it to scale very slowly with machine size. The directory organization also has ramifications for the performance of directory accesses since some directory data structures may require more hardware to implement than others, have more state bits to check, or require traversal of linked lists rather than more static data structures. The directory organization holds the state of the cache coherence protocol, but the protocol must also send messages back and forth between nodes to communicate protocol state changes, data requests, and data replies. Each protocol message sent over the network has a type or opcode associated with it, and each node takes a specific action based on the type of message it receives and the current state of the system. The set of message actions include reading and updating the directory state as necessary, handling all possible race conditions, transient states, and "corner cases" in the protocol, composing any necessary response messages, and correctly managing the central resources of the machine, such as virtual lanes in the network, in a deadlock-free manner. Because the actions of the protocol are intimately related to the machine's deadlock avoidance strategy, it is very easy to design a protocol that will livelock or deadlock. It is much more complicated to design and implement a high-performance protocol that is deadlock-free. Variants of three major cache coherence protocols have been implemented in commercial DSM machines.

3.2. Software Based Coherency Techniques

Using large cache blocks can reduce certain types of overheads in maintaining coherence as well as reduce the overall cache miss rates. However, larger cache blocks will increase the possibility of false-sharing. False sharing refers to the situation when 2 or more processors which do not really share any specific memory address, however they appear to share 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 eliminate 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 detect when a processor no longer accesses a cache line (or variable), and "self-invalidation" can be used to eliminate unnecessary invalidation signals. Migration of processes or threads from one node to another can lead to poor cache performances since the migration can cause "false" sharing: the original node where 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 threads migrate have been proposed. Software aided prefetching of cache lines is often used to reduce cache misses. In shared memory systems, prefetching may actually increase misses, unless it is possible to predict if a prefetched cache line will be invalidated before its use.

4. MEMORY PERFORMANCE IN MULTIPROCESSORS

Blocks selected for replacement in the cache need to be written back to main memory only if in the DIRTY state(written more than once and the only copy in any cache). When the processor immediately finds the data in cache then it results in cache hit. The proportion of accesses that result in a cache hit is known as the hit rate, and can be measure of the effectiveness of the cache for a given program or algorithm. While, when the data is not found in cache then it results in cache miss. Read misses delay execution because they require data to be transferred from memory much more slowly than the cache itself. Write misses may occur without such penalty, since the processor can continue execution while data is copied to main memory in the background.

The scheme works as follows:
1. Read miss
If another copy of the block exists that is in state DIRTY, the cache with that copy inhibits the memory from supplying the data and supplies the block itself, as well as writing the block back to main memory. If no cache has a DIRTY copy, the block comes from memory. All caches with a copy of the block set their state to VALID (Daniel Molka).

2. Write hit
If the block is already DIRTY, the write can proceed locally without delay. If the block is in state RESERVED, the write can also proceed without delay, and the state is changed to DIRTY. If the block is in state VALID, the word being written is written through to main memory (i.e., the bus is obtained, and a one-word write to the backing store takes place) and the local state is set to RESERVED. Other caches with a copy of that black (if any) observe the bus write and change the state of their block

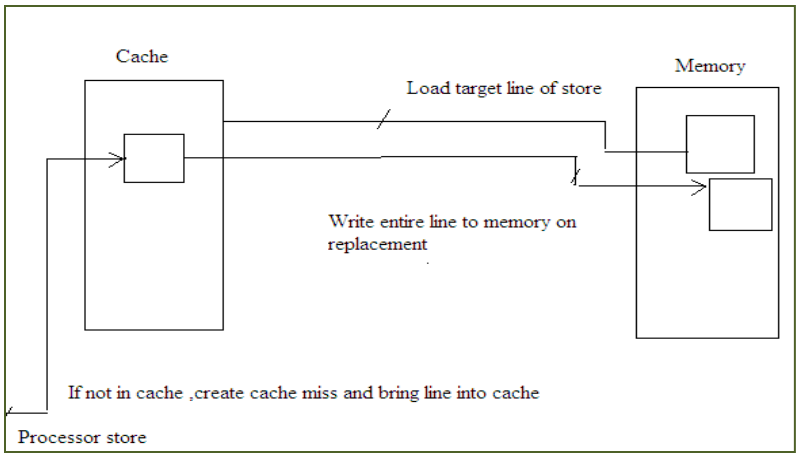
Cache
Memory
Load target line of store
Write entire line to memory on replacement
If not in cache ,create cache miss and bring line into cache
Processor store

Figure 7
Copyback cache

copies to INVALID. If the block is replaced in state RESERVED, it need not be written back, since the copy in main memory is current (James Archibald).

(3) Write miss
Like a read miss, the block is loaded from memory, or, if the block is DIRTY, from the cache that has the DIRTY copy, which then invalidates its copy. Upon seeing the write miss on the bus, all other caches with the block invalidate their copies. Once the block is loaded, the write takes place and the state is set to DIRTY(james archibald). "Today's microprocessors have complex memory subsystems with several cache levels. The efficient use of this memory hierarchy is crucial to gain optimal performance, especially on multicore processors"(Daniel Molka).
Multiprocessor computers can be categorized into two general divisions:

- **shared memory systems** (also known as tightly coupled systems)
- **distributed memory systems** (also known as loosely coupled systems)

Since, communication between processors is handled through the shared memory in shared memory systems, therefore; shared memory systems are generally easier to program than distributed memory systems. Also, there is no need of explicit message passing as well. On the other hand, shared memory systems generally require a more complex and costly interconnection network therefore; they also tend to be more expensive than distributed memory systems for a given level of peak performance.

5. CACHE IMPLEMENTATION ISSUES

5.1. Addressing Constraint

The access should be triggered as soon as the effective address of the memory reference becomes available, in order to minimize effective memory access time. In most computers, however, caches are addressed, as noted above, with the physical address, and thus there is a delay for translation. It is hard to avoid this delay completely; rather it can often be partially overlapped. Virtually addressed caches do not require address translation during cache access, but the fact that multiple virtual pages may be mapped to the same physical page greatly complicates their design.

5.1.1. Physical Address Cache

"Caches are organized as 2-dimensional arrays and are accessed in a two phase cycle .In the first phase, a cache set is selected by using a portion of the address known as the index bits. In the second phase, the remaining part of the address is used to make a further selection from within this cache set to yield either a cache miss determination or the requested data. This two phase access cycle means that only a portion of the address needs to be available at the onset of cache access. There are various techniques that exploit this two phase access cycle to enable a physically addressed cache to be accessed without requiring an extra address translation cycle. Because only the page number bits need to be translated, the un-translated bits are immediately available to start the access. For example, if the un-translated bits can be used to select the set, then for a 2K page size, we can overlap set selection with translation, and then does a J-way associative search among the elements of the set, for a cache of size J _ 2K. However, there is a practical limit to this approach because increasing the set-associativity provides only a fading return on cache hit ratio but adds hardware complexity and adversely impacts the access time. Rather than limiting the index bits to within the page offset, another approach is to increase the number of address bits available before address translation. A straightforward way to achieve this is to increase the page size. For many computer architectures, however, the page size is typically fixed and enlarging it would require substantial changes to both the architecture and the system software .In addition, it may lead to more memory fragmentation. In some newer architecture, however, the page size may be variable. A technique that can be used to increase the number of address bits available before address translation is to restrict the virtual to physical page mapping so that the low-order bits of the physical and virtual page numbers are identical. This amounts to implementing a set-associative rather than fully associative main memory. This technique, which has also been referred to as page coloring, may increase the number of page faults, although with associative of 8 and larger (in the main memory), the effect is likely to be negligible. Another way to make more address bits available before address translation is to predict the additional address bits .An example of a good predictor is the content of the base register that is used to compute the effective address .At the address generation stage, the low-order page address bits in the base register are used to index into a history-based prediction table to obtain the needed physical address bits. The base register content is an accurate predictor for the needed physical index bits because the displacement for computing the effective address is usually small and recently used pages tend to be re-referenced, i.e. there is locality of reference. Results reported in show that a 128-entry direct-mapped prediction table can achieve accuracy exceeding 98% for commercial workloads. Similar prediction schemes are presented in. The MIPS R6000 implements a TLB-slice which is essentially a predictor based on the low-order virtual page address" (Jih-Kwon Peir Windsor).

5.1.2. Virtual Address Cache

"Instead of using bits from the virtual address as a predictor for the physical address, a different approach is to use the virtual address to directly access the cache. This avoids the delay for translation; other TLB functions such as page protection, update, and reference in- formation can be resolved in parallel with cache access .In addition, all the addresses must be tagged with an address space identifier or else the cache must be purged on every task switch. The most serious drawback of the virtual address cache is that multiple virtual addresses may be mapped to the same physical address, i.e. synonyms may occur. Synonyms occur when code or data is shared. In addition, on some systems, the supervisor and its data structures exist in all the address spaces. Thus, in a virtual address cache, the absence of an address tag does not imply a miss because the data may be located elsewhere in the cache under a different virtual address. This is known as the synonym problem. The usual approach to handling synonyms is to prevent them from being present in the cache at the same time .Such an approach has the nice property that it adds no overhead to the frequent case of a cache hit. The only penalty comes in the less frequent case of a cache miss during which the cache has to be searched for any synonym entry. The way to

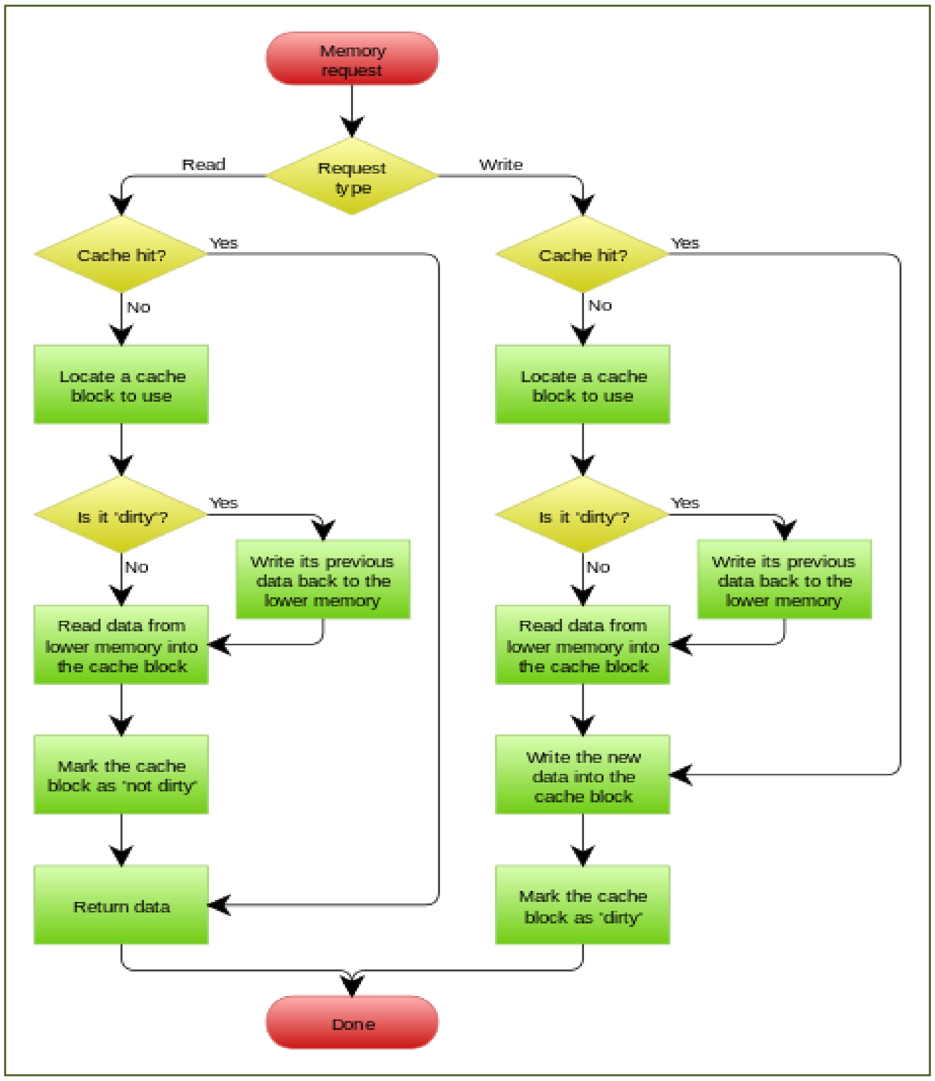
Memory request
Request type
Read
Write
Cache hit?
Yes
No
Locate a cache block to use
Is it 'dirty'?
Yes
No
Write its previous data back to the lower memory
Read data from lower memory into the cache block
Mark the cache block as 'not dirty'
Return data
Cache hit?
Yes
No
Locate a cache block to use
Is it 'dirty'?
Yes
No
Write its previous data back to the lower memory
Read data from lower memory into the cache block
Write the new data into the cache block
Mark the cache block as 'dirty'
Done

Figure 8

Flow chart showing the working of copyback cache

detect synonyms is to map the reque sted virtual address into its physical address and see if any of the virtual addresses in the cache maps into the 3 same physical address. In general, a reverse translation buffer (RTB) is needed in order for this approach to be feasible. Such a buffer is accessed with the physical address and indicates the cache line that is associated with that physical address. This in effect maintains a physical tag for each and every line present in the virtual address cache, meaning that a cache line has to be invalidated whenever its physical tag is replaced. One way to reduce the complexity in handling synonyms is to make sure that the index bits used to select the cache set are the same for both the physical and virtual addresses. In this case, the reverse mapping of cache lines can be implemented by simply associating both the virtual and physical address tags with each cache line. However, as mentioned earlier, this restricted page mapping may result in more page faults. Since the index bits are the most critical for cache access, a hybrid approach is to use virtual indices with physical tags. In this case, synonyms are mapped to a small number of sets determined by the number of index bits beyond the page offset. Software approaches have also been proposed to eliminate synonyms" (Jih-Kwon Peir Windsor).

5.1.3. Interesting Implementations

"Typically, a combination of the above techniques and other engineering solutions are used to circumvent the addressing constraint. For instance, the IBM-3090 has a unified 128KB L1 cache with physical tags. The tag array is organized as 32-way set-associative to limit the index bits to within the 4-KB page offset. The data array has a virtually-indexed 4-way set-associative design to avoid fetching all 32 double words out of the data array simultaneously. As a result, there is a primary set indexed by including the low-order 3 bits (bits 17-19) of the virtual page address and seven synonym sets indexed by using other combinations of the 3 bits. In case of a synonym hit, the correct data location is available from the tag comparison and the data can be fetched out in the next cycle. The synonym line is moved to the primary set afterwards. This solution becomes prohibitively expensive with larger caches because of the higher set associativity of the tag array. The recently announced IBM S/390-G4 CMOS mainframe processor has a 64KB 4-way set-associative unified L1 cache. With 4KB pages, two of the address bits (18:19) needed to index the cache are subject to translation. These two address bits are predicted prior to the cache access cycle . An Absolute Address History Table (AAHT) is used to maintain the recent associations between the physical address bits (18:19) and the values in the base and the index registers. At the address generation cycle, AAHT is accessed to obtain the predicted bits (18:19) for accessing the cache in the following cycle. The UltraSPARC-IIi has 16KB L1 instruction and data caches. The instruction cache is 2-way set-associative, physically indexed and tagged. However, for fast access time, the data cache is direct-mapped with virtual index bits and physical tags. The approach taken in the UltraSPARC-IIi is to rely on software to force synonym lines to have the same index bits. In cases where the synonyms cannot be mapped to the same index bits, the software either uses the data cache or turns o_ caching for the synonym pages. In a two-level cache design, a natural way of incorporating both the virtual and physical cache designs is to have a first-level virtual cache for fast access and a second-level physical cache for resolving synonyms. In this case, the tag array of the L2 cache acts as a means of locating L1 cache lines via their physical addresses. For instance, the MIPS R10000 has 32KB 2-way set-associative L1 instruction and data caches. Both caches are virtually indexed and physically tagged. The L2 cache is physically indexed and tagged to detect synonyms and to handle cache coherence requests. In order to support a 4 KB page size, two virtual bits are needed to index the L1 cache. The L2 tag array maintains these bits to enable L1 cache lookup" (Jih-Kwon Peir Windsor).

6. WRITE POLICIES

How is memory updated on a write? One could write a both cache and memory (write-through), or write only to the cache (copyback), updating the memory when the line is replaced. These two strategies are the basic cache write policies.

1. Write-policies

In a write-through policy, a write is directed at both the cache and the main memory for every CPU store. This has the advantage of maintaining a consistent (up-to-date) image of the program activity in main memory. It has the disadvantage of increasing memory, traffic for large caches, as those caches with low read read-miss rates may now find the memory traffic to be dominated by the write traffic (Figure 5 & 6).

2. Copy-back

In a copy-back policy, the entire line is replaced in main memory if a write has occurred into that line. A line which has been unaltered when replaced is simply discarded, as the main memories continue to retain the correct contents of that line. In copy-back, the selection of the line to be replace is usually unaffected by the write policy since that line is usually determined by the line replacement strategy. Thus when a read miss occur in copy back, either the new line is accessed from the main memory and placed in the cache. Copy-back caches require an additional bit, associated with each line in the cache directory. This “dirty bit” is set on if a write occurs anywhere in the line. The presence of the dirty bit indicate that the line must be written out completely to the main memory or replacement (Figure 7 & 8).

7. CONCLUSION

According to this research paper we can conclude that cache memory is very much important it generally reduces the time consumption because user don’t have to search the whole Main Memory, Main Memory consists of Lakhs of addresses of instructions. Secondly, we can say that it reduces the risks created by memory traversal, in other words we can say that it helps in memory management. It also reduces the chances of errors, therefore support Error Management. It also helps in disk management. In this we have also explained various cache memory implementation issues along with the coherency and their protocols. It also supports process management by diving whole process into modules where each module is executed separately.

REFERENCES

1. Implementation Issues in Modern Cache Memory Jih-Kwon Peir Windsor W. Hsu Alan Jay Smith Report No. UCB/CSD-98-1023 November 1998
2. Implementation Issues in Modern Cache Memory Jih-Kwon Peir Windsor W. Hsu Alan Jay Smith Report No. UCB/CSD-98-1023 November 1998 Computer Science Division (EECS), University of California Berkeley, California 94720
3. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System Daniel Molka, Daniel Hackenberg, Robert Sch¨one and Matthias S. M¨uller Center for Information Services and High Performance Computing (ZIH) Technische Universit¨at Dresden 01062 Dresden, Germany
4. BUS AND CACHE MEMORY ORGANIZATIONS FOR MULTIPROCESSORS by Donald Charles Winsor A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Electrical Engineering) in The University of Michigan 1989
5. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model JAMES ARCHIBALD and JEAN-LOUP BAER University of Washington
6. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System Daniel Molka, Daniel Hackenberg, Robert Sch¨one and Matthias S. M¨uller Center for Information Services and High Performance Computing (ZIH) Technische Universit¨at Dresden 01062 Dresden, Germany
7. Implementation Issues in Modern Cache Memory Jih-Kwon Peir Windsor W. Hsu Alan Jay Smith Report No. UCB/CSD-98-1023 November 1998
8. Cache Memories Krishna M. Kavi The University of Alabama in Huntsville
9. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model JAMES ARCHIBALD and JEAN-LOUP BAER University of Washington
10. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model JAMES ARCHIBALD and JEAN-LOUP BAER University of Washington
11. Implementation Issues in Modern Cache Memory Jih-Kwon Peir Windsor W. Hsu Alan Jay Smith Report No. UCB/CSD-98-1023 November 1998