Cache Memories

Krishna M. Kavi
The University of Alabama in Huntsville

The use of cache memories are so pervasive in today's computer systems it is difficult to imagine processors without them. Cache memories, along with virtual memories and processor registers form a continuum of memory hierarchies that rely on the principle of locality of reference. Most applications exhibit temporal and spatial localities among instructions and data. Spatial locality 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 likely be referenced in the near future (time-wise). Memory hierarchies are designed to keep most likely referenced items in the fastest devices. This results in an effective reduction in access time.

Cache Memories In Uniprocessors.

A cache memory can be viewed as a number of cache blocks (or cache lines), organized into a number of sets, each set containing one or more cache lines. 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 single cache line, while in a k-way set associative cache, each set contains k cache lines. The virtual address encountered by the processor 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 values 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 loaded with the missing data from main memory. Higher set associativity will reduce the number of cache misses caused by a program, but, direct mapped caches are more common because they are less complex and can use faster clock signals.

Reading versus Writing.

On a read access, it is possible to read the cache line at the same time that the tag is read and compared. On a hit, the required data is readily available, and on a miss, the read data can be discarded awaiting cache line replacement. However, on a write hit, it is necessary to first read the cache line, modify only the effected bytes before writing the line back to cache. It is possible to pipeline writes to hide the two steps required for a write. It is also necessary to address how the main memory is notified of the changes made to cache lines. In *write-through*, all writes to a cache line will also be written to main memory. Often buffers are used to process writes to main memory, to avoid blocking the processor on a write. In *write-back* (or *copy back*) the main memory is updated only when a cache line is replaced. Write back technique is more common since it eliminates some writes to main memory and reduces memory bus traffic. However, write-back can result in inconsistent 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 main memory.

Improving Performance.

Numerous methods have been proposed and implemented to enhance the performance of cache memories. Two factors influence 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 memory references. Miss penalty refers to the amount of time needed to handle a cache miss. Strategies for reducing miss rates include: victim caches, higher set-associativity, larger cache lines and prefetching of cache lines. Victim caches [4] are effective as they include a small fully associative cache that hold recently displaced (i.e. victims) cache lines so that future references to the victims can be satisfied by the victim cache. Strategies for reducing miss penalty include non-blocking caches and multi-level caches. Non-blocking caches permit access to cache lines while one or more cache misses are pending. These and other variation have found their implementation 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 system does not cause any problems. But techniques are needed to ensure that consistent data is available to all processors in a multiprocessor system. Cache coherency can be maintained either by hardware techniques or software techniques. We will first introduce hardware solutions.

Snoopy Protocols [1].

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 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 above, 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 other caches have the requested data in *Shared* state, the copies are invalidated, the write can complete with a single *Modified* copy. 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.

Instead of invalidating shared copies on a write, it may be possible to provide updated copies. The trade-off between Invalidation and Update centers around the misses caused by invalidations as compared to the increased bus traffic due to updates.

Directory Protocols[5]

Snoopy protocols rely on the ability to listen and broadcast invalidations on a common bus. However, the common bus also places a limit on the number of processing nodes in a SMP system. Large scale multiprocessor and distributed systems must use more complex interconnection mechanisms, such as multiple buses, N-dimensional grids, Crossbar switches and multistage interconnection networks (see Network Topologies and Network Architecture). New techniques are needed to assure that invalidation messages are received by all caches with copies of the shared data. This is normally achieved by keeping a directory with main memory units at each site. There exists one directory entry corresponding to each cache block, and the entry keeps track of shared copies, or the identification of the processor that contains modified data. On a read miss, a processor requests the memory unit for data. The request may go to a remote memory unit depending on the address. If the data is not modified, a copy is sent to the requesting cache, and the directory entry is modified to reflect the existence of a shared copy. If a modified copy exists at another cache, the new data is written back to the memory, a copy of the data is provided to the requesting cache, and the directory is marked as shared.

In order to handle writes, it is necessary to maintain state information with each cache block at local caches, somewhat similar to the Snoopy protocols. On a write hit, the write can proceed immediately if the state of the cache line is Modified. Otherwise (the state is Shared), Invalidation message is communicated to the memory unit, which in turn sends invalidation signals to all caches with shared copies. Only after the completion of this process 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 approach outlined here, the directory associated with each memory unit is responsible for tracking the shared copies and for sending invalidation signals. We can consider distributing the work as follows. On the first request for data, the memory unit supplies the requested data, and marks the directory with the requesting processor identification. Future read requests will be forwarded to the processor which has the copy. The processors with copies of the data are thus linked, and track all shared copies. On a write request, an invalidation signal is sent along the linked list to all shared copies. The memory unit can also send the identification of the writer so that invalidations can be acknowledged directly by the processors with shared copies. Scalable Coherence Interface (SCI) standard [3] uses a doubly linked list of shared copies. This permits a processor to remove itself from 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 performance of the directory based protocols. Hybrid techniques that combine Snoopy protocols with Directory 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 across clusters (see Cluster Computing).

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.

Conclusions.

Cache memories play a significant role in improving the performance of today’s computer systems. Numerous techniques for the use of caches and for maintaining coherent data have been proposed and implemented in commercial SMP systems. The use of cache memories in a distributed processing system, however, must be carefully understood to benefit from them. In a related research on memory consistency models, performance improvements are achieved by requiring data consistency only at synchronization points (see Memory Consistency). The research on cache memories in distributed systems is very active. Furthermore research is currently active on the use of cache memories in fully distributed systems, including web-based computing (see Client Caching, Server Caching).

Further Reading.

An excellent introduction and overview of cache memories can be found in [2]. This book also provides references to several empirical studies on cache memories in uniprocessor and multiprocessor systems. Selected papers on software based cache coherency in distributed systems are included in [6]. The research in cache memories very active and the reader can find more recent techniques for improving cache performance in annual conference proceedings, such as the International Symposium on Computer Architecture, High Performance Computer Architecture, Symposium on Architectural Support for Programming Languages and Operating Systems.

References.

[1] J. Archibald and J.-L. Baer. "Cache coherence protocols", *ACM Transactions on Computer Systems*, Vol. 4, No. 4, (Nov. 1986), pp 273-298.

[2] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*, 2nd Edition, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

[3] D. James. "The scalable coherent interface: Scaling to high-performance systems", *Proceedings of the Spring COMPCON-94*, IEEE Computer Society, Los Alamitos, CA, 1994, pp 64-71.

[4] N. Jouppi. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers", *Proceedings of the 17th Annual International Symposium on Computer Architecture*, IEEE Computer Society, Los Alamitos, CA, 1990, pp 364-373.

[5] D.J. Lilja. "Cache coherence in large scale shared-memory multiprocessors: Issues and comparison", *ACM Computing Surveys*, Vol. 25, No. 3 (Sept. 1993), pp 303-338.

[6] I. Tartalja and V. Milutinovic. *The Cache Coherence Problem In Shared-Memory Multiprocessors: Software Solutions*, IEEE Computer Society, Los Alamitos, CA, 1996

Cross Reference

Cache Coherence see Cache Memories
Copy Back see Cache Memories
Directory Protocols see Cache Memories
Direct-Mapped see Cache Memories
False Sharing see Cache Memories
MESI see Cache Coherence
Scalable Coherence Interface (SCI) see Cache Memories
Set Associative see Cache Memories
Snoopy Protocols see Cache Memories
Victim Cache see Cache Memories
Write Back see Cache Memories
Write Through see Cache Memories

Dictionary Terms

Victim Cache. A small (about 8 cache lines) fully associative buffer is added to direct mapped caches. When a cache line is displaced, the "victim" is stored in the buffer. A future reference to a victim may then be satisfied by the buffer. This approach has been shown to reduce conflict misses.

Prefetch Buffer. Cache memory performance can be improved by using a small buffer that is used to prefetch successive cache lines starting at the miss target.

Compulsory Misses. In most cases, the first reference to an address causes a cache miss since the program addresses are not yet loaded into cache. These are also called cold start misses or first reference misses. These misses can be minimized by techniques such as preloading a program addresses into the cache and other prefetching techniques.

Conflict Misses. It is often the case that several addresses of a program all map to the same set in the cache memory. When a program references addresses that all fall to the same set, a conflict for the cache lines of the set is said to occur. If cache blocks are replaced by newer references and later retrieved (replacing the newer references) cache performance can suffer. Higher set associativity can reduce the number of conflict misses.

Set Associative Mapping. A cache memory is organized as a number of sets where each set contains one or more cache lines. The virtual address encountered by the processor is used to locate a set; the address-tag is then compared against the tags associated with the cache lines belonging to the set; a tag match in any cache line indicates a "hit". In a k-way set-associative cache, each set contains k-cache lines. A direct mapped cache is 1-way set associative with one cache line per set. Higher set associativity (2, 4, or 8) can reduce the number of conflict misses encountered by a program.

Symmetric Multiprocessing (SMP). A small scale shared memory multiprocessor system relying on a common bus to interconnect processors to the common memory unit. All processors have equal access to the memory. They are also called Uniform Memory Access (UMA) multiprocessor systems. In most cases, any process (user or system) can execute on any of the processors in the system.

Scalable Coherence Interface (SCI). A standard protocol that defines how directory based approach to maintain coherent data across distributed memory units in a multiprocessing system.