International Journal of Information Technology and Management

Vol. VI, Issue No. I, February-2014, ISSN 2249-4510

A CRITICAL STUDY ON CACHE MEMORY MANAGEMENT AND ITS VARIOUS ASPECTS

AN INTERNATIONALLY INDEXED PEER REVIEWED & REFEREED JOURNAL

www.ignited.in

International Journal of Information Technology and Management
Vol. VI, Issue No. I, February-2014, ISSN 2249-4510

A Critical Study on Cache Memory Management and Its Various Aspects

Rishi Mathur[1] Dr. D. D. Aggarawal[2]

[1]Research Scholar, Bundelkhand University, Jhansi (U.P)

Abstract – Cache memories are used in modern, medium and high-speed CPUs to hold temporarily those portions of the contents of main memory which are {believed to be) currently in use. Since instructions and data in cache memories can usually be referenced in 10 to 25 percent of the time required to access main memory, cache memories permit the executing rate of the machine to be substantially increased. In order to function effectively, cache memories must be carefully designed and implemented. In this paper, we explain the various aspects of cache memories and discuss in some detail the design features and trade-offs.

----------♦----------

INTRODUCTION

A large number of original, trace-driven simulation results are presented. Consideration is given to practical implementation questions as well as to more abstract design issues. Specific aspects of cache memories that are investigated include: the cache fetch algorithm (demand versus prefetch), the placement and replacement algorithms, line size, store-through versus copy-back updating of main memory, cold-start versus warm-start miss ratios, multi-cache consistency, the effect of input/output through the cache, the behavior of split data/instruction caches, and cache size. Our discussion includes other aspects of memory system architecture, including translation look aside buffers. Throughout the paper, we use as examples the implementation of the cache in the Amdahl 470V/6 and 470V/7, the IBM 3081, 3033, and 370/168, and the DEC VAX 11/780. Cache memories are small, high-speed buffer memories used in modern computer systems to hold temporarily those portions of the contents of main memory which are (believed to be) currently in use. Information located in cache memory may be accessed in much less time than that located in main memory (for reasons discussed throughout this paper}. Thus, a central processing unit (CPU) with a cache memory needs to spend far less time waiting for instructions and operands to be fetched and/or stored. For example, in typical large, high-speed computers (e.g., Amdahl 470V/ 7, IBM 3033), main memory can be accessed in 300 to 600 nanoseconds; information can be obtained from a cache, on the other hand, in 50 to 100 nanoseconds. Since the performance of such machines is already limited in instruction execution rate by cache memory access time, the absence of any cache memory at all would produce a very substantial decrease in execution speed. Virtually all modern large computer systems have cache memories; for example, the Amdahl 470, the IBM 3081 [IBM82, REIL82, GUST82], 3033, 370/168, 360/195, the Univac 1100/80, and the Honeywell 66/ 80. Also, many medium and small size machines have cache memories; for example, the DEC VAX 11/780, 11/750 [ARMS81], and PDP-11/70 [SIRE76, SNOW78], and the Apollo, which uses a Motorola 68000 microprocessor. We believe that within two to four years, circuit speed and density will progress sufficiently to permit cache memories in one chip microcomputers. (On-chip addressable memory is planned for the Texas Instruments 99000 [LAFF81, ELEC81].) Even microcomputers could benefit substantially from an on-chip cache, since on-chip access times are much smaller than off-chip access times. Thus, the material presented in this paper should be relevant to almost the full range of computer architecture implementations. The success of cache memories has been explained by reference to the "property of locality" [DENN72].

The property of locality has two aspects, temporal and spatial. Over short periods of time, a program distributes its memory references non-uniformly over its address space, and which portions of the address space are favored remain largely the same for long periods of time. This first property, called temporal locality, or locality by time, means that the information which will be in use in the near future is likely to be in use already. This type of behavior can be expected from program loops in which both data and instructions are reused. The second property, locality by space, means that portions of the address space which are in use generally consist of a fairly small number of individually contiguous segments of that address space. Locality by space, then, means that the loci of reference of the program in the near future

are likely to be near the current loci of reference. This type of behavior can be expected from common knowledge of programs: related data items (variables, arrays) are usually stored together, and instructions are mostly executed sequentially. Since the cache memory buffers segments of information that have been recently used, the property of locality implies that needed information is also likely to be found in the cache. Optimizing the design of a cache memory generally has four aspects: (1) Maximizing the probability of finding a memory reference's target in the cache (the hit ratio), (2) minimizing the time to access information that is indeed in the cache {access time), (3) minimizing the delay due to a miss.

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.

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. It may be noted here that TLB cache is a part of the memory management unit (MMU) and not directly related to the CPU caches.

CACHE ENTRIES

Data is transferred between memory and cache in blocks of fixed size, called *cache lines*. When a cache line is copied from memory into the cache, a cache entry is created. The cache entry will include the copied data as well as the requested memory location (now called a tag).

When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache. The cache checks for the contents of the requested memory location in any cache lines that might contain that address. If the processor finds that the memory location is in the cache, a cache hit has occurred. However, if the processor does not find the memory location in the cache, a cache miss has occurred. In the case of a cache hit, the processor immediately reads or writes the data in the cache line. For a cache miss, the cache allocates a new entry and copies in data from main memory, then the request is fulfilled from the contents of the cache.

CACHE PERFORMANCE

The proportion of accesses that result in a cache hit is known as the hit rate, and can be a measure of the effectiveness of the cache for a given program or algorithm.

Read misses delay execution because of requiring data to be transferred from memory, which is much slower than reading from the cache. Write misses may occur without such penalty, since the processor can continue execution while data is copied to main memory in the background.

REPLACEMENT POLICIES

In order to make room for the new entry on a cache miss, the cache may have to evict one of the existing entries. The heuristic that it uses to choose the entry to evict is called the replacement policy. The fundamental problem with any replacement policy is that it must predict which existing cache entry is least likely to be used in the future. Predicting the future is difficult, so there is no perfect way to choose among the variety of replacement policies available.

One popular replacement policy, least-recently used (LRU), replaces the least recently accessed entry.

Marking some memory ranges as non-cacheable can improve performance, by avoiding caching of memory regions that are rarely re-accessed. This avoids the overhead of loading something into the cache without having any reuse. Cache entries may also be disabled or locked depending on the context.

WRITE POLICIES

If data is written to the cache, at some point it must also be written to main memory; the timing of this write is known as the write policy. In a write-through cache, every write to the cache causes a write to main memory. Alternatively, in a write-back or copy-back cache, writes are not immediately mirrored to the main memory, and the cache instead tracks which locations have been written over, marking them as dirty. The data in these locations is written back to the main memory only when that data is evicted from the cache. For this reason, a read miss in a write-back cache may sometimes require two memory accesses to service: one to first write the dirty location to main memory, and then another to read the new location from memory. Also, a write to a main memory location that is not yet mapped in a write-back cache may evict an already dirty location, thereby freeing that cache space for the new memory location.

There are intermediate policies as well. The cache may be write-through, but the writes may be held in a store data queue temporarily, usually so that multiple stores can be processed together (which can reduce bus turnarounds and improve bus utilization).

Cached data from the main memory may be changed by other entities (e.g. peripherals using direct memory access (DMA) or another core in a multi-core processor), in which case the copy in the cache may become out-of-date or stale.

Alternatively, when a CPU in a multiprocessor system updates data in the cache, copies of data in caches associated with other CPUs will become stale. Communication protocols between the cache managers that keep the data consistent are known as cache coherence protocols.

CPU STALLS

The time taken to fetch one cache line from memory (read latency) matters because the CPU will run out of things to do while waiting for the cache line. When a CPU reaches this state, it is called a stall. As CPUs become faster compared to main memory, stalls due to cache misses displace more potential computation; modern CPUs can execute hundreds of instructions in the time taken to fetch a single cache line from main memory.

Various techniques have been employed to keep the CPU busy during this time, including out-of-order execution in which the CPU (Pentium Pro and later Intel designs, for example) attempts to execute independent instructions after the instruction that is waiting for the cache miss data. Another technology, used by many processors, is simultaneous multithreading(SMT), or in Intel's terminology hyper-threading (HT), which allows an alternate thread to use the CPU core while the first thread waits for required CPU resources to become available.

CACHE ENTRY STRUCTURE

Cache row entries usually have the following structure:

tag	data block	flag bits

The *data block* (cache line) contains the actual data fetched from the main memory. The *tag* contains (part of) the address of the actual data fetched from the main memory. The flag bits are discussed below.

The "size" of the cache is the amount of main memory data it can hold. This size can be calculated as the number of bytes stored in each data block times the number of blocks stored in the cache. (The tag, flag and error correction code bits are not included in the size although they do affect the physical area of a cache.)

An effective memory address is split (MSB to LSB) into the tag, the index and the block offset.

tag	index	block offset

The index describes which cache row (which cache line) that the data has been put in. The index length is $\lceil \log_2(r) \rceil$ bits for *r* cache rows.

The block offset specifies the desired data within the stored data block within the cache row. Typically the effective address is in bytes, so the block offset length is $\lceil \log_2(b) \rceil$ bits, where *b* is the number of bytes per data block. The tag contains the most significant bits of the address, which are checked against the current row (the row has been retrieved by index) to see if it is the one we need or another, irrelevant memory location that happened to have the same index bits as the one we want. The tag length in bits is address_length - index_length - block_offset_length.

Some authors refer to the block offset as simply the "offset" or the "displacement".

EXAMPLE

The original Pentium 4 processor had a four-way set associative L1 data cache of 8 KB in size, with 64-byte cache blocks. Hence, there are 8 KB / 64 = 128 cache blocks. The number of sets is equal to the number of cache blocks divided by the number of ways of associativity, what leads to 128 / 4 = 32 sets, and hence 2^5 = 32 different indices. There are 2^6 = 64 possible offsets. Since the CPU address is 32 bits wide, this implies 21 + 5 + 6 = 32 and hence 21 bits for the tag field.

The original Pentium 4 processor also had an eight-way set associative L2 integrated cache 256 KB in size, with 128-byte cache blocks. This implies 17 + 8 + 7 = 32 and hence 17 bits for the tag field.

FLAG BITS

An instruction cache requires only one flag bit per cache row entry: a valid bit. The valid bit indicates whether or not a cache block has been loaded with valid data.

On power-up, the hardware sets all the valid bits in all the caches to "invalid". Some systems also set a valid bit to "invalid" at other times, such as when multi-master bus snooping hardware in the cache of one processor hears an address broadcast from some other processor, and realizes that certain data blocks in the local cache are now stale and should be marked invalid.

A data cache typically requires two flag bits per cache line – a valid bit and a dirty bit. Having a dirty bit set indicates that the associated cache line has been changed since it was read from main memory ("dirty"), meaning that the processor has written data

to that line and the new value has not propagated all the way to main memory.

DIRECT-MAPPED CACHE

In this cache organization, each location in main memory can go in only one entry in the cache. Therefore, a direct-mapped cache can also be called a "one-way set associative" cache. It does not have a replacement policy as such, since there is no choice of which cache entry's contents to evict. This means that if two locations map to the same entry, they may continually knock each other out. Although simpler, a direct-mapped cache needs to be much larger than an associative one to give comparable performance, and it is more unpredictable. Let *x* be block number in cache, *y* be block number of memory, and *n* be number of blocks in cache, then mapping is done with the help of the equation $x = y \bmod n$.

TWO-WAY SET ASSOCIATIVE CACHE

If each location in main memory can be cached in either of two locations in the cache, one logical question is: *which one of the two?* The simplest and most commonly used scheme, shown in the right-hand diagram above, is to use the least significant bits of the memory location's index as the index for the cache memory, and to have two entries for each index. One benefit of this scheme is that the tags stored in the cache do not have to include that part of the main memory address which is implied by the cache memory's index. Since the cache tags have fewer bits, they require fewer transistors, take less space on the processor circuit board or on the microprocessor chip, and can be read and compared faster. Also LRUI's especially simple since only one bit needs to be stored for each pair.

SPECULATIVE EXECUTION

One of the advantages of a direct mapped cache is that it allows simple and fast speculation. Once the address has been computed, the one cache index which might have a copy of that location in memory is known. That cache entry can be read, and the processor can continue to work with that data before it finishes checking that the tag actually matches the requested address.

The idea of having the processor use the cached data before the tag match completes can be applied to associative caches as well. A subset of the tag, called a *hint*, can be used to pick just one of the possible cache entries mapping to the requested address. The entry selected by the hint can then be used in parallel with checking the full tag.

TWO-WAY SKEWED ASSOCIATIVE CACHE

Other schemes have been suggested, such as the *skewed cache* where the index for May 0 is direct, as above, but the index for May 1 is formed with a hash function. A good hash function has the property that addresses which conflict with the direct mapping tend not to conflict when mapped with the hash function, and so it is less likely that a program will suffer from an unexpectedly large number of conflict misses due to a pathological access pattern. The downside is extra latency from computing the hash function. Additionally, when it comes time to load a new line and evict an old line, it may be difficult to determine which existing line was least recently used, because the new line conflicts with data at different indexes in each way; LRU tracking for non-skewed caches is usually done on a per-set basis. Nevertheless, skewed-associative caches have major advantages over conventional set-associative ones.

PSEUDO-ASSOCIATIVE CACHE

A true set-associative cache tests all the possible ways simultaneously, using something like a content addressable memory. A pseudo-associative cache tests each possible way one at a time. A hash-rehash cache and a column-associative cache are examples of a pseudo-associative cache.

In the common case of finding a hit in the first way tested, a pseudo-associative cache is as fast as a direct-mapped cache, but it has a much lower conflict miss rate than a direct-mapped cache, closer to the miss rate of a fully associative cache.

REFERENCES

- Nathan N. Sadler and Daniel J. Sorin. "Choosing an Error Protection Scheme for a Microprocessor's L1 Data Cache". 2006. p. 4.
- John L. Hennessy, David A. Patterson. "Computer Architecture: A Quantitative Approach". 2011. ISBN 0-12-383872-X, ISBN 978-0-12-383872-8. page B-9.
- David A. Patterson, John L. Hennessy. "Computer organization and design: the hardware/software interface". 2009. ISBN 0-12-374493-8, ISBN 978-0-12-374493-7 "Chapter 5: Large and Fast: Exploiting the Memory Hierarchy". p. 484.
- Gene Cooperman. "Cache Basics". 2003.
- Ben Dugan. "Concerning Caches". 2002.
- Harvey G. Cragon. "Memory systems and pipelined processors". 1996. ISBN 0-86720-474-5,ISBN 978-0-86720-474-2. "Chapter 4.1: Cache Addressing, Virtual or Real" p. 209

- "Cache design" (PDF). *ucsd.edu*. 2010-12-02. p. 10–15. Retrieved 2014-02-24.
- IEEE Xplore - Phased set associative cache design for reduced power consumption. Ieeexplore.ieee.org (2009-08-11). Retrieved on 2013-07-30.
- André Seznec. "A Case for Two-Way Skewed-Associative Caches".doi:10.1145/173682.165152. Retrieved 2007-12-13.