Scientific Journal Impact Factor (SJIF) : 1.711
International Journal of Modern Trends in Engineering and Research
ISSN (ONLINE) : 2349 - 9745 ISSN (PRINT) : 2393 - 8161

A Review on Cache Memory with Multiprocessor System

Chirag R. Patel[1], Rajesh H. Davda[2]
[1,2] *Computer Engineering Department, C. U. Shah College of Engineering & Technology, Wadhwan (Gujarat)*

Abstract— The development of caches and caching is one of the most significant events in the history of computing. Virtually every modern CPU core from ultra-low power chips like the ARM Cortex-A5 to the highest-end Intel Core i7 use caches. Even higher-end microcontrollers often have small caches or offer them as options — the performance benefits are too significant to ignore, even in ultra low-power designs. Caching was invented to solve a significant problem. In the early decades of computing, main memory was extremely slow and incredibly expensive — but CPUs weren't particularly fast, either. In this paper we have discuss basic functionality of caches memory and its organization in computer system. This paper also includes effectiveness of cache memory with Temporal and spatial locality and different mapping function like Direct, Associative and set-associative mapping and their comparison. This paper also shows the common algorithms that are use for designing of cache memory.

Keywords— Cache, mapping, CPU, RAM.

I. INTRODUCTION

CPUs are today much more sophisticated than they were only 25 years ago. In those days, the frequency of the CPU core was at a level equivalent to that of the memory bus. Memory access was only a bit slower than register access. But this changed dramatically in the early 90s, when CPU designers increased the frequency of the CPU core but the frequency of the memory bus and the performance of RAM chips did not increase proportionally. It is possible but it is not economical. RAM as fast as current CPU cores is orders of magnitude more expensive than any dynamic RAM.

Hence, it is important to devise a scheme that reduces the time needed to access the necessary information. Since the speed of main memory unit is limited by electronic and packaging constraints, the solution must be sought in a different architectural arrangement. An efficient solution is to use a fast cache memory which essentially makes the main memory appear to the processor to be faster than it really is.

A. Memory Hierarchies

In Memory Hierarchies, processor or CPU core is not directly with main memory. In even earlier systems, the cache was attached to the system bus just like the CPU and the main memory. This was more a hack than a real solution. All loads and stores have to go through the cache. The connection between the CPU core and the cache is a special, fast connection. In a simplified representation, the main memory and the cache are connected to the system bus which can also be used for communication with other components of the system.

Even though computers for the last several decades have used the von Neumann architecture, experience has shown that it is of advantage to separate the caches used for code and for data. Intel has used separate code and data caches since 1993 and never looked back. The memory regions needed for code and data are pretty much independent of each other, which is why independent caches work better. In recent years another advantage emerged: the instruction decoding step for the most common processors is slow; caching decoded instructions can speed up the execution,

especially when the pipeline is empty due to incorrectly predicted or impossible-to-predict branches. Fig. 1.1 shows the memory hierarchies with its characteristics.

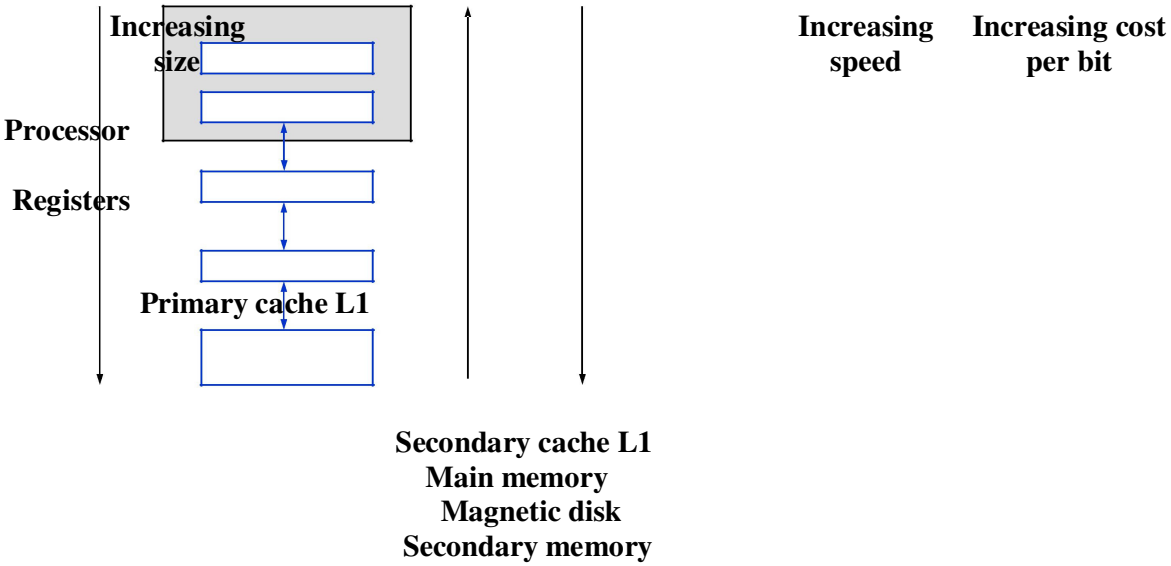

Figure 1.1 Memory Hierarchies

B. Processor with Level-3 Cache

Soon after the introduction of the cache, the system got more complicated. The speed difference between the cache and the main memory increased again, to a point that another level of cache was added, bigger and slower than the first-level cache. Only increasing the size of the first-level cache was not an option for economical reasons. Today, there are even machines with three levels of cache in regular use. A system with such a processor looks like Figure 1.2. With the increase on the number of cores in a single CPU the number of cache levels might increase in the future even more

Figure 1.2 shows three levels of cache and introduces the nomenclature we will use in the remainder of the document. L1d is the level 1 data cache, L1i the level 1 instruction cache, etc. Note that this is a schematic; the data flow in reality need not pass through any of the higher-level caches on the way from the core to the main memory. CPU designers have a lot of freedom designing the interfaces of the caches. For programmers these design choices are invisible.

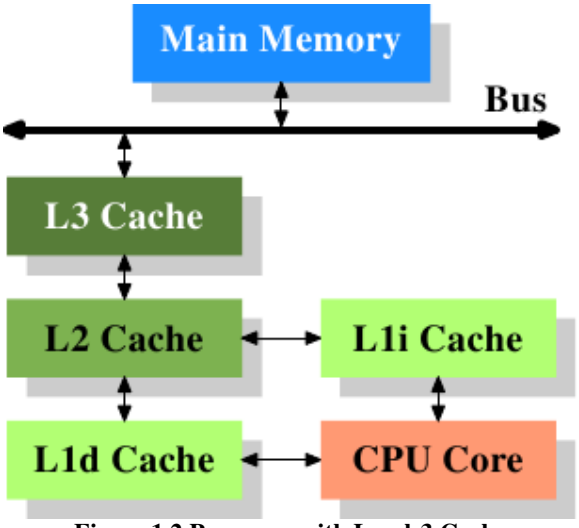

Figure 1.2 Processor with Level-3 Cache

C. Multi Processor, Multi-core, Multi-thread

In addition we have processors which have multiple cores and each core can have multiple "threads". The difference between a core and a thread is that separate cores have separate copies of (almost {Early multi-core processors even had separate 2nd level caches and no 3rd level cache.}) all the hardware resources. The cores can run completely independently unless they are using the same resources—e.g., the connections to the outside—at the same time. Threads, on the other hand, share almost all of the processor's resources. Intel's implementation of threads has only separate registers for the threads and even that is limited, some registers are shared. The complete picture for a modern CPU therefore looks like Figure 1.3

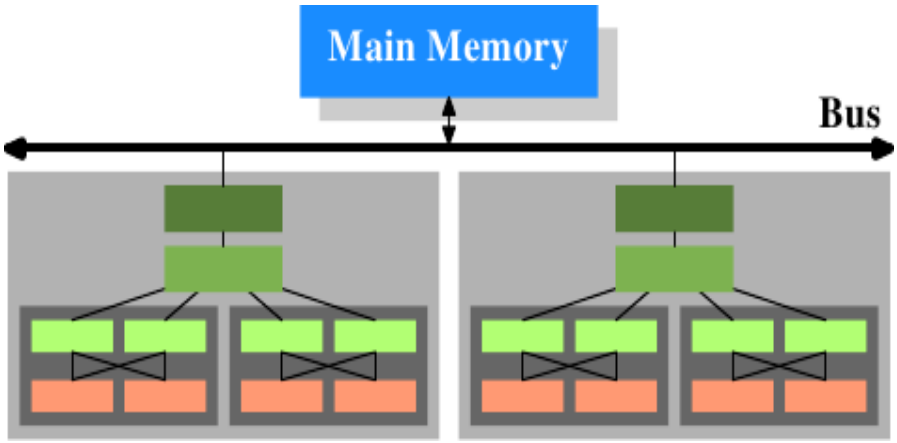

Figure 1.3 Multi Processor, multi-core, multi-thread

II. EFFECTIVENESS OF THE CACHE MEMORY

The effectiveness of the cache mechanism is based on a property of computer programs called locality of reference. Analysis of programs shows that most of their execution time is spent on routines in which many instructions are executed repeatedly. These instructions may constitute a simple loop, nested loops, or a few procedures that repeatedly call each other. Memory instructions in localized areas of the program are executed repeatedly during some time period, and the remainder of the program is accessed relatively infrequently. This is referred to as locality of reference.

A. Cache function

The general function of a cache is accomplished by creating an association between the main primary memory and the cache memory. First, main memory is divided into blocks equal to the total size of the cache. Then both main memory and the cache itself are divided into smaller units called lines. A particular line in each block of main memory is always associated with the same line in the cache. In other words, if a cache is 16K in size, then lines 0, 16K + 0, 32K + 0, etc of main memory will all be associated with line 0 of the cache. When a particular main memory location is accessed, its address is used to find its line identifier in main memory and the equivalent line in the cache where that memory location would be cached. If the cache line holds a copy of the line from the correct block of main memory, called a hit, the cache version of the memory is used.

If the cache line contains a line from a different block of main memory or no valid code at all, called a miss, then the correct line of main memory containing the memory of interest must be read into the cache. If the cache line did have valid memory from another block, it may have to be written out first.

B. Cache design

The simplest cache design to accomplish the function described above is a table with three fields.

- The first field contains a flag indicating whether the line of data is valid. This is an issue when the computer system first comes up. The first access to any memory will automatically be a cache

miss. However, the information in the cache is garbage and should not be first written out to main memory that may contain valid data.

- The second field contains the block address, called the tag, of the main memory where the line of data in the cache came from (or is going to).
- The third field contains the line of memory from the main memory. The length of this line is usually between 4 and 16 bytes or cells. The number of bytes is usually equal to the data bus size or some natural multiple of. This allows efficient moving of data between the cache line and the main memory.

The number of rows (lines) in the cache is determined by the size of the cache divided by the number of bytes in a cache line. So a 64K cache with 16 byte lines contains 4K lines. An important point to remember is that while all memory locations (bytes) on a single cache line are contiguous and come from a single block of main memory, each cache line in the cache is independent of all others and any two adjacent cache lines may be from different main memory blocks.

Determining cache line

To calculate the cache information from a memory address, use the following formulas.
Tag = int(Memory address / cache size)
Cache line = int((Memory address mod(%) cache size) / line length)
When a memory address is accessed, the calculations above are performed. The appropriate line in the cache is then checked for valid data and if valid, the tag or block id stored in the tag field is compared to the calculated tag. If the cache line contains valid data and the tags match, a hit has occurred. If the cache line is invalid or contains a line from a different block, a miss has occurred and the system must take additional steps to put the correct information in the cache.

C. Temporal and Spatial Localities

The temporal aspect of the locality of reference suggests that whenever an information item (instruction or data) is first needed, this item should be brought into the cache where it will hopefully remain until it is needed again.

The spatial aspect of the locality of reference suggests that instead of fetching just one item from the main memory to the cache, it is useful to fetch several items that reside at adjacent addresses as well. We will use the term block to refer to a set of continuous address locations of some size. Another item that is often used to refer to a cache block is cache line

III. CACHE MEMORY ALGORITHM

Cache algorithm is frequently called replacement algorithms or replacement policies. When the cache is full and a memory word that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word.

The collection of rules for making this decision constitutes the *replacement algorithm.*

A. Belady's Algorithm

The *most* efficient caching algorithm would be to always discard the information that will not be needed for the longest time in the future. 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.

B. Least Frequently Used (LFU)

Least Frequently Used. Discard the line that as been referenced the fewest times since being loaded. This based on the idea that this line probably won't be used again for a while and there is a good chance that the line being loaded will be referenced often. Requires counter hardware.

C. Least Recently Used (LRU)

Discard the line that has not been accessed in the longest time. This based again on the idea of progression and that the system is probably done with that portion of code or data. This also requires additional circuitry to do bookkeeping.

D. Most Recently Used (MRU)

Discards, in contrast to LRU, the most recently used items first. MRU cache algorithms have more hits than LRU due to their tendency to retain older data. MRU algorithms are most useful in situations where the older an item is, the more likely it is to be accessed than LRU due to their tendency to retain older data. MRU algorithms are most useful in situations where the older an item is; the more likely it is to be accessed.

E. Random Replacement (RR)

Randomly selects a candidate item and discards it to make space when necessary. This algorithm does not require keeping any information about the access history. For its simplicity, it has been used in ARM processors. It admits efficient stochastic simulation.

IV. MAPPING FUNCTION

Usually, the cache memory can store a reasonable number of blocks at any given time, but this number is small compared to the total number of blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a mapping function.

Cache mapping functions: Direct mapping, Associative mapping and Set-associative mapping.

To discuss possible methods for specifying where memory blocks are placed in the cache, we use a specific small example Consider a cache consisting of 128 blocks of 16 words each, for a total of 2048 (2K) words, and assume that the main memory is addressable by a 16-bit address. The main memory has 64K words, which we will view as 4K blocks of 16 words each.

A. Direct Mapping

This method is simplest way of mapping. Main memory is divided in block. Block j of the main memory is mapped onto block j modulo 128 of the cache – consider a cache of 128 blocks of 16 words each. Consider a memory of 64Kwords divided into 4096 blocks. Direct mapping`s performance is directly proportional to the Hit ratio.

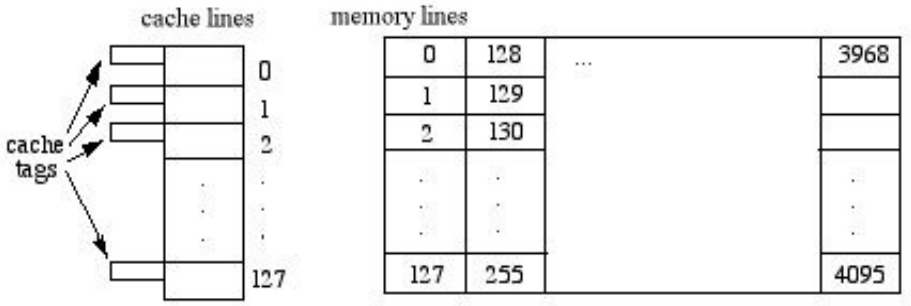

Figure 4.1 Direct Mapping

Tag |Block| Word

5	7	4

Main Memory address

- Advantages:
  - Low cost; doesn't require an associative memory in hardware
  - Uses less cache space

- Disadvantages:
- Contention with main memory data with same index bits.

B. Associative Mapping

In this type of mapping the associative memory is used to store content and addresses both of the memory word. This enables the placement of the any word at any place in the cache memory. It is considered to be the fastest and the most flexible mapping form.

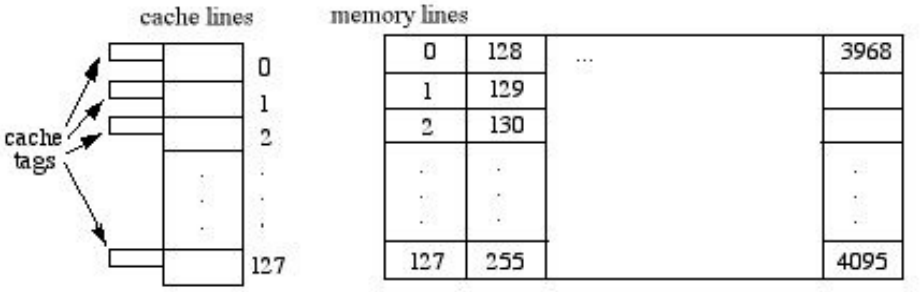

Figure 4.2 Associative Mapping

Tag | Word

12	4

Main Memory address

C. Set-Associative Mapping

This scheme is a compromise between the direct and associative schemes described above. Here, the cache is divided into sets of tags, and the set number is directly mapped from the memory address (e.g., memory line j is mapped to cache set j mod 64), as suggested by the diagram below:

Here, the "Tag" field identifies one of the $2^6 = 64$ different memory lines in each of the $2^6 = 64$ different "Set" values. Since each cache set has room for only two lines at a time, the search for a match is limited to those two lines (rather than the entire cache). If there's a match, we have a hit and the read or write can proceed immediately. Otherwise, there's a miss and we need to replace one of the two cache lines by this line before reading or writing into the cache

In set-associative mapping, when the number of lines per set is n, the mapping is called n-way associative. For instance, the above example is 2-way associative.

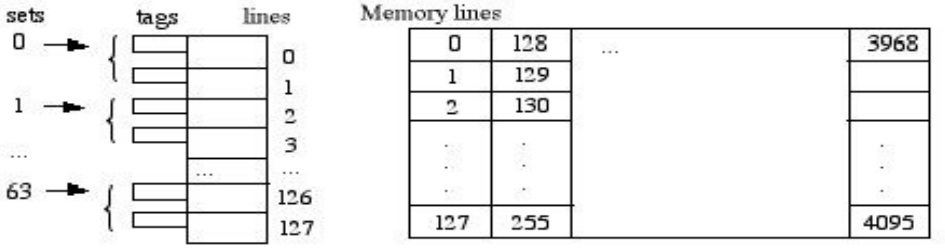

Figure 4.3 Set-Associative Mapping

Tag |Block| Word

6	6	4

Main Memory address

V. CONCLUSION

Cache memory is small and fast memory between Main memory and Microprocessor used to improve access time. The effectiveness of the cache mechanism is based on a property of computer

programs called locality of reference that is spatial and temporal locality. There are different cache memory mapping techniques from that best hit ratio is provided by associative technique and best searching speed is provided by direct mapping technique. As per use of data there are various replacement policies available in cache memory.

REFERENCES

[1] Jin-Fu Li, "The Memory System"
[2] Barry Wilkinson, "Multiprocessor Systems", 2000.
[3] Michael Smaili, "Cache and Virtual Memory Replacement Algorithms", Spring 2008
[4] www.wikipedia.org

