International Journal of Research

International Journal of Research (IJR)

e-ISSN: 2348-6848, p- ISSN: 2348-795X Volume 2, Issue 06, June 2015
Available at http://internationaljournalofresearch.org

Analysis of the computer caching scheme

Dr. Arinze S. Nwaeze
Dept. of Computer Science & Info. Tech. Caritas University, Amorji-Nike, Enugu

Abstract

Cache (pronounced cash) memory is extremely fast memory that is built into a computer's central processing unit (CPU), or located next to it on a separate chip. The CPU uses cache memory to store instructions that are repeatedly required to run programs, improving overall system speed. The advantage of cache memory is that the CPU does not have to use the motherboard's system bus for data transfer. Whenever data must be passed through the system bus, the data transfer speed slows down to the motherboard's capability. The CPU can process data much faster by avoiding the bottleneck created by the system bus. Data is transferred between memory and cache in blocks of fixed sizes, 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 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. 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 order to make room for the new entry on a cache miss, the cache may have to evict one of the existing entries. In a simple cache scheme, when a miss occurs, the contents of the cache will be written out. It is possible for code and data to be in the same memory line in different blocks of memory. If this happens, especially in a multitasking architecture, it could cause constant misses if a loop were accessing a particular block of memory. It is common for code of each task loaded to be aligned at the beginning of a memory segment which often correlates to the block size of the cache. As each task is activated, a miss occurs. The more tasks loaded the more often the miss. Caches can be designed to avoid or limit pitfalls like this. This thesis looks at ways to avoid pitfalls such as described. It also looks at the heuristic that the cache uses to choose the entry to evict. The bottleneck created by the system bus as the CPU processes data, will also be discussed. Furthermore, this thesis will look at the entire caching scheme, including the cache lines, the replacement policy, cache mapping, cache associativity, and when not to use the caching technology.

Keywords:

Cache-lines; Cache-mapping; Associativity; Spatial Locality; Temporal Locality; Sequentiality; LFU; MRU

Introduction

Memory (RAM) is often the bottleneck when executing code. This is an important fact to remember. You may have, say, 2 GHz machines, but typical RAM cannot retrieve instructions from memory at that speed.

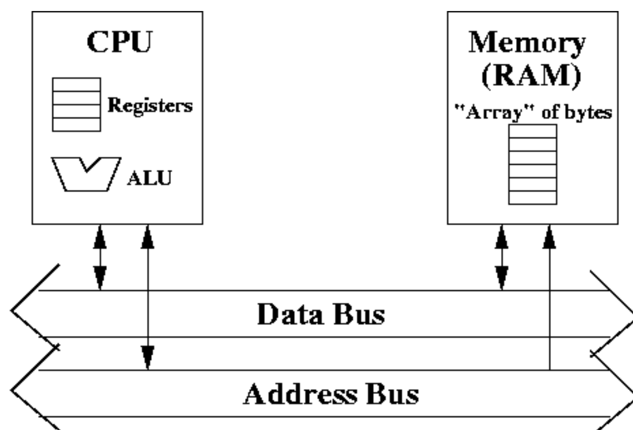
A solution is to use small, but fast memories. For example, accessing registers is very quick, while accessing RAM is comparatively slow. This suggests that if we want speed, we need more registers, and hope that most data stay in the registers, and that we rarely need to access RAM.

This is not a reasonable assumption because registers simply do not hold enough memory.

However, if we can use small, fast memory, then it is going to be called *cache*, and *cache* memory is faster than RAM, but slower than registers, and we can keep the most commonly used data (or instructions) in the cache. Cache can hold more data (typically 512 K or so) than a register (typically 128 bytes).

There is a basic principle about memory. If you want lots of memory, it will not cost you much but it is slow to access data. If you want very fast memory, it will be small (holding little data) and expensive per byte of memory. The goal of the cache is to put frequently accessed data in a place where it can be more quickly and efficiently accessed. If a web page can be read from local disk instead of going onto the Internet, the page can be displayed faster by the browser.

In the simplest model of a computer, there is a CPU, and there is physical memory (RAM) and there is a bus to connect the two. Over the years, computers have become more sophisticated, and implementation of memory systems has also become more sophisticated.

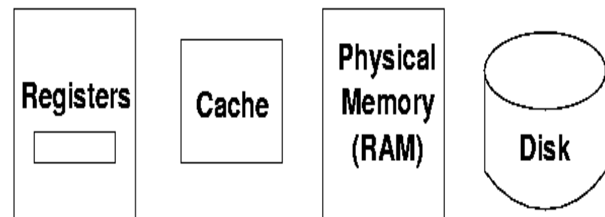


In particular, RISC (Reduced Instruction Set Computing) machines became popular. Even though Intel processors dominate the market, they use a lot of RISC ideas.

Basically, in RISC, some complex instructions were eliminated in favor of running several simpler instructions that did the same task. Initially, RISC used as few instructions as they could get away with. Over time, people realized this was not a reasonable way to develop an Industry Standard Architecture (ISA). The decision about which instructions to keep and which instructions to get rid of was based on benchmark programs. If simulations showed that including an instruction in the ISA would improve performance (speed) in many typical programs, then the instruction was kept in the ISA. Otherwise, it was discarded. As a result, a typical RISC program might contain many more instructions than its equivalent CISC (Complex Instruction Set Computing) program, since it may take more RISC instructions to run the equivalent CISC code.

Since about 1990, cache has become increasingly important. It lies between registers and physical memory in terms of size and speed. From slowest to fastest: *disk, RAM, cache, registers.* Normally, one thinks of memory in three levels: *registers, RAM* (physical memory) and *disk*. This forms part of the memory hierarchy.

Memory Hierarchy



On the left are registers. They give you the fastest access time.

Cache Replacement Policies

Consider that you are planning to write a very long paper on the history of Democracy in many different countries. You start with United Kingdom, and work your way to America, Australia, Russia, Asia, Nigeria, etc. You divided your paper so that each chapter covers one democracy.

You have a computer on your desk, and get your books online. You can download only so many books because your computer is limited in memory. The advantage of downloading the books onto your own computer is so that you can access the materials quickly. Downloading the books is quite slow, so you hope not to do it very often. If it were not for the fact that you collect related books and write from those books on each chapter, you might be forever downloading books. Even though the majority of books remain at the central library site, the books you need are on your computer, which is quick to access.

What happens if you have already downloaded 10 books (assuming maximum available space), and you want the 11th book? You will need to delete one of your books off your computer, to make room for the next book. Which one should you pick? You could pick the one that you have not read in the longest time. That policy is called "*least recently used*". Or you could see which book you have had on your computer the longest time. That policy is called "*first in first out*". Or perhaps the one you have looked at the least number of times. That is called "*least frequently used*". In any case, you use some policy to decide which book to get rid of to make space for the new book.

Extending this illustration

Let us assume your computer can still store 10 books. There are two sites you can access books. There is the *Virtual Central Library* (VCL), which has any book that you might want, and there is also a *Local Library* (LL).

Whenever you want to access a book, you will look in the Local Library (LL) first. If it is there, you will copy the book to your own computer. If it is not there, the LL will contact the Virtual Central Library (VCL). The LL will copy the book to its own library (possibly removing a book from the LL to make space), and then you will copy the book from the LL to your own computer. The LL is much smaller than the VCL, but can store more books than you can on your computer. Thus, the LL serves as a *middle man*. If the book you want is in the LL, you access it much faster than if you have to download it from the Virtual Central Library (VCL).

If it takes 1 second to access the LL, and 100 seconds to access the VCL when you are unable to find the book in the LL, you spend 100 seconds getting a copy from the VCL to the LL, and one more second to get it from the LL to your local computer. Thus, it is actually slower to use the LL if the book you are looking for is not there.

Cache works very similarly. Basically, you want some data (or an instruction) at some address. This data either appears in the cache (which is like the LL) or it is not. If it is not there, you need to access the data from RAM (which is like the VCL). This data is then copied to the cache, and then from the cache to the registers. Thus, the registers act like your local computer.

Cache Replacement Algorithms (Policy)

A cache algorithm is a detailed list of instructions that decides which items should be discarded in a computer's cache of information. 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 algorithm or 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.

Suppose you have a cache miss, and all the slots in the cache are being used. You will now have to pick a slot to get rid of. You are *evicting* a cache line. Which slot should you pick?

There are many replacement policies, including; *LRU, LFU, FIFO*.

LRU (Least Recently Used):
One popular replacement policy is the least-recently used (LRU), which 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. A good approximation to this algorithm is based on the observation those slots that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, slots that have not been used for ages will probably remain unused for a long time. This idea suggests a realizable algorithm: when a *miss* occurs, throw out the slot that has been unused for the longest time. This strategy is called **LRU** (**Least Recently Used**) caching.

LFU (Least Frequently Used)
This algorithm discards the least recently used items first. In other words, it discards the slot that been used the least often. This algorithm requires keeping track of what was used and when. It is expensive if one wants to make sure the algorithm always discards *the* least recently used item. General implementations of this technique require keeping *age bits* for cache-lines and tracking the *Least Recently Used* cache-line based on age-bits. In such an implementation, every time a cache-line is used, the age of all other cache-lines changes.

FIFO (First In First Out)
First In First Out algorithm picks the slot that has been in the cache the longest (which is NOT the same as LRU).

These policies often require additional hardware to indicate time of use or frequency of use.

When you are programming in assembly, you are not even aware of the cache. Much of the actual interactions with cache are handled by the hardware and the operating system.

Cache Line

A *cache line* is the *unit* of data you transfer to a cache. The size of this unit (or chunk of memory) is called the cache-line size. Common cache-line sizes are 32, 64, 128 bytes.

Data is transferred between memory and cache in blocks of fixed sizes, 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 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. In the case of 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. The concept of caching occurs everywhere in the computer world. Anytime you have to access data from a location that is slow, you may want to cache a copy at a location which allows you to access the data faster.

For example, suppose you visit a particular webpage at a website. Initially, that webpage may be slow to download. Once downloaded, a local copy of it is kept on your hard drive. Thus, your computer's hard drive (or possibly memory) is being used as a cache for that webpage. This can sometimes create problem, especially if the content of the webpage is always being updated.

These days, hard disks also have memory that serves as cache. Recall that hard disks are used to store files. Recall that hard disks are also slow. So, one idea is to use RAM as a cache for the disk. As files are accessed, they are placed in a special area in the RAM just for files. When you want to edit a file, you check in the RAM first, and if it is not there, then you check on the disk. The operating system does that automatically for you. So, as you can see, the principle of caching is to use faster memory to store frequently accessed data.

Cache efficiency pitfalls

In a simple cache scheme, when a *miss* occurs, the contents of the cache will be written out. It is possible for code and data to be in the same memory line in different blocks of memory. If this happens, especially in a multitasking architecture, it could cause constant misses if a loop were accessing a particular block of memory. It is common for code of each task loaded to be aligned at the beginning of a memory segment which often correlates to the block size of the cache. As each task is activated, a miss occurs. The more tasks loaded the more often the miss.

Caches can be designed to avoid or limit pitfalls like this.

The cache can be split such that you have separate caches for data and code. This allows the same line from the code segment and the data segment to be loaded at the same time. It also allows the system to skip the write back on miss for the code portion of the cache. In splitting the cache, the designer must decide whether to double the cache size or half the number of data and cache lines that can be cached at any time.

The idea of splitting the cache can be taken one step further and create additional complete sets of ‘N’cache lines. Now there is a set of ‘N’ cache lines available for each line in memory. A cache of this type is called an N-way set associative cache, where ‘N’ is the number of duplicate cache lines. This cache requires additional circuits to search the complete set for a particular line. Additionally, the size of the expensive cache memory has increased by ‘N’. In practice, a 2-way or 4-way cache is the best trade off for improved efficiency vs. cost. With additional lines in a line set, any memory reference has to check each copy of the line for a hit. This takes a little longer but the chances of a hit have gone up by ‘N’. In most modern computer architectures, the code and data are kept in separate places in memory.

The hierarchy of Cache

Cache has become so useful, most modern CPUs often have two levels of cache. They are called *L1* and *L2* caches for level 1 and level 2 cache.

Here's how to visualize an L2 cache. Imagine a *State Library* (SL), which is bigger than the LL, but smaller than the VCL. Every book in the LL is in the State Library SL. However, the SL has more books than the LL. Every book the SL

is also in the VCL. Thus, LL is a subset of SL, which is a subset of VCL.

To access a book, you go to the LL. If it is not there, you check the SL. If it is not there, you go to the VCL. That data gets copied to the SL, then back to the LL, and finally to your computer.

In principle, you can carry this idea of caching many, many levels. Each level is a subset of the next level. The higher the level, the more memory, and the slower it gets. For example, we could label the levels of memory as L1, L2, L3, etc. where all the data in level L_1 is a subset of that in level L_2 and so on. When you look for data at a particular address, you search for it at the smallest level, and if you could not find it there, you look at the next level. Once you find the data you are looking for, you copy it down to the lower levels, possibly getting rid of a *cache line* in the process.

The memory hierarchy would not be very effective if two facts about programs were not true. Programs exhibit *spatial locality* and *temporal locality*.

- **Spatial locality:** Spatial locality says that if you access memory address **x**, you are likely to access memory address **x** plus or minus **Delta** where **Delta** is a small number. That is, you are likely to access addresses very near **x** in the near future. Programs exhibit spatial locality. For example, when you accessing data from arrays, you are accessing memory locations that are contiguous. Also, instructions usually run sequentially (with the occasional branch or jump). Since instructions are contiguous in memory, they exhibit spatial locality. When you run one instruction, you are likely to run the next one too.
- **Temporal locality:** Temporal locality says that if you access memory address **x** at time **t**, you are like to access memory address **x** at time **t +/- Delta**. That is, you are likely to access the same address sometime soon. This happens when you run code in a loop. You access the same instructions over and over. If you process an array in a loop, then you access array elements over and over again too.

How Locality impacts caching

We can go back to our library example. Suppose you wish to access random books from the VCL. Each time you download a book, you would look in the local library. Assuming it is random, the probability that the book is in the LL would be low (since it stores only a small subset of the VCL). So you would have to go to the Virtual Central Library to find the book. Since accessing the VCL is very slow, you pay a fairly large cost (in time) for accessing books.

Furthermore, you have to copy that book from the VCL to the LL too. We copy the book to the LL, so it can be cached, and accessed again in the near future. However, since you are accessing books randomly, the books in the LL would not be used in the near future, and you will always have to access the VCL. In effect, the LL is useless to you immediately. You get benefit from using the Local Library if you are accessing the books found in the LL all the time. When you search for the same books over and over, they appear in the LL, and access time is much smaller to access the LL than the VCL.

To apply the analogy to hardware, if you want to access data and it is found in cache, you get savings in access time, since accessing data (or instructions) found in the cache is much quicker than accessing main memory, all the time. On the other hand, if you do not access data in the

International Journal of Research (IJR)
e-ISSN: 2348-6848, p- ISSN: 2348-795X Volume 2, Issue 06, June 2015
Available at http://internationaljournalofresearch.org

cache very often, then the cache is not useful, and may in fact, cause a small delay than not having the cache at all.

The principles of **spatial** and **temporal** locality are applied to cache design. Temporal locality is simple. You simply place data in the cache, because it is likely to be accessed in the near future. Since cache access is quicker than physical memory (RAM) access, placing data you just used into the cache should make it much quicker to access in the near future.

What about spatial locality? This says that if you access address **X**, you are likely to access memory address **x** plus or minus **Delta** where **Delta** is a small number addresses near **X** too.

Spatial locality suggests the following strategy when accessing data. Load the data from many contiguous addresses near **X** from RAM into the cache. That is, copy a block of data from memory to cache. This idea would not be so great if it were not quicker to access the entire block at once, as opposed to accessing the bytes one at a time. This makes some sense if you think about it. Suppose you were ordering books. If ordering 10 books at once took just as much time as ordering each book individually, then there is no need to order 10 books at a time. Just order them whenever you want them.

How Cache is organized (Cache mapping)

Cache is organized in three basic manners, namely: *Fully Associative, Direct-mapped, Set Associative.*

In **fully associative mapping**, instead of hard-allocating cache lines to particular memory locations, it is possible to design the cache so that any line can store the content of any memory location. When a request is made to the cache, the requested address is compared in a directory against all entries in the directory. If the requested address is found (*a directory hit*), the corresponding location in the cache is fetched and returned to the processor; otherwise, a *miss* occurs.

In a **direct mapped cache**, lower order line address bits are used to access the directory. Since multiple line addresses map into the same location in the cache directory, the upper line address bits (tag bits) must be compared with the directory address to ensure a hit. If a comparison is not valid, the result is a cache miss, or simply a miss. The simplest way to allocate the cache to the system memory is to determine how many cache lines there are (16,384 for example) and just chop the system memory into the same number of chunks. Then each chunk gets the use of one cache line. This is called *direct mapping*. So if we have 64 MB of main memory addresses, each cache line would be shared by 4,096 memory addresses (64 M divided by 16 K).

The **set associative cache** operates in a fashion somewhat similar to the direct-mapped cache. This is a compromise between the direct mapped and fully associative designs. In this case the cache is broken into sets where each set contains "N" cache lines, let's say 4. Then, each memory address is assigned a set, and can be cached in any one of those 4 locations within the set that it is assigned to. In other words, *within each set* the cache is associative, and thus the name.
This design means that there are "N" possible places that a given memory location may be in the cache. The tradeoff is that there are "N" times as many memory locations competing for the same "N" lines in the set. Let's suppose in our example that we are using a 4-way set associative cache. So instead of a single block of 16,384 lines, we have 4,096 sets with 4 lines in each. Each of these sets is shared by 16,384 memory addresses (64 M divided by 4 K) instead of 4,096 addresses as in the case of the direct mapped cache. So there is more to share

(4 lines instead of 1) but more addresses sharing it (16,384 instead of 4,096).

Indeed, the *direct-mapped* and *fully associative* caches are just special cases of the *set associative* cache. This is because, if you set your "N" to 1, then you have 1-way set associative cache. That means there is only one line per set, which is the same as a *direct-mapped* cache because each memory address is back to pointing to only one possible cache location. On the other hand, suppose you set your "N" really large; say, you set "N" to be equal to the number of lines in the cache (16,384 in our example). If you do this, then you only have one set, containing all of the cache lines, and every memory location points to that huge set. This means that any memory address can be in any line, and you are back to a *fully associative* cache.

Cache Slots

When you choose a slot to be evicted, you look at the dirty bit. A *slot* consists of the following:

- **V,** the valid bit, indicating whether the slot holds valid data. If **V = 1**, then the data is valid. If **V = 0**, the data is not valid. Initially, it is invalid until data is placed into it.
- **D,** the dirty bit. This bit only has meaning if **V = 1**. This indicates that the data in the slot has been modified (written to) or not. If **D = 1**, the data has been modified since being in the cache. If **D = 0**, then the data is the same as it was when it first entered the cache.
- **Tag:** The tag represents the upper bits of the address, which includes the copied data as well as the requested memory location. The *tag* contains (part of) the address of the actual data fetched from the main memory.
- **Cache Line** This is the actual data itself. There are N bytes, where N is a power of 2. We will also call this the **data block**.

Cache Hits and Cache Misses

When you look for data at a given address, and find it stored in cache, then you have a *cache hit*. If you do not find the data, then it is a *cache miss*. You want to maximize cache hits, because then you have much improved performance (speed). But what happens in a cache miss?
As you run your program, it needs data at an address. This can either be the address of actual data being loaded or stored, or it can be the address of an instruction.

You look for the address to see if it is one of the addresses in cache. If the address (and its contents) is NOT in the cache, then you must access it from main memory. This means that you must copy the data from memory to the cache. Since the cache is a small subset of main memory, there may be data that you need to remove from the cache (just as you had to remove books from your computer, if you needed other books to replace it).

Categories of Cache Misses

We can subdivide cache misses into one of three categories: *compulsory miss, conflict miss*, and *capacity miss*.

Compulsory miss (or **cold miss**) occurs when there is little or no data in the cache. Initially, you are going to have a cache miss, no matter how big the cache is, and no matter how many bytes the cache line contains. Once a program has been running, the cache becomes more fully utilized, and such misses don't occur.

Conflict miss usually occurs in direct-mapped caches. Two cache lines may map to the same cache slot, even as there may be empty slots

that could store both cache lines. Such a conflict would force an unnecessary eviction of a cache line, but this is the price you pay for direct-mapped cache. Set-associative caches have this problem too, to a lesser degree. Fully associative caches avoid this kind of problem.

Capacity miss can be one of two sorts. One is a miss that would not have occurred if the cache size were larger (more slots). The other occurs with the number of bytes in a cache line. Suppose you are processing a large *int* array. If the cache line has 16 bytes, you will get a cache miss after processing 4 elements of the array. Had the cache line been larger, you would not have had this miss. These misses also occur in *virtual memory*.

Types of Caches

Caches are differentiated by the way they handle updates to the cache. Caching systems typically come in one of three forms: *Write-through, Write-around, Write-back.*

Write-through: Updates (writes) to the data being cached are written both to the cache and to the primary copy. A write is not considered complete until the write to the primary is completed. This says that you update main memory at the same time you update cache. *Write-through* caches tend to be the most common because they provide a good balance of performance and reliability. A block of data can get into the cache by an application either reading or writing data. Once data is read or written, any subsequent read of the block is quickly returned to the application from the cache instead of slower primary storage.

Write-around: The cache contains only copies of data that have been READ. Updates to data are not written to the cache. They go *only* to primary storage. A *write-around* cache behaves just like a write-through cache, except that the cache cannot receive blocks of data from a write operation. A write cannot cause a block to initially be placed into the cache, but if a write causes an update to a block already residing in the cache, then the cached block is updated. Write-around caches are used when the application is writing a lot of data that is seldom accessed shortly after the write occurs. This approach avoids populating the cache with data that might not be frequently accessed. There are few, select systems that benefit significantly from write-around caches.

Write-back: Updates are written only to the cache. Afterwards, and in the background, the cached data is used to update the primary storage. This says that you update main memory only when the cache line is evicted. Otherwise, only update the cache

All caches have one additional similarity; they all are of finite size and therefore need to manage their limited ability to store active data. All caches have replacement algorithms that determine when more recently accessed data should be retained and therefore manage when older data can be safely released from the cache and the space reclaimed. A *write-back* cache stores both read and write requests directly in the cache, so it can significantly speed up *both* reading and writing operations. However, write-back caches have the added complexity of ensuring that the cached write block eventually reaches primary storage. This approach requires handling a number of error conditions to protect against data corruption if the cache and the primary storage get out of synch. This added complexity and risk reduces the popularity of write-back caches, so they are deployed much less frequently than write-through or write-around caches.

Summary

One major advantage of cache is that it holds or stores data that are frequently used by the processor. It is a mechanism used to store

International Journal of Research (IJR)
e-ISSN: 2348-6848, p- ISSN: 2348-795X Volume 2, Issue 06, June 2015
Available at http://internationaljournalofresearch.org

frequently accessed data in fast memory. Cache is a subset of the data found in RAM, which means any data you find in cache, you can also find in RAM. When data at an address is needed, the CPU first searches in the cache. If it finds it there, then a load or store is performed to or from a register. If a cache miss occurs, then a cache line needs to be copied from main memory to a slot in a cache, which may require the eviction of a slot. In an inclusive scheme, L1 data would have to be stored into the L2 cache. If the L2 is write back (as is the case with pretty much any CPU), then it frees the CPU core up since the write back logic is part of the L2 cache, not the core.

Ideally, you want the percentage of cache hits to be high, to give the illusion that all the useful data is in the cache, and being accessed at the speed of the cache, as opposed to the speed of main memory. How effective this is depends on how much spatial and temporal locality, the mapping and replacement policy, of the scheme.

Conclusion

One problem caches have to address is keeping the data copies consistent. Cache is always stored for later use. If you are using something stored, the chance is that you are not getting the newest version of it. When a web page is cached, for example, what happens if that original page gets modified? Subsequent reads of that data require checking that the cached copy is current. Caches can also be used to speed writes. In this case, the caching technology must ensure that when the cached copy is updated, the primary copy of data is always protected and up-to-date. Caching must also take into account the possibility that the primary copy of the data has been modified from another source, rendering the copy in the cache now obsolete.

Caching only benefits situations where data is accessed repeatedly. In situations where data is accessed only once and not again for a very long time (such as a single person playing music or watching a movie), caching provides very little benefit. But in the case of a web server that is responding to requests for the same web pages over and over, caching can dramatically improve performance.

Ideally, an exclusive cache scheme will give you the most available room to store information. However, the scheme requires more logic and, similar to increasing associativity, there comes a point when the logic required to perform the task is not justified by the benefits. This is especially true when the lower level cache is much smaller in size compared to the higher levels of cache. The case of the original Pentium4 is instructive. It has a separate high speed trace cache and a small L1 data cache with a large unified L2 cache. So, Intel would have needed to put in logic to save some space. P4 is optimized by design for streaming data. In that case, an exclusive cache hierarchy would be useless since you would be filling the L2 cache with new information on every bus cycle. The 8kB is probably sufficient to store basic data for simple, non-streaming threads.

References

[1.] Anderson T., Gupta A, and Martonosi M. (2013). Computer System Laboratory. Stanford, CA.

[2.] Bianchi, G., Detti, A., Caponi, A., and Melazzi, N.B., (2013*). Check before storing: what is the performance price of content integrity verification in lru caching?.* SIGCOMM Comput. Comm. Rev. , vol. 43, pp. 59–67.

[3.] Castro, R.S, Lago, A.P and Silva, D.D (2003). *Adaptive Compressed Caching: Design and Implementation.* In Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing.

[4.] Che, H., Tung, Y, and Wang, Z (2002). *Hierarchical Web caching systems: modeling, design and experimental results*. IEEE JSAC, vol. 20, no. 7, pp. 1305–1314, 2002.

[5.] Daintith, J. (2014). *Memory hierarchy*. A Dictionary of Computing. *Encyclopedia.com.*URL:http://www.encyclopedia.com/doc/1O11-memoryhierarchy.html

[6.] Dan, A. and Towsley, D. (1990). *An approximate analysis of the LRU and FIFO buffer replacement schemes.* SIGMETRICS Perform. Eval. Rev. , vol. 18, pp. 143–152.

[7.] Fink, M. (2014). *The end of a necessary evil: collapsing the memory hierarchy*. Retrieved 9/04/2015. 10:44 am. URL: http://www8.hp.com/hpnext/posts/end-necessary-evil-collapsing-memory-hierarchy#.VSZJRy4k3XQ

[8.] Fricker, C., Robert, P., and Roberts, J. (2012). *A versatile and accurate approximation for lru cache performance* ITC.

[9.] Gallo, M., Kauffmann, B., Muscariello, L., Simonian, A., and Tanguy, C., (2012). *Performance evaluation of the random replacement policy for networks of caches.* SIGMETRICS Perf. Eval. Rev. , vol. 40(1), pp. 395–396.

[10.] Gupta, A., and Rothberg, E. (1982) *Parallel ICCG on a Hierarchical Memory Multiprocessor:Addressing the Triangular Solve Bottleneck*. Technical Report CSL-TR-90-449, Stanford University Computer Systems Laboratory.

[11.] Higbee L. *Quick and easy cache performance analysis*. ACM Sigarch Computer Architecture News. Vol 18, No. 2, 1990. Pages 33-44. NY, USA

[12.] Intel Corporation (2002). *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*. URL:ftp://download.intel.com/design/PentiumII/manuals/24319002.pdf

[13.] Jiang, W. Ioannidis, S. Massouli´e, L. and Picconi, F. (2012). *Orchestrating massively distributed CDNs.* ACM CoNEXT.

[14.] Rosenthal, D. (2014). *Bringing data infrastructures to horizon2020. Economic Sustainability of Digital Preservation*. 3rd UPDATE Conference. Amsterdam, The Netherlands. p. 10.

[15.] Rosenthal, D.S.H. (2010). *Keeping Bits Safe: How Hard Can It Be?*. ACM Queue. Retrieved 5/04/2014. URL:http://queue.acm.org/detail.cfm?id=1866298

[16.] Rosensweig, E., Kurose, J., and Towsley, D. (2010). *Approximate Models for General Cache Networks.* INFOCOM.

[17.] Shibu, A. (2014). *Different types of mappings used in cache memory*. Computer Architecture & Design. CareerRide.com.www.careerride.com/view.aspx?id=2274. *Retrieved 8/04/2014*

[18.] Wikipedia. (2013). *Performance Tuning*.URL:http://en.wikipedia.org/wiki/Performance_tuning