Cache in a Memory Hierarchy

Paulo J. D. Domingues

Curso de Especialização em Informática
Universidade do Minho
pado@clix.pt

Abstract: In the past two decades, the steady increase on processor performance was not followed by a similar improvement on memory technology. This lead computer designers to build a memory hierarchy where the top level is built up of very fast but also very small memory devices and the bottom level by slower and larger memories. This overview focuses the top levels, the cache memory, and the way they largely improve system performance without significantly increase its cost: what are the constrains on cache size, the advantages and disadvantages of using several levels of cache opposed to just one, the mapping and writing policies, the differences between some cache organizations; which techniques can be used to improve miss rate, miss penalty and hit time.

1 Introduction

The memory hierarchy available on a computer system depends on its use: a desktop computer has a single user and usually runs one application at a time, server computers are used by several users at a time to simultaneously run several applications, embedded systems typically run only one application, possibly without an operating system and requiring a small memory. This document focuses on desktop computers, although many of the presented solutions also apply to the other computer systems.

The main goal of a cache memory [1] is to speedup a computer system without significantly increasing its cost. This can be achieved with multiple layers of cache memory, where a small but fast memory device is used to hide slower and larger memory devices. The goal is to provide a low cost memory system almost as fast as the fastest memory level on the hierarchy.

When using a cache, the system first checks the cache to see if an item is in there. If there is, it is called a *cache hit*; if not, a *cache miss* occurs and the computer must wait for a round trip from the larger, slower memory in the hierarchy below. Usually data in one level is also found in the level below.

Efficiency in a memory hierarchy is due to two principles, the *temporal locality* and *spatial locality*. It means that in a fairly large program, only small adjacent portions of the code or data are used at any given time. As strange as it may seem, this works for the majority of programs. Even if the executable is 10 megabytes in size, only a handful of bytes from that program are in use at any time, and their rate of repetition is very high. This is why a small cache can efficiently cache a large memory system. Therefore, it is not worth to construct a computer using only the fastest (and expensive) memory devices, if it is possible to deliver most of this effectiveness for a fraction of the cost.

2 Levels

Why having more than one level of cache? The reason is that the faster the memory is, the more expensive it gets. Furthermore [3], the larger the cache is slower it gets due to the time needed to access the requested address, especially for higher clock cycles.

Modern computer systems use at least two levels of cache, located inside the processor chip. The second level used to be off-chip but now is usually on-chip too. The reasons are that a smaller cache means:

- shorter path for physical signal propagation, since light takes one nanosecond to travel one foot in vacuum;
- less decoding time for logical interpretation;
- fewer items to choose between;
- shorter codes to distinguish between them;
- higher-grade technology can be employed.

2.1 Level 1

Level 1 or primary cache is the fastest memory on the PC with an access time of 10s of *ns*. It is built directly into the processor itself and is very small, generally from 8 KB to 64 KB, with eventually larger amounts on the newer processors, but always extremely fast running at the same speed as the processor. If the processor requests information and can find it in the level 1 cache, that is the best case and the most common, the system does not have to wait more than one or two clock cycles.

There are two different ways that the processor can organize its primary cache: some processors have a single cache to handle both command instructions and program data; this is called a *unified cache* as opposite to a *separate cache* where data and instruction are kept separate. In some implementations the capabilities of the data and instruction caches may be slightly different. For example [1], on the Pentium 4 the data cache can use the write-back policy, while the instruction cache is write-through. Overall the performance difference between integrated and separate primary caches is not significant.

2.2 Level 2

The level 2 cache is a secondary cache to the level 1, being larger and slightly slower. It is used to catch recent accesses that are not caught by the level 1 cache and it is usually 64 KB to 2 MB in size. It used to be placed either on the motherboard or on a daughter board that inserts into the motherboard, but it is now usually on-chip.

Level 2 Components: The level 2 cache is composed of two main components. These may not be physically located in the same chips, but they represent how the cache logically works [2]:

- *The Data Store*: This is where the cached information is actually kept. When reference is made to "storing something in the cache" or "retrieving something from the cache", this is where the actual data goes to or comes from. When someone says that the cache is 256 KB or 512 KB, they are referring to the size of the data store. The larger the store, the more information can be cached and the more likelihood of the cache being able to satisfy a request.

- *The Tag RAM*: This is a small area of memory used by the cache to keep track of where in memory the entries in the data store belong. The size of the tag RAM influence how much of main memory can be cached.

2.3 Unified *vs.* Separate Data and Instruction Caches

Almost all level 2 caches work on both data and processor instructions (code, programs). They do not differentiate between the two because they view both as just memory addresses. However, many processors use a split design for their level 1 cache. For example [1], the Intel "Classic" Pentium (P54C) processor uses an 8 KB cache for data, and a separate 8 KB cache for program instructions. This is more efficient due to the way the processor is designed, and does not really affect performance very much compared to a single 16 KB cache, though it might lead to a very slightly lower hit ratio. Each of these caches can have different characteristics. For example they can use different mapping techniques.

3 Mapping

A very important factor in determining the effectiveness of the cache relates to how the cache is mapped to the system memory, since there are many different ways to allocate the storage in a cache to the memory addresses it serves. As an example, a system with 512 KB of L2 cache and 128 MB of main memory: how to decide to map the 16,384 address lines in the cache on to the 128 MB memory?

There are three different ways that this mapping can generally be done [1], [3]. The choice of the mapping technique is so critical to the design that the cache is often named after this choice.

3.1 Direct Mapped Cache

The simplest and fastest way to allocate the cache to the system memory is to determine how many cache lines there are (16,384 in the example) and just cut the system memory into the same number of fractions. Then each fraction gets the use of one cache line. This is called *direct mapping*. So if we have 128 MB of main memory addresses, each cache line would be shared by 8,102 memory addresses (128 MB divided by 16 K).

3.2 Fully Associative Cache

Instead of hard-allocating cache lines to particular memory locations, it is possible to design the cache so that any line can store the contents of any memory location. This leads to very complex and expensive implementation exponentially increasing with size.

3.3 N-Way Set Associative Cache

"N" is a number, typically 2, 4, 8 etc. 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. To each memory address a set is assigned which can be cached in any one of those “N” cache lines within the set that is assigned to. In other words, *within each set* the cache is associative.

This design means that there are "N" possible places that a given memory location may be in the cache. The trade-off is that there are "N" times as many memory locations competing for the same "N" lines in the set. In the previous example, using a 4-way set associative cache replaces a single block of 16,384 lines, by 4,096 sets with 4 lines in each. Each of these sets is shared by 32,768 memory addresses (128 MB divided by 4 K) instead of 8,102 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 8,102).

Conceptually, the direct mapped and fully associative caches are just "special cases" of the N-way set associative cache. You can set "N" to 1 to make a "1-way" set associative cache. If you do this, then 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, if "N" is equal to the number of lines in the cache (16,384 in the example), then there is only one set containing all the cache lines and every memory location points to that huge set.

3.4 Comparing the Cache Mapping Techniques

There is a critical trade-off in cache performance that has led to the creation of the various cache mapping techniques described before. In order for the cache to have good performance, it should maximize both of the following:

- *Hit Ratio*: increase as much as possible the probability of the cache containing the memory addresses that the processor wants. Otherwise much of the benefit of caching are lost, because there will be too many misses.
- *Search Speed*: to be able to determine as quickly as possible if have scored a hit in the cache. Otherwise a small amount of time is lost on every access, hit *or* miss, while searching the cache.

Table 1: Perfomance comparison between cache mapping techniques

Cache Type	Hit Ratio	Search Speed
Direct Mapped	Good	Best
Fully Associative	Best	Moderate
N-Way Set Associative, N>1	Very Good, Better as N Increases	Good, Worse as N Increases

The direct mapped and set associative caches are the most common. Direct mapping is used more for level 2 caches, while the higher-performance set-associative cache is found more commonly on the smaller primary caches contained within processors.

4 Replacement of a Cache Line

Whenever a miss occur, a cache line must be replaced with the requested data by the cache controller. If the cache is direct-mapped there is no difficult decision to be made, since there is only one place to where the new data can be copy. If the cache is associative, one out of several lines that can be replaced. The controller can use one of the following three strategies [1]:

- *Randomly* decide which line to replace; this it the simplest and usually less effective method to implement.
- *Least-recently used*: trusting the past can predict the future; this strategy need to keep track of the access to lines, becoming increasingly expensive as the number of lines increase.
- *First in, first out*: it is an approximation of the previous one, determining the oldest line (which is not necessarily the least recently used).

5 Write Policy

Besides caching reads *from* memory, the system can also cache writes *to* memory. The handling of the address bits and the cache lines is not too different from the read process. However, there are two really different methods for the cache to handle writes and an evolution of the second one [1], [3].

5.1 Write-back Cache

Also called "copy back" cache, this policy is "full" write caching of the system memory. When a write is made to system memory at a location that is currently cached, the new data is only written to the cache and marked as modified, not actually written to the system memory. Later, if another memory location needs to use the cache line where this data is stored, it is saved ("written back") to the system memory and then the line can be used by the new address.

It is difficult to implement than the following methods, not assuring a “clean” cache, meaning that there is different data in cache and in memory and a read miss can result in a write to memory of the newer data to be replaced.

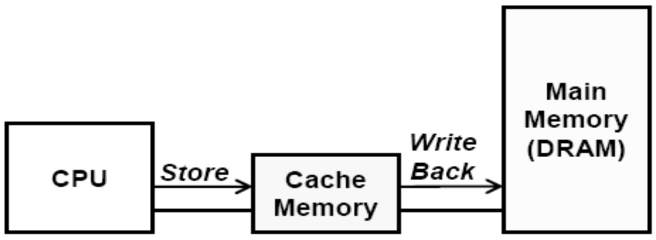

Fig. 1: Write-back cache

5.2 Write-through Cache

With this method every time the processor writes to a cached memory location, both the cache and the underlying memory location are updated. This is really sort of like "half caching" of writes; the data is just written in the cache in case it is needed to be read by the processor soon, but the write itself is not actually cached because it still have to initiate a memory write operation each time.

This simplified data coherency once the next memory level has the current copy of the data, which is especially important in multiprocessor environment.

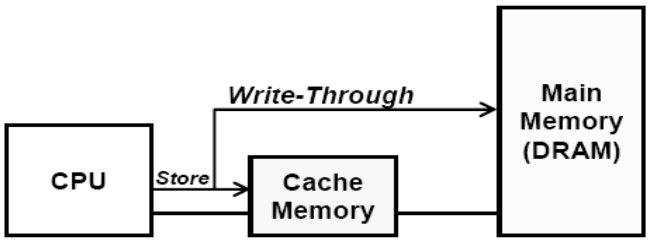

Fig. 2: Write-through cache

5.3 Buffered Write-through

This technique expands the previous one, improving it by providing zero-wait state write operation for both hits and misses. When a write occurs, buffered write-through caches cheat the processor into thinking that the information was written to memory in zero wait states, when in fact the write has not been performed yet. The look-through cache controller stares the entire write operation in a buffer, enabling it to complete the memory writes later without impacting the processor performance.

6 Cache Transfer Technologies

6.1 Cache Bursting

In a typical level 2 cache, each cache line contains 32 bytes and transfers to and from the cache, which occurs 32 bytes (256 bits) at a time. The normal transfer paths are not so wide, which means several (four normally) transfers are done in sequence. Because the transfers are from consecutive memory locations there is no need to specify a different address after the first one; this makes the second, third and fourth accesses extremely fast. This high-performance access is called "*bursting*" or using the cache in "*burst mode*". All modern level 2 caches use this type of access. The timing, in clock cycles, to perform this quadruple read is normally stated as "x-y-y-y". For example, with ‘3-1-1-1" timing the first read takes 3 clock cycles and the next three take 1 each, for a total of 6. Obviously, the lower these numbers, the better [1].

6.2 Asynchronous Cache

The oldest and slowest type of cache timing is asynchronous cache. Asynchronous means that transfers are not tied to the system clock. A request is sent to the cache, and the cache responds. This happens independently of what the system clock (on the memory bus) is doing. This is similar to how most system memory works (typical FPM or EDO memories are also asynchronous and relatively slow, for this reason).

Because asynchronous cache is not tied to the system clock, it can have problems dealing with faster clock speeds. At very slow speeds like 33 MHz it is capable of 2-1-1-1 timing (which is very good) but at slightly superior speeds like 60 or 66 MHz, drops down to 3-2-2-2 (which is pretty bad). For this reason, asynchronous cache was commonly found on 486 class motherboards but was not generally used on Pentium and later class machines.

6.3 Synchronous Burst Cache

Unlike asynchronous cache, which operates independently of the system clock, synchronous cache is tied to the memory bus clock. On each tick of the system clock, a transfer can be done to or from the cache (if it is ready). This means that it is capable of handling faster system speeds without slowing down the way asynchronous cache does. However, the faster the system runs, the faster the SRAM chips have to be, in order to keep up. Otherwise timing problems (crashes, lockups) will occur.

Even this type of cache slows down at very high speeds. It is capable of 2-1-1-1 operations up to 66 MHz, but then it slows down to 3-2-2-2 at higher speeds. Synchronous burst cache never quite caught on because pipelined burst cache was developed at around the same time and taken the market away from synchronous burst before the it could really get going.

6.4 Pipelined Burst (PLB) Cache

Pipelining is a technology commonly used in processors to increase performance. In the pipelined burst (PLB) cache it is used in a similar way. PLB cache adds special circuitry that allows the four data transfers that occur in a "burst" to be done partially at the same time. In essence, the second transfer begins before the first transfer is completed.

Because of the complexity of the circuitry, a bit more time is required initially to set up the "pipeline". For this reason, pipelined burst cache is slightly slower than synchronous burst cache for the initial read, requiring 3 clock cycles instead of 2 for sync burst. However, this parallelism allows PLB cache to burst at a single clock cycle for the remaining 3 transfers even up to very high clock speeds. PLB cache is now the standard for almost all quality Pentium class motherboards.

7 Improving Performance

7.1 Reducing Miss Penalty

Miss penalty is the additional time required because of a miss. The penalty for L1 misses served from L2 is typically 5 to 10 cycles. The penalty for L1 misses served from main memory is typically 25 to 100 cycles; reducing this value can increase the overall performance [1], [2]. The methods that can be used to reduce it are:

- *Multilevel Caches*. This allows having the first cache level small enough to match the fastest CPU's clock and the second level large enough to capture many of the access that would go into main memory.
- *Critical Word First and Early Restart*. Trusting the CPU normally only needs one word of a line at a time, the cache controller requests the missed word from memory and sends it to the CPU as soon as it arrives, letting the CPU continue execution while retrieves the rest of the words from memory to fill the line. Using only *early restart*, words are fetched normally from memory, but as soon as the requested word arrives are immediately sent to the CPU.
- *Giving Priority to Read Miss over Writes*. Instead of writing a dirty line into memory and then reading a new line, the dirty line is written into a buffer allowing the read to start and only then the dirty line is written in memory.

- *Merging Write Buffer*. Like the one before, it also uses buffers, improving its performance. When a write to memory is needed and the buffer is empty the word is copied into it. If the buffer is filled, the addresses are compared and when matching both data are combined without having to write immediately to memory. When the write has to take place all words are written, taking less time that written one at a time.
- *Victim Caches*. Basically the idea is to save what is to be discarded, once it may be needed again. This is achieved placing a small, fully associative cache between L1 and L2 (the AMD uses an eight entries victim cache).

7.2 Reducing Miss Rate

The miss rate is the fraction of memory references during the execution of a program, or a part of a program that miss. Because of the increasing gap between processor and memory, this is the primary cause of obtaining a performance worse that primarily expected.

They are classified into three categories: *Compulsory* (when accessing a block for the first time, it can not be in cache); *Capacity* (when the cache can not contain all blocks needed for a program execution) and *Conflict* (when a block is discarded and later needed again).

There are several ways the miss rate can be reduced:

- *Increasing Block Size*. Because of spatial locality it is likely that the next word be after the last one. The side effect is that increasing block size also increases miss penalty.
- *Using Larger Caches*. It sure reduces capacity misses but increases cost and hit time. Thus is mostly used in off-chip caches.
- *Through Higher Associativity*. Miss rate decrease with associativity but is not need to have a full associative cache once that an eight-way set associative cache is as effective as a full associative cache. On the other hand, the miss rate of a direct-mapping *N* size cache is similar to a 2-way set associative cache with size *N/2*. The more a cache is associative the higher the hit time.
- *Using Way Prediction and Pseudo Associative Caches*. To assure *way prediction* extra bit are needed to early set the multiplexor to select the desired block, meaning that only a tag comparison is performed in that clock cycle. Another approach is to check first one part of the cache and on a miss check the other part. If the data is found only half a miss happened.
- *With Compiler Optimizations*. This is the only technique without any hardware changes. To reduce the miss on instructions the compiler can reorder procedures in memory. To reduce miss of data there are four main categories of improvement: *Merging Arrays* (one array of elements is better that several single arrays); *Loop Interchange* (change the nesting of loops to access the data in the same order as in memory); *Loop Fusion* (combine two independent loops that have same looping and some variables overlap); *Blocking* (improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows

7.3 Reducing Miss Penalty or Rate via Parallelism

The following techniques overlap the execution of instructions with activity in the memory hierarchy.

- *Nonblocking Caches***,** *to Reduce Stalls on Cache Misses*. On pipeline computers that allow out-of-order completion, the CPU should not stall on a cache miss, being able to continue fetching instructions on a data cache miss. With this method the cache can continue to supply cache hits during a miss.
- *H/W Prefetching of Instructions and Data*. An extra block is placed on the stream buffer, which is checked on miss. Typically the processor on a cache misses fetches besides the requested blocks, the next one. It needs an extra memory bandwidth that may be used without penalty.
- *Compiler-Controlled Prefetching*. It is an option to h/w prefetching, performed by the compiler, which can load data into registers, or into cache. Like prefetching costs time, this prefetching should save more than it cost.

7.4 Reducing Hit Time

Hit time is how long it takes to deliver a word in the cache to the CPU, including the time for set selection, line identification, and word selection. Hit time is typically 1 to 2 clock cycle for L1 caches.

The techniques to reduce the hit time are:

- *Small and Simple Caches***.** Most of the time during a cache hit is consumed reading the tag memory with the index portion of the address and comparing it with the address. It is know that small hardware is faster so time decreases as cache size also decreases.

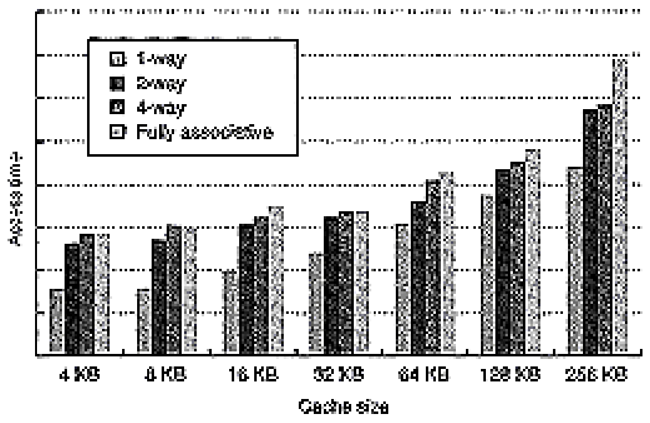

Fig. 3: Access time comparison for different cache sizes and associativity

- *Avoiding Address Translation during Indexing of the Cache***.** Every cache needs to translate the virtual address from the CPU into a physical memory address. Like the processor treats main memory like any other hierarchy levels, the address of the virtual memory that exists on disk has to be mapped onto main memory. Trying to make the common case the fastest one, we should use virtual addresses for the cache (hits are more common than misses)
- *Pipelined Cache Access*. This makes the effective latency of L1 cache multiple of clock cycles (four clock cycles in Pentium 4).
- *Trace Caches*. Instead of limiting the instructions in a static cache block to special locality, a trace cache finds a dynamic sequence of instructions *including taken branches* to load into a cache block. This lets to increase the instruction-level parallelism.

7.5 Main Memory

Being the next level down in the hierarchy, it satisfies the demands of cache besides serving as the I/O interface. There are two characteristics that can be measure: *latency* and *bandwidth.* Latency affects primarily cache (in the miss penalty), while bandwidth has more effect on I/O and multiprocessors. Nevertheless it is easier to improve memory bandwidth than latency and with the growing of L2 caches bandwidth has also become important to caches [1].

There are three techniques to improve memory bandwidth:

- *Wider Main Memory*. At twice the width we need half the memory accesses. Once CPU needs one word at a time, there is the need for a multiplexor between the cache and the CPU, or between L1 and L2 cache. This method also increases costs while needs a larger bus.
- *Simple Interleaved Memory*. A way of taking advantage of parallelism of having several chips in a memory system organized in *banks*. Each bank can be one word width (no need to change the bus) but sending addresses to several banks allows them all to read simultaneously.
- *Independent Memory Banks*. A generalization of interleaving is to allow multiple independent accesses where multiple memory controllers allow banks to operate independently. Each bank needs a separate address lines and perhaps a separate data bus.

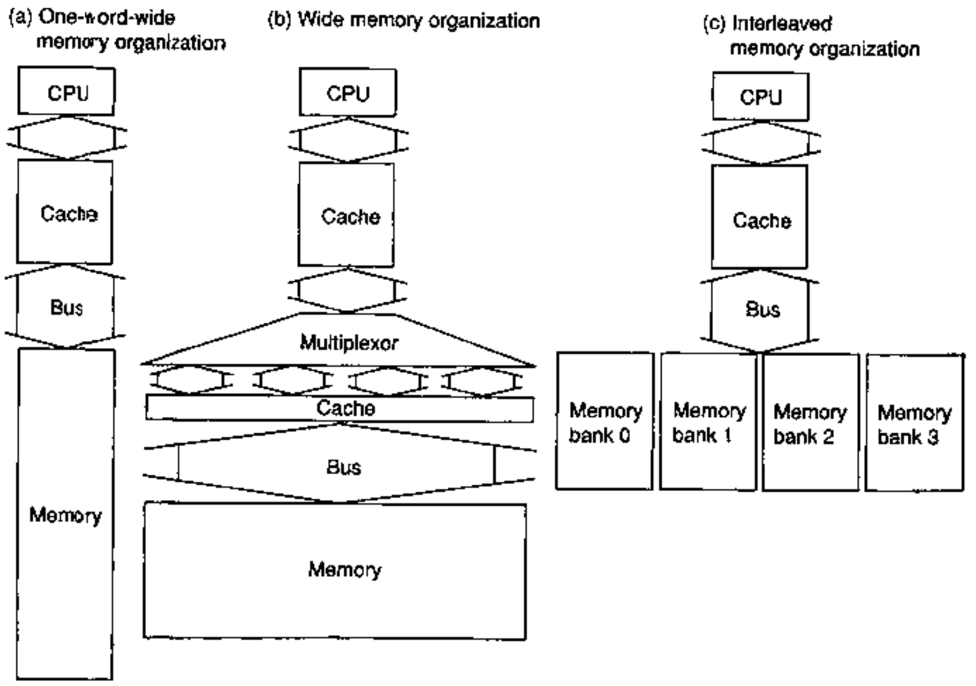

Fig. 4: Different memory organizations. a) One-word-wide; b) Wide; c) Interleaved

8 Coherence Problem

In a multiprocessor system, where each processor has it is own cache, the problems of cache coherence grows. There can be multiple copies of the same data in different caches; each CPU modifies locally their cache making cache and memory incoherent. Consequently when one processor request data from memory it might get wrong data [1].

Some solutions to this problem are:

- *Declare shared data to be non-cacheable* (by software), meaning that all access to share data are made to main memory which is against the purpose of caching;
- *Cache on read, but not cache writes cycles*. Writing to main memory the cache is not update but the data is marked as invalid. It is still possible that one CPU has in his cache old data;
- *Use bus snooping* to maintain coherence on write back to main memory. This works by making each CPU updates his own cache. If data not cached anywhere, just update memory and if data is cached locally, update cache and memory. Each other cache monitors (snoops) the common address bus for access to data that it holds. If they detect access, each cache marks his copy as invalid. At the end of the write, memory is correct and data cached in one processor only.
- *Use directory based*: cache lines maintain extra two bits per processor to maintain clean/dirty status bits [5].

9 Conclusions

Near sixty years ago engineers predicted that computers will need unlimited quantities of memory working as fast as possible. Nowadays actual engineers have achieved a solution to this problem by creating a memory hierarchy that keeps the costs at a satisfactory level while accompanying the growing performance of the processors. The name given to the small memories on top of the hierarchy was cache and was so successful that his name is given to almost all memory systems that contain the most used data of a large memory system, in a hidden way to the end user.

Most people keep a small phone book near the phone. This is no more no less than a cache of the global phone book and works more or less the same way.

So far, most computer systems use two level of cache, being probable that this number might increase in the future as the gap between CPU and memory increases. With the growing number of distributed systems, the next step might be in the cache coherence and the improvement of keeping data coherence in several computers.

References

[1] John L. Hennessy & David A. Patterson: Computer Architecture – A Quantitative Approach. 3thd Ed. Morgan Kaufmann Publishers (2003)

[2] Don Anderson, Tom Shanley: Pentium™ Processor System Architecture. Inc. MindShare (1995)

[3] William Stallings: Computer Organization and Architecture. 5th Ed. Copyrighted Material. (2000)

[4] Randal E. Bryant, David R. O'Hallaron: Computer Systems A Programmer's Perspective. Prentice Hall (2001)

[5] Michael Wu, Sarita Adve, Willy Zwaenepoel: The Cache Directory. ISCA (1997)

[6] Intel: Intel Pentium 4 Processor – Product Briefs

[7] Manuel E. Acacio, José González, José M. Garcia and José Duato, “Owner Prediction for Accelerating Cache-to-Cache Transfer Misses in a cc-NUMA Architecture” Universidad de Murcia, Spain.

[8] Philip Machanick and Zunaid Patel; School of Computer Science University of the Witwatersrand - “Further Cache and TLB Investigation of the RAMpage Memory Hierarchy”