

How The Memory Cache Works

By Gabriel Torres on September 12, 2007

Page 1 of 9

AMD 50x15

See how we're bridging the digital divide. Subscribe to 50x15 today!

50x15.com

Ads by Google

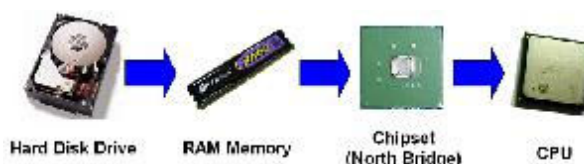
Introduction

The memory cache is high-speed memory available inside the CPU in order to speed up access to data and instructions stored in RAM memory. In this tutorial we will explain how this circuit works in an easy to follow language.

A computer is completely useless if you don't tell the processor (i.e. the CPU) what to do. This is done thru a program, which is a list of instructions telling the CPU what to do.

The CPU fetches programs from the RAM memory. The problem with the RAM memory is that when it's power is cut, it's contents are lost – this classifies the RAM memory as a “volatile” medium. Thus programs and data must be stored on non-volatile media (i.e. where the contents aren't lost after your turn your PC off) if you want to have them back after you turn off your PC, like hard disk drives and optical media like CDs and DVDs.

When you double click an icon on Windows to run a program, the program, which is usually stored on the computer's hard disk drive, is loaded into the RAM memory, and then from the RAM memory the CPU loads the program thru a circuit called memory controller, which is located inside the chipset (north bridge chip) on Intel processors or inside the CPU on AMD processors. On Figure 1 we summarize this (for AMD processors please ignore the chipset drawn).



click to enlarge

Figure 1: How stored data is transferred to the CPU.

The CPU can't fetch data directly from hard disk drives because they are too slow for it, even if you consider the fastest hard disk drive available. Just to give you some idea of what we are talking about, a SATA-300 hard disk drive – the fastest kind of hard disk drive available today for the regular user – has a maximum theoretical transfer rate of 300 MB/s. A CPU running internally at 2 GHz with 64-bit internal datapaths* will transfer data internally at 16 GB/s – over 50 times faster.

* Translation: the paths between the CPU internal circuits. This is rough math just to give you an idea, because CPUs have several different datapaths inside the CPU, each one having different lengths. For example, on AMD processors the datapath between the L2 memory cache and the L1 memory cache is 128-bit wide, while on current Intel CPUs this datapath is 256-bit wide. If you got confused don't worry. This is just to explain that the number we published in the above paragraph isn't fixed, but the CPU is always a lot faster than hard disk drives.

The difference in speed comes from the fact that hard disk drives are mechanical systems, which are slower than pure electronics systems, as mechanical parts have to move for the data to be retrieved (which is far slower than moving electrons around). RAM memory, on the other hand, is 100% electronic, thus faster than hard disk drives and optimally as fast as the CPU.

And here is the problem. Even the fastest RAM memory isn't as fast as the CPU. If you take DDR2-800 memories, they transfer data at 6,400 MB/s – 12,800 MB/s if dual channel mode is used. Even though this number is somewhat close to the 16 GB/s from the previous example, as current CPUs are capable of fetching data from the L2 memory cache at 128- or 256-bit rate, we are talking

about 32 GB/s or 64 GB/s if the CPU works internally at 2 GHz. Don't worry about what the heck "L2 memory cache" is right now, we will explain it later. All we want is that you get the idea that the RAM memory is slower than the CPU.

By the way, transfer rates can be calculated using the following formula (on all examples so far "data per clock" is equal to "1"):

Transfer rate = width (number of bits) x clock rate x data per clock / 8

The problem is not only the transfer rate, i.e. the transfer speed, but also latency. Latency (a.k.a. "access time") is how much time the memory delays in giving back the data that the CPU asked for – this isn't instantaneous. When the CPU asks for an instruction (or data) that is stored at a given address, the memory delays a certain time to deliver this instruction (or data) back. On current memories, if it is labeled as having a CL (CAS Latency, which is the latency we are talking about) of 5, this means that the memory will deliver the asked data only after five memory clock cycles – meaning that the CPU will have to wait.

Waiting reduces the CPU performance. If the CPU has to wait five memory clock cycles to receive the instruction or data it asked for, its performance will be only 1/5 of the performance it would get if it were using a memory capable of delivering data immediately. In other words, when accessing a DDR2-800 memory with CL5, the performance the CPU gets is the same as a memory working at 160 MHz (800 MHz / 5). In the real world the performance decrease isn't that much because memories work under a mode called burst mode where from the second data on, data can be delivered immediately, if it is stored on a contiguous address (usually the instructions of a given program are stored in sequential addresses). This is expressed as "x-1-1-1" (e.g. "5-1-1-1" for the memory in our example), meaning that the first data is delivered after five clock cycles but from the second data on data can be delivered in just one clock cycle – if it is stored on a contiguous address, like we said.

Originally at <http://www.hardwaresecrets.com/article/481/1> Pages (9): [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) » ... [Last](#) »

© 2004-6, Hardware Secrets, LLC. All Rights Reserved.

Total or partial reproduction of the contents of this site, as well as that of the texts available for downloading, be this in the electronic media, in print, or any other form of distribution, is expressly forbidden. Those who do not comply with these copyright laws will be indicted and punished according to the International Copyrights Law.

We do not take responsibility for material damage of any kind caused by the use of information contained in Hardware Secrets.

How The Memory Cache Works

By Gabriel Torres on September 12, 2007

Page 2 of 9

Free Part Info Lookup - 2.5M Datasheets, Crosses, Distri's
 Try Our Free Search - No Reg www.supplyframe.com

Ads by Google

Dynamic RAM vs. Static RAM

There are two types of RAM memory: dynamic (DRAM) and static (SRAM). The RAM memory used on the PC is dynamic. On this kind of memory each bit of data is stored inside the memory chip in a tiny capacitor. Capacitors are very small components, meaning that millions of them can be manufactured on a small area – this is called “high density”. On the other hand capacitors lose their charge after some time, so dynamic memories need a recharge process called refresh, which occurs from time to time. During this period data cannot be read or written. Dynamic memory is also cheaper than static memory and also consumes far less power. But, as we saw, on dynamic RAM data is not readily available and it can’t work as fast as the CPU.

Static memory, on the other hand, can work as fast as the CPU, because each data bit is stored on a circuit called flip-flop, which can also deliver data with zero or very small latency, because flip-flops do not require refresh cycles. The problem is that flip-flops require several transistors to be made, i.e. they are far bigger than a single capacitor. This means that on the same area where on a static memory only one flip-flop exists, on a dynamic memory there are hundreds of capacitors. Thus static memories provide a lower density – chips have lower capacity. The other two problems with static memory: it is more expensive and it consumes more power – thus it heats more.

On the table below we summarize the main differences between dynamic RAM (DRAM) and static RAM (SRAM).

| Feature | Dynamic RAM (DRAM) | Static RAM (SRAM) |
|--------------------------|--------------------|-------------------|
| Storage circuit | Capacitor | Flip-flop |
| Transfer speed | Slower than CPU | Same as CPU |
| Latency | High | Low |
| Density | High | Low |
| Power Consumption | Low | High |
| Cost | Cheap | Expensive |

Even though static RAM is faster than dynamic RAM, its disadvantages prevent it from being used as the main RAM circuit.

The solution found to lower the impact of using a RAM memory that is slower than the CPU was using a small amount of static RAM between the CPU and the RAM memory. This technique is called memory cache and nowadays this small amount of static memory is located inside the CPU.

The memory cache copies most recently accessed data from the RAM memory to the static memory and tries to guess what data the CPU will ask next, loading them to the static memory before the CPU actually ask for it. The goal is to make the CPU to access the memory cache instead of accessing directly the RAM memory, since it can retrieve data from the memory cache immediately or almost immediately, while it has to wait when accessing data located on RAM. The more the CPU accesses the memory cache instead of the RAM, the fastest the system will be. We will explain exactly how the memory cache works in just a moment.

By the way, here we are using the terms “data” and “instruction” interchangeably because what is stored inside each memory address doesn’t make any difference to the memory.

How The Memory Cache Works

By Gabriel Torres on September 12, 2007

Page 3 of 9

Increase App PerformanceDistributed caching for your application - download now.
www.gigaspace.com**AMD Athlon™ 64 X2**Black Edition processor 5000+ w/ Tunable Performance. Learn more!
www.amd.com/unleash

History of Memory Cache on PCs

This section is only for those interested on the historic aspects of memory cache. If you are not interested on this subject you can simply jump to the next page.

Memory cache was first used on PCs at the 386DX timeframe. Even though the CPU itself didn't have memory cache inside, its support circuitry – i.e. the chipset – had a memory cache controller. Thus the memory cache at this time was external to the CPU and thus was optional, i.e. the motherboard manufacturer could add it or not. If you had a motherboard without memory cache your PC would be far slower than a PC with this circuit. The amount of available memory cache varied as well depending on the motherboard model and typical values for that time were 64 KB and 128 KB. At this time the memory cache controller used an architecture known as "write-through", where for write operations – i.e. when the CPU wants to store data in memory – the memory cache controller updates the RAM memory immediately.

With the 486DX processor Intel added a small amount (8 KB) of memory cache inside the CPU. This internal memory cache was called L1 (level 1) or "internal", while the external memory cache was called L2 (level 2) or "external". The amount and existence of the external memory cache depended on the motherboard model. Typical amounts for that time were 128 KB and 256 KB. Later 486 models added the "write back" cache architecture, which is used until today, where for write operations the RAM memory isn't updated immediately, the CPU stores the data on the cache memory and the memory controller updates the RAM memory only when a cache miss occurs.

Then with the first Pentium processor Intel created two separated internal memory caches, one for instructions and another for data (at the time with 8 KB each). This architecture is still used to date, and that is why you sometimes see the L1 memory cache being referred as 64 KB + 64 KB, for example – this is because there are one 64 KB instruction L1 cache and one 64 KB data L1 cache. Of course we will explain later what is the difference between the two. At that time the L2 memory cache continued to be located on the motherboard, so its amount and existence depended on the motherboard model. Of course having a system without memory cache was insane. Typical amounts for that time were 256 KB and 512 KB.

On AMD side K5, K6 and K6-2 processors used this same architecture, with K6-III having a third memory cache (L3, level 3).

The problem with the L2 memory cache being external is that it is accessed with a lower clock rate, because since 486DX2 the CPU internal clock rate is different from the CPU external clock rate. While a Pentium-200 worked internally at 200 MHz, it accessed its L2 memory cache at 66 MHz, for example.

Then with P6 architecture Intel moved the memory cache from the motherboard to inside the CPU – what allowed the CPU to access it with its internal clock rate –, except on Pentium II, where the memory cache was not located inside the CPU but on the same printed circuit board where the CPU was soldered to (this printed circuit board was located inside a cartridge), running at half the CPU internal clock rate, and on Celeron-266 and Celeron-300, which had no memory cache at all (and thus they were the worst-performing CPUs in PC history).

This same architecture is used until today: both L1 and L2 memory caches are located inside the CPU running at the CPU internal clock rate. So the amount of memory cache you can have on your system will depend on the CPU model you have; there is no way to increase the amount of memory cache without replacing the CPU.

Originally at <http://www.hardwaresecrets.com/article/481/3>

Pages (9): [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#)
[»](#) ... [Last](#) [»](#)

How The Memory Cache Works

By Gabriel Torres on September 12, 2007

Page 4 of 9

Pentium4 code optimizer

Optimizes gcc generated Pentium4 code. Up to 84% speed improvements.
www.dalsoft.com

Fast Fingerprint Terminal

FVC2006 No.1 fingerprint algorithm, Color LCD, WiFi LAN, dual CPU, USB
www.supremainc.com

JBoss in Action

Configuring the Application Server
Download a Free Chapter Now!
www.manning.com

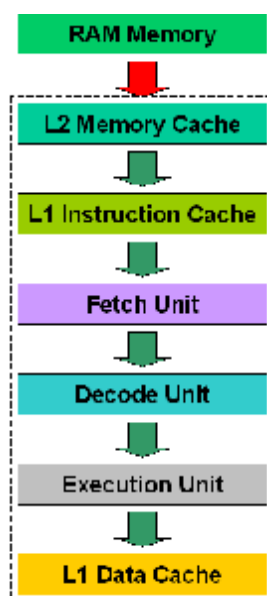
Secure your DNS

Stop Pharming and DNS Attacks
Secure, High Performance,
Always-On
www.nominum.com

Ads by Google

Meet The Memory Cache

On Figure 2 you can see a basic block diagram from a generic single-core CPU. Of course the actual block diagram will vary depending on the CPU and you can read our tutorials for each CPU line to take a look on their real block diagram ([Inside Pentium 4 Architecture](#), [Inside Intel Core Microarchitecture](#) and [Inside AMD64 Architecture](#)).



click to enlarge

Figure 2: Basic block diagram of a CPU.

The dotted line on Figure 2 represents the CPU body, as the RAM memory is located outside the CPU. The datapath between the RAM memory and the CPU is usually 64-bit wide (or 128-bit when dual channel memory configuration is used), running at the memory clock or the CPU external clock (or memory bus clock, in the case of AMD processors), which one is lower. We have already taught you how to calculate the memory transfer rate on the first page of this tutorial.

All the circuits inside the dotted box run at the CPU internal clock. Depending on the CPU some of its internal parts can even run at a higher clock rate. Also, the datapath between the CPU units can be wider, i.e. transfer more bits per clock cycle than 64 or 128. For example, the datapath between the L2 memory cache and the L1 instruction cache on modern processors is usually 256-bit wide. The datapath between the L1 instruction cache and the CPU fetch unit also varies depending on the CPU model – 128 bits is a typical value, but at the end of this tutorial we will present a table containing the main memory cache specs for the CPUs available on the market today. The higher the number the bits transferred per clock cycle, the fast the transfer will be done (in other words, the transfer rate will be higher).

In summary, all modern CPUs have three memory caches: L2, which is bigger and found between the RAM memory and the L1 instruction cache, which holds both instructions and data; the L1 instruction cache, which is used to store instructions to be executed by the CPU; and the L1 data cache, which is used to store data to be written back to the memory.

L1 and L2 means "Level 1" and "Level 2", respectively, and refers to the distance they are from the CPU core (execution unit). One common doubt is why having three separated cache memories (L1

data cache, L1 instruction cache and L2 cache).

Making zero latency static memory is a huge challenge, especially with CPUs running at very high clock rates. Since manufacturing near zero latency static RAM is very hard, the manufacturer uses this kind of memory only on the L1 memory cache. The L2 memory cache uses a static RAM that isn't as fast as the memory used on the L1 cache, as it provides some latency, thus it is a little bit slower than L1 memory cache.

Pay attention to Figure 2 and you will see that L1 instruction cache works as an "input cache", while L1 data cache works as an "output cache". L1 instruction cache – which is usually smaller than L2 cache – is particularly efficient when the program starts to repeat a small part of it (loop), because the required instructions will be closer to the fetch unit.

This is rarely mentioned, but the L1 instruction cache is also used to store other data besides the instructions to be decoded. Depending on the CPU it can be also used to store some pre-decode data and branching information (in summary, control data that will speed up the decoding process) and sometimes the L1 instruction cache is bigger than announced, because the manufacturer didn't add the extra space available for these extra pieces of information.

On the specs page of a CPU the L1 cache can be found with different kinds of representation. Some manufacturers list the two L1 cache separately (some times calling the instruction cache as "I" and the data cache as "D"), some add the amount of the two and writes "separated" – so a "128 KB, separated" would mean 64 KB instruction cache and 64 KB data cache –, and some simply add the two and you have to guess that the amount is total and you should divide by two to get the capacity of each cache. The exception, however, goes to CPUs based on Netburst microarchitecture, i.e. Pentium 4, Pentium D, Pentium 4-based Xeon and Pentium 4-based Celeron CPUs.

Processors based on Netburst microarchitecture don't have a L1 instruction cache, instead they have a trace execution cache, which is a cache located between the decode unit and the execution unit, storing instructions already decoded. So, the L1 instruction cache is there, but with a different name and a different location. We are mentioning this here because this is a very common mistake, to think that Pentium 4 processors don't have L1 instruction cache. So when comparing Pentium 4 to other CPUs people would think that its L1 cache is much smaller, because they are only counting the 8 KB L1 data cache. The trace execution cache of Netburst-based CPUs is of 150 KB and should be taken in account, of course.

Originally at <http://www.hardwaresecrets.com/article/481/4> Pages (9): [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) » ... [Last](#) »

© 2004-6, Hardware Secrets, LLC. All Rights Reserved.

Total or partial reproduction of the contents of this site, as well as that of the texts available for downloading, be this in the electronic media, in print, or any other form of distribution, is expressly forbidden. Those who do not comply with these copyright laws will be indicted and punished according to the International Copyrights Law.

We do not take responsibility for material damage of any kind caused by the use of information contained in Hardware Secrets.

How The Memory Cache Works

By Gabriel Torres on September 12, 2007

Page 5 of 9

AMD Athlon™ 64 X2

Black Edition processor 5000+ w/ Tunable Performance. Learn more!
www.amd.com/unleash

Intel Processor Info

Get the latest Intel processor blogs, white papers, and much more!
Hardware.ITtoolbox.com

Tsunami Flyer 6725S

Escolha a cor. Intel® Centrino® Duo. Só 1.230€
www.tsunami.pt

CompactPCI Express

Enclosures, backplanes, CPU boards and I/O boards for all applications
www.onestopsystems.com

Ads by Google

L2 Memory Cache on Multi-Core CPUs

On CPUs with more than one core the L2 cache architecture varies a lot, depending on the CPU.

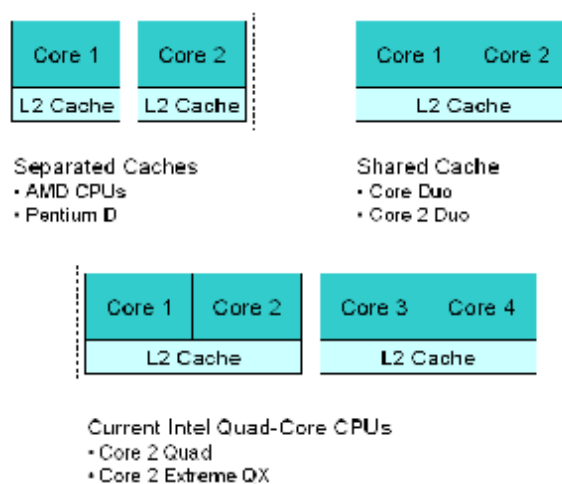
On Pentium D and AMD dual-core CPUs based on K8 architecture each CPU core has its own L2 memory cache. So each core works as if it were a completely independent CPU.

On Intel dual-core CPUs based on Core and Pentium M microarchitectures, there is only L2 memory cache, which is shared between the two cores.

Intel says that this shared architecture is better, because on the separated cache approach at some moment one core may run out of cache while the other may have unused parts on its own L2 memory cache. When this happens, the first core must grab data from the main RAM memory, even though there was empty space on the L2 memory cache of the second core that could be used to store data and prevent that core from accessing the main RAM memory. So on a Core 2 Duo processor with 4 MB L2 memory cache, one core may be using 3.5 MB while the other 512 KB (0.5 MB), contrasted to the fixed 50%-50% division used on other dual-core CPUs.

On the other hand, current quad-core Intel CPUs like Core 2 Extreme QX and Core 2 Quad use two dual-core chips, meaning that this sharing only occurs between cores 1 & 2 and 3 & 4. In the future Intel plans to launch quad-core CPUs using a single chip. When this happens the L2 cache will be shared between the four cores.

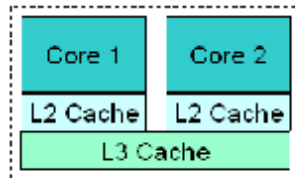
On Figure 3 you can see a comparison between these three L2 memory cache solutions.



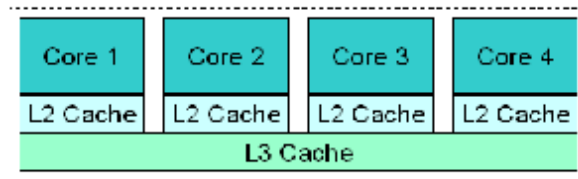
click to enlarge

Figure 3: Comparison between current L2 memory cache solutions on current multi-core CPUs.

AMD processors based on K10 architecture will have a shared L3 memory cache inside the CPU, being a hybrid between the other two approaches. This is shown on Figure 4. The size of this cache will depend on the CPU model, just like what happens with the size of L2 cache.



K10-based dual-core CPU



K10-based quad-core CPU

click to enlarge

Figure 4: K10 cache architecture.

Originally at <http://www.hardwaresecrets.com/article/481/5> Pages (9): [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) » ... [Last](#) »

© 2004-6, Hardware Secrets, LLC. All Rights Reserved.

Total or partial reproduction of the contents of this site, as well as that of the texts available for downloading, be this in the electronic media, in print, or any other form of distribution, is expressly forbidden. Those who do not comply with these copyright laws will be indicted and punished according to the International Copyrights Law.

We do not take responsibility for material damage of any kind caused by the use of information contained in Hardware Secrets.

How The Memory Cache Works

By Gabriel Torres on September 12, 2007

Page 6 of 9

Increase App Performance

Distributed caching for your application - download now.
www.gigaspace.com


Computer Cache Memory

Check our Selected Sites on Computer Cache Memory
www.SelectaSites.com

Apple Memory

100% Guaranteed Compatible Apple Memory Upgrades. Free Same Day Ship
MemorySuppliers.com/AppleMemor

CompactPCI

Enclosures, backplanes, CPU/IO bds Order PCIe cables
 **1-877-438-2724**
www.onestopsystems.com

Ads by Google

How It Works

Here is how the memory cache works. The CPU fetch unit looks for the next instruction to be executed in the L1 instruction cache. If it isn't there, it will look for it on the L2 cache. Then, if it is not there, it will have to go to the RAM memory to fetch the instruction.

We call a "hit" when the CPU loads a required instruction or data from the cache, and we call a "miss" if the required instruction or data isn't there and the CPU needs to access the system RAM memory directly.

Of course when you turn your PC on the caches are empty, so accessing the RAM memory is required – this is an inevitable cache miss. But after the first instruction is loaded, the show begins.

When the CPU loads an instruction from a certain memory position, a circuit called memory cache controller loads into the memory cache a small block of data below the current position that the CPU has just loaded. Since usually programs flow in a sequential way, the next memory position the CPU will request will probably be the position immediately below the memory position that it has just loaded. Since the memory cache controller already loaded some data below the first memory position read by the CPU, the next data will probably be inside the memory cache, so the CPU doesn't need to go outside to grab the data: it is already loaded inside in the memory cache embedded in the CPU, which it can access at its internal clock rate.

This amount of data is called line and it is usually 64 bytes long (more on that on the next page).

Besides loading this small amount of data, the memory controller is always trying to guess what the CPU will ask next. A circuit called prefetcher, for example, loads more data located after these first 64 bytes from RAM into the memory cache. If the program continues to load instructions and data from memory positions in a sequential way, the instructions and data that the CPU will ask next will be already loaded into the memory cache.

So we can summarize how the memory cache works as:

1. The CPU asks for instruction/data stored in address "a".
2. Since the contents from address "a" aren't inside the memory cache, the CPU has to fetch it directly from RAM.
3. The cache controller loads a line (typically 64 bytes) starting at address "a" into the memory cache. This is more data than the CPU requested, so if the program continues to run sequentially (i.e. asks for address a+1) the next instruction/data the CPU will ask will be already loaded in the memory cache.
4. A circuit called prefetcher loads more data located after this line, i.e. starts loading the contents from address a+64 on into the cache. To give you a real example, Pentium 4 CPUs have a 256-byte prefetcher, so it loads the next 256 bytes after the line already loaded into the cache.

If programs always run sequentially the CPU would never need to fetch data directly from the RAM memory – except to load the very first instruction – as the instructions and data required by the CPU would always be inside the memory cache before the CPU would ask for them.

However programs do not run like this, from time to time they jump from one memory position to

another. The main challenge of the cache controller is trying to guess what address the CPU will jump, loading the content of this address into the memory cache before the CPU asks for it in order to avoid the CPU having to go to the RAM memory, what slows the system down. This task is called branch predicting and all modern CPUs have this feature.

Modern CPUs have a hit rate of at least 80%, meaning that at least 80% of the time the CPU isn't accessing the RAM memory directly, but the memory cache instead.

Originally at <http://www.hardwaresecrets.com/article/481/6> Pages (9): [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) » ... [Last](#) »

© 2004-6, Hardware Secrets, LLC. All Rights Reserved.

Total or partial reproduction of the contents of this site, as well as that of the texts available for downloading, be this in the electronic media, in print, or any other form of distribution, is expressly forbidden. Those who do not comply with these copyright laws will be indicted and punished according to the International Copyrights Law.

We do not take responsibility for material damage of any kind caused by the use of information contained in Hardware Secrets.

How The Memory Cache Works

By Gabriel Torres on September 12, 2007

Page 7 of 9

Computer Cache Memory

Check our Selected Sites on Computer Cache Memory
www.SelectaSites.com

CPU Waterblocks in the UK

Unbeatable prices on Watercooling
 Largest range of cooling systems
www.SpecialTech.co.uk

Dynamic Data Cache

MySQL Embedded Server in
 Network Management
www.mysql.com

Ram Stress Test

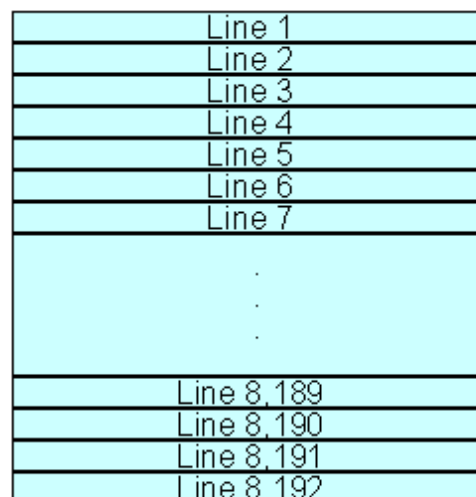
Signs, Symptoms & Treatments.
 Get Free Health Information
 Online.
www.healthline.com

Ads by Google

Memory Cache Organization

The memory cache is divided internally into lines, each one holding from 16 to 128 bytes, depending on the CPU. On the majority of current CPUs the memory cache is organized in 64-byte lines (512 bits), so we will be considering a memory cache using 64-byte lines in our examples throughout this tutorial. On the last page we will present the main memory cache specs for all CPUs currently found on the market.

So a 512 KB L2 memory cache is divided into 8,192 lines. Keep in mind that 1 KB is 2^{10} or 1,024 bytes and not 1,000 bytes, so the math is $524,288 / 64 = 8,192$. We will be considering a single-core CPU with 512 KB L2 memory cache in our examples. On Figure 5 we illustrate this memory cache internal organization.



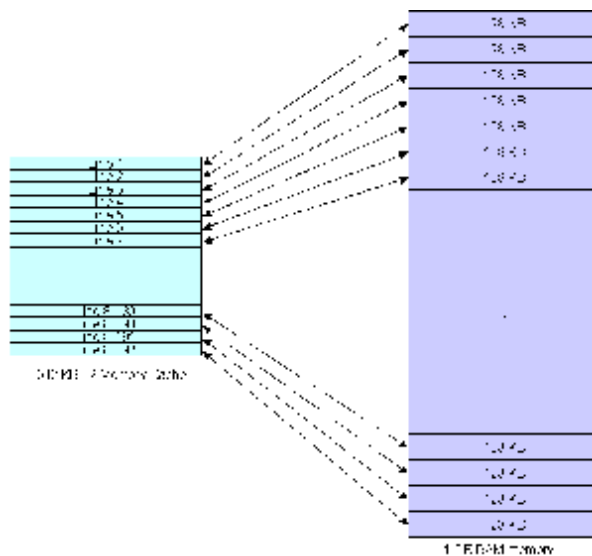
512 KB L2 Memory Cache

Figure 5: How a 512 KB L2 memory cache is organized.

The memory cache can work under three different configurations: direct mapping, fully associative and set associative (a.k.a. n-way set associative). The later is the most used configuration nowadays, but let's take a look on how these three configurations work.

Direct Mapping

Direct mapping is the simplest way of creating a memory cache. In this configuration the main RAM memory is divided into the same number of lines there are inside the memory cache. If we have a system with 1 GB of RAM, this 1 GB will be divided into 8,192 blocks (assuming the memory cache uses the configuration we describe above), each one with 128 KB ($1,073,741,824 / 8,192 = 131,072$ – keep in mind that 1 GB is 2^{30} bytes, 1 MB is 2^{20} bytes and 1 KB is 2^{10} bytes). If our system had 512 MB the memory would be also divided into 8,192 blocks, but this time each one would have 64 KB. And so on. We illustrate this organization on Figure 6.



click to enlarge

Figure 6: How direct mapping cache works.

The main advantage of direct mapping is that it is the easiest configuration to implement.

When the CPU asks for a given address from the RAM memory (e.g. address 1,000), the cache controller will load a line (64 bytes) from the RAM memory and store this line on the memory cache (i.e. addresses 1,000 thru 1,063, assuming we are using 8-bit addressing scheme just to help our examples). So if the CPU asks again the contents of this address or of the next few addresses right after this address (i.e. any address from 1,000 to 1,063) they will be already inside the cache.

The problem is that if the CPU needs two addresses that are mapped to the same cache line, a cache miss will occur (this problem is called collision or conflict). Continuing our example, if the CPU asks address 1,000 and then asks address 2,000, a cache miss will occur because these two addresses are inside the same block (the first 128 KB), and what was inside the cache was a line starting from address 1,000. So the cache controller will load the line from address 2,000 and store it on the first line of the memory cache, cleaning the previous contents, in our case the line from address 1,000.

The problem goes on. If the program has a loop that is more than 64 bytes long, there will be a cache miss for the entire duration of the loop.

For example, if the loop goes from address 1,000 to address 1,100, the CPU will have to load all instructions directly from the RAM memory thru the duration of the loop. This will happen because the cache will have the contents from addresses 1,000 thru 1,063 and when the CPU asks for the contents from address 1,100 it will have to go the RAM memory, and the cache controller will load addresses 1,100 thru 1,163. When the CPU asks address 1,000 back it will have to go back to the RAM memory, as the cache doesn't have the contents from address 1,000 anymore. If this loop is executed 1,000 times, the CPU will have to go to the RAM memory 1,000 times.

That is why direct mapping cache is the least efficient cache configuration and not used anymore – at least on PCs.

Fully associative

On fully associative configuration, on the other hand, there is no hard linking between the lines of the memory cache and the RAM memory locations. The cache controller can store any address. Thus the problems described above don't happen. This configuration is the most efficient one (i.e. the one with the highest hit rate).

On the other hand, the control circuit is far more complex, as it needs to keep track of what memory locations are loaded inside the memory cache. That is why a hybrid solution – called set associative – is the most used one nowadays.

Originally at <http://www.hardwaresecrets.com/article/481/7> Pages (9): [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) » ... [Last](#) »

How The Memory Cache Works

By Gabriel Torres on September 12, 2007

Page 8 of 9

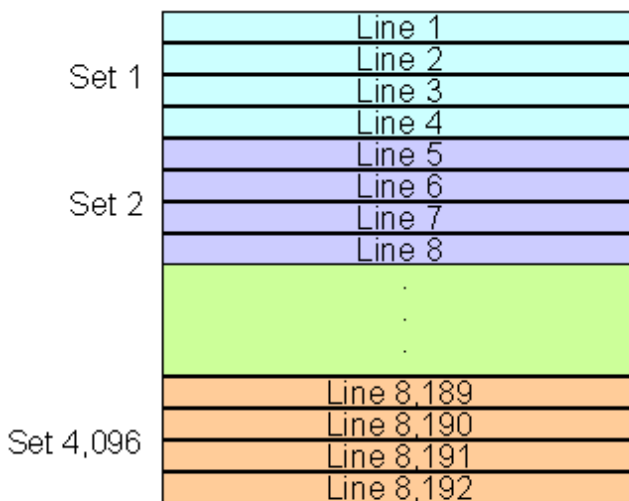
| | | | |
|--|--|--|---|
| <p>Computer Cache Memory Check our Selected Sites on Computer Cache Memory www.SelectaSites.com</p> | <p>Memory Upgrade → Absolutely Guaranteed Compatible! Fast Shipping, 30 Days Money Back. www.MemorySuppliers.com</p> | <p>RAMCHECK Memory Tester Test DDR2, DDR memory in seconds. For PC manufacturers and PC techs. www.memorytesters.com</p> | <p>Dynamic Data Cache MySQL Embedded Server in Network Management www.mysql.com</p> |
|--|--|--|---|



n-Way Set Associative Cache

On this configuration the memory cache is divided in several blocks (sets) containing "n" lines each.

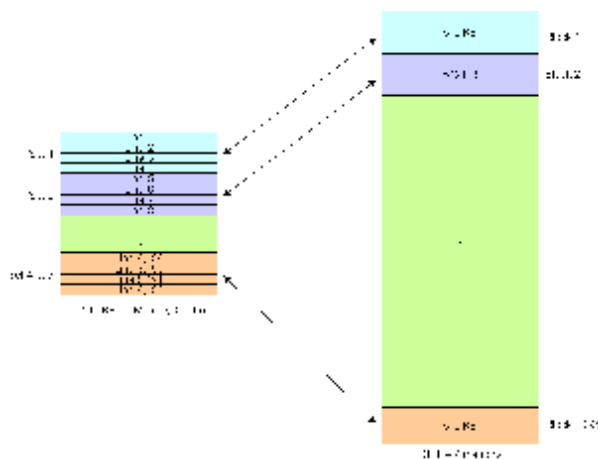
So on a 4-way set associative cache the memory cache will have 2,048 blocks containing four lines each (8,192 lines / 4), on a 2-way set associative cache the memory cache will have 4,096 blocks containing 2 lines each and on a 16-way set associative cache the memory cache will have 512 blocks containing 16 lines each. Here we are continuing our example of a 512 KB L2 memory cache divided into 8,192 64-byte lines. Depending on the CPU the number of blocks will be different, of course.



512 KB L2 Memory Cache

Figure 7: 512 KB L2 memory cache configured as 4-way set associative.

Then the main RAM memory is divided in the same number of blocks available in the memory cache. Keeping the 512 KB 4-way set associative example, the main RAM would be divided into 2,048 blocks, the same number of blocks available inside the memory cache. Each memory block is linked to a set of lines inside the cache, just like in the direct mapped cache. With 1 GB RAM, the memory would be divided into 2,048 blocks with 512 KB each, see Figure 8.



click to enlarge

Figure 8: 512 KB L2 memory cache configured as 4-way set associative.

As you see the mapping is very similar to what happens with the direct mapped cache, the difference is that for each memory block there is now more than one line available on the memory cache. Each line can hold the contents from any address inside the mapped block. On a 4-way set associative cache each set on the memory cache can hold up to four lines from the same memory block.

With this approach the problems presented by the direct mapped cache are gone (both the collision problem and the loop problem we describe on the previous page). At the same time, the set associative cache is easier to implement than the full associative cache, since its control logic is simpler. Because of that this is nowadays the most common cache configuration, even though it provides a lower performance compared to the full associative one.

Of course we still have a limited number of available slots inside each memory cache set for each memory block – four on a 4-way configuration. After these four slots are taken, the cache controller will have to free one of them to store the next instruction loaded from the same memory block.

When we increase the number of ways a set associative memory cache has – for example, from 4-way to 8-way configuration –, we have more slots available on each set, but if we keep the same amount of cache memory the size of each memory block is also increased. Continuing our example, moving from 4-way to 8-way would make our 1 GB RAM memory to be divided into 1,024 1 MB blocks. So this move would increase the number of available slots on each set, but now each set would be in charge of a bigger memory block.

There is a lot of academic discussion regarding what is the perfect balance between the number of sets and the memory block size and there is no definitive answer – Intel and AMD use different configurations, as you will see on next page.

So what happens if we have a bigger memory cache? Keeping the above example, if we increased the L2 memory cache from 512 KB to 1 MB (the only way to do that would be by replacing the CPU), what would happen is that we would have 16,384 64-byte lines in our memory cache, what would give us 4,096 sets with four lines each. Our 1 GB RAM memory would be divided into 4,096 256 KB blocks. So basically what happens is that the size of each memory block is lowered, increasing the chance of the requested data to be inside the memory cache – in other words, increasing the cache size lowers the cache miss rate.

However, increasing the memory cache isn't something that guarantees increase in performance. Increasing the size of the memory cache assures that more data will be cached, but the question is whether the CPU is using this extra data or not. For example, suppose a single-core CPU with 4 MB L2 cache. If the CPU is using heavily 1 MB but not so heavily the other 3 MB (i.e. the most accessed instructions are taking up 1 MB and on the other 3 MB the CPU cached instructions are not being called so much), chance is that this CPU will have a similar performance of an identical CPU but with 2 MB or even 1 MB L2 cache.

Originally at <http://www.hardwaresecrets.com/article/481/8> Pages (9): [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) » ... [Last](#) »

© 2004-6, Hardware Secrets, LLC. All Rights Reserved.

Total or partial reproduction of the contents of this site, as well as that of the texts available for downloading, be this in the electronic media, in print, or any other form of distribution, is expressly forbidden. Those who do not comply with these copyright laws will be indicted and punished according to the International Copyrights Law.

We do not take responsibility for material damage of any kind caused by the use of information contained in Hardware Secrets.

How The Memory Cache Works

By Gabriel Torres on September 12, 2007

Page 9 of 9

Free Component Searchbox

Meta-Search 1,100 Manufacturers - Add to Your Website in Seconds.

SupplyFrame.com

Ads by Google

Memory Cache Configuration on Current CPUs

Below we present you a reference table containing the main memory cache specs for the main CPUs available on the market today.

| CPU | L1 Instruction | L1 Data | L2 |
|-------------------------------|--|--|---|
| Athlon 64 | 64 KB 2-way set associative 64-byte lines 128-bit datapath with L2 128-bit datapath with fetch unit | 64 KB 2-way set associative 64-byte lines 128-bit datapath with L2 | 512 KB or 1 MB 16-way set associative 64-byte lines 128-bit datapath with L1 data 128-bit datapath with L1 instruction |
| Athlon 64 FX | 64 KB per core 2-way set associative 64-byte lines 128-bit datapath with L2 128-bit datapath with fetch unit | 64 KB per core 2-way set associative 64-byte lines 128-bit datapath with L2 | 1 MB per core 16-way set associative 64-byte lines 128-bit datapath with L1 data 128-bit datapath with L1 instruction |
| Athlon 64 X2 | 64 KB per core 2-way set associative 64-byte lines 128-bit datapath with L2 128-bit datapath with fetch unit | 64 KB per core 2-way set associative 64-byte lines 128-bit datapath with L2 | 512 KB or 1 MB per core 16-way set associative 64-byte lines 128-bit datapath with L1 data 128-bit datapath with L1 instruction |
| Sempron (sockets 754 and AM2) | 64 KB 2-way set associative 64-byte lines 128-bit datapath with L2 128-bit datapath with fetch unit | 64 KB 2-way set associative 64-byte lines 128-bit datapath with L2 | 128 KB or 256 KB 16-way set associative 64-byte lines 128-bit datapath with L1 data 128-bit datapath with L1 instruction |
| Opteron | 64 KB per core 2-way set associative 128-bit datapath with L2 128-bit datapath with fetch unit | 64 KB per core 2-way set associative 64-byte lines 128-bit datapath with L2 | 1 MB per core 16-way set associative 64-byte lines 128-bit datapath with L1 data 128-bit datapath with L1 instruction |
| Pentium 4 | N/A * | 8 KB 4-way set associative 64-byte lines 256-bit datapath with L2 | 256 KB, 512 MB or 1 MB 8-way set associative 128-byte lines 64-bit datapath with fetch |

| | | | |
|-------------------|--|---|--|
| | | | unit 256-bit datapath with L1 data |
| Pentium D | N/A * | 16 KB 4-way set associative 64-byte lines 256-bit datapath with L2 | 1 MB or 2 MB per core 8-way set associative 128-byte lines 64-bit datapath with fetch unit 256-bit datapath with L1 data |
| Core 2 Duo | 32 KB 64-byte lines 256-bit datapath with fetch unit | 32 KB 64-byte lines 256-bit datapath with L2 | 2 MB or 4 MB 8-way set associative 64-byte lines 256-bit datapath with L1 data |
| Pentium Dual Core | 32 KB 64-byte lines 256-bit datapath with fetch unit | 32 KB 64-byte lines 256-bit datapath with L2 | 1 MB 8-way set associative 64-byte lines 256-bit datapath with L1 data |

* There is a 150 KB trace cache on these processors. This cache is located between the decoder unit and the execution unit. Thus the fetch unit grabs data directly from L2 memory cache.

We didn't include Xeon and Celeron processors on the above table because there are several different Xeon and Celeron models around based on different architectures. Celeron and Xeon based on Netburst microarchitecture (i.e. based on Pentium 4) will have the same specs as Pentium 4 but with different L2 cache size, while Celeron and Xeon based on Core microarchitecture (i.e. based on Core 2 Duo) will have the same specs as Core 2 Duo but with different L2 cache size.

Originally at <http://www.hardwaresecrets.com/article/481/9> Pages (9): [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) » ... [Last](#) »

© 2004-6, Hardware Secrets, LLC. All Rights Reserved.

Total or partial reproduction of the contents of this site, as well as that of the texts available for downloading, be this in the electronic media, in print, or any other form of distribution, is expressly forbidden. Those who do not comply with these copyright laws will be indicted and punished according to the International Copyrights Law.

We do not take responsibility for material damage of any kind caused by the use of information contained in Hardware Secrets.