Understanding CPU Caching and Performance

Back when Ars first started, Intel had just released the first Celeron processor aimed at the low end market, and since it lacked an off-die backside cache like its cousin the PII it turned out to be *extremely* overclockable. The Celeron really pushed the overclocking craze into the mainstream in a big way, and Ars got its start by providing seats on the bandwagon to anyone with a web browser and a desire to learn the hows and whys of Celeron overclocking. Frank Monroe's Celeron Overclocking FAQ was one of the most relentlessly popular articles on Ars for what seemed like forever, and "Celeron" and "overclocking" were the two main search terms that brought people in from Yahoo, which at the time was our number one referrer. (In fact, some time ago my girlfriend mentioned that she had actually come across Ars via Yahoo and read the OC FAQ years before she and I ever even met.)

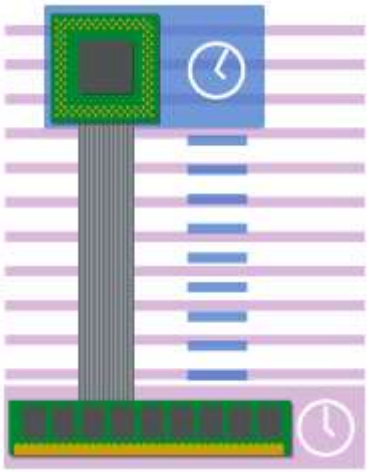
Along with its overclockability, there was one peculiar feature of the "cacheless wonder," as the Celeron was then called, that blew everyone's mind: it performed almost as well on Quake benchmarks as the cache-endowed PII. What became evident in the ensuing round of newsgroup and BBS speculation over this phenomenon was that few people actually understood *how* caching works to improve performance. My suspicion is that this situation hasn't changed a whole lot since the Celeron's heyday. However, what has changed since then is the relative importance of caching in system design. Despite the introduction of RAMBUS, DDR, and other next-gen memory technologies, CPU clockspeed and performance have grown significantly faster than main memory performance. As a result L1, L2, and even L3 caches have become a major factor in preventing relatively slow RAM from holding back overall system performance by failing to feed code and data to the CPU at a high enough rate.

The current article is intended as a general introduction to CPU caching and performance. The article covers fundamental cache concepts like spatial and temporal locality, set associativity, how different types of applications use the cache, the general layout and function of the memory hierarchy, et cetera, et cetera. Building on all of this, the next installment will address real-world examples of caching and memory subsystems in Intel's P4 and Motorola's G4e-based systems. (I hope to include some discussion of the XServe hardware, so stay tuned.) But for those who've wondered why cache size matters more in some applications than in others, or what people mean when they talk about "tag RAM," then this article is for you.

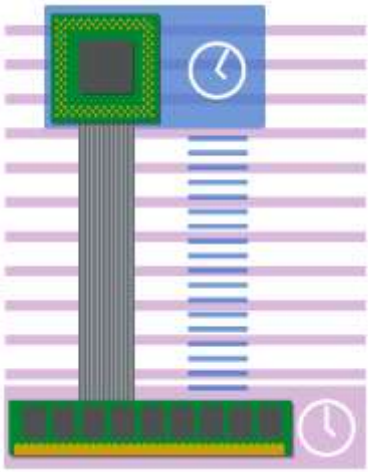
Caching basics

In order to really understand the role of caching in system design, it helps to think of the CPU and memory subsystem as operating on a consumer-producer (or client-server) model: the CPU consumes information provided to it by the hard disks and RAM, which act as producers. Driven by innovations in process technology and processor design, CPUs have increased their ability to consume at a significantly higher rate than the memory subsystem has increased its ability to produce. The problem is that CPU clock cycles have gotten shorter at a faster rate than memory

and bus clock cycles, so the number of CPU clock cycles that the processor has to wait before main memory can fulfill its requests for data has increased. So with each CPU clockspeed increase, memory is getting further and further away from the CPU in terms of the number of CPU clock cycles.



Slower CPU Clock



Faster CPU Clock

To visualize the effect that this widening speed gap has on overall system performance, imagine the CPU as a downtown furniture maker's workshop and the main memory as a lumberyard that keeps getting moved further and further out into the suburbs. Even if we start using bigger trucks to cart all the wood, it's still going to take longer from the time the workshop places an order to the time that order gets filled.

Note: I'm not the first person to use a workshop and warehouse analogy to explain caching. The most famous example of such an analogy is the Thing King game, which I first saw in this book by Peter van der Linden (reviewed).

Sticking with our furniture workshop analogy, one solution to this problem would be to rent out a smaller warehouse in-town and store the most recently requested types of lumber, there. This smaller, closer warehouse would act as a cache for the workshop, and we could keep a driver on-hand who could run out at a moment's notice and quickly pick up whatever we need from the warehouse. Of course, the bigger our warehouse the better, because it allows us to store more types of wood, thereby increasing the likelihood that the raw materials for any particular order will be on-hand when we need them. In the event that we need a type of wood that isn't in the closer warehouse, we'll have to drive all the way out of town to get it from our big, suburban warehouse. This is bad news, because unless our furniture workers have another task to work on while they're waiting for our driver to return with the lumber, they're going to sit around in the break room smoking and watching Oprah. And we *hate* paying people to watch Oprah.

2. The memory hierarchy.

I'm sure you've figured it out already, but the warehouse in our analogy is the level 1 (or L1) cache. The L1 can be accessed very quickly by the CPU, so it's a good place to keep the code and data that the CPU is most likely to request, next. (In a moment, we'll talk in more detail about how the L1 can "predict" what the CPU will probably want.) The L1's quick access time is a result of the fact that it's made of the fastest and most expensive type of static RAM, or SRAM. Since each SRAM memory cell is made up of four to six transistors (compared to the one-transistor-per-cell configuration of DRAM) its cost-per-bit is quite high. This high cost-per-bit means that we generally can't afford to have a very large L1 cache unless we really want to drive up the total cost of the system.

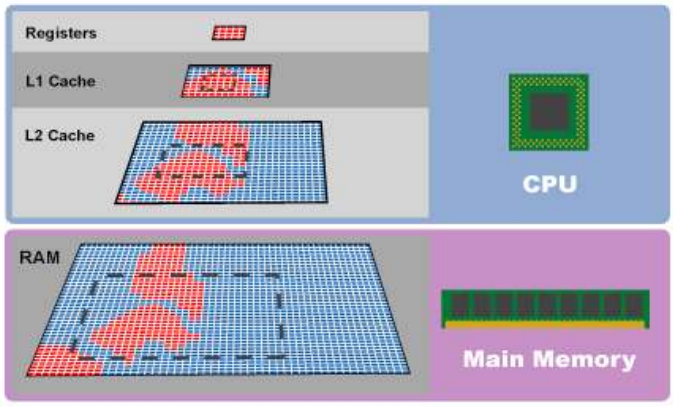
In most current CPUs the L1 sits on the same piece of silicon as the rest of the processor. In terms of the warehouse analogy, this is a bit like having the warehouse on the same block as the workshop. This has the advantage of giving the CPU some very fast, very close storage, but the disadvantage is that now the main memory (our suburban warehouse) is just as far away from the L1 cache as it is from the processor. So if some data that the CPU needs isn't in the L1, a situation called a **cache miss**, it's going to take quite a while to retrieve that data from memory. Furthermore, remember that as the processor gets faster, the main memory gets "further" away all the time. So while our warehouse may be on the same block as our workshop, the lumber yard has now moved not just out of town but out of the state. For an ultra-high-clockrate processor like the P4, being forced to wait for data to load from main memory in order to complete an operation is like our workshop having to wait a few days for lumber to ship in from out of state.

Check out the table below, which shows common latency and size information for the various levels of the memory hierarchy. (These numbers are a bit old. The current numbers are smaller and shrinking all the time). Notice the large gap in access times between the L1 cache and the main memory. For a 1 GHz CPU a 50 ns wait means 50 wasted clock cycles. Ouch. To see the kind of effect such stalls have on a modern, hyperpipelined processor, check out Part I of my P4 vs. G4e comparison.

The solution to this dilemma is to add more cache. At first you might think we could get more cache by enlarging the L1, but as I said above the cost considerations are a major factor limiting L1 cache size. In terms of our workshop analogy, we could say that rents are much higher in-town than in the suburbs, so we can't afford much warehouse space without the cost of rent eating into our bottom line to the point where the added costs of the warehouse space would outweigh the benefits of increased worker productivity. So we have to fine tune the amount of warehouse space that we rent by weighing all the costs and benefits so that we get the maximum output for the least cost.

A better solution than adding more L1 would be to rent some cheaper, larger warehouse space right outside of town to act as a cache for the L1 cache. Hence processors like the P4 and G4e have a Level 2 (L2) cache that sits *between* the L1 cache and main memory. The L2 cache usually contains all of the data that's in the L1 plus, some extra. The common way to describe this situation is to say that the L1 cache **subsets** the L2 cache, because the L1 contains a subset of the data in the

L2. In the picture below, the red cells are the code and data for the process that the CPU is currently running. The blue cells are unrelated to the currently active process. (This diagram will become more clear once we talk about locality of reference.)



This series of caches, starting with the page file on the hard disk (the lumber yard in our analogy) and going all the way up to the registers on the CPU (the workshop's workbenches), is called a *cache hierarchy*. As we go up the cache hierarchy towards the CPU, the caches get smaller, faster, and more expensive to implement; conversely, as we go down the cache hierarchy they get larger, cheaper, and much slower. The data contained in each level of the hierarchy is usually mirrored in the level below it, so that for a piece of data that's in the L1 cache there are usually copies of that same data in the L2, main memory, and page file. So each level in the hierarchy subsets the level below it. We'll talk later about how all of those copies are kept in sync.

Level	Access Time	Typical Size	Technology	Managed By
Registers	1-3 ns	1 KB	Custom CMOS	Compiler
Level 1 Cache (on-chip)	2-8 ns	8 KB-128 KB	SRAM	Hardware
Level 2 Cache (off-chip)	5-12 ns	0.5 MB - 8 MB	SRAM	Hardware
Main Memory	10-60 ns	64 MB - 1 GB	DRAM	Operating System
Hard Disk	3,000,000 - 10,000,000 ns	20 - 100 GB	Magnetic	Operating System/User

As you can see from the above table, each level of the hierarchy is controlled by a different part of the system. Data is promoted up the hierarchy or demoted down the hierarchy based on a number of different criteria, but in the remainder of this article we'll only concern ourselves with the top levels of the hierarchy. Also, again note that these numbers are kind of old. I cobbled them together from a chart I found on the web (check the bibliography) and the Hennessy and Patterson book. More detailed numbers for P4 and G4e-based systems will be forthcoming in the next installment.

3. Locality.

Example: A Byte's Brief Journey Through the Memory Hierarchy

For the sake of example, let's say the CPU issues a LOAD instruction that tells the memory subsystem to load a piece of data (in this case, a single byte) into one of its registers. First, the request goes out to the L1 cache, which is checked to see if it contains the requested data. If the L1 cache does not contain the data and therefore cannot fulfill the request--a situation called a **cache miss**--then the request propagates down to the L2 cache. If the L2 cache does not contain the desired byte, then the request begins the relatively long trip out to main memory. If main memory doesn't contain the data, then we're in big trouble, because then it has to be paged in from the hard disk, an act which can take a relative eternity in CPU time.

Let's assume that the requested byte is found in main memory. Once located, the byte is copied from main memory, along with a bunch of its neighboring bytes in the form of a **cache block** or **cache line**, into the L2 and L1 caches. When the CPU requests this same byte again it will be waiting for it there in the L1 cache, a situation called a **cache hit**.

Computer architects usually divide misses up into three different types depending on the situation that brought about the miss. I'll introduce these three types of misses at appropriate points over the course of the article, but I can talk about the first one right now. A **compulsory miss** is a cache miss that occurs because the desired data was never in the cache and therefore must be paged in for the first time in a program's execution. It's called a "compulsory" miss because, barring the use of certain specialized tricks like data prefetching, it's the one type of miss that just can't be avoided. All cached data must be brought into the cache for the very first time at some point, and the occasion for doing so is normally a compulsory miss.

The two other types of misses are misses that result when the CPU requests data that was previously in the cache but has been **evicted** for some reason or other. We'll discuss evictions later.

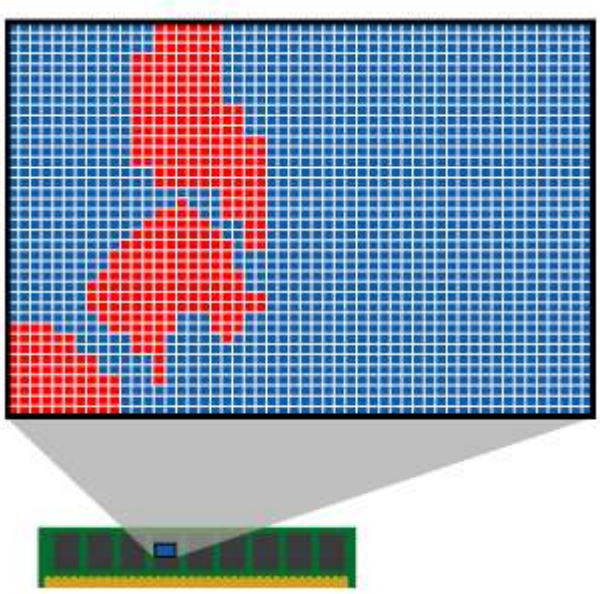
How different applications use the cache

There's one very simple principle that's basic to how caches work: *locality of reference*. We generally find it useful to talk about two types of locality of reference: spatial locality and temporal locality. Spatial locality is a fancy way of labeling the general rule that if the CPU needs an item from memory at any given moment, it's likely to need its neighbors, next. Temporal locality is the name we give to the general rule that if an item in memory was accessed once, it's likely to be accessed again in the near future. Depending on the type of application, both code and data streams can exhibit spatial and temporal locality.

Spatial locality

The concept of locality of reference is probably easy enough to understand without

much explanation, but just in case it's not immediately clear let's take a moment to look closer. Spatial locality is the easiest type of locality to understand, because most of us have used media applications like mp3 players, DVD players, and other types of apps whose datasets consist of large, ordered files. Consider an MP3 file, which has a bunch of blocks of data that are consumed by the processor in sequence from the file's beginning to its end. If the CPU is running Winamp and it has just requested second 1:23 of a 5 minute MP3, then you can be reasonably certain that next it's going to want seconds 1:24, 1:25, and so on. This is the same with a DVD file, and with many other types of media files like images, Autocad drawings, and Quake levels. All of these applications operate on large arrays of sequentially ordered data that gets ground through in sequence by the CPU again and again.



In the picture above, the red cells are related chunks of data in the memory array. This picture shows a program with fairly good spatial locality, since the red cells are clumped closely together. In an application with poor spatial locality, the red cells would be randomly distributed among the unrelated blue cells.

4. Temporal locality.

For data, business apps like word processors often have great spatial locality. If you think about it, few people open six or seven documents in a word processor and quickly alternate between them typing one or two words in each. Most of us just open up one or two relatively modest sized files and work in them for a while without skipping around too much in the same file. So spatial locality is just a way of saying that related chunks of data tend to clump together in memory, and since they're related they also tend to be processed together in batches by the CPU.

Spatial locality also applies to code just like it does to data, since most well-written code tries to avoid jumps and branches so that the processor can execute through large contiguous blocks uninterrupted. Games, simulations, and media processing applications tend to have decent spatial locality for code, since such applications often feature small blocks of code operating repeatedly on very large datasets.

When it comes to spatial locality of code for business apps, the picture is more mixed. If you think about the way that you use a Word processor, it's easy to see what's going on. As we create a document most of us are constantly pressing different formatting buttons and invoking different menu options. For instance, we might format one word in italics, change the paragraph spacing, and then save the file, all in sequence. Each of these actions invokes a very different part of the code in a large application like MS Word; it's not too likely that the code for the File->Save menu option is stored right next to the code that formats a font in italics. So the way that we use a word processor forces the CPU to jump around from place to place in memory to retrieve the correct code. However, the code for each individual action (i.e. saving a file, formatting a font, etc.) usually hangs together in a fairly large, spatially localized chunk--very much like a little sub-program within the larger application. So while the code for the File->Save action might be quite far away from the code for the italics formatting option, both of these chunks of code exhibit good spatial locality as small programs in their own right.

What this means for a cache designer is that business apps need large caches to be able to collect up all of the most frequently used clumps of code that correspond to the most frequently executed actions and pack them together in the cache. If the cache is too small, then the different clumps get swapped out as different actions are performed. If the cache is large enough, then all of these sub-programs fit and there's little swapping needed. Incidentally, this is why business apps performed so poorly on the cacheless Celeron.

Cache lines or "blocks"

Caches take advantage of spatial locality in two primary ways. First, when the CPU requests a particular piece of data from the memory subsystem, that piece gets fetched and loaded into the L1 cache along with some of its nearest neighbors. The actual piece of data that was requested is called the **critical word**, and the surrounding group of bytes that gets fetched along with it is called a **cache line** or **cache block**. So by fetching not only the critical word but also a group of its neighbors and loading them into the cache, the CPU is prepared to take advantage

of the fact that those neighboring bytes are the most likely to be the ones it will need to process next.

The other way that caches can take advantage of spatial locality is through a trick called **prefetching**. We'll talk about this more in a subsequent article.

Temporal locality

Consider a simple Photoshop filter that inverts an image to produce a negative; there's a small piece of code that performs the same inversion on each pixel starting at one corner and going in sequence all the way across and down to the opposite corner. This code is just a small loop that gets executed repeatedly on each pixel, so it's an example of code that is reused again and again. Media apps, games, and simulations, since they use lots of small loops that iterate through very large datasets, have excellent temporal locality for code.

However, it's important to note that these kinds of apps have extremely poor temporal locality for data. Returning to our MP3 example, a music file is usually played through once in sequence and none of its parts are repeated. This being the case, it's actually kind of a waste to store any of that file in the cache, since it's only going to stop off there temporarily before passing through to the CPU. When an app fills up the cache with data that doesn't really need to be cached because it won't be used again and as a result winds up bumping out of the cache data that will be reused, that app is said to "pollute the cache." Media apps, games, and the like are big cache polluters, which is why they weren't too affected by the original Celeron's lack of cache. Because they were streaming data through the CPU at a very fast rate, they didn't actually even care that their data wasn't being cached. Since this data wasn't going to be needed again anytime soon, the fact that it wasn't in a readily accessible cache didn't really matter.

The primary way in which caches take advantage of temporal locality is probably obvious by this point: caches provide a place to store code and data that the CPU is currently working with. By "working with," I mean that the CPU has used it once and is likely to use it again. A group of related blocks of code and/or data that the CPU uses and reuses over a period of time in order to complete a task is called a **working set**. Depending on the size of a task's working set and the number of operations that it takes the CPU to complete that task, spatial and temporal locality--and with them the cache hierarchy--will afford a greater or lesser performance increase on that task.

The other way that caches can benefit from temporal locality is by implementing an intelligent **replacement policy**. But we'll talk more about this after the next section.

5. Associativity

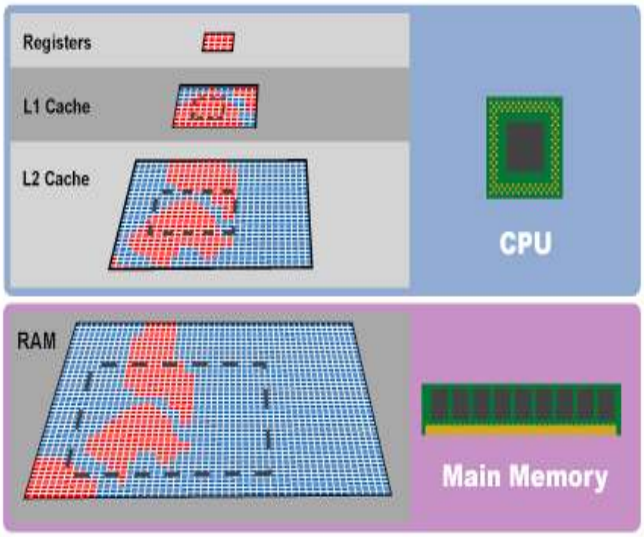
Locality: Conclusions

One point that should be apparent from the preceding discussion is that temporal locality implies spatial locality, but not vice versa. That is to say, data that is reused is always related (and therefore collects into spatially localized clumps in memory), but data that is related is not always reused. An open text file is an example of reusable data that occupies a localized region of memory, and an MP3 file is an example of non-reusable (or streaming) data that also occupies a localized region of memory.

The other thing that you probably noticed from the above section is the fact that memory access patterns for code and memory access patterns for data are often very different within the same application. For instance, media applications had excellent temporal locality for code but poor temporal locality for data. This fact has inspired many cache designers to split the L1 cache into two regions, one for code and one for data. The code half of the cache is called the **instruction cache**, or **i-cache**, while the data half of the cache is called the **data cache**, or **d-cache**. This partitioning can result in significant performance gains, depending on the size of the cache, the types of applications normally run on the system, and a variety of other factors.

Cache organization: block frames and blocks

As I mentioned earlier, caches fetch data in chunks called blocks (or "lines"), and each of these blocks fits into a special slot called a "block frame." The blocks form the basic unit of cache organization. RAM is also organized into blocks of the same size as the cache's blocks, and cache designers can choose from a few different schemes for governing which RAM blocks can be stored in which of the cache's block frames. Such a scheme is called a **cache placement policy** because it dictates where in the cache a block from memory can be placed.



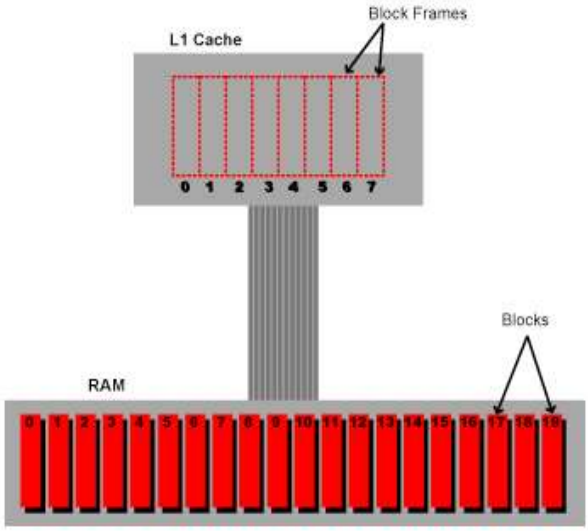
So a RAM block is fetched into the cache and stored in one of the block frames.

When the CPU requests a byte from a particular RAM block, it needs to be able to determine three things very quickly: 1. whether or not the needed block is actually in the cache (i.e. whether there is *cache hit* or *cache miss*); 2. the location of the block within the cache (in the case of a cache hit); and 3. the location of the desired byte within the block (again, in the case of a cache hit). The caches accommodate all three needs by associating a special piece of memory, called a tag, with each block frame in the cache. The tag fields allow the CPU to determine the answer to all three of these questions, but just how quickly that answer comes depends on a variety of factors.

Tags are stored in a special type of memory called the "tag RAM." This memory has to be made of very fast SRAM because it can take some time to search it for the location of the desired block. The larger the cache the greater the number of blocks, and the greater the number of blocks the more tag RAM you need and the longer it can take to find the correct block. Thus the tag RAM can add unwanted latency to the cache. As a result, we not only have to use the fastest RAM available for the tag RAM, but we also have to be smart about how we use tags to map RAM blocks to block frames. In the section below, I'll introduce the three general options for doing such mapping, and I'll discuss some of the pros and cons of each option.

Fully associative, n-way, and direct-mapped caches

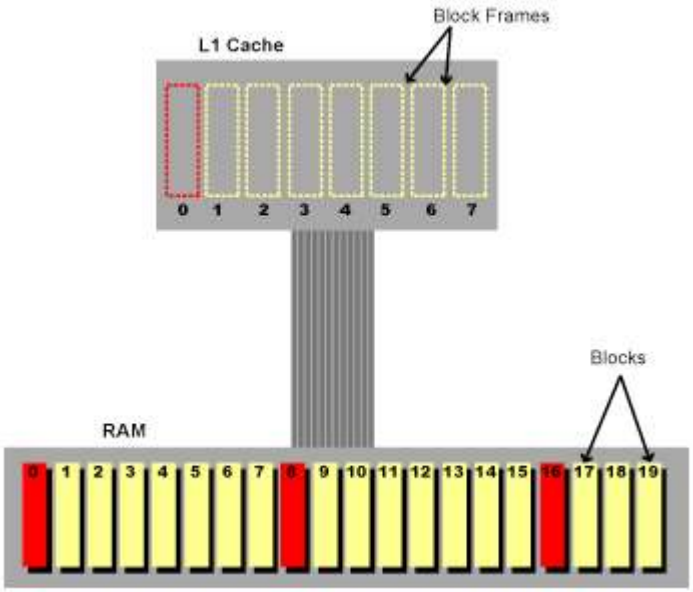
The most conceptually simple scheme for mapping RAM blocks to cache block frames is called "fully associative" mapping. Under this scheme, any RAM block can be stored in any available block frame.



The problem with this scheme is that if you want to retrieve a specific block from the cache, you have to check the tag of every single block frame in the *entire* cache because the desired block could be in any of the frames. Since large caches can have thousands of block frames, this tag searching can add considerable delay (latency) to a fetch. Furthermore, the larger the cache the worse the delay gets, since there are more block frames and hence more block tags to check on each fetch.

6. More associativity.

Another, more popular way of organizing the cache is to use a "direct mapping." In a direct-mapped cache, each block frame can cache only a certain subset of the blocks in main memory.

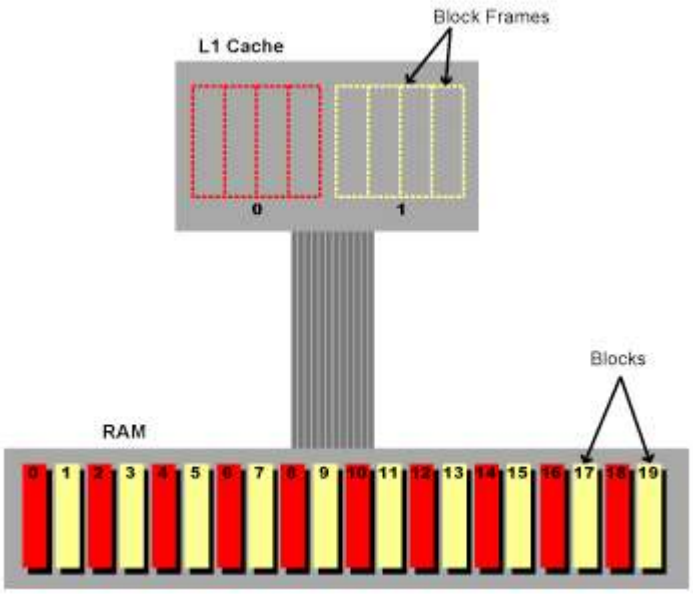


In the above diagram, each of the red blocks (blocks 0, 8, and 16) can be cached only in the red block frame (frame 0). Likewise, blocks 1, 9, and 17 can be cached only in frame 1, blocks 2, 10, and 18 can be cached only in frame 2, and so on. Hopefully, the pattern here is apparent: each frame caches every eighth block of main memory. As a result, the potential number of locations for any one block is greatly narrowed, and therefore the number of tags that must be checked on each fetch is greatly reduced. So, for example, if the CPU needs a byte from blocks 0, 8, or 16, it knows that it only has to check frame 0 to determine if the desired block is in the cache and to retrieve it if it is. This is much faster and more efficient than checking every frame in the cache.

There are some drawbacks to this scheme, though. For instance, what if blocks 0-3 and 8-11 combine to form an 8-block "working set" that the CPU wants to load into the cache and work on for a while. The cache is 8 blocks long, but since it's direct-mapped it can only store four of these particular blocks at a time. Remember, blocks 0 and 8 have to go in the same frame, as do blocks 1 and 9, 2 and 10, and 3 and 11. As a result, the CPU must load only four blocks of this 8-block set at a time, and swap them out as it works on the set. If the CPU wants to work on this set 8-block set for a long time, then that could mean a lot of swapping. Meanwhile, half of the cache is going completely unused! So while direct-mapped caches are almost always faster than fully associative caches due to the shortened amount of time it takes to locate a cached block, they can still be inefficient under some circumstances.

Note that the kind of situation described above, where the CPU would like to store multiple blocks but it can't because they all require the same frame, is called a **collision**. In the preceding example, blocks 0 and 8 are said to collide, since they both want to fit into frame 0 but can't. Misses that result from such collisions are called **conflict misses**, the second of the three types of cache miss that I mentioned earlier.

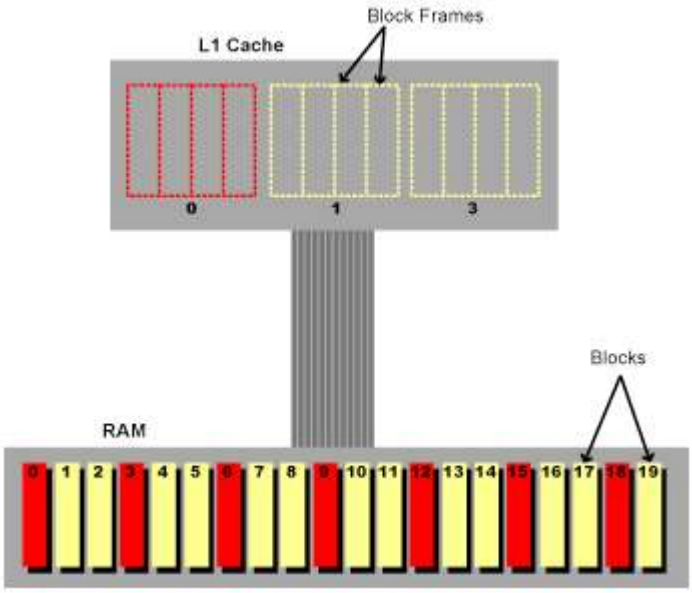
One way to get some of the benefits of direct-mapped caches while lessening the amount of wasted cache space due to collisions is to restrict the caching of main memory blocks to a subset of the available cache frames. To see what I mean, take a look at the diagram below, which represents a four-way associative cache.



In this diagram, any of the red blocks can go anywhere in the red set of frames (set 0) and any of the light yellow blocks can go anywhere in the light yellow set of frames (set 1). You can think of it like this: we took a fully associative cache and cut it in two, restricting half the main memory blocks to one side and half the main memory blocks to the other. This way, the odds of a collision are greatly reduced versus the direct-mapped cache, but we still don't have to search all the tags on every fetch like we did with the fully associative cache. For any given fetch, we need search only a single, four-block set to find the block we're looking for, which in this case amounts to half the cache.

The cache pictured above is said to be "four-way associative" because the cache is divided into "sets" of four frames each. Since the above cache has only 8 frames, then it can accommodate only two sets. A larger cache could accommodate more sets, reducing the odds of a collision even more. Furthermore, since all the sets consist of exactly four blocks, no matter how big the cache gets we'll only ever have to search through four frames (or one set) to find any given block. This means that as the cache gets larger and the number of sets it can accommodate increases, the more efficient the tag searches get. Think about it. In a cache with three sets, only 1/3 of the cache (or one set) needs to be searched for a given block. In a cache with four sets, only 1/4 of the cache is searched. In a cache with 100, four-block sets,

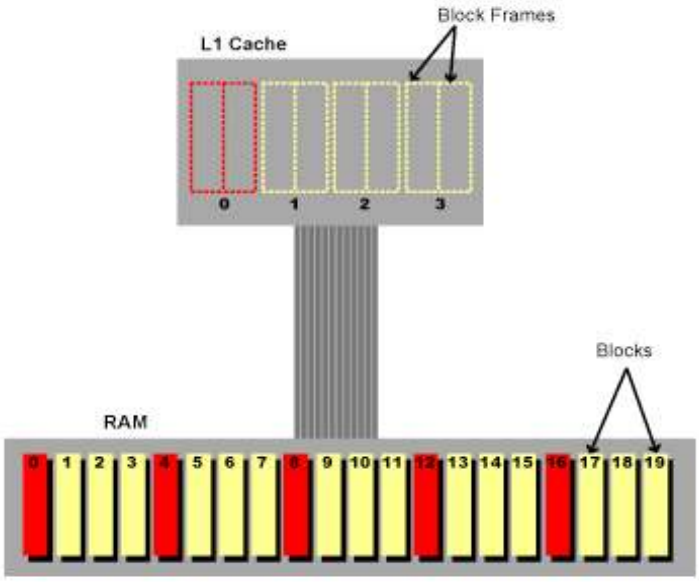
only 1/100 of the cache needs to be searched. So the relative search efficiency scales with the cache size.



Furthermore, if we increase the number of sets in the cache while keeping the amount of main memory constant, then we can decrease the odds of a collision even more. Notice that in the above diagram there are fewer red main memory blocks competing for space in set 0.

7. Eviction Policies.

Another way to increase the number of sets in the cache without actually increasing the cache size is to reduce the number of blocks in each set. Check out the diagram below, which shows a two-way set associative cache.



To get a better idea of what's going on, let's briefly compare the two-way associative cache to both the direct-mapped and the four-way cache. For the sake of comparison, let's assume that the cache size and the memory size both stay constant. And as you read the comparison, keep in mind that since each increase in the level of associativity (i.e. from two-way to four-way, or from four-way to eight-way) also increases the number of tags that must be checked in order to locate a specific block, an increase in associativity also means an increase in the cache's latency.

Two-way vs. direct-mapped

With the two-way cache, the number of potential collisions (and hence the miss rate) is reduced versus the direct-mapped scheme. However, the number of tags that must be searched for each fetch is twice as high. Depending on the relative sizes of the cache and main memory, this may or may not increase the cache's overall latency to the point that the decreased conflict miss rate is worth it.

Two-way vs. four-way

Though a two-way cache's latency is less than that of the four-way scheme, its number of potential collisions (and hence its miss rate) is higher than that of the four-way cache. Just as with the preceding comparison, how a two-way associative cache compares to a four-way associative cache depends on just how much latency the increase in associativity ends up costing versus the decrease in conflict miss rate.

Associativity: Conclusions

In general, it turns out that when you factor in current technological conditions (i.e. the speed of tag RAM, the range of sizes that most caches fall into, etc.) some level of associativity less than or equal to eight-way turn out to be optimal for most caches. Any more than eight-way associativity and the cache's latency is increased so much that the decrease in miss rate isn't worth it. Any less than two-way associativity and the number of collisions often increase the miss rate to the point that the decrease in latency isn't worth it. There are some direct-mapped caches out there, though.

Before I conclude the discussion of associativity, there are two minor bits of information that I should include for the sake of completeness. First, though you've probably already figured it out, a direct-mapped cache is simply a one-way associative cache and a fully associative cache is an *n*-way associative cache where *n* is equal to the total number of blocks in the cache.

The second thing you should know is the formula for computing which set in which an arbitrary block of memory should be stored in an *n*-way associative cache. From page 377 of Henessey and Patterson, the cache placement formula is as follows:

(*Block address*) MOD (*Number of sets in cache*)

I recommend trying this formula out on some of the examples above. It seems like it might be a boring and pointless exercise, but if you take 5 minutes and place a few blocks using this simple formula in conjunction with the preceding diagrams, every thing I've said in this section will really fall into place for you in a "big picture" kind of way. And it's actually more fun to do that it probably sounds.

Temporal and Spatial Locality Revisited: Replacement/Eviction Policies and Block Sizes

Replacement/Eviction policies

Caches can increase the amount of benefit they derive from temporal locality by implementing an intelligent **replacement policy** (also called, conversely, an **eviction policy**). A replacement policy dictates which of the blocks currently in the cache will be replaced by any new block that gets fetched in. (Or, another way of putting it would be that an eviction policy dictates which of the blocks currently in the cache will be evicted in order to make room for any new block that is fetched in.) One way to do things would be to pick a block at random to be replaced. Other alternatives would be FIFO (First in, First Out) or LIFO (Last In, First Out), or some other such variation. However, none of these policies take account of the fact that any block that was recently used is likely to be used again, soon. With these algorithms, we wind up evicting blocks that will be used again shortly, thereby increasing the cache miss rate and eating up valuable memory bus bandwidth with a bunch of unnecessary fetches.

The optimal replacement policy that doesn't involve predicting the future would be to evict the block that has gone the longest period of time without being used, or the Least Recently Used (LRU) block. The logic here is that if a block hasn't been

used in a while, it's less likely to be part of the current working set than a block that was more recently used.

An LRU algorithm, though ideal, isn't quite feasible to implement in real life. Control logic that checks each cache block to determine which one is the least recently used not only adds complexity to the cache design, but such a check would take up valuable time and thereby add unwanted latency to each replacement. Most caches wind up implementing some type of pseudo-LRU algorithm that approximates true LRU by marking blocks as more and more "dirty" the longer they sit unused in the cache. When a new block is fetched into the cache, the dirtiest blocks are the first to be replaced.

Sometimes, blocks that aren't all that dirty get replaced by newer blocks just because there isn't enough room in the cache to contain the entire working set. A miss that results when a block containing needed data has been evicted from the cache due to a lack of cache capacity is called a **capacity miss**. This is the third and final type of cache miss.

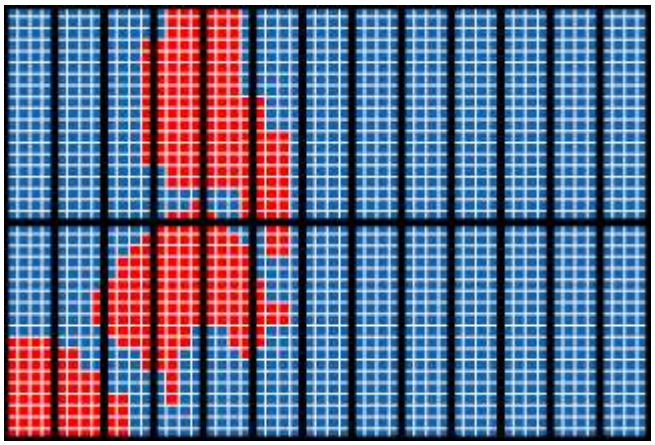
8. Conclusions.

Block Sizes

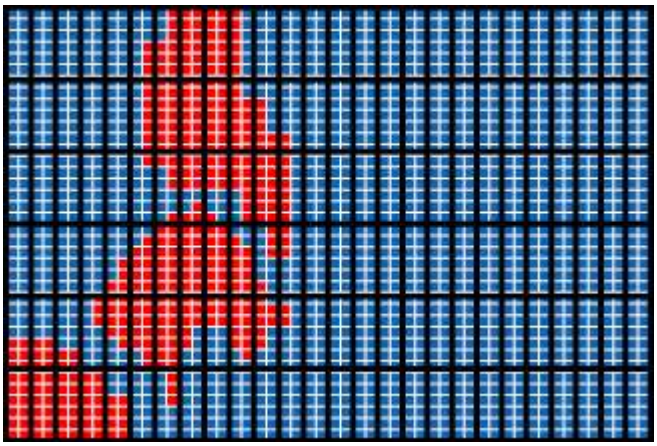
In the section on spatial locality I mentioned that storing whole blocks is one way that caches take advantage of spatial locality of reference. Now that we know a little more about how caches are organized internally, we can look a bit closer at the issue of block size. You might think that as cache sizes increase you could take even better advantage of spatial locality by making block sizes even bigger. Surely fetching more bytes per block into the cache would decrease the odds that some part of the working set will be evicted because it resides in a different block. This is true, to some extent, but we have to be careful. If we increase the block size while keeping the cache size the same, then we decrease the number of blocks that the cache can hold. Fewer blocks in the cache means fewer sets, and fewer sets means that collisions and therefore misses are more likely. And of course, with fewer blocks in the cache the likelihood that any particular block that the CPU needs will be available in the cache decreases.

The upshot of all this is that smaller block sizes allow us to exercise more fine-grained control of the cache. We can trace out the boundaries of a working set with a higher resolution by using smaller cache blocks. If our cache blocks are too large, we wind up with a lot of wasted cache space because many of the blocks will contain only a few bytes from the working set while the rest is irrelevant junk. If we think of this issue in terms of cache pollution, we can say that large cache blocks are more prone to pollute the cache with non-reusable data than small cache blocks.

The following image shows the memory map we've been using, with large block sizes.



This next image shows the same map, but with the block sizes decreased. Notice how much more control the smaller blocks allow over cache pollution.



The other problems with large block sizes are bandwidth-related. Since the larger the block size the more data is fetched with each LOAD, large block sizes can really eat up memory bus bandwidth, especially if the miss rate is high. So a system has to have plenty of bandwidth if it's going to make good use of large cache blocks. Otherwise, the increase in bus traffic can increase the amount of time it takes to fetch a cache block from memory, thereby adding latency to the cache.

Write Policies: Write through vs. Write back

So far, this entire article has dealt with only one type of memory traffic: loads, or requests for data from memory. I've only talked about loads because they make up the vast majority of memory traffic. The remainder of memory traffic is made up of stores, which in simple uniprocessor systems are much easier to deal with. In this section, we'll cover how to handle stores in single-processor systems with just an L1 cache. When you throw in more caches and multiple processors, things get more complicated than I want to go into, here.

Once a retrieved piece of data is modified by the CPU, it must be stored or written back out to main memory so that the rest of the system has access to the most up-to-date version of it. There are two ways to deal with such writes to memory. The first way is to immediately update all the copies of the modified data in each level of the hierarchy to reflect the latest changes. So a piece of modified data would be written to the L1 and main memory so that all of its copies are current. Such a policy for handling writes is called **write through**, since it writes the modified data through to all levels of the hierarchy.

A write through policy can be nice for multiprocessor and I/O-intensive system designs, since multiple clients are reading from memory at once and all need the most current data available. However, the multiple updates per write required by this policy can greatly increase memory traffic. For each STORE, the system must update multiple copies of the modified data. If there's a large amount of data that has been modified, then that could eat up quite a bit of memory bandwidth that could be used for the more important LOAD traffic.

The alternative to write through is **write back**, and it can potentially result in less memory traffic. With a write back policy, changes propagate down to the lower levels of the hierarchy as cache blocks are evicted from the higher levels. So an

updated piece of data in an L1 cache block will not be updated in main memory until it's evicted from the L1.

Conclusions

There is much, much more that can be said about caching, and this article has covered only the basic concepts. In the next article, we'll look in detail at the caching and memory systems of both the P4 and the G4e. This will provide an opportunity note only to fill in the preceding, general discussion with some real-world specifics, but also to introduce some more advanced caching concepts like data prefetching and cache coherency.

Bibliography

- David A. Patterson and John L. Hennessy, *Computer Architecture: A Quantitative Approach*. Second Edition. Morgan Kaufmann Publishers, Inc.: San Francisco, 1996.
- Dennis C. Lee, Patrick J. Crowley, Jean-Loup Baer, Thomas E. Anderson, and Brian N. Bershad, "Execution Characteristics of Desktop Applications on Windows NT." 1998. http://citeseer.nj.nec.com/lee98execution.html
- Institute for System-Level Integration, "Chapter 5: The Memory Hierarchy."
- Manish J. Bhatt, "Locality of Reference." Proceedings of the 4th Pattern Languages of Programming Conference. 1997. http://st-www.cs.uiuc.edu/users/hanmer/PLoP-97/
- James R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic." 25 Years ISCA: Retrospectives and Reprints 1998: 255-262
- Luiz Andre Barroso, Kourosh Gharachorloo, and Edouard Bugnion, "Memory System Characterization of Commercial Workloads." *Proceedings of the 25th International Symposium on Computer Architecture*. June 1998.

Revision History

Date	Version	Changes
7/9/2002	1.0	Release
7/9/2002	1.1	Page 3 was missing, so I've added it in.