

Understanding CPU Caching and Performance

by Hellazon

Caching is one of the most important concepts in computer systems. Understanding how caching works is the key to understanding system performance on all levels. On a PC, caching is everywhere. We'll focus our attention here on CPU-related cache issues; specifically, how code and data are cached for maximum performance. We'll start at the top and work our way down, stopping just before we get to the RAM.

Feeding the beast

L1, L2, RAM, page files, etc... Why all these caches? The answer is simple: you've got to feed the beast if you want it to work. When there's a large access-time or transfer rate disparity between the producer (the disk or RAM) and the consumer (the CPU), we need caches to speed things along. The CPU needs code and data to crunch, and it needs it now. There are two overriding factors in setting up a caching scheme: cost and speed. Faster storage costs more (add that to the list of things that are certain, along with death and taxes). As you move down the storage hierarchy from on-chip L1 to hard disk swap-file, you'll notice that access time and latency increase as the cost-per-bit decreases. So we put the fastest, most expensive, and lowest-capacity storage closest to the CPU, and work our way out by adding slower, cheaper, and higher-capacity storage. What you wind up with is a pyramid-shaped caching hierarchy.

So what it all boils down to is this: if the CPU needs something, it checks its fastest cache. If what it needs isn't there, it checks the next fastest cache. If that's no good, then it checks the next fastest cache . . . all the way down the storage hierarchy. The trick is to make sure that the data that's used most often is closest to the top of the hierarchy (in the smallest, fastest, and most expensive cache), and the data that's used the least often is near the bottom (in the largest, slowest, and cheapest cache).

Most discussions on caching break things down according to issues in cache design, e.g. "cache coherency and consistency", "caching algorithms", etc. I'm not going to do that here. If you want to read one of those discussions, then you should buy a good textbook. The approach I'll take here is to start at the top of the cache hierarchy and work my way down, explaining in as much detail as I can each cache's role in enhancing system performance. In particular, I'll focus on how code and data work with and against this caching scheme.

One more thing, I look forward to getting feedback on this article. As always, if I'm out of line, then please feel free to correct me. And as always, leave the 'tude at the door. We all try to be professional and courteous here; we'll respect you if you respect us. That having been said, lets get on with the show.

First, the registers

The first and most important cache is the on-chip register set. Now, I know you're not used to thinking of the registers as a cache, but they certainly fit the description. Whenever the ALU (Arithmetic Logic Unit) needs two numbers to crunch and a place to put the results, it uses the registers. These registers are the fastest form of storage on the CPU, and they're also the most expensive. They're expensive in terms of the logic it takes to implement them. More registers means more chip real estate, and chip real estate is the priciest real estate there is (with Boston and NYC running second and third respectively). However, having more registers makes parallelizing code easier.

Parallel Power

The beauty of modern superscalar architectures is that they can perform more than one operation at once. Now there's more than one reason you'd want to do this. Some types of problems can be solved quicker if you break them up into smaller parts and then do all the smaller parts all at once. For instance, say you wanted to find all the prime numbers between 1 and 100. One way to do it would be to start at 1 and check each consecutive

integer for primeness until you get to 100. However, there's a better way to do this if you have the resources: Split the problem up 10 ways among 10 friends. Give each of your 10 friends 10 numbers to check for primeness, and let them all do it at the same time. Assuming all your friends work at the same speed, then you'll get the answers ten times as fast. That's parallel processing.

The problem with parallel processing is that only certain types of problems lend themselves to this type of processing. Some problems you just can't break up. For instance, if you have a string of complex calculations (or decisions), and each calculation (or decision) depends on the one before it for at least one of its inputs, then you can only do those one at a time in sequence. Or can you? Here's where branch prediction and large register sets come in. What branch prediction does is let you use subsets of registers to execute down multiple branch paths simultaneously. So while you're waiting for that calculation to finish, you can make a guess as to what its outcome will be and then go on with the sequence using a different register subset while the calculation you were waiting on is still wrapping up. Of course if you guess wrong, then you've got to throw out all of those speculative results, which means you've just wasted your resources on worthless calculations. Not a good thing. Modern branch prediction algorithms are very good, though, and bad guesses don't happen often.

When you need data from a register to do a calculation but the data in that register is unstable or is still subject to unfinished calculations, this is called a dependency. Dependencies are bad, and the more registers you have, the easier it is to avoid them. Branch prediction sounds wonderful, but the problem with it is that it takes a lot of logic. Prediction logic must unravel code as it's running, look for dependencies, and make predictions. Often, that logic could be better used elsewhere. When IA-64 comes out . . . nope, I'd better save that for the next article.

The L1 cache -- It's the code, stupid

I'm not going to talk about write policy, or unified vs. instruction/data caches. These things affect performance, but there are excellent write-ups of them elsewhere. For a good introduction to L1 cache technology, check out the PCGuide.

What I want to discuss here is how code and data interact with the cache to affect performance. For this particular discussion, I'm going to take the Pentium II's L1 cache as my starting point. The PII's L1 is split into a 16KB instruction cache and a 16KB data cache. That's all we need to know about the PII's L1 to get started.

The L1 cache's job is to keep the CPU supplied with instructions and the registers supplied with data. Whenever the CPU needs its next instruction, it should be able to look into the L1 cache and find it instantly. When the CPU doesn't find the instruction in the L1 cache, this event is called a cache miss. The CPU then has to go out to the L2 and hunt down that instruction. If it ain't in the L2, the CPU goes out to RAM, and so on. The same goes for data.

The ideal caching algorithm knows which bits of code and data will be needed next, and it goes out and gets them. Of course, such an ideal algorithm can't actually be implemented, so you have to do your best to guess what's coming up. Modern caching algorithms are pretty good at guessing, generating hit rates of over 90%.

Caching Code

The key to caching instructions is to know what code is being used. Lucky for all of us, code exhibits a property called locality of reference. This phrase is a fancy way of saying that after you execute an instruction, the odds that the next instruction you'll need to execute is stored somewhere very nearby in memory are damn good. But the type of code you're running affects those odds. Some code gives better odds on easily finding the next instruction, and some gives worse.

Business Apps

With a business app like MS Word, there are lots of buttons and menus and pretty things to play with on the screen. You could press any one of them at any time, and completely different bits of code from completely different parts of memory would have to be put into the L1. There's no way of knowing whether the user is going to stop typing and fire up the "insert table wizard", or the annoying "Office Assistant" (does anybody else hate that thing?), or the drawing tools. Such are the joys of event-driven programming. Now, within a particular menu selection or wizard, there's a pretty high degree of locality of reference. Lines of code that do related tasks tend to be near each other. But overall, an office program like Word tends to call on code from all over the map. In a situation like this, you can see why the number of L1 cache misses would be pretty high. It's just tough to predict what code will need to execute next when the user switches tasks so often.

3D Games/Simulation/Image Processing

So now let's talk about 3D Games like Quake and uh...what's that other game...Unplayable, Unoriginal, Uncool, Unstable,...oh! Unreal! (I'd blocked it out of my mind.) These games, and any similar program, make very good use of the L1 cache. Why is that? There are a number of reasons, but one thing probably accounts for most of their success: matrices.

Simulations use matrices like you wouldn't believe. 3D simulations use matrices for a whole host of tasks, including transformation and translation. Translating or transforming a 3D object requires iterating through various matrices. Even other types of simulations like neural networks use matrix math. All a neural net really consists of is a set of matrices describing the weights on connections between neurons, along with input and output vectors. When the network learns, these weights have to be updated, a task which requires you to iterate through the weight matrices.

Image processing programs are also matrix intensive. When an image is loaded into memory, it's converted into a matrix of RGB values. Histograms, filters, edge detection, and other image processing functions are all done by iterating through the image matrix and changing the values in it. To iterate through a matrix you need lots of loops, and these loops often fit in the L1 cache. So when you're operating on matrix data you're using a small amount of code to crunch a large amount of data. The algorithms which this code implements are loop-based and very sequential (those of you who've sat through a differential equations course know what I'm talking about). Actually, most numerical methods programming involves highly sequential code which doesn't jump around too much.

Some Clarifications

I can already see people writing in and explaining to me in detail how the network code or some other piece of a game like Quake involves all sorts of user input and unpredictability just like an office app. This is true, but it's also true that the part of a simulation/game like Quake that is the most CPU-intensive is the rendering engine. And the rendering engine is a fairly sequential, loopy, and predictable mathematical beast—just exactly the sort of animal that makes excellent use of the L1 cache.

Caching Data

The differences between business apps and simulations in terms of code caching pretty much apply to data caching, as well. I haven't written any business apps, so I won't say any more about them than I've already said. If anybody out there has written a word processor or something and would like to explain how it uses the L1 cache, you're welcome to send in a good explanation, and we'll post it.

I may not have written any business apps, but I sure as hell have written simulations and math routines out the waz (my undergraduate focus was on math and programming). So I do know something about how these types of programs use data. As in the above discussion on code, the most important thing you can say about these

types of programs is that they use lots of matrices. Most good programmers know how to use double pointers to create a dynamically allocated matrix which stores its data in a grid of contiguous memory addresses. Since the matrix data are stored in contiguous memory addresses, and they get iterated through a lot, then you can imagine how easy it is to load a whole hunk of a matrix into the L1 and crunch on it for a while. And when you get done with that piece, you know right where the next one is coming from.

The L2 cache

The L2 isn't as glamorous (in my opinion) as the L1. It's gotten lots of press lately though, so I guess I should say a bit about it. It sort of occupies this in-between limbo between the L1 cache (which is very important for increasing performance) and the RAM (which is essential for the machine to even function at all). Basically, the only reason that the L2 is there is in case there's a miss in the L1. If the cache hit rate for the L1 were 100%, then we wouldn't need the L2. The reason the L2 does actually get press is because the L1 does miss, and when it misses it hurts to have to go all the way out to RAM to get the needed data. So the L2 sort of acts as a way-station between the L1 and the RAM. That way the penalty for an L1 miss isn't quite as stiff. And now that I've explained what types of things affect whether or not the L1 misses, it should be pretty obvious why the L2 is more important in business apps than in games.

A Word About FP and INT Performance and Cache Usage

A business application like Word is integer, not floating-point, intensive. This fact, along with Word's poor performance on the cacheless Celeron, misled some folks into thinking that integer intensiveness and L2 cache usage are somehow related. This isn't necessarily the case. It just so happens that applications that are integer-intensive also happen to have lower locality of reference than floating-point-intensive apps.

Understanding CPU Caching and Performance: Part II

by Hellazon

I received some thoughtful responses to my first article on caching, entitled, aptly enough, Understanding CPU Caching and Performance. A few of you raised some issues that really need to be addressed. So, let me get right into the reader email so we can get a strive on for some communal enlightenment.

Smart vs. dumb caches

One reader wrote in to further clarify the difference between the hit rates in the L1 and L2. Dennis Gorrie writes:

"The one comment I have is about the description of caches. For example, not all caches have a 90% hit rate. Of course, it does depend on the application, but even for a given application, some caches do much better than others. Typically, an L1 cache will get over 90% hit rate, while the L2 cache is much lower. This is because the L1 is a 'smarter' cache than the L2 cache. Smarter equates to more logic, more real-estate, and more expensive, as you have pointed out, but also a higher hit rate, even though only a few KB compared to the L2 cache. The fact that an L1 cache can achieve such a high hit rate even though it is a fraction of the size of the L2 cache is a dramatic reminder of the difference your caching algorithm can make."

I didn't go into this issue, but it's a good point. The L1 does indeed use a better (and more complex) caching algorithm than the L2, an algorithm which results in much higher hit rates. I'm not going to say too much about the specific algorithms used because you can find a good description of them elsewhere.

Here's a good write-up of the locality model from one of my old textbooks.

From Operating System Concepts: Fifth Edition, by Silberschatz and Galvin, Addison-Wesley 1998,

The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together. A program is generally composed of several different localities, which may overlap.

For example, when a subroutine is called, it defines a new locality. In this locality, memory references are made to the instructions of the subroutine, its local variables, and a subset of the global variables. When the subroutine is exited, the process leaves this locality, since the local variables and instructions of the subroutine are no longer in active use. We may return to this locality later. Thus, we see that localities are defined by the program structure and its data structures. The locality model states that all programs will exhibit this basic memory reference structure. Note that the locality model is the unstated principle behind all the caching discussions so far in this book. If accesses to any types of data were random rather than patterned, caching would be useless.

Reading this, you can see that the LRU algorithm depends solely on locality of reference. Let's look again at the statement "...it keeps track of what has been used most often, regardless of locality, and keeps that in the cache if at all possible. When it needs to load new instructions/data, it throws out the least used cache elements and replaces them with the new one." Think about this idea for a second: the only reason it makes sense to keep the most-used code in the cache and throw out the least-used code is because code that has been used once is more likely to be used again. Why? Because code comes from a particular locality, and if you can drop that locality into the cache, then you can use it for a while before the program switches localities. So Dennis' statement should actually read "...it keeps track of what has been used most often, because of locality, and keeps that in the cache if at all possible."

Also, I'd like to clarify one other part of the reader's response. He says "you can't really make 'branch prediction' better, as you always have a 50/50 chance of being right or wrong." There are ways to improve branch prediction, but that's beyond the scope of this discussion (maybe I'll devote an article to branch prediction).

Once again, a good clarification of my point, but I have to take issue with the assertion that Set Associative LRU doesn't depend on locality of reference. All caching schemes depend on the locality model. If data access were purely random, no caching scheme would work.

FP and INT performance and cache usage

At the end of my article, I claimed that FP- or integer-intensiveness doesn't affect cache usage. One reader wrote in with some reasons why this might not actually be true. Don wrote:

Another thing I thought of is that there are floating point divides in 3D which take more than 30cycles to complete (divides are not pipelined). Therefore, at 5-1-1-1 for 100MHZ SDRAM it takes 32CPU cycles to get 16bytes of data from main memory. It would take 60+cycles to divide those (4-32bit numbers) and 32cycles to retrieve from memory. So, having the data in cache results in only a 1/3 speed improvement. On the other hand performing integer operations on 16 ascii characters, could take as little as 16 cycles. In this case the difference between having the data in L1 cache is 16/48, or a $\frac{3}{4}$ speed improvement. Put the data in a 100MHZ L2 cache (2-1-

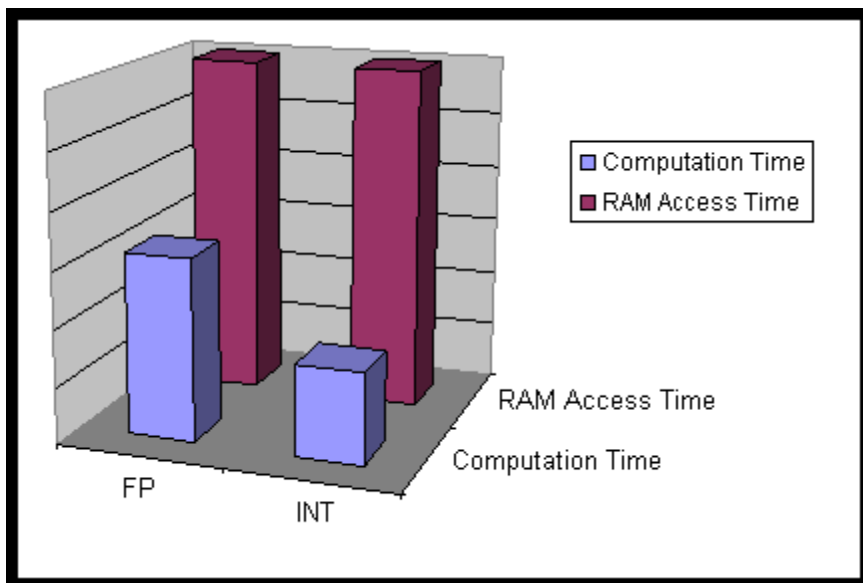
1-1 or 20 - 400MHZ cycle) and the difference is 36/48 or 1/3 faster. So, because even the smallest Word document can't be held in L1 cache, that would explain why the cacheless Celeron was so bad at Office apps. Then, consider that a lot of the benchmarking involves copy and paste operations which are definitely memory intensive having the data in any cache is even more important.

I'm not going to check his numbers, but I think the gist of Don's comments is probably on target. Let me lay this out in simpler terms, and we can see how plausible it sounds. Near the beginning of my article, I said: "When there's a large access-time or transfer rate disparity between the producer (the disk or RAM) and the consumer (the CPU), we need caches to speed things along." Now, when you're crunching those floating-point numbers, the CPU can take quite a few cycles to do its work. Not only do FP ops take many cycles to complete, but some of them also can't be pipelined. So when it's grinding away at those floating point matrices in Quake, it's huffing and puffing and dragging along.

In contrast, when you feed it some integers to crunch it flies right through them. Not only do integer ops take fewer cycles to complete, you can do more of them at once. The CPU can sail through an array of integers, but it gets bogged down in an array of floating point numbers.

Now, here's the key part. Regardless of whether the CPU is running, walking, or crawling, the time it takes to access RAM is the same. If you need to get 32 bits from RAM (one 32-bit float or two 16-bit integers), it's going to take you a certain amount of time to grab those 32 bits regardless of what you intend on doing with them once you get them.

We could make the following chart:



If caching exists to take up the slack when there's a disparity between the speed at which the producer (RAM) produces and the speed at which the consumer (CPU) consumes, then--it would seem--if that disparity is lessened, the cache won't make as much of a difference. There isn't as large a speed disparity between the producer and the consumer, so the delays the caches are intended to minimize aren't as critical.

So it seems I was wrong to assert that caching and FP/integer-intensiveness are unrelated issues. I think the argument laid out above is pretty plausible, but I may be over-simplifying things. There could be other issues lurking around which make this whole discussion way off the mark. If anyone can give some solid objections to the above argument, I'd be glad to hear them and give them some air time here.

The L3 cache on the K6+3D and the possibilities for Mendocino

Dark Chyld raised a question in the forum about the L3 cache that AMD is planning on implementing on their K6-3 supported motherboards. This whole L3 cache thing is part of AMD's drive to keep the Super 7 architecture competitive with Intel's various Slot-based offerings. Consider this quote from W.J. Sanders' keynote address at the Microprocessor Forum:

In the second half of 1998, we expect to introduce an even more powerful processor for the visual computing platform, the AMD-K6+ 3D processor, with clock speeds up to 400 megahertz. The AMD-K6+ 3D processor will add 256 kilobytes of on-chip backside L2 cache running at the full speed of the processor while maintaining mechanical Socket 7/Super 7 compatibility.

What will happen here is that the L2 cache will move up the speed/cost/size hierarchy and leave a space there on the motherboard for another cache—the L3. Or perhaps a better way to think of it is that the cache we normally know as L2 is staying there on the mother-board, but it's getting renamed "L3" because they're adding another cache on the chip and calling it the "L2". So now there will be three caches between the CPU and RAM. One wonders if they'll add some smarts to the L2 cache and have it use an algorithm like LRU, or will they just leave it direct-mapped. For more information on AMD's plans for Super 7, check out this link.

The same reader also mentioned the possibility that Intel could release a Mendocino-based system with an L3 on the motherboard. This is possible, but you'd think that if they were planning to do this that they'd have already mentioned as part of their roadmap. But then again, who the heck understands Intel's labyrinthine roadmap, anyway? Maybe this idea will come up. It sounds like an excellent idea.

Conclusions and credits

One of the readers raised a question in the forum about caching and SMP machines. I said I'd address it here, but I've since decided that SMP needs its own article. For more information on caching, I'd recommend the PCGuide. It can give you the lowdown on the various caching algorithms.

I'd like to send props out to the readers who wrote in, and the folks in the forum who contributed insightful questions and comments, especially the people who I've used here (Don, Dennis Gorrie, and Dark Chyld). I'd also like to thank my boy Stephen for giving me some preliminary feedback on the article, and I want to send mad props out to Dr. Damage for his brilliant layout job (he also did the RISC vs. CISC series, and the previous caching article).