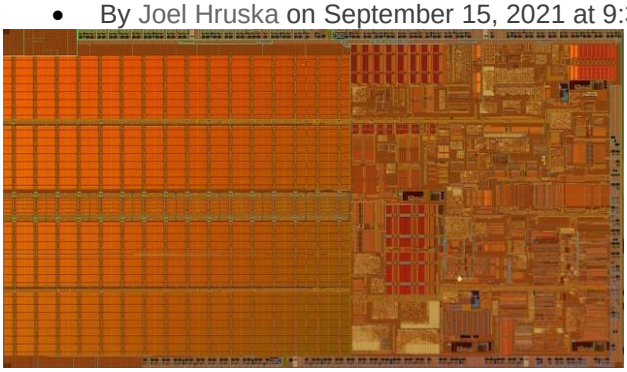
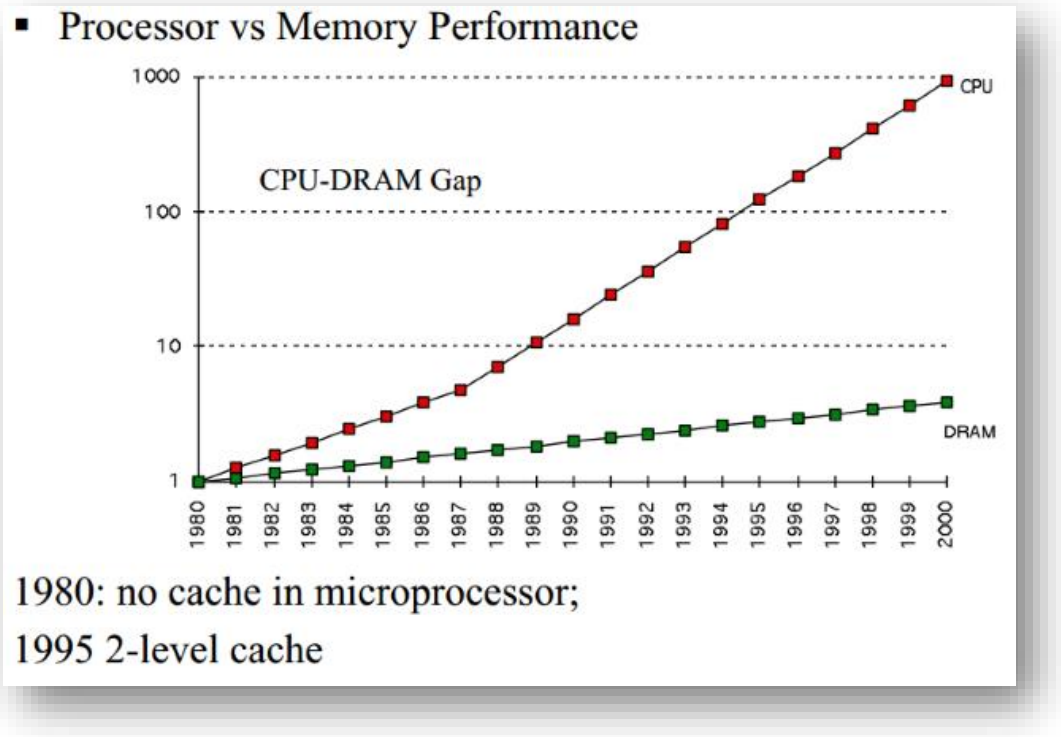
How L1 and L2 CPU Caches Work, and Why They're an Essential Part of Modern Chips

- By Joel Hruska on September 15, 2021 at 9:38 am



The development of caches and caching is one of the most significant events in the history of computing. Virtually every modern CPU core from ultra-low power chips like the ARM Cortex-A5 to the highest-end Intel Core i9 use caches. Even higher-end microcontrollers often have small caches or offer them as options — the performance benefits are too large to ignore, even in ultra-low-power designs.

Caching was invented to solve a significant problem. In the early decades of computing, main memory was extremely slow and incredibly expensive, but CPUs weren't particularly fast, either. Starting in the 1980s, the gap began to widen quickly. Microprocessor clock speeds took off, but memory access times improved far less dramatically. As this gap grew, it became increasingly clear that a new type of fast memory was needed to bridge the gap.

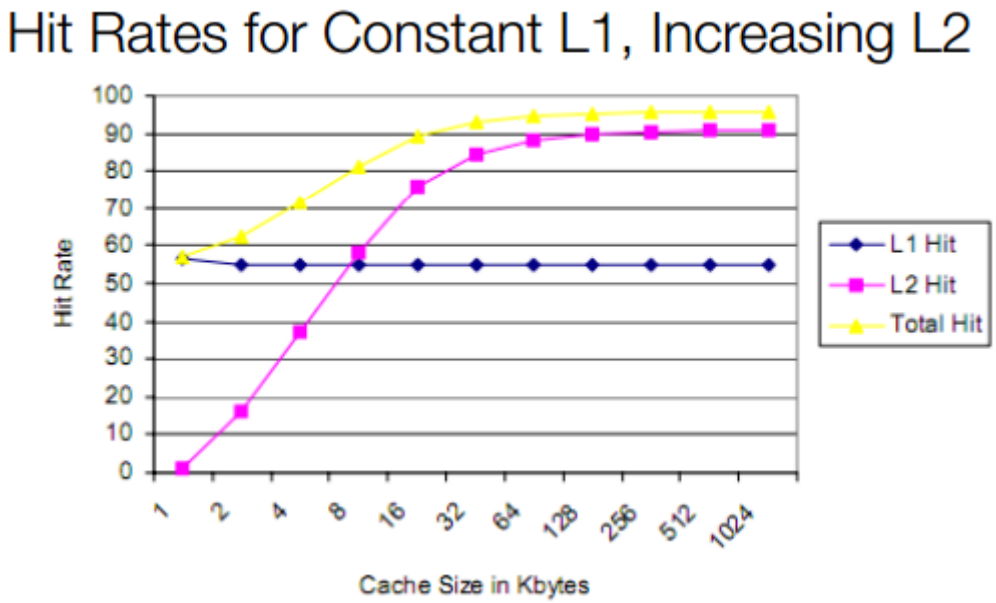


While it only runs up to 2000, the growing discrepancies of the 1980s led to the development of the first CPU caches.

How Caching Works

CPU caches are small pools of memory that store information the CPU is most likely to need next. Which information is loaded into cache depends on sophisticated algorithms and certain assumptions about programming code. The goal of the cache system is to ensure that the CPU has the next bit of data it will need already loaded into cache by the time it goes looking for it (also called a cache hit).

A cache miss, on the other hand, means the CPU has to go scampering off to find the data elsewhere. This is where the L2 cache comes into play — while it's slower, it's also much larger. Some processors use an inclusive cache design (meaning data stored in the L1 cache is also duplicated in the L2 cache) while others are exclusive (meaning the two caches never share data). If data can't be found in the L2 cache, the CPU continues down the chain to L3 (typically still on-die), then L4 (if it exists) and main memory (DRAM).

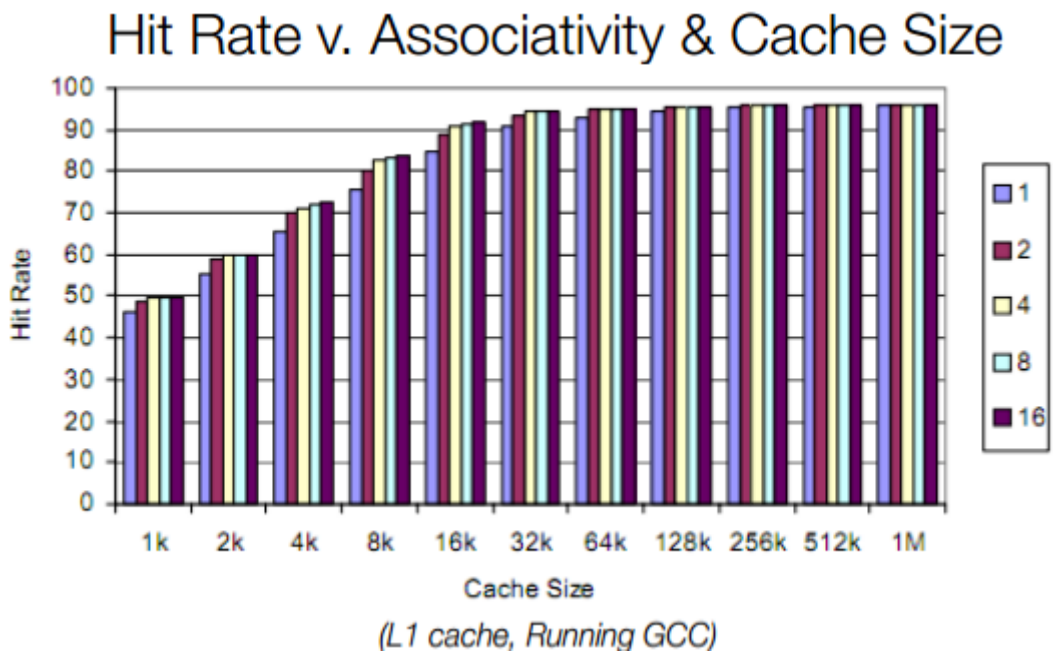


This chart shows the relationship between an L1 cache with a constant hit rate, but a larger L2 cache. Note that the total hit rate goes up sharply as the size of the L2 increases. A larger, slower, cheaper L2 can provide all the benefits of a large L1, but without the die size and power consumption penalty. Most modern L1 cache rates have hit rates far above the theoretical 50 percent shown here — Intel and AMD both typically field cache hit rates of 95 percent or higher.

The next important topic is the set-associativity. Every CPU contains a specific type of RAM called tag RAM. The tag RAM is a record of all the memory locations that can map to any given block of cache. If a cache is fully associative, it means that any block of RAM data can be stored in any block of cache. The advantage of such a system is that the hit rate is high, but the search time is extremely long — the CPU has to look through its entire cache to find out if the data is present before searching main memory.

At the opposite end of the spectrum, we have direct-mapped caches. A direct-mapped cache is a cache where each cache block can contain one and only one block of main memory. This type of cache can be searched extremely quickly, but since it maps 1:1 to memory locations, it has a low hit rate. In between these two extremes are *n*-way associative caches. A 2-way associative cache (Piledriver's L1 is 2-way) means that each main memory block can map to one of two cache blocks. An eight-way associative cache means that each block of main memory could be in one of eight cache blocks. Ryzen's L1 instruction cache is 4-way associative, while the L1 data cache is 8-way set associative.

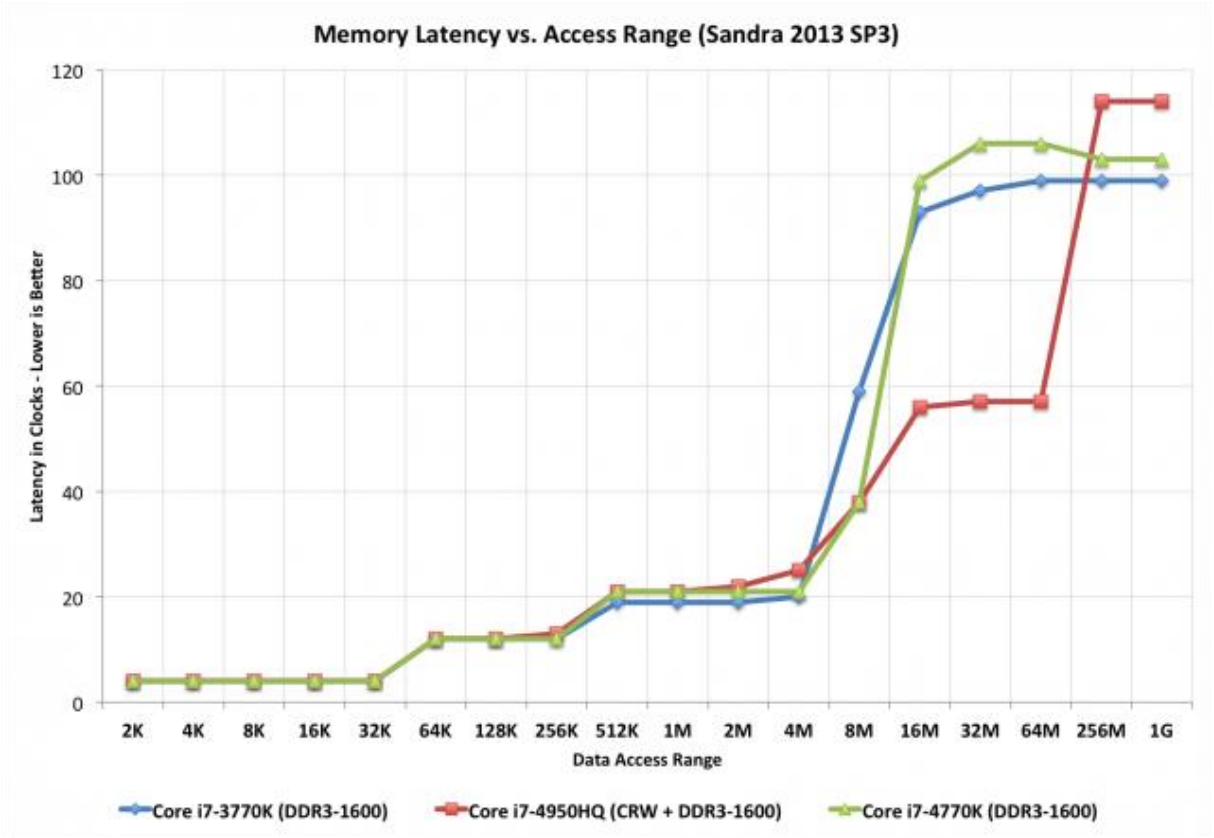
The next two slides show how hit rate improves with set associativity. Keep in mind that things like hit rate are highly particular — different applications will have different hit rates.



(L1 cache, Running GCC)

Why CPU Caches Keep Getting Larger

So why add continually larger caches in the first place? Because each additional memory pool pushes back the need to access main memory and can improve performance in specific cases.



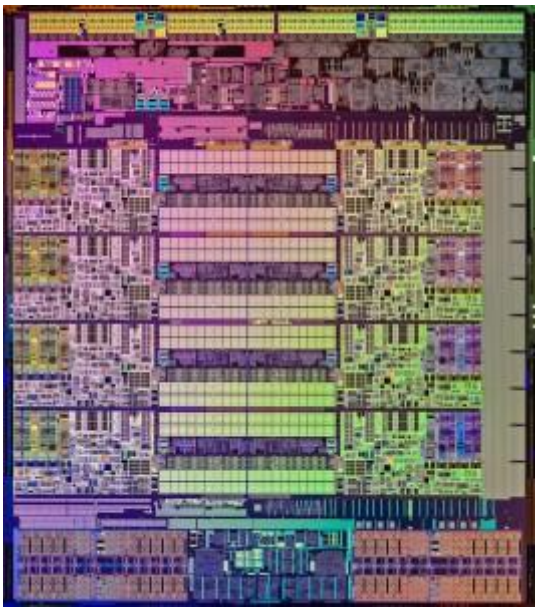
This chart from Anandtech's Haswell review is useful because it illustrates the performance impact of adding a huge (128MB) L4 cache as well as the conventional L1/L2/L3 structures. Each stair step represents a new level of cache. The red line is the chip with an L4 — note that for large file sizes, it's still almost twice as fast as the other two Intel chips.

It might seem logical, then, to devote huge amounts of on-die resources to cache — but it turns out there's a diminishing marginal return to doing so. Larger caches are both slower and more expensive. At six transistors per bit of SRAM (6T), cache is also expensive (in terms of die size, and therefore dollar cost). Past a certain point, it makes more sense to spend the chip's power budget and transistor count on more execution units, better branch prediction, or additional cores. At the top of the story, you can see an image of the Pentium M (Centrino/Dothan) chip; the entire left side of the die is dedicated to a massive L2 cache. This was the case in the last days of single-threaded CPUs, now that we have multi-core chips and GPU's on-die in many cases, a smaller percentage of the overall CPU is dedicated to cache.

How Cache Design Impacts Performance

The performance impact of adding a CPU cache is directly related to its efficiency or hit rate; repeated cache misses can have a catastrophic impact on CPU performance. The following example is vastly simplified but should serve to illustrate the point.

Imagine that a CPU has to load data from the L1 cache 100 times in a row. The L1 cache has a 1ns access latency and a 100 percent hit rate. It, therefore, takes our CPU 100 nanoseconds to perform this operation.



Haswell-E die shot (click to zoom in). The repetitive structures in the middle of the chip are 20MB of shared L3 cache.

Now, assume the cache has a 99 percent hit rate, but the data the CPU actually needs for its 100th access is sitting in L2, with a 10-cycle (10ns) access latency. That means it takes the CPU 99 nanoseconds to perform the first 99 reads and 10 nanoseconds to perform the 100th. A 1 percent reduction in hit rate has just slowed the CPU down by 10 percent.

In the real world, an L1 cache typically has a hit rate between 95 and 97 percent, but the *performance* impact of those two values in our simple example isn't 2 percent — it's 14 percent. Keep in mind, we're assuming the missed data is always sitting in the L2 cache. If the data has been evicted from the cache and is sitting in main memory, with an access latency of 80-120ns, the performance difference between a 95 and 97 percent hit rate could nearly double the total time needed to execute the code.

Back when AMD's Bulldozer family was compared with Intel's processors, the topic of cache design and performance impact came up a great deal. It's not clear how much of Bulldozer's lackluster performance could be blamed on its relatively slow cache subsystem — in addition to having relatively high latencies, the Bulldozer family also suffered from a high amount of cache *contention.* Each Bulldozer/Piledriver/Steamroller module shared its L1 instruction cache, as shown below:

Cache	Bulldozer	Piledriver	Steamroller
Level 1 code	64 kB, 2-way, 64 B line size, shared between two cores.	64 kB, 2-way, 64 B line size, shared between two cores.	96 kB, 3-way, 64 B line size, shared between two cores.
Level 1 data	16 kB, 4-way, 64 B line size, per core. Latency 3-4 clocks.	16 kB, 4-way, 64 B line size, per core. Latency 3-4 clocks.	16 kB, 4-way, 64 B line size, per core. Latency 3-4 clocks.
Level 2	1 - 2 MB, 16-way, 64 B line size, shared between two cores. Latency 21 clocks. Read throughput 1 per 4 clock. Write throughput 1 per 12 clock.	2 MB, 16-way, 64 B line size, shared between two cores. Latency 20 clocks. Read throughput 1 per 4 clock. Write throughput 1 per 12 clock.	2 MB, 16-way, 64 B line size, shared between two cores. Latency 19 clocks. Read throughput 1 per 4 clock. Write throughput 1 per 6 clock.
Level 3	0 - 8 MB, 64-way, 64 B line size, shared between all cores. Latency 87 clock. Read throughput 1 per 15 clock. Write throughput 1 per 21 clock.	0 - 8 MB, 64-way, 64 B line size, shared between all cores. Latency 87 clock. Read throughput 1 per 15 clock. Write throughput 1 per 21 clock.	None

Table 14.4. Cache sizes on AMD Bulldozer, Piledriver and Steamroller

A cache is contended when two different threads are writing and overwriting data in the same memory space. It hurts the performance of both threads — each core is forced to spend time writing its own preferred data into the L1, only for the other core promptly overwrite that information. AMD'S older Steamroller still gets whacked by this problem, even though AMD increased the L1 code cache to 96KB and made it three-way associative instead of two. Later Ryzen CPUs do not share cache in this fashion and do not suffer from this problem.

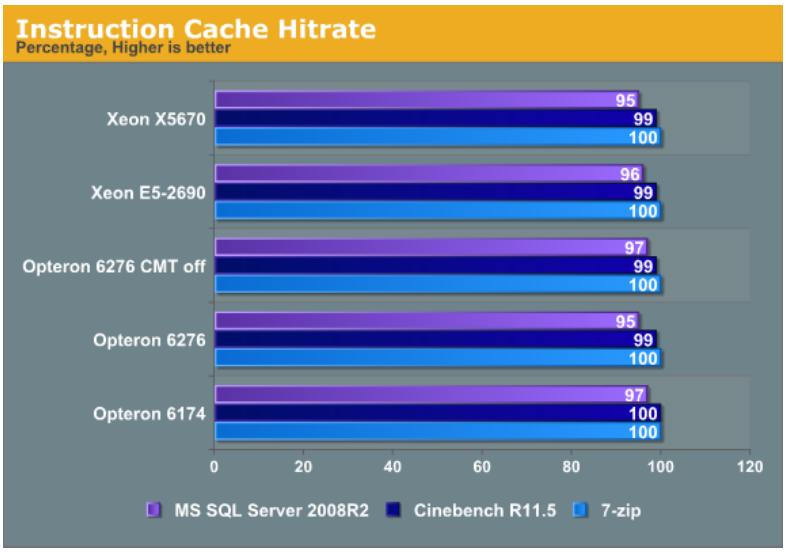
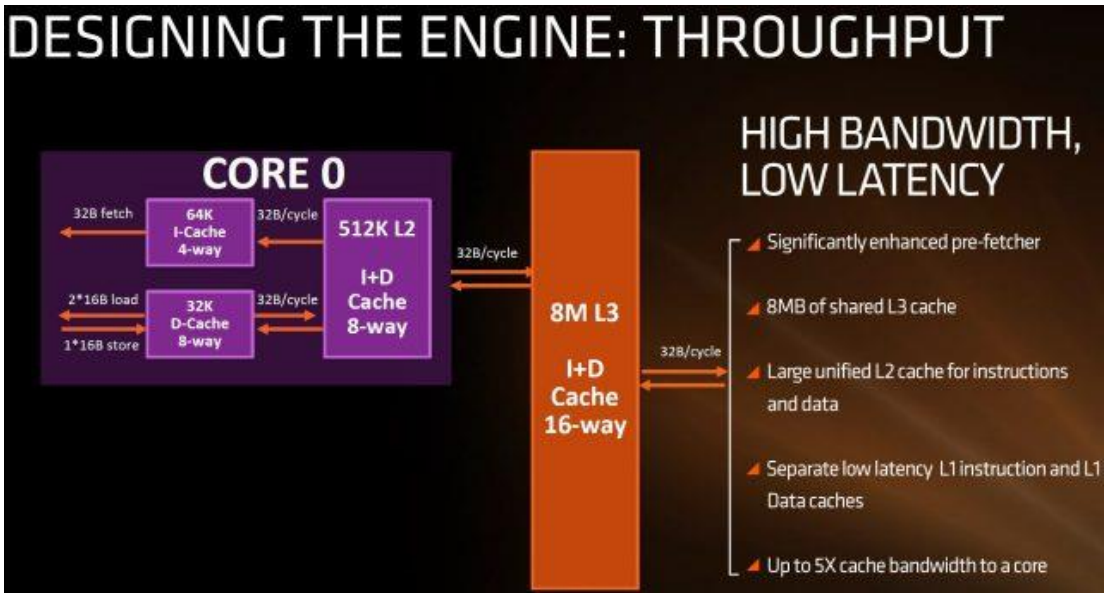


Image by Anandtech

This graph shows how the hit rate of the Opteron 6276 (an original Bulldozer processor) dropped off when both cores were active, in at least some tests. Clearly, however, cache contention isn't the only problem — the 6276 historically struggled to outperform the 6174 even when both processors had equal hit rates.

Zen 2 does not have these kinds of weaknesses today, and the overall cache and memory performance of Zen and Zen 2 is much better than the older Piledriver architecture.

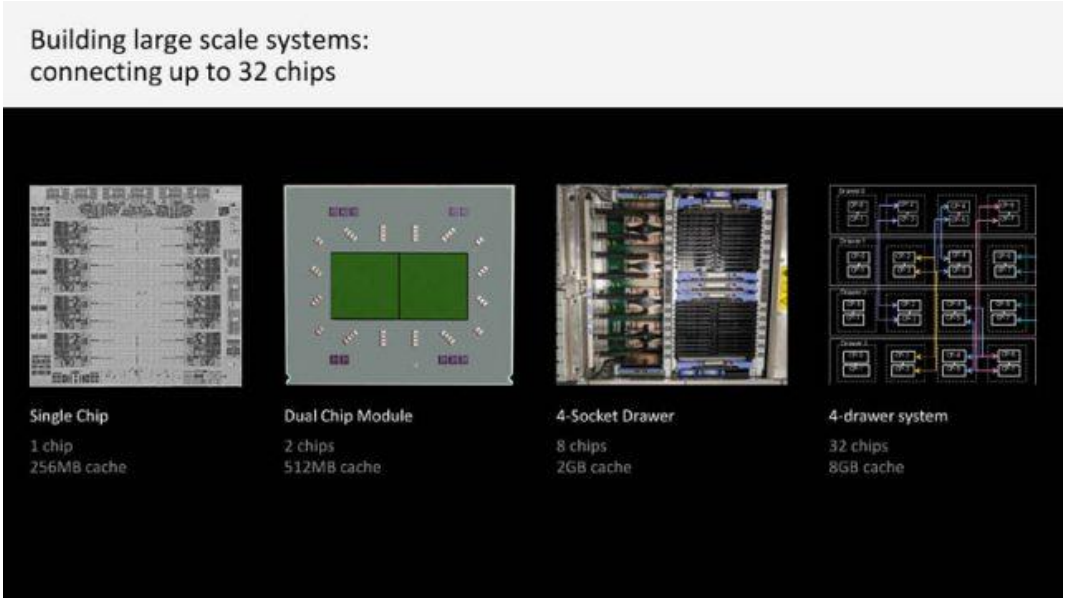


From AMD's initial Ryzen debut. Ryzen uses a very different cache system than previous AMD CPUs.

Modern CPUs also often have a very small "L0" cache, which is often just a few KB in size and is used for storing micro-ops. AMD and Intel both use this kind of cache; Zen had a 2,048 µOP cache, while Zen 2 has a 4,096 µOP cache. These tiny cache pools operate under the same general principles as L1 and L2, but represent an even-smaller pool of memory that the CPU can access at even lower latencies than L1. Often companies will adjust these capabilities against each other. Zen 1 and Zen+ (Ryzen 1xxx, 2xxx, 3xxx APUs) have a 64KB L1 instruction cache that's 4-way set associative and a 2,048 µOP L0 cache. Zen 2 (Ryzen 3xxx desktop CPUs, Ryzen Mobile 4xxx) has a 32KB L1 instruction cache that's 8-way set associative and a 4,096 µOP cache. Doubling the set associativity and the size of the µOP cache allowed AMD to cut the size of the L1 cache in half. These kinds of trade-offs are common in CPU designs.

Future Innovations

Recently, IBM debuted its Telum microprocessor with an interesting and unusual cache structure. Telum has the usual L1 and L2, but instead of a physical L3, the CPU deploys a virtual L3. Instead of completely evicting L2 data that the CPU doesn't believe it needs any longer, the next-generation CPU evicts it into the L2 cache of a different CPU on the same slice of silicon, and marks it as L3 data. Each core has its own 32MB L2 and the virtual L3 across the entire chip is 256MB. IBM can even share this capability across multi-chip systems, creating a virtual L4 with a total of 8192MB of data storage.



This type of virtual cache system is something unique — it doesn't have an equivalent on the x86 side of the equation — and it's an interesting example of how companies are pushing the envelope of cache design. While AMD's V-Cache is designed to provide additional L3, not L2, we'd be remiss not to mention it at all. AMD's recent achievement and a capability expected to show up on future Zen CPUs is a large L3 cache vertically mounted on top of existing chiplets and connected to them through in-die silicon vias. Cache may be 40 years old at this point, but manufacturers and designers are still finding ways to improve it and expand its utility.

Caching Out

Cache structure and design are still being fine-tuned as researchers look for ways to squeeze higher performance out of smaller caches. So far, manufacturers like Intel and AMD haven't dramatically pushed for larger caches or taken designs all the way out to an L4 yet. There are some Intel CPUs with onboard EDRAM that have what amounts to an L4 cache, but this approach is unusual. That's why we used the Haswell example above, even though that CPU is older. Presumably, the benefits of a large L4 cache do not yet outweigh the costs for most use-cases.

Regardless, cache design, power consumption, and performance will be critical to the performance of future processors, and substantive improvements to current designs could boost the status of whichever company can implement them.