CPU cache

A **CPU cache** is a hardware cache used by the central processing unit (CPU) of a computer to reduce the average cost (time or energy) to access data from the main memory.[1] A cache is a smaller, faster memory, located closer to a processor core, which stores copies of the data from frequently used main memory locations. Most CPUs have a hierarchy of multiple cache levels (L1, L2, often L3, and rarely even L4), with separate instruction-specific and data-specific caches at level 1.

Other types of caches exist (that are not counted towards the "cache size" of the most important caches mentioned above), such as the translation lookaside buffer (TLB) which is part of the memory management unit (MMU) which most CPUs have.

Contents

Overview

When trying to read from or write to a location in the main memory, the processor checks whether the data from that location is already in the cache. If so, the processor will read from or write to the cache instead of the much slower main memory.

Most modern desktop and server CPUs have at least three independent caches: an **instruction cache** to speed up executable instruction fetch, a **data cache** to speed up data fetch and store, and a translation lookaside buffer(TLB) used to speed up virtual-to-physical address translation for both executable instructions and data. A single TLB can be provided for access to both instructions and data, or a separate Instruction TLB (ITLB) and data TLB (DTLB) can be provided.[2] The data cache is usually organized as a hierarchy of more cache levels (L1, L2, etc.; see also multi-level caches below). However, the TLB cache is part of the memory management unit (MMU) and not directly related to the CPU caches.

History

The first CPUs that used a cache had only one level of cache; unlike later level 1 cache, it was not split into L1d (for data) and L1i (for instructions). Split L1 cache started in 1976 with the IBM 801 CPU,[3][4] achieved mainstream in 1993 with the Intel Pentium and in 1997 the embedded CPU market with the ARMv5TE. In 2015, even sub-dollar SoC split the L1 cache. They also have L2 caches and, for larger processors, L3 caches as well. The L2 cache is usually not split and acts as a common repository for the already split L1 cache. Every core of a multi-core processor has a dedicated L1 cache and is usually not shared between the cores. The L2 cache, and

higher-level caches, may be shared between the cores. L4 cache is currently uncommon, and is generally on (a form of) dynamic random-access memory (DRAM), rather than on static random-access memory (SRAM), on a separate die or chip (exceptionally, the form, eDRAM is used for all levels of cache, down to L1). That was also the case historically with L1, while bigger chips have allowed integration of it and generally all cache levels, with the possible exception of the last level. Each extra level of cache tends to be bigger and optimized differently.

Caches (like for RAM historically) have generally been sized in powers of: 2, 4, 8, 16 etc. KiB; when up to MiBsizes (i.e. for larger non-L1), very early on the pattern broke down, to allow for larger caches without being forced into the doubling-in-size paradigm, with e.g. Intel Core 2 Duo with 3 MiB L2 cache in April 2008. Much later however for L1 sizes, that still only count in small number of KiB, however IBM zEC12 from 2012 is an exception, to gain unusually large 96 KiB L1 data cache for its time, and e.g. the IBM z13 having a 96 KiB L1 instruction cache (and 128 KiB L1 data cache),[5] and Intel Ice Lake-based processors from 2018, having 48 KiB L1 data cache and 48 KiB L1 instruction cache. In 2020, some Intel Atom CPUs (with up to 24 cores) have (multiple of) 4.5 MiB and 15 MiB cache sizes.[6][7]

Cache entries

Data is transferred between memory and cache in blocks of fixed size, called *cache lines* or *cache blocks*. When a cache line is copied from memory into the cache, a cache entry is created. The cache entry will include the copied data as well as the requested memory location (called a tag).

When the processor needs to read or write a location in memory, it first checks for a corresponding entry in the cache. The cache checks for the contents of the requested memory location in any cache lines that might contain that address. If the processor finds that the memory location is in the cache, a **cache hit** has occurred. However, if the processor does not find the memory location in the cache, a **cache miss** has occurred. In the case of a cache hit, the processor immediately reads or writes the data in the cache line. For a cache miss, the cache allocates a new entry and copies data from main memory, then the request is fulfilled from the contents of the cache.

Policies

Replacement policies

To make room for the new entry on a cache miss, the cache may have to evict one of the existing entries. The heuristic it uses to choose the entry to evict is called the replacement policy. The fundamental problem with any replacement policy is that it must predict which existing cache entry is least likely to be used in the future. Predicting the future is difficult, so there is no perfect method to choose among the variety of replacement policies available. One popular replacement policy, least-recently used (LRU), replaces the least recently accessed entry.

Marking some memory ranges as non-cacheable can improve performance, by avoiding caching of memory regions that are rarely re-accessed. This avoids the overhead of loading something into the cache without having any reuse. Cache entries may also be disabled or locked depending on the context.

Write policies

If data is written to the cache, at some point it must also be written to main memory; the timing of this write is known as the write policy. In a write-through cache, every write to the cache causes a write to main memory. Alternatively, in a write-back or copy-back cache, writes are not immediately mirrored to the main memory, and the cache instead tracks which locations have been written over, marking them as dirty. The data in these locations is written back to the main memory only when that data is evicted from the cache. For this reason, a read miss in a write-back cache may sometimes require two memory accesses to service: one to first write the dirty location to main memory, and then another to read the new location from memory. Also, a write to a main memory location that is not yet mapped in a write-back cache may evict an already dirty location, thereby freeing that cache space for the new memory location.

There are intermediate policies as well. The cache may be write-through, but the writes may be held in a store data queue temporarily, usually so multiple stores can be processed together (which can reduce bus turnarounds and improve bus utilization).

Cached data from the main memory may be changed by other entities (e.g., peripherals using direct memory access(DMA) or another core in a multi-core processor), in which case the copy in the cache may become out-of-date or stale. Alternatively, when a CPU in a multiprocessor system updates data in the cache, copies of data in caches associated with other CPUs become stale. Communication protocols between the cache managers that keep the data consistent are known as cache coherence protocols.

Cache performance

Cache performance measurement has become important in recent times where the speed gap between the memory performance and the processor performance is increasing exponentially. The cache was introduced to reduce this speed gap. Thus knowing how well the cache is able to bridge the gap in the speed of processor and memory becomes important, especially in high-performance systems. The cache hit rate and the cache miss rate play an important role in determining this performance. To improve the cache performance, reducing the miss rate becomes one of the necessary steps among other steps. Decreasing the access time to the cache also gives a boost to its performance.

CPU stalls

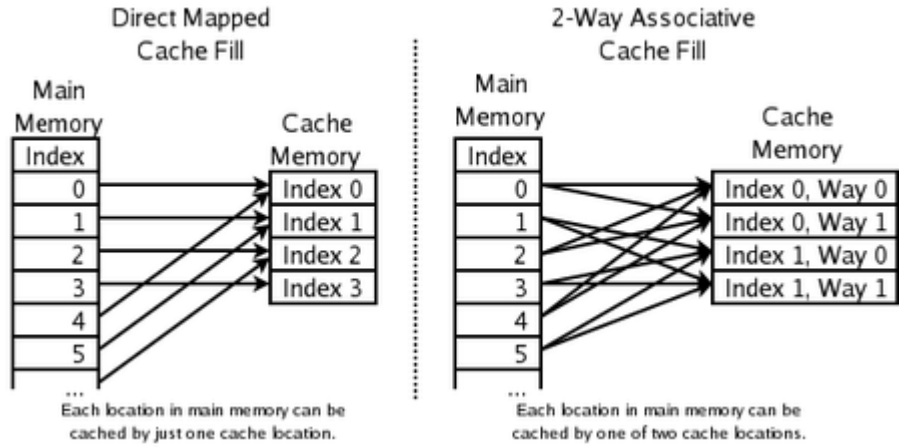
The time taken to fetch one cache line from memory (read latency due to a cache miss) matters because the CPU will run out of things to do while waiting for the cache line. When a CPU reaches this state, it is called a stall. As CPUs become faster compared to main memory, stalls due to cache misses displace more potential computation; modern CPUs can execute hundreds of instructions in the time taken to fetch a single cache line from main memory.

Various techniques have been employed to keep the CPU busy during this time, including out-of-order execution in which the CPU attempts to execute independent instructions after the instruction that is waiting for the cache miss data. Another technology, used by many processors, is simultaneous multithreading (SMT), which allows an alternate thread to use the CPU core while the first thread waits for required CPU resources to become available.

Associativity

The placement policy decides where in the cache a copy of a particular entry of main memory will go. If the placement policy is free to choose any entry in the cache to hold the copy, the cache is called *fully associative*. At the other extreme, if each entry in main memory can go in just one place in the cache, the cache is *direct mapped*. Many caches implement a compromise in which each entry in main memory can go to any one of N

places in the cache, and are described as N-way set associative.[8] For example, the level-1 data cache in an AMD Athlon is two-way set associative, which means that any particular location in main memory can be cached in either of two locations in the level-1 data cache.



An illustration of different ways in which memory locations can be cached by particular cache locations

Choosing the right value of associativity involves a trade-off. If there are ten places to which the placement policy could have mapped a memory location, then to check if that location is in the cache, ten cache entries must be searched. Checking more places takes more power and chip area, and potentially more time. On the other hand, caches with more associativity suffer fewer misses (see conflict misses, below), so that the CPU wastes less time reading from the slow main memory. The general guideline is that doubling the associativity, from direct mapped to two-way, or from two-way to four-way, has about the same effect on raising the hit rate as doubling the cache size. However, increasing associativity more than four does not improve hit rate as much,[9] and are generally done for other reasons (see virtual aliasing, below). Some CPUs can dynamically reduce the associativity of their caches in low-power states, which acts as a power-saving measure.[10]

In order of worse but simple to better but complex:

- Direct mapped cache – good best-case time, but unpredictable in worst case
- Two-way set associative cache
- Two-way skewed associative cache[11]
- Four-way set associative cache
- Eight-way set associative cache, a common choice for later implementations
- 12-way set associative cache, similar to eight-way
- Fully associative cache – the best miss rates, but practical only for a small number of entries

Direct-mapped cache

In this cache organization, each location in main memory can go in only one entry in the cache. Therefore, a direct-mapped cache can also be called a "one-way set associative" cache. It does not have a placement policy as such, since there is no choice of which cache entry's contents to evict. This means that if two locations map to the same entry, they may continually knock each other out. Although simpler, a direct-mapped cache needs to be much larger than an associative one to give comparable performance, and it is more unpredictable. Let x be block number in cache, y be block number of memory, and n be number of blocks in cache, then mapping is done with the help of the equation $x = y \bmod n$.

Two-way set associative cache

If each location in main memory can be cached in either of two locations in the cache, one logical question is: *which one of the two?* The simplest and most commonly used scheme, shown in the right-hand diagram above, is to use the least significant bits of the memory location's index as the index for the cache memory, and to have two entries for each index. One benefit of this scheme is that the tags stored in the cache do not have to include that part of the main memory address which is implied by the cache memory's index. Since the cache tags have fewer bits, they require fewer transistors, take less space on the processor circuit board or on the microprocessor chip, and can be read and compared faster. Also LRU is especially simple since only one bit needs to be stored for each pair.

Speculative execution

One of the advantages of a direct mapped cache is that it allows simple and fast speculation. Once the address has been computed, the one cache index which might have a copy of that location in memory is known. That cache entry can be read, and the processor can continue to work with that data before it finishes checking that the tag actually matches the requested address.

The idea of having the processor use the cached data before the tag match completes can be applied to associative caches as well. A subset of the tag, called a *hint*, can be used to pick just one of the possible cache entries mapping to the requested address. The entry selected by the hint can then be used in parallel with checking the full tag. The hint technique works best when used in the context of address translation, as explained below.

Two-way skewed associative cache

Other schemes have been suggested, such as the *skewed cache*,[11] where the index for way 0 is direct, as above, but the index for way 1 is formed with a hash function. A good hash function has the property that addresses which conflict with the direct mapping tend not to conflict when mapped with the hash function, and so it is less likely that a program will suffer from an unexpectedly large number of conflict misses due to a pathological access pattern. The downside is extra latency from computing the hash function.[12] Additionally, when it comes time to load a new line and evict an old line, it may be difficult to determine which existing line was least recently used, because the new line conflicts with data at different indexes in each way; LRU tracking for non-skewed caches is usually done on a per-set basis. Nevertheless, skewed-associative caches have major advantages over conventional set-associative ones.[13]

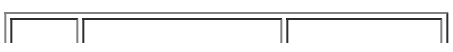
Pseudo-associative cache

A true set-associative cache tests all the possible ways simultaneously, using something like a content addressable memory. A pseudo-associative cache tests each possible way one at a time. A hash-rehash cache and a column-associative cache are examples of a pseudo-associative cache.

In the common case of finding a hit in the first way tested, a pseudo-associative cache is as fast as a direct-mapped cache, but it has a much lower conflict miss rate than a direct-mapped cache, closer to the miss rate of a fully associative cache. [12]

Cache entry structure

Cache row entries usually have the following structure:



tag	data block	flag bits

The *data block* (cache line) contains the actual data fetched from the main memory. The *tag* contains (part of) the address of the actual data fetched from the main memory. The flag bits are discussed below.

The "size" of the cache is the amount of main memory data it can hold. This size can be calculated as the number of bytes stored in each data block times the number of blocks stored in the cache. (The tag, flag and error correction code bits are not included in the size,[14] although they do affect the physical area of a cache.)

An effective memory address which goes along with the cache line (memory block) is split (MSB to LSB) into the tag, the index and the block offset.[15][16]

tag	index	block offset

The index describes which cache set that the data has been put in. The index length is $\lceil \log_2(s) \rceil$ bits for s cache sets.

The block offset specifies the desired data within the stored data block within the cache row. Typically the effective address is in bytes, so the block offset length is $\lceil \log_2(b) \rceil$ bits, where b is the number of bytes per data block. The tag contains the most significant bits of the address, which are checked against all rows in the current set (the set has been retrieved by index) to see if this set contains the requested address. If it does, a cache hit occurs. The tag length in bits is as follows:

```
tag_length = address_length - index_length - block_offset_length
```

Some authors refer to the block offset as simply the "offset"[17] or the "displacement".[18][19]

Example

The original Pentium 4 processor had a four-way set associative L1 data cache of 8 KiB in size, with 64-byte cache blocks. Hence, there are 8 KiB / 64 = 128 cache blocks. The number of sets is equal to the number of cache blocks divided by the number of ways of associativity, what leads to 128 / 4 = 32 sets, and hence $2^5 = 32$ different indices. There are $2^6 = 64$ possible offsets. Since the CPU address is 32 bits wide, this implies 32 - 5 - 6 = 21 bits for the tag field.

The original Pentium 4 processor also had an eight-way set associative L2 integrated cache 256 KiB in size, with 128-byte cache blocks. This implies 32 - 8 - 7 = 17 bits for the tag field.[17]

Flag bits

An instruction cache requires only one flag bit per cache row entry: a valid bit. The valid bit indicates whether or not a cache block has been loaded with valid data.

On power-up, the hardware sets all the valid bits in all the caches to "invalid". Some systems also set a valid bit to "invalid" at other times, such as when multi-master bus snooping hardware in the cache of one processor hears an address broadcast from some other processor, and realizes that certain data blocks in the local cache are now stale and should be marked invalid.

A data cache typically requires two flag bits per cache line – a valid bit and a dirty bit. Having a dirty bit set indicates that the associated cache line has been changed since it was read from main memory ("dirty"), meaning that the processor has written data to that line and the new value has not propagated all the way to main memory.

Cache miss

A cache miss is a failed attempt to read or write a piece of data in the cache, which results in a main memory access with much longer latency. There are three kinds of cache misses: instruction read miss, data read miss, and data write miss.

Cache read misses from an *instruction* cache generally cause the largest delay, because the processor, or at least the thread of execution, has to wait (stall) until the instruction is fetched from main memory. *Cache read misses* from a *data* cache usually cause a smaller delay, because instructions not dependent on the cache read can be issued and continue execution until the data is returned from main memory, and the dependent instructions can resume execution. *Cache write misses* to a *data* cache generally cause the shortest delay, because the write can be queued and there are few limitations on the execution of subsequent instructions; the processor can continue until the queue is full. For a detailed introduction to the types of misses, see cache performance measurement and metric.

Address translation

Most general purpose CPUs implement some form of virtual memory. To summarize, either each program running on the machine sees its own simplified address space, which contains code and data for that program only, or all programs run in a common virtual address space. A program executes by calculating, comparing, reading and writing to addresses of its virtual address space, rather than addresses of physical address space, making programs simpler and thus easier to write.

Virtual memory requires the processor to translate virtual addresses generated by the program into physical addresses in main memory. The portion of the processor that does this translation is known as the memory management unit (MMU). The fast path through the MMU can perform those translations stored in the translation lookaside buffer (TLB), which is a cache of mappings from the operating system's page table, segment table, or both.

For the purposes of the present discussion, there are three important features of address translation:

- *Latency:* The physical address is available from the MMU some time, perhaps a few cycles, after the virtual address is available from the address generator.
- *Aliasing:* Multiple virtual addresses can map to a single physical address. Most processors guarantee that all updates to that single physical address will happen in program order. To deliver on that guarantee, the processor must ensure that only one copy of a physical address resides in the cache at any given time.
- *Granularity:* The virtual address space is broken up into pages. For instance, a 4 GiB virtual address space might be cut up into 1,048,576 pages of 4 KiB size, each of which can be independently mapped. There may be multiple page sizes supported; see virtual memory for elaboration.

Some early virtual memory systems were very slow because they required an access to the page table (held in main memory) before every programmed access to main memory.[NB 1] With no caches, this effectively cut the speed of memory access in half. The first hardware cache used in a computer system was not actually a data or instruction cache, but rather a TLB.[21]

Caches can be divided into four types, based on whether the index or tag correspond to physical or virtual addresses:

- *Physically indexed, physically tagged* (PIPT) caches use the physical address for both the index and the tag. While this is simple and avoids problems with aliasing, it is also slow, as the physical address must be looked up (which could involve a TLB miss and access to main memory) before that address can be looked up in the cache.
- *Virtually indexed, virtually tagged* (VIVT) caches use the virtual address for both the index and the tag. This caching scheme can result in much faster lookups, since the MMU does not need to be consulted first to determine the physical address for a given virtual address. However, VIVT suffers from aliasing problems, where several different virtual addresses may refer to the same physical address. The result is that such addresses would be cached separately despite referring to the same memory, causing coherency problems. Although solutions to this problem exist [22] they do not work for standard coherence protocols. Another problem is homonyms, where the same virtual address maps to several different physical addresses. It is not possible to distinguish these mappings merely by looking at the virtual index itself, though potential solutions include: flushing the cache after a context switch, forcing address spaces to be non-overlapping, tagging the virtual address with an address space ID (ASID). Additionally, there is a problem that virtual-to-physical mappings can change, which would require flushing cache lines, as the VAs would no longer be valid. All these issues are absent if tags use physical addresses (VIPT).
- *Virtually indexed, physically tagged* (VIPT) caches use the virtual address for the index and the physical address in the tag. The advantage over PIPT is lower latency, as the cache line can be looked up in parallel with the TLB translation, however the tag cannot be compared until the physical address is available. The advantage over VIVT is that since the tag has the physical address, the cache can detect homonyms. Theoretically, VIPT requires more tags bits because some of the index bits could differ between the virtual and physical addresses (for example bit 12 and above for 4 KiB pages) and would have to be included both in the virtual index and in the physical tag. In practice this is not an issue because, in order to avoid coherency problems, VIPT caches are designed to have no such index bits (e.g., by limiting the total number of bits for the index and the block offset to 12 for 4 KiB pages); this limits the size of VIPT caches to the page size times the associativity of the cache.
- *Physically indexed, virtually tagged* (PIVT) caches are often claimed in literature to be useless and non-existing.[23] However, the MIPS R6000 uses this cache type as the sole known implementation.[24] The R6000 is implemented in emitter-coupled logic, which is an extremely fast technology not suitable for large memories such as a TLB. The R6000 solves the issue by putting the TLB memory into a reserved part of the second-level cache having a tiny, high-speed TLB "slice" on chip. The cache is indexed by the physical address obtained from the TLB slice. However, since the TLB slice only translates those virtual address bits that are necessary to index the cache and does not use any tags, false cache hits may occur, which is solved by tagging with the virtual address.

The speed of this recurrence (the *load latency*) is crucial to CPU performance, and so most modern level-1 caches are virtually indexed, which at least allows the MMU's TLB lookup to proceed in parallel with fetching the data from the cache RAM.

But virtual indexing is not the best choice for all cache levels. The cost of dealing with virtual aliases grows with cache size, and as a result most level-2 and larger caches are physically indexed.

Caches have historically used both virtual and physical addresses for the cache tags, although virtual tagging is now uncommon. If the TLB lookup can finish before the cache RAM lookup, then the physical address is available in time for tag compare, and there is no need for virtual tagging. Large caches, then, tend to be physically tagged, and only small, very low latency caches are virtually tagged. In recent general-purpose CPUs, virtual tagging has been superseded by vhints, as described below.

Homonym and synonym problems

A cache that relies on virtual indexing and tagging becomes inconsistent after the same virtual address is mapped into different physical addresses (homonym), which can be solved by using physical address for tagging, or by storing the address space identifier in the cache line. However, the latter approach does not help against the synonym problem, in which several cache lines end up storing data for the same physical address. Writing to such locations may update only one location in the cache, leaving the others with inconsistent data. This issue may be solved by using non-overlapping memory layouts for different address spaces, or otherwise the cache (or a part of it) must be flushed when the mapping changes.[25]

Virtual tags and vhints

The great advantage of virtual tags is that, for associative caches, they allow the tag match to proceed before the virtual to physical translation is done. However, coherence probes and evictions present a physical address for action. The hardware must have some means of converting the physical addresses into a cache index, generally by storing physical tags as well as virtual tags. For comparison, a physically tagged cache does not need to keep virtual tags, which is simpler. When a virtual to physical mapping is deleted from the TLB, cache entries with those virtual addresses will have to be flushed somehow. Alternatively, if cache entries are allowed on pages not mapped by the TLB, then those entries will have to be flushed when the access rights on those pages are changed in the page table.

It is also possible for the operating system to ensure that no virtual aliases are simultaneously resident in the cache. The operating system makes this guarantee by enforcing page coloring, which is described below. Some early RISC processors (SPARC, RS/6000) took this approach. It has not been used recently, as the hardware cost of detecting and evicting virtual aliases has fallen and the software complexity and performance penalty of perfect page coloring has risen.

It can be useful to distinguish the two functions of tags in an associative cache: they are used to determine which way of the entry set to select, and they are used to determine if the cache hit or missed. The second function must always be correct, but it is permissible for the first function to guess, and get the wrong answer occasionally.

Some processors (e.g. early SPARCs) have caches with both virtual and physical tags. The virtual tags are used for way selection, and the physical tags are used for determining hit or miss. This kind of cache enjoys the latency advantage of a virtually tagged cache, and the simple software interface of a physically tagged cache. It bears the added cost of duplicated tags, however. Also, during miss processing, the alternate ways of the cache line indexed have to be probed for virtual aliases and any matches evicted.

The extra area (and some latency) can be mitigated by keeping *virtual hints* with each cache entry instead of virtual tags. These hints are a subset or hash of the virtual tag, and are used for selecting the way of the cache from which to get data and a physical tag. Like a virtually tagged cache, there may be a virtual hint match but physical tag mismatch, in which case the cache entry with the matching hint must be evicted so that cache accesses after the cache fill at this address will have just one hint match. Since virtual hints have fewer bits than virtual tags distinguishing them from one another, a virtually hinted cache suffers more conflict misses than a virtually tagged cache.

Perhaps the ultimate reduction of virtual hints can be found in the Pentium 4 (Willamette and Northwood cores). In these processors the virtual hint is effectively two bits, and the cache is four-way set associative. Effectively, the hardware maintains a simple permutation from virtual address to cache index, so that no content-addressable memory (CAM) is necessary to select the right one of the four ways fetched.

Page coloring

Large physically indexed caches (usually secondary caches) run into a problem: the operating system rather than the application controls which pages collide with one another in the cache. Differences in page allocation from one program run to the next lead to differences in the cache collision patterns, which can lead to very large differences in program performance. These differences can make it very difficult to get a consistent and repeatable timing for a benchmark run.

To understand the problem, consider a CPU with a 1 MiB physically indexed direct-mapped level-2 cache and 4 KiB virtual memory pages. Sequential physical pages map to sequential locations in the cache until after 256 pages the pattern wraps around. We can label each physical page with a color of 0–255 to denote where in the cache it can go. Locations within physical pages with different colors cannot conflict in the cache.

Programmers attempting to make maximum use of the cache may arrange their programs' access patterns so that only 1 MiB of data need be cached at any given time, thus avoiding capacity misses. But they should also ensure that the access patterns do not have conflict misses. One way to think about this problem is to divide up the virtual pages the program uses and assign them virtual colors in the same way as physical colors were assigned to physical pages before. Programmers can then arrange the access patterns of their code so that no two pages with the same virtual color are in use at the same time. There is a wide literature on such optimizations (e.g. loop nest optimization), largely coming from the High Performance Computing (HPC) community.

The snag is that while all the pages in use at any given moment may have different virtual colors, some may have the same physical colors. In fact, if the operating system assigns physical pages to virtual pages randomly and uniformly, it is extremely likely that some pages will have the same physical color, and then locations from those pages will collide in the cache (this is the birthday paradox).

The solution is to have the operating system attempt to assign different physical color pages to different virtual colors, a technique called *page coloring*. Although the actual mapping from virtual to physical color is irrelevant to system performance, odd mappings are difficult to keep track of and have little benefit, so most approaches to page coloring simply try to keep physical and virtual page colors the same.

If the operating system can guarantee that each physical page maps to only one virtual color, then there are no virtual aliases, and the processor can use virtually indexed caches with no need for extra virtual alias probes during miss handling. Alternatively, the OS can flush a page from the cache whenever it changes from one virtual color to another. As mentioned above, this approach was used for some early SPARC and RS/6000 designs.

Cache hierarchy in a modern processor

Modern processors have multiple interacting on-chip caches. The operation of a particular cache can be completely specified by the cache size, the cache block size, the number of blocks in a set, the cache set replacement policy, and the cache write policy (write-through or write-back).[17]

Memory hierarchy of an AMD Bulldozer server

While all of the cache blocks in a particular cache are the same size and have the same associativity, typically the "lower-level" caches (called Level 1 cache) have a smaller number of blocks, smaller block size, and fewer blocks in a

set, but have very short access times. "Higher-level" caches (i.e. Level 2 and above) have progressively larger numbers of blocks, larger block size, more blocks in a set, and relatively longer access times, but are still much faster than main memory.

Cache entry replacement policy is determined by a cache algorithm selected to be implemented by the processor designers. In some cases, multiple algorithms are provided for different kinds of work loads.

Specialized caches

Pipelined CPUs access memory from multiple points in the pipeline: instruction fetch, virtual-to-physical address translation, and data fetch (see classic RISC pipeline). The natural design is to use different physical caches for each of these points, so that no one physical resource has to be scheduled to service two points in the pipeline. Thus the pipeline naturally ends up with at least three separate caches (instruction, TLB, and data), each specialized to its particular role.

Victim cache

A **victim cache** is a cache used to hold blocks evicted from a CPU cache upon replacement. The victim cache lies between the main cache and its refill path, and holds only those blocks of data that were evicted from the main cache. The victim cache is usually fully associative, and is intended to reduce the number of conflict misses. Many commonly used programs do not require an associative mapping for all the accesses. In fact, only a small fraction of the memory accesses of the program require high associativity. The victim cache exploits this property by providing high associativity to only these accesses. It was introduced by Norman Jouppi from DEC in 1990.[26]

Intel's *Crystalwell*[27] variant of its Haswell processors introduced an on-package 128 MB eDRAM Level 4 cache which serves as a victim cache to the processors' Level 3 cache.[28] In the Skylake microarchitecture the Level 4 cache no longer works as a victim cache.[29]

Trace cache

One of the more extreme examples of cache specialization is the **trace cache** (also known as *execution trace cache*) found in the Intel Pentium 4 microprocessors. A trace cache is a mechanism for increasing the instruction fetch bandwidth and decreasing power consumption (in the case of the Pentium 4) by storing traces of instructionsthat have already been fetched and decoded.[30]

A trace cache stores instructions either after they have been decoded, or as they are retired. Generally, instructions are added to trace caches in groups representing either individual basic blocks or dynamic instruction traces. The Pentium 4's trace cache stores micro-operations resulting from decoding x86 instructions, providing also the functionality of a micro-operation cache. Having this, the next time an instruction is needed, it does not have to be decoded into micro-ops again.[31]:63–68

Write Coalescing Cache (WCC)

Write Coalescing Cache[32] is a special cache that is part of L2 cache in AMD's Bulldozer microarchitecture. Stores from both L1D caches in the module go through the WCC, where they are buffered and coalesced. The WCC's task is reducing number of writes to the L2 cache.

Micro-operation (μop or uop) cache

A **micro-operation cache** (**μop cache**, **uop cache** or **UC**)[33] is a specialized cache that stores micro-operations of decoded instructions, as received directly from the instruction decoders or from the instruction cache. When an instruction needs to be decoded, the μop cache is checked for its decoded form which is re-used if cached; if it is not available, the instruction is decoded and then cached.

One of the early works describing μop cache as an alternative frontend for the Intel P6 processor family is the 2001 paper *"Micro-Operation Cache: A Power Aware Frontend for Variable Instruction Length ISA"*.[34] Later, Intel included μop caches in its Sandy Bridge processors and in successive microarchitectures like Ivy Bridge and Haswell.[31]:121–123[35] AMD implemented a μop cache in their Zen microarchitecture.[36]

Fetching complete pre-decoded instructions eliminates the need to repeatedly decode variable length complex instructions into simpler fixed-length micro-operations, and simplifies the process of predicting, fetching, rotating and aligning fetched instructions. A μop cache effectively offloads the fetch and decode hardware, thus decreasing power consumption and improving the frontend supply of decoded micro-operations. The μop cache also increases performance by more consistently delivering decoded micro-operations to the backend and eliminating various bottlenecks in the CPU's fetch and decode logic.[34][35]

A μop cache has many similarities with a trace cache, although a μop cache is much simpler thus providing better power efficiency; this makes it better suited for implementations on battery-powered devices. The main disadvantage of the trace cache, leading to its power inefficiency, is the hardware complexity required for its heuristic deciding on caching and reusing dynamically created instruction traces.[37]

Branch target instruction cache

A **branch target cache** or **branch target instruction cache**, the name used on ARM microprocessors,[38] is a specialized cache which holds the first few instructions at the destination of a taken branch. This is used by low-powered processors which do not need a normal instruction cache because the memory system is capable of delivering instructions fast enough to satisfy the CPU without one. However, this only applies to consecutive instructions in sequence; it still takes several cycles of latency to restart instruction fetch at a new address, causing a few cycles of pipeline bubble after a control transfer. A branch target cache provides instructions for those few cycles avoiding a delay after most taken branches.

This allows full-speed operation with a much smaller cache than a traditional full-time instruction cache.

Smart cache

Smart cache is a level 2 or level 3 caching method for multiple execution cores, developed by Intel.

Smart Cache shares the actual cache memory between the cores of a multi-core processor. In comparison to a dedicated per-core cache, the overall cache miss rate decreases when not all cores need equal parts of the cache space. Consequently, a single core can use the full level 2 or level 3 cache, if the other cores are inactive.[39]Furthermore, the shared cache makes it faster to share memory among different execution cores.[40]

Multi-level caches

Another issue is the fundamental tradeoff between cache latency and hit rate. Larger caches have better hit rates but longer latency. To address this tradeoff, many computers use multiple levels of cache, with small fast caches backed up by larger, slower caches. Multi-level caches generally operate by checking the fastest, *level 1* (**L1**)

cache first; if it hits, the processor proceeds at high speed. If that smaller cache misses, the next fastest cache (*level 2*, **L2**) is checked, and so on, before accessing external memory.

As the latency difference between main memory and the fastest cache has become larger, some processors have begun to utilize as many as three levels of on-chip cache. Price-sensitive designs used this to pull the entire cache hierarchy on-chip, but by the 2010s some of the highest-performance designs returned to having large off-chip caches, which is often implemented in eDRAM and mounted on a multi-chip module, as a fourth cache level. In rare cases, such as in the mainframe CPU IBM z15 (2019), all levels down to L1 are implemented by eDRAM, replacing SRAM entirely (for cache, SRAM is still used for registers). The ARM-based Apple M1 has a 192 KB L1 cache for each of the four high-performance cores, an unusually large amount; however the four high-efficiency cores only have 128 KB.

The benefits of L3 and L4 caches depend on the application's access patterns. Examples of products incorporating L3 and L4 caches include the following:

- Alpha 21164 (1995) has 1 to 64 MB off-chip L3 cache.
- IBM POWER4 (2001) has off-chip L3 caches of 32 MB per processor, shared among several processors.
- Itanium 2 (2003) has a 6 MB unified level 3 (L3) cache on-die; the Itanium 2 (2003) MX 2 module incorporates two Itanium 2 processors along with a shared 64 MB L4 cache on a multi-chip modulethat was pin compatible with a Madison processor.
- Intel's Xeon MP product codenamed "Tulsa" (2006) features 16 MB of on-die L3 cache shared between two processor cores.
- AMD Phenom II (2008) has up to 6 MB on-die unified L3 cache.
- Intel Core i7 (2008) has an 8 MB on-die unified L3 cache that is inclusive, shared by all cores.
- Intel Haswell CPUs with integrated Intel Iris Pro Graphics have 128 MB of eDRAM acting essentially as an L4 cache.[41]

Finally, at the other end of the memory hierarchy, the CPU register file itself can be considered the smallest, fastest cache in the system, with the special characteristic that it is scheduled in software—typically by a compiler, as it allocates registers to hold values retrieved from main memory for, as an example, loop nest optimization. However, with register renaming most compiler register assignments are reallocated dynamically by hardware at runtime into a register bank, allowing the CPU to break false data dependencies and thus easing pipeline hazards.

Register files sometimes also have hierarchy: The Cray-1 (circa 1976) had eight address "A" and eight scalar data "S" registers that were generally usable. There was also a set of 64 address "B" and 64 scalar data "T" registers that took longer to access, but were faster than main memory. The "B" and "T" registers were provided because the Cray-1 did not have a data cache. (The Cray-1 did, however, have an instruction cache.)

Multi-core chips

When considering a chip with multiple cores, there is a question of whether the caches should be shared or local to each core. Implementing shared cache inevitably introduces more wiring and complexity. But then, having one cache per *chip*, rather than *core*, greatly reduces the amount of space needed, and thus one can include a larger cache.

Typically, sharing the L1 cache is undesirable because the resulting increase in latency would make each core run considerably slower than a single-core chip. However, for the highest-level cache, the last one called before accessing memory, having a global cache is desirable for several reasons, such as allowing a single core to use the whole cache, reducing data redundancy by making it possible for different processes or threads to share

cached data, and reducing the complexity of utilized cache coherency protocols.[42] For example, an eight-core chip with three levels may include an L1 cache for each core, one intermediate L2 cache for each pair of cores, and one L3 cache shared between all cores.

Shared highest-level cache, which is called before accessing memory, is usually referred to as the *last level cache*(LLC). Additional techniques are used for increasing the level of parallelism when LLC is shared between multiple cores, including slicing it into multiple pieces which are addressing certain ranges of memory addresses, and can be accessed independently.[43]

Separate versus unified

In a separate cache structure, instructions and data are cached separately, meaning that a cache line is used to cache either instructions or data, but not both; various benefits have been demonstrated with separate data and instruction translation lookaside buffers.[44] In a unified structure, this constraint is not present, and cache lines can be used to cache both instructions and data.

Exclusive versus inclusive

Multi-level caches introduce new design decisions. For instance, in some processors, all data in the L1 cache must also be somewhere in the L2 cache. These caches are called *strictly inclusive*. Other processors (like the AMD Athlon) have *exclusive* caches: data is guaranteed to be in at most one of the L1 and L2 caches, never in both. Still other processors (like the Intel Pentium II, III, and 4) do not require that data in the L1 cache also reside in the L2 cache, although it may often do so. There is no universally accepted name for this intermediate policy;[45][46] two common names are "non-exclusive" and "partially-inclusive".

The advantage of exclusive caches is that they store more data. This advantage is larger when the exclusive L1 cache is comparable to the L2 cache, and diminishes if the L2 cache is many times larger than the L1 cache. When the L1 misses and the L2 hits on an access, the hitting cache line in the L2 is exchanged with a line in the L1. This exchange is quite a bit more work than just copying a line from L2 to L1, which is what an inclusive cache does.[46]

One advantage of strictly inclusive caches is that when external devices or other processors in a multiprocessor system wish to remove a cache line from the processor, they need only have the processor check the L2 cache. In cache hierarchies which do not enforce inclusion, the L1 cache must be checked as well. As a drawback, there is a correlation between the associativities of L1 and L2 caches: if the L2 cache does not have at least as many ways as all L1 caches together, the effective associativity of the L1 caches is restricted. Another disadvantage of inclusive cache is that whenever there is an eviction in L2 cache, the (possibly) corresponding lines in L1 also have to get evicted in order to maintain inclusiveness. This is quite a bit of work, and would result in a higher L1 miss rate.[46]

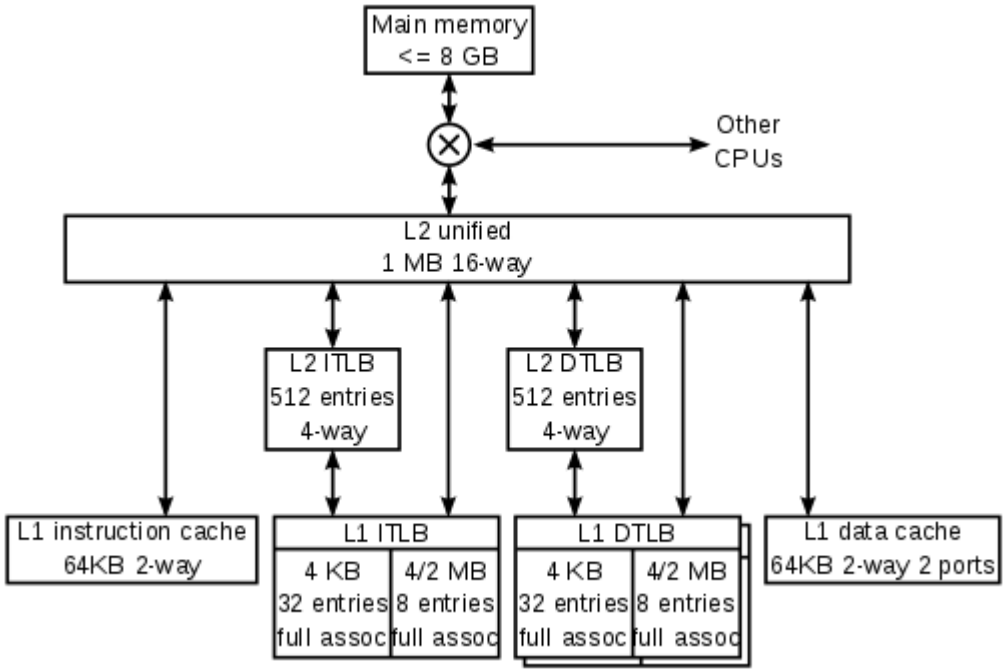
Another advantage of inclusive caches is that the larger cache can use larger cache lines, which reduces the size of the secondary cache tags. (Exclusive caches require both caches to have the same size cache lines, so that cache lines can be swapped on a L1 miss, L2 hit.) If the secondary cache is an order of magnitude larger than the primary, and the cache data is an order of magnitude larger than the cache tags, this tag area saved can be comparable to the incremental area needed to store the L1 cache data in the L2.[47]

Scratchpad memory

Scratchpad memory (SPM), also known as scratchpad, scratchpad RAM or local store in computer terminology, is a high-speed internal memory used for temporary storage of calculations, data, and other work in progress.

Example: the K8

To illustrate both specialization and multi-level caching, here is the cache hierarchy of the K8 core in the AMD Athlon 64 CPU.[48]



Cache hierarchy of the K8 core in the AMD Athlon 64 CPU.

The K8 has four specialized caches: an instruction cache, an instruction TLB, a data TLB, and a data cache. Each of these caches is specialized:

- The instruction cache keeps copies of 64-byte lines of memory, and fetches 16 bytes each cycle. Each byte in this cache is stored in ten bits rather than eight, with the extra bits marking the boundaries of instructions (this is an example of predecoding). The cache has only parity protection rather than ECC, because parity is smaller and any damaged data can be replaced by fresh data fetched from memory (which always has an up-to-date copy of instructions).
- The instruction TLB keeps copies of page table entries (PTEs). Each cycle's instruction fetch has its virtual address translated through this TLB into a physical address. Each entry is either four or eight bytes in memory. Because the K8 has a variable page size, each of the TLBs is split into two sections, one to keep PTEs that map 4 KB pages, and one to keep PTEs that map 4 MB or 2 MB pages. The split allows the fully associative match circuitry in each section to be simpler. The operating system maps different sections of the virtual address space with different size PTEs.
- The data TLB has two copies which keep identical entries. The two copies allow two data accesses per cycle to translate virtual addresses to physical addresses. Like the instruction TLB, this TLB is split into two kinds of entries.
- The data cache keeps copies of 64-byte lines of memory. It is split into 8 banks (each storing 8 KB of data), and can fetch two 8-byte data each cycle so long as those data are in different banks. There are two copies of the tags, because each 64-byte line is spread among all eight banks. Each tag copy handles one of the two accesses per cycle.

The K8 also has multiple-level caches. There are second-level instruction and data TLBs, which store only PTEs mapping 4 KB. Both instruction and data caches, and the various TLBs, can fill from the large **unified** L2 cache. This cache is exclusive to both the L1 instruction and data caches, which means that any 8-byte line can only be

in one of the L1 instruction cache, the L1 data cache, or the L2 cache. It is, however, possible for a line in the data cache to have a PTE which is also in one of the TLBs—the operating system is responsible for keeping the TLBs coherent by flushing portions of them when the page tables in memory are updated.

The K8 also caches information that is never stored in memory—prediction information. These caches are not shown in the above diagram. As is usual for this class of CPU, the K8 has fairly complex branch prediction, with tables that help predict whether branches are taken and other tables which predict the targets of branches and jumps. Some of this information is associated with instructions, in both the level 1 instruction cache and the unified secondary cache.

The K8 uses an interesting trick to store prediction information with instructions in the secondary cache. Lines in the secondary cache are protected from accidental data corruption (e.g. by an alpha particle strike) by either ECCor parity, depending on whether those lines were evicted from the data or instruction primary caches. Since the parity code takes fewer bits than the ECC code, lines from the instruction cache have a few spare bits. These bits are used to cache branch prediction information associated with those instructions. The net result is that the branch predictor has a larger effective history table, and so has better accuracy.

More hierarchies

Other processors have other kinds of predictors (e.g., the store-to-load bypass predictor in the DEC Alpha 21264), and various specialized predictors are likely to flourish in future processors.

These predictors are caches in that they store information that is costly to compute. Some of the terminology used when discussing predictors is the same as that for caches (one speaks of a **hit** in a branch predictor), but predictors are not generally thought of as part of the cache hierarchy.

The K8 keeps the instruction and data caches **coherent** in hardware, which means that a store into an instruction closely following the store instruction will change that following instruction. Other processors, like those in the Alpha and MIPS family, have relied on software to keep the instruction cache coherent. Stores are not guaranteed to show up in the instruction stream until a program calls an operating system facility to ensure coherency.

Tag RAM

In computer engineering, a *tag RAM* is used to specify which of the possible memory locations is currently stored in a CPU cache.[49][50] For a simple, direct-mapped design fast SRAM can be used. Higher associative cachesusually employ content-addressable memory.

Implementation

Cache **reads** are the most common CPU operation that takes more than a single cycle. Program execution time tends to be very sensitive to the latency of a level-1 data cache hit. A great deal of design effort, and often power and silicon area are expended making the caches as fast as possible.

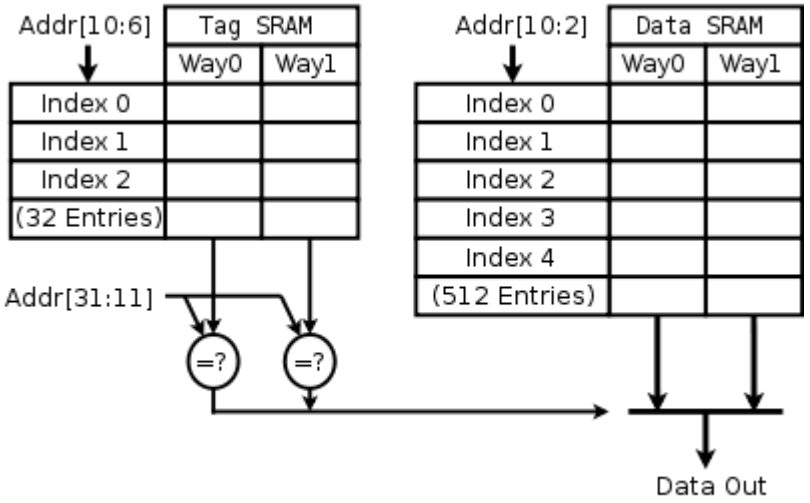
The simplest cache is a virtually indexed direct-mapped cache. The virtual address is calculated with an adder, the relevant portion of the address extracted and used to index an SRAM, which returns the loaded data. The data is byte aligned in a byte shifter, and from there is bypassed to the next operation. There is no need for any tag checking in the inner loop – in fact, the tags need not even be read. Later in the pipeline, but before the load

instruction is retired, the tag for the loaded data must be read, and checked against the virtual address to make sure there was a cache hit. On a miss, the cache is updated with the requested cache line and the pipeline is restarted.

An associative cache is more complicated, because some form of tag must be read to determine which entry of the cache to select. An N-way set-associative level-1 cache usually reads all N possible tags and N data in parallel, and then chooses the data associated with the matching tag. Level-2 caches sometimes save power by reading the tags first, so that only one data element is read from the data SRAM.

The adjacent diagram is intended to clarify the manner in which the various fields of the address are used. Address bit 31 is most significant, bit 0 is least significant. The diagram shows the SRAMs, indexing, and multiplexing for a 4 KB, 2-way set-associative, virtually indexed and virtually tagged cache with 64 byte (B) lines, a 32-bit read width and 32-bit virtual address.

Because the cache is 4 KB and has 64 B lines, there are just 64 lines in the cache, and we read two at a time from a Tag SRAM which has 32 rows, each with a pair of 21 bit tags. Although any function of virtual address bits 31 through 6 could be used to index the tag and data SRAMs, it is simplest to use the least significant bits.



4KB, 2-way set-associative 64B line cache read path

Read path for a 2-way associative cache

Similarly, because the cache is 4 KB and has a 4 B read path, and reads two ways for each access, the Data SRAM is 512 rows by 8 bytes wide.

A more modern cache might be 16 KB, 4-way set-associative, virtually indexed, virtually hinted, and physically tagged, with 32 B lines, 32-bit read width and 36-bit physical addresses. The read path recurrence for such a cache looks very similar to the path above. Instead of tags, vhints are read, and matched against a subset of the virtual address. Later on in the pipeline, the virtual address is translated into a physical address by the TLB, and the physical tag is read (just one, as the vhint supplies which way of the cache to read). Finally the physical address is compared to the physical tag to determine if a hit has occurred.

Some SPARC designs have improved the speed of their L1 caches by a few gate delays by collapsing the virtual address adder into the SRAM decoders. See Sum addressed decoder.

History

The early history of cache technology is closely tied to the invention and use of virtual memory. Because of scarcity and cost of semi-conductor memories, early mainframe computers in the 1960s used a complex hierarchy of physical memory, mapped onto a flat virtual memory space used by programs. The memory technologies would span semi-conductor, magnetic core, drum and disc. Virtual memory seen and used by programs would be flat and caching would be used to fetch data and instructions into the fastest memory ahead of processor access. Extensive studies were done to optimize the cache sizes. Optimal values were found to depend greatly on the programming language used with Algol needing the smallest and Fortran and Cobol needing the largest cache sizes.

In the early days of microcomputer technology, memory access was only slightly slower than register access. But since the 1980s[51] the performance gap between processor and memory has been growing. Microprocessors have advanced much faster than memory, especially in terms of their operating frequency, so memory became a performance bottleneck. While it was technically possible to have all the main memory as fast as the CPU, a more economically viable path has been taken: use plenty of low-speed memory, but also introduce a small high-speed cache memory to alleviate the performance gap. This provided an order of magnitude more capacity—for the same price—with only a slightly reduced combined performance.

First TLB implementations

The first documented uses of a TLB were on the GE 645[52] and the IBM 360/67,[53] both of which used an associative memory as a TLB.

First instruction cache

The first documented use of an instruction cache was on the CDC 6600.[54]

First data cache

The first documented use of a data cache was on the IBM System/360 Model 85.[55]

In 68k microprocessors

The 68010, released in 1982, has a "loop mode" which can be considered a tiny and special-case instruction cache that accelerates loops that consist of only two instructions. The 68020, released in 1984, replaced that with a typical instruction cache of 256 bytes, being the first 68k series processor to feature true on-chip cache memory.

The 68030, released in 1987, is basically a 68020 core with an additional 256-byte data cache, an on-chip memory management unit (MMU), a process shrink, and added burst mode for the caches. The 68040, released in 1990, has split instruction and data caches of four kilobytes each. The 68060, released in 1994, has the following: 8 KB data cache (four-way associative), 8 KB instruction cache (four-way associative), 96-byte FIFO instruction buffer, 256-entry branch cache, and 64-entry address translation cache MMU buffer (four-way associative).

In x86 microprocessors

As the x86 microprocessors reached clock rates of 20 MHz and above in the 386, small amounts of fast cache memory began to be featured in systems to improve performance. This was because the DRAM used for main memory had significant latency, up to 120 ns, as well as refresh cycles. The cache was constructed from more expensive, but significantly faster, SRAM memory cells, which at the time had latencies around 10 ns - 25 ns. The early caches were external to the processor and typically located on the motherboard in the form of eight or nine DIP devices placed in sockets to enable the cache as an optional extra or upgrade feature.

Some versions of the Intel 386 processor could support 16 to 256 KB of external cache.

With the 486 processor, an 8 KB cache was integrated directly into the CPU die. This cache was termed Level 1 or L1 cache to differentiate it from the slower on-motherboard, or Level 2 (L2) cache. These on-motherboard caches were much larger, with the most common size being 256 KB. The popularity of on-motherboard cache

continued through the Pentium MMX era but was made obsolete by the introduction of SDRAM and the growing disparity between bus clock rates and CPU clock rates, which caused on-motherboard cache to be only slightly faster than main memory.

The next development in cache implementation in the x86 microprocessors began with the Pentium Pro, which brought the secondary cache onto the same package as the microprocessor, clocked at the same frequency as the microprocessor.

On-motherboard caches enjoyed prolonged popularity thanks to the AMD K6-2 and AMD K6-III processors that still used Socket 7, which was previously used by Intel with on-motherboard caches. K6-III included 256 KB on-die L2 cache and took advantage of the on-board cache as a third level cache, named L3 (motherboards with up to 2 MB of on-board cache were produced). After the Socket 7 became obsolete, on-motherboard cache disappeared from the x86 systems.

The three-level caches were used again first with the introduction of multiple processor cores, where the L3 cache was added to the CPU die. It became common for the total cache sizes to be increasingly larger in newer processor generations, and recently (as of 2011) it is not uncommon to find Level 3 cache sizes of tens of megabytes.[56]

Intel introduced a Level 4 on-package cache with the Haswell microarchitecture. *Crystalwell*[27] Haswell CPUs, equipped with the GT3e variant of Intel's integrated Iris Pro graphics, effectively feature 128 MB of embedded DRAM (eDRAM) on the same package. This L4 cache is shared dynamically between the on-die GPU and CPU, and serves as a victim cache to the CPU's L3 cache.[28]

In ARM microprocessors

Apple M1 CPU has 128 or 192 KB instruction L1 cache for each core (important for latency/single-thread performance), depending on core type, unusually large for L1 cache of any CPU type, not just for a laptop, while the total cache memory size is not unusually large (the total is more important for throughput), for a laptop, and much larger total (e.g. L3 or L4) sizes are available in IBM's mainframes.

Current research

Early cache designs focused entirely on the direct cost of cache and RAM and average execution speed. More recent cache designs also consider energy efficiency,[57] fault tolerance, and other goals.[58][59] Researchers have also explored use of emerging memory technologies such as eDRAM (embedded DRAM) and NVRAM (non-volatile RAM) for designing caches.[60]

There are several tools available to computer architects to help explore tradeoffs between the cache cycle time, energy, and area; the CACTI cache simulator[61] and the SimpleScalar instruction set simulator are two open-source options. Modeling of 2D and 3D SRAM, eDRAM, STT-RAM, ReRAM and PCM caches can be done using the DESTINY tool.[62]

Multi-ported cache

A multi-ported cache is a cache which can serve more than one request at a time. When accessing a traditional cache we normally use a single memory address, whereas in a multi-ported cache we may request N addresses at a time – where N is the number of ports that connected through the processor and the cache. The benefit of this is that a pipelined processor may access memory from different phases in its pipeline. Another benefit is that it allows the concept of super-scalar processors through different cache levels.

See also

- Branch predictor
- Cache (computing)
- Cache algorithms
- Cache coherency
- Cache control instructions
- Cache hierarchy
- Cache placement policies
- Cache prefetching
- Dinero (Cache simulator by University of Wisconsin System)
- Instruction unit
- Locality of reference
- Memoization
- Memory hierarchy
- Micro-operation
- No-write allocation
- Scratchpad RAM
- Sum addressed decoder
- Write buffer

Notes

1. The very first paging machine, the Ferranti Atlas[20][21] had no page tables in main memory; there was an associative memory with one entry for every 512 word page frame of core.

References

1. Gabriel Torres (September 12, 2007). "How The Cache Memory Works".
2. "A Survey of Techniques for Architecting TLBs", Concurrency and Computation, 2016.
3. Smith, Alan Jay (September 1982). "Cache Memories" (PDF). *Computing Surveys*. **14** (3): 473–530. doi:10.1145/356887.356892. S2CID 6023466.
4. "Altering Computer Architecture is Way to Raise Throughput, Suggests IBM Researchers". *Electronics*. **49** (25): 30–31. December 23, 1976.
5. "IBM z13 and IBM z13s Technical Introduction" (PDF). IBM. March 2016. p. 20.
6. "Product Fact Sheet: Accelerating 5G Network Infrastructure, from the Core to the Edge". *Intel Newsroom* (Press release). Retrieved 2020-04-12. "L1 cache of 32KB/core, L2 cache of 4.5MB per 4-core cluster and shared LLC cache up to 15MB."
7. Smith, Ryan. "Intel Launches Atom P5900: A 10nm Atom for Radio Access Networks". *www.anandtech.com*. Retrieved 2020-04-12.
8. "Cache design" (PDF). *ucsd.edu*. 2010-12-02. p. 10–15. Retrieved 2014-02-24.
9. IEEE Xplore - Phased set associative cache design for reduced power consumption. Ieeexplore.ieee.org (2009-08-11). Retrieved on 2013-07-30.
10. Sanjeev Jahagirdar; Varghese George; Inder Sodhi; Ryan Wells (2012). "Power Management of the Third Generation Intel Core Micro Architecture formerly codenamed Ivy Bridge" (PDF). *hotchips.org*. p. 18. Retrieved 2015-12-16.

11. André Seznec (1993). "A Case for Two-Way Skewed-Associative Caches". *ACM SIGARCH Computer Architecture News*. **21** (2): 169–178. doi:10.1145/173682.165152.
12. C. Kozyrakis. "Lecture 3: Advanced Caching Techniques" (PDF). Archived from the original (PDF) on September 7, 2012.
13. Micro-Architecture "Skewed-associative caches have ... major advantages over conventional set-associative caches."
14. Nathan N. Sadler; Daniel J. Sorin (2006). "Choosing an Error Protection Scheme for a Microprocessor's L1 Data Cache" (PDF). p. 4.
15. John L. Hennessy; David A. Patterson (2011). *Computer Architecture: A Quantitative Approach*. p. B-9. ISBN 978-0-12-383872-8.
16. David A. Patterson; John L. Hennessy (2009). *Computer Organization and Design: The Hardware/Software Interface*. p. 484. ISBN 978-0-12-374493-7.
17. Gene Cooperman (2003). "Cache Basics".
18. Ben Dugan (2002). "Concerning Cache".
19. Harvey G. Cragon. "Memory systems and pipelined processors". 1996. ISBN 0-86720-474-5, ISBN 978-0-86720-474-2. "Chapter 4.1: Cache Addressing, Virtual or Real" p. 209 [1]
20. Sumner, F. H.; Haley, G.; Chenh, E. C. Y. (1962). "The Central Control Unit of the 'Atlas' Computer". *Information Processing 1962*. IFIP Congress Proceedings. Proceedings of IFIP Congress 62. Spartan.
21. Kilburn, T.; Payne, R. B.; Howarth, D. J. (December 1961). "The Atlas Supervisor". *Computers - Key to Total Systems Control*. Conferences Proceedings. 20 Proceedings of the Eastern Joint Computer Conference Washington, D.C. Macmillan. pp. 279–294.
22. Kaxiras, Stefanos; Ros, Alberto (2013). *A New Perspective for Efficient Virtual-Cache Coherence*. *40th International Symposium on Computer Architecture (ISCA)*. pp. 535–547. CiteSeerX 10.1.1.307.9125. doi:10.1145/2485922.2485968. ISBN 9781450320795. S2CID 15434231.
23. "Understanding Caching". *Linux Journal*. Retrieved 2010-05-02.
24. Taylor, George; Davies, Peter; Farmwald, Michael (1990). "The TLB Slice - A Low-Cost High-Speed Address Translation Mechanism". CH2887-8/90/0000/0355$01.OO.
25. Timothy Roscoe; Andrew Baumann (2009-03-03). "Advanced Operating Systems Caches and TLBs (263-3800-00L)" (PDF). *systems.ethz.ch*. Archived from the original (PDF) on 2011-10-07. Retrieved 2016-02-14.
26. N.P.Jouppi. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers." - 17th Annual International Symposium on Computer Architecture, 1990. Proceedings., doi:10.1109/ISCA.1990.134547
27. "Products (Formerly Crystal Well)". Intel. Retrieved 2013-09-15.
28. "Intel Iris Pro 5200 Graphics Review: Core i7-4950HQ Tested". AnandTech. Retrieved 2013-09-16.
29. Ian Cutress (September 2, 2015). "The Intel Skylake Mobile and Desktop Launch, with Architecture Analysis". AnandTech.
30. Anand Lal Shimpi (2000-11-20). "The Pentium 4's Cache – Intel Pentium 4 1.4 GHz & 1.5 GHz". AnandTech. Retrieved 2015-11-30.
31. Agner Fog (2014-02-19). "The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers" (PDF). *agner.org*. Retrieved 2014-03-21.
32. David Kanter (August 26, 2010). "AMD's Bulldozer Microarchitecture - Memory Subsystem Continued". *Real World Technologies*.
33. David Kanter (September 25, 2010). "Intel's Sandy Bridge Microarchitecture - Instruction Decode and uop Cache". *Real World Technologies*.

34. Baruch Solomon; Avi Mendelson; Doron Orenstein; Yoav Almog; Ronny Ronen (August 2001). "Micro-Operation Cache: A Power Aware Frontend for Variable Instruction Length ISA" (PDF). *ISLPED'01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design (IEEE Cat. No.01TH8581)*. Intel. pp. 4–9. doi:10.1109/LPE.2001.945363. ISBN 978-1-58113-371-4. S2CID 195859085. Retrieved 2013-10-06.
35. Anand Lal Shimpi (2012-10-05). "Intel's Haswell Architecture Analyzed". AnandTech. Retrieved 2013-10-20.
36. Ian Cutress (2016-08-18). "AMD Zen Microarchitecture: Dual Schedulers, Micro-Op Cache and Memory Hierarchy Revealed". AnandTech. Retrieved 2017-04-03.
37. Leon Gu; Dipti Motiani (October 2003). "Trace Cache" (PDF). Retrieved 2013-10-06.
38. Kun Niu (28 May 2015). "How does the BTIC (branch target instruction cache) work?". Retrieved 7 April 2018.
39. "Intel Smart Cache: Demo". Intel. Retrieved 2012-01-26.
40. "Inside Intel Core Microarchitecture and Smart Memory Access". Intel. 2006. p. 5. Archived from the original (PDF) on 2011-12-29. Retrieved 2012-01-26.
41. "Intel Iris Pro 5200 Graphics Review: Core i7-4950HQ Tested". AnandTech. Retrieved 2014-02-25.
42. Tian Tian; Chiu-Pi Shih (2012-03-08). "Software Techniques for Shared-Cache Multi-Core Systems". Intel. Retrieved 2015-11-24.
43. Oded Lempel (2013-07-28). "2nd Generation Intel Core Processor Family: Intel Core i7, i5 and i3" (PDF). *hotchips.org*. p. 7–10,31–45. Retrieved 2014-01-21.
44. Chen, J. Bradley; Borg, Anita; Jouppi, Norman P. (1992). "A Simulation Based Study of TLB Performance". *SIGARCH Computer Architecture News*. **20** (2): 114–123. doi:10.1145/146628.139708.
45. "Explanation of the L1 and L2 Cache". *amecomputers.com*. Retrieved 2014-06-09.
46. Ying Zheng; Brian T. Davis; Matthew Jordan (2004-06-25). "Performance Evaluation of Exclusive Cache Hierarchies" (PDF). Michigan Technological University. Retrieved 2014-06-09.
47. Aamer Jaleel; Eric Borch; Malini Bhandaru; Simon C. Steely Jr.; Joel Emer (2010-09-27). "Achieving Non-Inclusive Cache Performance with Inclusive Caches" (PDF). *jaleels.org*. Retrieved 2014-06-09.
48. "AMD K8". *Sandpile.org*. Archived from the original on 2007-05-15. Retrieved 2007-06-02.
49. "Cortex-R4 and Cortex-R4F Technical Reference Manual". arm.com. Retrieved 2013-09-28.
50. "L210 Cache Controller Technical Reference Manual". arm.com. Retrieved 2013-09-28.
51. Mahapatra, Nihar R.; Venkatrao, Balakrishna (1999). "The processor-memory bottleneck: problems and solutions" (PDF). *Crossroads*. **5** (3es): 2–es. doi:10.1145/357783.331677. S2CID 11557476. Retrieved 2013-03-05.
52. *GE-645 System Manual* (PDF). General Electric. January 1968. Retrieved 2020-07-10.
53. *IBM System/360 Model 67 Functional Characteristics* (PDF). Third Edition. IBM. February 1972. GA27-2719-2.
54. James E. Thornton (October 1964), "Parallel operation in the control data 6600" (PDF), *Proc. of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*
55. IBM (June 1968). *IBM System/360 Model 85 Functional Characteristics* (PDF). SECOND EDITION. A22-6916-1.
56. "Intel® Xeon® Processor E7 Family". Intel. Retrieved 2013-10-10.
57. Sparsh Mittal (March 2014). "A Survey of Architectural Techniques For Improving Cache Power Efficiency". *Sustainable Computing: Informatics and Systems*. **4** (1): 33–43. doi:10.1016/j.suscom.2013.11.001.
58. Sally Adee (2009). "Chip Design Thwarts Sneak Attack on Data".

59. Zhenghong Wang; Ruby B. Lee (November 8–12, 2008). *A novel cache architecture with enhanced performance and security* (PDF). 41st annual IEEE/ACM International Symposium on Microarchitecture. pp. 83–93. Archived from the original (PDF) on March 6, 2012.
60. Sparsh Mittal; Jeffrey S. Vetter; Dong Li (June 2015). "A Survey Of Architectural Approaches for Managing Embedded DRAM and Non-volatile On-chip Caches". *IEEE Transactions on Parallel and Distributed Systems*. **26** (6): 1524–1537. doi:10.1109/TPDS.2014.2324563. S2CID 14583671.
61. "CACTI". *Hpl.hp.com*. Retrieved 2010-05-02.
62. "3d_cache_modeling_tool / destiny". *code.ornl.gov*. Retrieved 2015-02-26.

External links

- Memory part 2: CPU caches – an article on lwn.net by Ulrich Drepper describing CPU caches in detail
- Evaluating Associativity in CPU Caches – Hill and Smith (1989) – introduces capacity, conflict, and compulsory classification
- Cache Performance for SPEC CPU2000 Benchmarks – Hill and Cantin (2003) – This reference paper has been updated several times. It has thorough and lucidly presented simulation results for a reasonably wide set of benchmarks and cache organizations.
- Memory Hierarchy in Cache-Based Systems – by Ruud van der Pas, 2002, Sun Microsystems – a nice introductory article to CPU memory caching
- A Cache Primer – by Paul Genua, P.E., 2004, Freescale Semiconductor, another introductory article
- An 8-way set-associative cache – written in VHDL
- Understanding CPU caching and performance – an article on Ars Technica by Jon Stokes
- IBM POWER4 processor review – an article on ixbtlabs by Pavel Danilov
- What is Cache Memory and its Types!
- Memory Caching – a Princeton University lecture