

# Cache (computing)

In computing, a **cache** /kæʃ/ *KASH*,<sup>[1]</sup> is a hardware or software component that stores data so future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation, or the duplicate of data stored elsewhere. A *cache hit* occurs when the requested data can be found in a cache, while a *cache miss* occurs when it cannot. Cache hits are served by reading data from the cache, which is faster than recomputing a result or reading from a slower data store; thus, the more requests can be served from the cache, the faster the system performs.

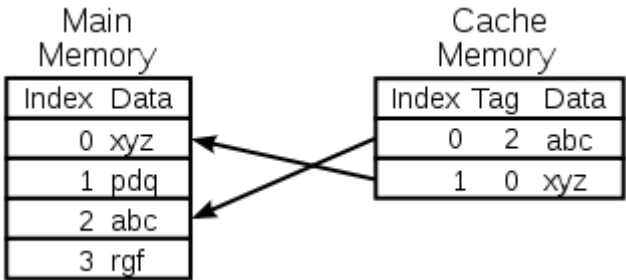


Diagram of a CPU memory cache operation

To be cost-effective and to enable efficient use of data, caches must be relatively small. Nevertheless, caches have proven themselves in many areas of computing because access patterns in typical computer applications exhibit the locality of reference. Moreover, access patterns exhibit temporal locality if data is requested again that has been recently requested already, while spatial locality refers to requests for data physically stored close to data that has been already requested.

## Contents

### Motivation

- Latency
- Throughput

### Operation

- Writing policies

### Examples of hardware caches

- CPU cache
- GPU cache
- DSPs
- Translation lookaside buffer

### Software caches

- Disk cache
- Web cache
- Memoization
- Other caches

### Buffer vs. cache

### See also

### References

### Further reading

## Motivation

There is an inherent trade-off between size and speed (given that a larger resource implies greater physical distances) but also a tradeoff between expensive, premium technologies (such as SRAM) vs cheaper, easily mass-produced commodities (such as DRAM or hard disks).

The buffering provided by a cache benefits both bandwidth and latency:

## Latency

A larger resource incurs a significant latency for access – e.g. it can take hundreds of clock cycles for a modern 4 GHz processor to reach DRAM. This is mitigated by reading in large chunks, in the hope that subsequent reads will be from nearby locations. Prediction or explicit prefetching might also guess where future reads will come from and make requests ahead of time; if done correctly the latency is bypassed altogether.

## Throughput

The use of a cache also allows for higher throughput from the underlying resource, by assembling multiple fine grain transfers into larger, more efficient requests. In the case of DRAM, this might be served by a wider bus. Imagine a program scanning bytes in a 32bit address space, but being served by a 128bit off chip data bus; individual uncached byte accesses would only allow 1/16th of the total bandwidth to be used, and 80% of the data movement would be addresses. Reading larger chunks reduces the fraction of bandwidth required for transmitting address information.

## Operation

---

Hardware implements cache as a block of memory for temporary storage of data likely to be used again. Central processing units (CPUs) and hard disk drives (HDDs) frequently use a cache, as do web browsers and web servers.

A cache is made up of a pool of entries. Each entry has associated data, which is a copy of the same data in some *backing store*. Each entry also has a tag, which specifies the identity of the data in the backing store of which the entry is a copy.

When the cache client (a CPU, web browser, operating system) needs to access data presumed to exist in the backing store, it first checks the cache. If an entry can be found with a tag matching that of the desired data, the data in the entry is used instead. This situation is known as a **cache hit**. So, for example, a web browser program might check its local cache on disk to see if it has a local copy of the contents of a web page at a particular URL. In this example, the URL is the tag, and the contents of the web page is the data. The percentage of accesses that result in cache hits is known as the **hit rate** or **hit ratio** of the cache.

The alternative situation, when the cache is consulted and found not to contain data with the desired tag, has become known as a **cache miss**. The previously uncached data fetched from the backing store during miss handling is usually copied into the cache, ready for the next access.

During a cache miss, the CPU usually ejects some other entry in order to make room for the previously uncached data. The heuristic used to select the entry to eject is known as the replacement policy. One popular replacement policy, "least recently used" (LRU), replaces the least recently used entry (see cache algorithm). More efficient caches compute use frequency against the size of the stored contents, as well as the latencies and throughputs for both the cache and the backing store. This works well for larger amounts of data, longer latencies and slower throughputs, such as experienced with a hard drive and the Internet, but is not efficient for use with a CPU cache.

## Writing policies

When a system writes data to cache, it must at some point write that data to the backing store as well. The timing of this write is controlled by what is known as the *write policy*.

There are two basic writing approaches:

- **Write-through:** write is done synchronously both to the cache and to the backing store.
- **Write-back** (also called *write-behind*): initially, writing is done only to the cache. The write to the backing store is postponed until the cache blocks containing the data are about to be modified/replaced by new content.

A write-back cache is more complex to implement, since it needs to track which of its locations have been written over, and mark them as *dirty* for later writing to the backing store. The data in these locations are written back to the backing store only when they are evicted from the cache, an effect referred to as a *lazy write*. For this reason, a read miss in a write-back cache (which requires a block to be replaced by another) will often require two memory accesses to service: one to write the replaced data from the cache back to the store, and then one to retrieve the needed data.

Other policies may also trigger data write-back. The client may make many changes to data in the cache, and then explicitly notify the cache to write back the data.

No data is returned on write operations, thus there are two approaches for situations of write-misses:

- **Write allocate** (also called *fetch on write*): data at the missed-write location is loaded to cache, followed by a write-hit operation. In this approach, write misses are similar to read misses.
- **No-write allocate** (also called *write-no-allocate* or *write around*): data at the missed-write location is not loaded to cache, and is written directly to the backing store. In this approach, only the reads are being cached.

Both write-through and write-back policies can use either of these write-miss policies, but usually they are paired in this way:<sup>[2]</sup>

- A write-back cache uses write allocate, hoping for subsequent writes (or even reads) to the same location, which is now cached.
- A write-through cache uses no-write allocate. Here, subsequent writes have no advantage, since they still need to be written directly to the backing store.

Entities other than the cache may change the data in the backing store, in which case the copy in the cache may become out-of-date or *stale*.

Alternatively, when the client updates the data in the cache, copies of those data in other caches will become stale. Communication protocols between the cache managers which keep the data consistent are known as coherency protocols.

## Examples of hardware caches

### CPU cache

Small memories on or close to the CPU can operate faster than the much larger main memory. Most CPUs since the 1980s have used one or more caches, sometimes in cascaded levels; modern high-end embedded, desktop and server microprocessors may have as many as six types of cache (between levels and functions),<sup>[3]</sup> Examples of caches with a specific function are the D-cache and I-cache and the translation lookaside buffer for the MMU.



A write-through cache with no-write allocation



A write-back cache with write allocation

## GPU cache

Earlier graphics processing units (GPUs) often had limited read-only texture caches, and introduced morton order swizzled textures to improve 2D cache coherency. Cache misses would drastically affect performance, e.g. if mipmapping was not used. Caching was important to leverage 32-bit (and wider) transfers for texture data that was often as little as 4 bits per pixel, indexed in complex patterns by arbitrary UV coordinates and perspective transformations in inverse texture mapping.

As GPUs advanced (especially with GPGPU compute shaders) they have developed progressively larger and increasingly general caches, including instruction caches for shaders, exhibiting increasingly common functionality with CPU caches.<sup>[4]</sup> For example, GT200 architecture GPUs did not feature an L2 cache, while the Fermi GPU has 768 KB of last-level cache, the Kepler GPU has 1536 KB of last-level cache,<sup>[4]</sup> and the Maxwell GPU has 2048 KB of last-level cache. These caches have grown to handle synchronisation primitives between threads and atomic operations, and interface with a CPU-style MMU.

## DSPs

Digital signal processors have similarly generalised over the years. Earlier designs used scratchpad memory fed by DMA, but modern DSPs such as Qualcomm Hexagon often include a very similar set of caches to a CPU (e.g. Modified Harvard architecture with shared L2, split L1 I-cache and D-cache).<sup>[5]</sup>

## Translation lookaside buffer

A memory management unit (MMU) that fetches page table entries from main memory has a specialized cache, used for recording the results of virtual address to physical address translations. This specialized cache is called a translation lookaside buffer (TLB).<sup>[6]</sup>

# Software caches

---

## Disk cache

While CPU caches are generally managed entirely by hardware, a variety of software manages other caches. The page cache in main memory, which is an example of disk cache, is managed by the operating system kernel.

While the disk buffer, which is an integrated part of the hard disk drive, is sometimes misleadingly referred to as "disk cache", its main functions are write sequencing and read prefetching. Repeated cache hits are relatively rare, due to the small size of the buffer in comparison to the drive's capacity. However, high-end disk controllers often have their own on-board cache of the hard disk drive's data blocks.

Finally, a fast local hard disk drive can also cache information held on even slower data storage devices, such as remote servers (web cache) or local tape drives or optical jukeboxes; such a scheme is the main concept of hierarchical storage management. Also, fast flash-based solid-state drives (SSDs) can be used as caches for slower rotational-media hard disk drives, working together as hybrid drives or solid-state hybrid drives (SSHDs).

## Web cache

Web browsers and web proxy servers employ web caches to store previous responses from web servers, such as web pages and images. Web caches reduce the amount of information that needs to be transmitted across the network, as information previously stored in the cache can often be re-used. This reduces bandwidth and processing requirements of the web server, and helps to improve responsiveness for users of the web.<sup>[7]</sup>

Web browsers employ a built-in web cache, but some Internet service providers (ISPs) or organizations also use a caching proxy server, which is a web cache that is shared among all users of that network.

Another form of cache is P2P caching, where the files most sought for by peer-to-peer applications are stored in an ISP cache to accelerate P2P transfers. Similarly, decentralised equivalents exist, which allow communities to perform the same task for P2P traffic, for example, Corelli.<sup>[8]</sup>

## Memoization

A cache can store data that is computed on demand rather than retrieved from a backing store. Memoization is an optimization technique that stores the results of resource-consuming function calls within a lookup table, allowing subsequent calls to reuse the stored results and avoid repeated computation.

## Other caches

The BIND DNS daemon caches a mapping of domain names to IP addresses, as does a resolver library.

Write-through operation is common when operating over unreliable networks (like an Ethernet LAN), because of the enormous complexity of the coherency protocol required between multiple write-back caches when communication is unreliable. For instance, web page caches and client-side network file system caches (like those in NFS or SMB) are typically read-only or write-through specifically to keep the network protocol simple and reliable.

Search engines also frequently make web pages they have indexed available from their cache. For example, Google provides a "Cached" link next to each search result. This can prove useful when web pages from a web server are temporarily or permanently inaccessible.

Another type of caching is storing computed results that will likely be needed again, or memoization. For example, ccache is a program that caches the output of the compilation, in order to speed up later compilation runs.

Database caching can substantially improve the throughput of database applications, for example in the processing of indexes, data dictionaries, and frequently used subsets of data.

A distributed cache<sup>[9]</sup> uses networked hosts to provide scalability, reliability and performance to the application.<sup>[10]</sup> The hosts can be co-located or spread over different geographical regions.

## Buffer vs. cache

---

The semantics of a "buffer" and a "cache" are not totally different; even so, there are fundamental differences in intent between the process of caching and the process of buffering.

Fundamentally, caching realizes a performance increase for transfers of data that is being repeatedly transferred. While a caching system may realize a performance increase upon the initial (typically write) transfer of a data item, this performance increase is due to buffering occurring within the caching system.

With read caches, a data item must have been fetched from its residing location at least once in order for subsequent reads of the data item to realize a performance increase by virtue of being able to be fetched from the cache's (faster) intermediate storage rather than the data's residing location. With write caches, a performance increase of writing a data item may be realized upon the first write of the data item by virtue of the data item immediately being stored in the cache's intermediate storage, deferring the transfer of the data item to its residing storage at a later stage or else occurring as a background process. Contrary to strict buffering, a caching process must adhere to a (potentially distributed) cache coherency protocol in order to maintain consistency between the cache's intermediate storage and the location where the data resides. Buffering, on the other hand,

- reduces the number of transfers for otherwise novel data amongst communicating processes, which amortizes overhead involved for several small transfers over fewer, larger transfers,
- provides an intermediary for communicating processes which are incapable of direct transfers amongst each other, or
- ensures a minimum data size or representation required by at least one of the communicating processes involved in a transfer.

With typical caching implementations, a data item that is read or written for the first time is effectively being buffered; and in the case of a write, mostly realizing a performance increase for the application from where the write originated. Additionally, the portion of a caching protocol where individual writes are deferred to a batch of writes is a form of buffering. The portion of a caching protocol where individual reads are deferred to a batch of reads is also a form of buffering, although this form may negatively impact the performance of at least the initial reads (even though it may positively impact the performance of the sum of the individual reads). In practice, caching almost always involves some form of buffering, while strict buffering does not involve caching.

A buffer is a temporary memory location that is traditionally used because CPU instructions cannot directly address data stored in peripheral devices. Thus, addressable memory is used as an intermediate stage. Additionally, such a buffer may be feasible when a large block of data is assembled or disassembled (as required by a storage device), or when data may be delivered in a different order than that in which it is produced. Also, a whole buffer of data is usually transferred sequentially (for example to hard disk), so buffering itself sometimes increases transfer performance or reduces the variation or jitter of the transfer's latency as opposed to caching where the intent is to reduce the latency. These benefits are present even if the buffered data are written to the buffer once and read from the buffer once.

A cache also increases transfer performance. A part of the increase similarly comes from the possibility that multiple small transfers will combine into one large block. But the main performance-gain occurs because there is a good chance that the same data will be read from cache multiple times, or that written data will soon be read. A cache's sole purpose is to reduce accesses to the underlying slower storage. Cache is also usually an abstraction layer that is designed to be invisible from the perspective of neighboring layers.

## See also

---

- Cache prefetching
- Cache algorithms
- Cache coherence
- Cache coloring
- Cache hierarchy
- Cache-oblivious algorithm
- Cache stampede
- Cache language model
- Database cache
- Dirty bit
- Disk buffer
- Cache manifest in HTML5
- Five-minute rule
- Materialized view
- Pipeline burst cache
- Temporary file

## References

---

1. "Cache" (<http://www.oxforddictionaries.com/definition/english/cache>). *Oxford Dictionaries*. Oxford Dictionaries. Retrieved 2 August 2016.

2. John L. Hennessy; David A. Patterson (16 September 2011). *Computer Architecture: A Quantitative Approach* (<https://books.google.com/books?id=v3-1hVwHnHwC&pg=SL2-PA12>). Elsevier. pp. B–12. ISBN 978-0-12-383872-8. Retrieved 25 March 2012.
3. "intel broad well core i7 with 128mb L4 cache" (<http://wccfttech.com/intel-broadwell-core-i7-5775c-128mb-l4-cache-and-skylake-core-i7-6700k-flagship-processors-available-retail/>). Mentions L4 cache. Combined with separate L-Cache and TLB, this brings the total 'number of caches (levels+functions) to 6
4. S. Mittal, "A Survey of Techniques for Managing and Leveraging Caches in GPUs ([https://www.academia.edu/6940614/A\\_Survey\\_of\\_Techniques\\_for\\_Managing\\_and\\_Leveraging\\_Caches\\_in\\_GPUs](https://www.academia.edu/6940614/A_Survey_of_Techniques_for_Managing_and_Leveraging_Caches_in_GPUs))", JCSC, 23(8), 2014.
5. "qualcom Hexagon DSP SDK overview" (<https://developer.qualcomm.com/software/hexagon-dsp-sdk/dsp-processor>).
6. Frank Uyeda (2009). "Lecture 7: Memory Management" (<http://cseweb.ucsd.edu/classes/su09/cse120/lectures/Lecture7.pdf>) (PDF). *CSE 120: Principles of Operating Systems*. UC San Diego. Retrieved 2013-12-04.
7. Multiple (wiki). "Web application caching" ([http://docforge.com/wiki/Web\\_application/Caching](http://docforge.com/wiki/Web_application/Caching)). *Docforge*. Retrieved 2013-07-24.
8. Gareth Tyson; Andreas Mauthe; Sebastian Kaune; Mu Mu; Thomas Plagemann. *Corelli: A Dynamic Replication Service for Supporting Latency-Dependent Content in Community Networks* (<https://web.archive.org/web/20150618193018/http://comp.eprints.lancs.ac.uk/2044/1/MMCN09.pdf>) (PDF). MMCN'09. Archived from the original (<http://comp.eprints.lancs.ac.uk/2044/1/MMCN09.pdf>) (PDF) on 2015-06-18.
9. Paul, S; Z Fei (1 February 2001). "Distributed caching with centralized control". *Computer Communications*. **24** (2): 256–268. doi:10.1016/S0140-3664(00)00322-4 (<https://doi.org/10.1016%2FS0140-3664%2800%2900322-4>).
10. Khan, Iqbal. "Distributed Caching On The Path To Scalability". *MSDN* (July 2009).

## Further reading

- "What Every Programmer Should Know About Memory" (<https://people.freebsd.org/~lstewart/articles/cpumemory.pdf>) by Ulrich Drepper
- "Caching in the Distributed Environment" (<http://msdn.microsoft.com/en-us/library/dd129907.aspx>)

Retrieved from "[https://en.wikipedia.org/w/index.php?title=Cache\\_\(computing\)&oldid=823003383](https://en.wikipedia.org/w/index.php?title=Cache_(computing)&oldid=823003383)"

**This page was last edited on 29 January 2018, at 19:19.**

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.