



Direct Mapped Cache

Related terms:

Cache Memory, Fully Associative Cache, Memory Access, Set-Associative Cache, Spatial Locality, Associativity, Memory Address, Memory Location

Memory Systems

Sarah L. Harris, David Money Harris, in Digital Design and Computer Architecture, 2016

Block Size

The previous examples were able to take advantage only of temporal locality, because the block size was one word. To exploit spatial locality, a cache uses larger blocks to hold several consecutive words.

The advantage of a block size greater than one is that when a miss occurs and the word is fetched into the cache, the adjacent words in the block are also fetched. Therefore, subsequent accesses are more likely to hit because of spatial locality. However, a large block size means that a fixed-size cache will have fewer blocks. This may lead to more conflicts, increasing the miss rate. Moreover, it takes more time to fetch the missing cache block after a miss, because more than one data word is fetched from main memory. The time required to load the missing block into the cache is called the *miss penalty*. If the adjacent words in the block are not accessed later, the effort of fetching them is wasted. Nevertheless, most real programs benefit from larger block sizes.

Figure 8.12 shows the hardware for a $C = 8$ -word direct mapped cache with a $b = 4$ -word block size. The cache now has only $B = C/b = 2$ blocks. A direct mapped cache

has one block in each set, so this cache is organized as two sets. Thus, only $\log_2 2 = 1$ bit is used to select the set. A multiplexer is now needed to select the word within the block. The multiplexer is controlled by the $\log_2 4 = 2$ *block offset bits* of the address. The most significant 27 address bits form the tag. Only one tag is needed for the entire block, because the words in the block are at consecutive addresses.

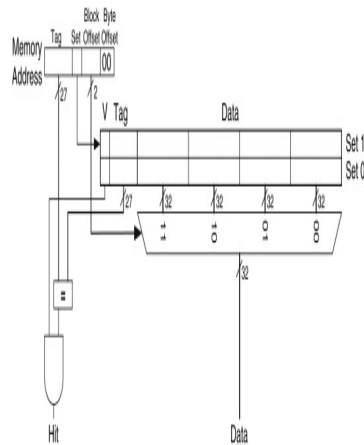


Figure 8.12. Direct mapped cache with two sets and a four-word block size

Figure 8.13 shows the cache fields for address 0x8000009C when it maps to the direct mapped cache of Figure 8.12. The byte offset bits are always 0 for word accesses. The next $\log_2 b = 2$ block offset bits indicate the word within the block. And the next bit indicates the set. The remaining 27 bits are the tag. Therefore, word 0x8000009C maps to set 1, word 3 in the cache. The principle of using larger block sizes to exploit spatial locality also applies to associative caches.



Figure 8.13. Cache fields for address 0x8000009C when mapping to the cache of Figure 8.12

Example 8.9

Spatial Locality with a Direct Mapped Cache

Repeat Example 8.6 for the eight-word direct mapped cache with a four-word block size.

Solution

Figure 8.14 shows the contents of the cache after the first memory access. On the first loop iteration, the cache misses on the access to memory address 0x4. This access loads data at addresses 0x0 through 0xC into the cache block. All subsequent accesses (as shown for address 0xC) hit in the cache. Hence, the miss rate is $1/15 = 6.67\%$.

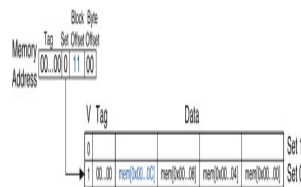


Figure 8.14. Cache contents with a block size b of four words

[Read full chapter](#)

URL: <https://www.sciencedirect.com/science/article/pii/B978012800056400008X>

μarch optimization advice

Jim Jeffers, ... Avinash Sodani, in Intel Xeon Phi Processor High Performance Programming (Second Edition), 2016

Direct Mapped MCDRAM Cache

The MCDRAM cache is a convenient way to increase memory bandwidth. As a memory side cache, it can automatically cache recently used data and provide much higher bandwidth than what DDR memory can achieve. When MCDRAM is placed in cache mode, it is a direct mapped cache. This means that multiple memory locations map to a single place in the cache. Because of this, a simple first optimization for a program is to turn on

the MCDRAM cache. Some applications that heavily utilize a few GB of memory could see performance improvements of up to 4 x. Because of the simplicity of this—no source code changes, and the large possible performance benefits, moving from DDR only to MCDRAM cache mode should be one of the first performance optimizations to try.

There are a few scenarios where enabling the cache could reduce performance. One case is when the MCDRAM cache is not able to hold the accessed working set. If an application streams through 64 GB of memory without reuse, then checking the MCDRAM cache (and missing) will only increase latency.

Another thing to note is that the direct mapped cache uses the physical address, not the linear address. Even if an address is contiguous in the linear/virtual address, the physical addresses that the OS gives to the application memory are not required to be. This can cause cache contention when using a significant portion of the MCDRAM cache. This is likely to reduce the peak memory bandwidth achievable. This can vary from run to run, as how the OS allocates pages can change from run to run. Monitoring the cache hit rate events from `perfmon` can be instructive in diagnosing this.

If MCDRAM cache is enabled, every modified line in the tile caches (L1 or L2 cache) must have an entry in the MCDRAM cache. If it is not in the MCDRAM cache, then the line will be downgraded to “shared” from “modified” in the tile cache. There is a very small probability that a pair of lines that are frequently read and written will map to the same MCDRAM set. This could cause a pair of reads that would normally hit in the L1 caches to become reads that need to go to DDR. This would cause a pair of threads to become substantially slower than the other threads in the chip. Due to linear to physical mapping variation, this behavior could vary from run to run, making it difficult to diagnose.

This case is very hard to create, but the way we employed to forcibly create this case was with two threads read and write their local stacks. Conceptually, any data location that is commonly read and written would work, but register spills to the stack are the most frequent case. If the stacks are offset by a multiple of 16 GB in physical memory, they would collide into the same MCDRAM cache set. A run-time that forced all thread stacks to allocate into a contiguous hardware memory region would avoid this from occurring. There is hardware to reduce the frequency of set conflicts from occurring. The probability of hitting this scenario on a given node is extremely small. The best clue to detecting this is that a pair of threads on the same chip are significantly slower than all other threads during a program phase—the threads that collide should vary from run to run, happen rarely, and only when MCDRAM cache mode is enabled.

[Read full chapter](#)

URL: <https://www.sciencedirect.com/science/article/pii/B9780128091944000065>

CPUs

Marilyn Wolf, in [Computers as Components \(Third Edition\)](#), 2012

3.5.1 Caches

Caches are widely used to speed up reads and writes in memory systems. Many [microprocessor](#) architectures include caches as part of their definition. The cache speeds up average memory access time when properly used. It increases the variability of memory access times—accesses in the cache will be fast, while access to locations not cached will be slow. This variability in performance makes it especially important to understand how caches work so that we can better understand how to predict cache performance and factor these variations into system design.

Cache controllers

A cache is a small, fast memory that holds copies of some of the contents of main memory. Because the cache is fast, it provides higher-speed access for the CPU; but because it is small, not all requests can be satisfied by the cache, forcing the system to wait for the slower main memory. Caching makes sense when the CPU is using only a relatively small set of memory locations at any one time; the set of active locations is often called the **working set**.

Figure 3.6 shows how the cache supports reads in the memory system. A **cache controller** mediates between the CPU and the memory system comprised of the cache and main memory. The cache controller sends a memory request to the cache and main memory. If the requested location is in the cache, the cache controller forwards the location's contents to the CPU and aborts the main memory request; this condition is known as a **cache hit**. If the location is not in the cache, the controller waits for the value from main memory and forwards it to the CPU; this situation is known as a **cache miss**.

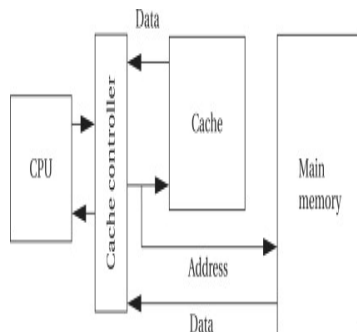


Figure 3.6. The cache in the memory system.

We can classify cache misses into several types depending on the situation that generated them:

- a **compulsory miss** (also known as a **cold miss**) occurs the first time a location is used,

- a **capacity miss** is caused by a too-large working set, and
- a **conflict miss** happens when two locations map to the same location in the cache.

Memory system performance

Even before we consider ways to implement caches, we can write some basic formulas for memory system performance. Let h be the **hit rate**, the probability that a given memory location is in the cache. It follows that $1 - h$ is the **miss rate**, or the probability that the location is not in the cache. Then we can compute the average memory access time as

$$t_{av} = ht_{cache} + (1 - h)t_{main}, \text{ (Eq. 3.1)}$$

where t_{cache} is the access time of the cache and t_{main} is the main memory access time. The memory access times are basic parameters available from the memory manufacturer. The hit rate depends on the program being executed and the cache organization, and is typically measured using simulators. The best-case memory access time (ignoring cache controller overhead) is t_{cache} , while the worst-case access time is t_{main} . Given that t_{main} is typically 50 to 75 ns, while t_{cache} is at most a few nanoseconds, the spread between worst-case and best-case memory delays is substantial.

First- and second-level cache

Modern CPUs may use multiple levels of cache as shown in Figure 3.7. The first-level cache (commonly known as L1 cache) is closest to the CPU, the **second-level cache (L2 cache)** feeds the first-level cache, and so on. In today's microprocessors, the first-level cache is often on-chip and the second-level cache is off-chip, although we are starting to see on-chip second-level caches.

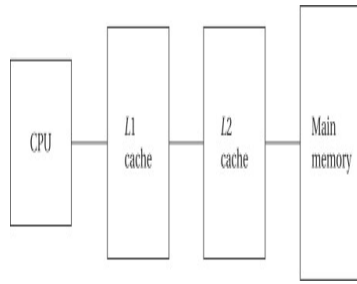


Figure 3.7. A two-level cache system.

The second-level cache is much larger but is also slower. If h_1 is the first-level hit rate and h_2 is the rate at which access hit the second-level cache, then the average access time for a two-level cache system is

$$t_{av} = h_1 t_{L1} + (h_2 - h_1) t_{L2} + (1 - h_2) t_{MM} \quad (\text{Eq. 3.2})$$

As the program's working set changes, we expect locations to be removed from the cache to make way for new locations. When set-associative caches are used, we have to think about what happens when we throw out a value from the cache to make room for a new value. We do not have this problem in direct-mapped caches because every location maps onto a unique block, but in a set-associative cache we must decide which set will have its block thrown out to make way for the new block. One possible replacement policy is least recently used (LRU), that is, throw out the block that has been used farthest in the past. We can add relatively small amounts of hardware to the cache to keep track of the time since the last access for each block. Another policy is random replacement, which requires even less hardware to implement.

Cache organization

The simplest way to implement a cache is a **direct-mapped cache**, as shown in Figure 3.8. The cache consists of cache **blocks**, each of which includes a tag to show which memory location is represented by this block, a data field holding the contents of that memory, and a valid tag to show whether the contents of this cache block are valid. An address is

divided into three sections. The index is used to select which cache block to check. The tag is compared against the tag value in the block selected by the index. If the address tag matches the tag value in the block, that block includes the desired memory location. If the length of the data field is longer than the minimum addressable unit, then the lowest bits of the address are used as an offset to select the required value from the data field. Given the structure of the cache, there is only one block that must be checked to see whether a location is in the cache—the index uniquely determines that block. If the access is a hit, the data value is read from the cache.

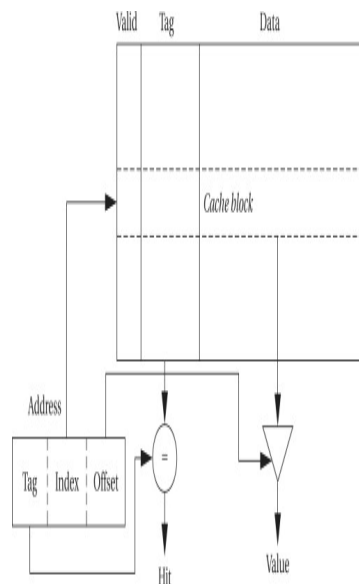


Figure 3.8. A direct-mapped cache.

Writes are slightly more complicated than reads because we have to update main memory as well as the cache. There are several methods by which we can do this. The simplest scheme is known as **write-through**—every write changes both the cache and the corresponding main memory location (usually through a write buffer). This scheme ensures that the cache and main memory are consistent, but may generate some additional main memory traffic. We can reduce the number of times we write to main memory by using a **write-back** policy: If we write only when we remove a location

from the cache, we eliminate the writes when a location is written several times before it is removed from the cache.

The direct-mapped cache is both fast and relatively low cost, but it does have limits in its caching power due to its simple scheme for mapping the cache onto main memory. Consider a direct-mapped cache with four blocks, in which locations 0, 1, 2, and 3 all map to different blocks. But locations 4, 8, 12, ... all map to the same block as location 0; locations 1, 5, 9, 13, ... all map to a single block; and so on. If two popular locations in a program happen to map onto the same block, we will not gain the full benefits of the cache. As seen in Section 5.7, this can create program performance problems.

The limitations of the direct-mapped cache can be reduced by going to the **set-associative** cache structure shown in Figure 3.9. A set-associative cache is characterized by the number of **banks** or **ways** it uses, giving an n -way set-associative cache. A set is formed by all the blocks (one for each bank) that share the same index. Each set is implemented with a direct-mapped cache. A cache request is broadcast to all banks simultaneously. If any of the sets has the location, the cache reports a hit. Although memory locations map onto blocks using the same function, there are n separate blocks for each set of locations. Therefore, we can simultaneously cache several locations that happen to map onto the same cache block. The set-associative cache structure incurs a little extra overhead and is slightly slower than a direct-mapped cache, but the higher hit rates that it can provide often compensate.

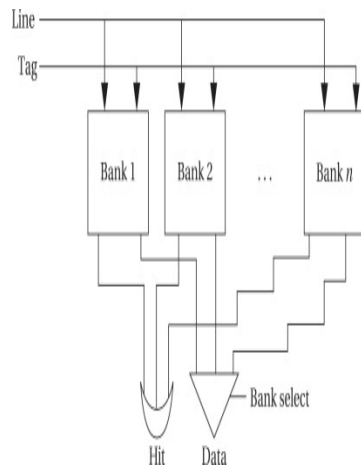


Figure 3.9. A set-associative cache.

The set-associative cache generally provides higher hit rates than the direct-mapped cache because conflicts between a small set of locations can be resolved within the cache. The set-associative cache is somewhat slower, so the CPU designer has to be careful that it doesn't slow down the CPU's cycle time too much. A more important problem with set-associative caches for embedded program design is predictability. Because the time penalty for a cache miss is so severe, we often want to make sure that critical segments of our programs have good behavior in the cache. It is relatively easy to determine when two memory locations will conflict in a direct-mapped cache. Conflicts in a set-associative cache are more subtle, and so the behavior of a set-associative cache is more difficult to analyze for both humans and programs.

Example 3.8 compares the behavior of direct-mapped and set-associative caches.

Example 3.8

Direct-Mapped versus Set-Associative Caches

For simplicity, let's consider a very simple caching scheme. We use two bits of the address as the tag. We compare a direct-mapped cache with four blocks and a two-way set-associative cache with four sets, and we use LRU replacement to make it easy to compare the two caches.

Here are the contents of memory, using a three-bit address for simplicity:

Address	Data
000	0101
001	1111
010	0000
011	0110
100	1000
101	0001
110	1010
111	0100

We will give each cache the same pattern of addresses (in binary to simplify picking out the index): 001, 010, 011, 100, 101, and 111. To understand how the direct-mapped cache works, let's see how its state evolves.

After 001 access:

Block	Tag	Data
00	—	—
01	0	1111
10	—	—
11	—	—

After 010 access:

Block	Tag	Data
00	—	—
01	0	1111
10	0	0000
11	—	—

After 011 access:

Block	Tag	Data
00	—	—
01	0	1111
10	0	0000
11	0	0110

After 100 access (notice that the tag bit for this entry is 1):

Block	Tag	Data
00	1	1000
01	0	1111
10	0	0000
11	0	0110

After 101 access (overwrites the 01 block entry):

Block	Tag	Data
00	1	1000
01	1	0001
10	0	0000
11	0	0110

After 111 access (overwrites the 11 block entry):

Block	Tag	Data
00	1	1000
01	1	0001

10	0	0000
11	1	0100

We can use a similar procedure to determine what ends up in the two-way set-associative cache. The only difference is that we have some freedom when we have to replace a block with new data. To make the results easy to understand, we use a least-recently-used replacement policy. For starters, let's make each bank the size of the original direct-mapped cache. The final state of the two-way set-associative cache follows:

Block	Bank 0 tag	Bank 0 data	Bank 1 tag	Bank 1 data
00	1	1000	—	—
01	0	1111	1	0001
10	0	0000	—	
11	0	0110	1	0100

Of course, this isn't a fair comparison for performance because the two-way set-associative cache has twice as many entries as the direct-mapped cache. Let's use a two-way, set-associative cache with two sets, giving us four blocks, the same number as in the direct-mapped cache. In this case, the index size is reduced to one bit and the tag grows to two bits.

Block	Bank 0 tag	Bank 0 data	Bank 1 tag	Bank 1 data
0	01	0000	10	1000
1	10	0111	11	0100

In this case, the cache contents significantly differ from either the direct-mapped cache or the four-block, two-way set-associative cache.

The CPU knows when it is fetching an instruction (the PC is used to calculate the address, either directly or indirectly) or data. We can therefore choose whether to cache instructions, data, or both. If cache space is limited, instructions are the highest priority for caching because they will usually provide the highest hit rates. A cache that holds both instructions and data is called a **unified cache**.

ARM caches

Various ARM implementations use different cache sizes and organizations [Fur96]. The ARM600 includes a 4-Kbyte, 64-way (wow!) unified instruction/data cache. The StrongARM uses a 16-Kbyte, 32-way instruction cache with a 32-byte block and a 16-Kbyte, 32-way data cache with a 32-byte block; the data cache uses a write-back strategy.

C55x caches

The C5510, one of the models of C55x, uses a 16-Kbyte instruction cache organized as a two-way set-associative cache with four 32-bit words per line. The instruction cache can be disabled by software if desired. It also includes two RAM sets that are designed to hold large contiguous blocks of code. Each RAM set can hold up to 4-Kbytes of code organized as 256 lines of four 32-bit words per line. Each RAM has a tag that specifies what range of addresses are in the RAM; it also includes a tag valid field to show whether the RAM is in use and line valid bits for each line.

Read full chapter

URL: <https://www.sciencedirect.com/science/article/pii/B9780123884367000039>

CACHES

ANDREW N. SLOSS, ... CHRIS WRIGHT,
in ARM System Developer's Guide, 2004

12.2.3 THE RELATIONSHIP BETWEEN CACHE AND MAIN MEMORY

Having a general understanding of basic cache memory architecture and how the cache controller works provides enough information to discuss the relationship that a cache has with main memory.

Figure 12.5 shows where portions of main memory are temporarily stored in cache memory. The figure represents the simplest form of cache, known as a *direct-mapped* cache. In a direct-mapped cache each addressed location in main memory maps to a single location in cache memory. Since main memory is much larger than cache memory, there are many addresses in main memory that map to the same single location in cache memory. The figure shows this relationship for the class of addresses ending in 0x824.

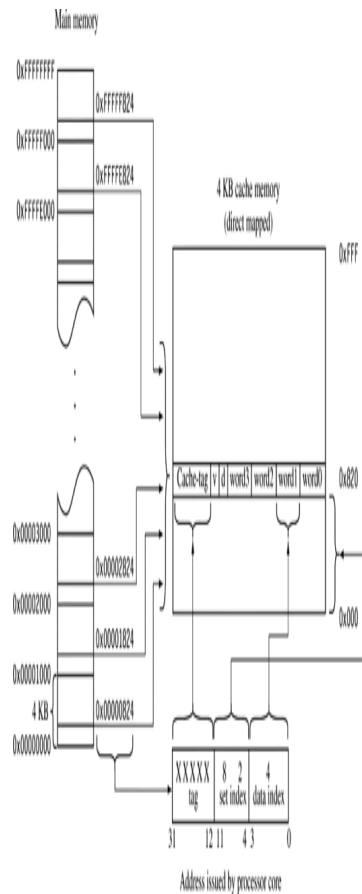


Figure 12.5. How main memory maps to a direct-mapped cache.

The three bit fields introduced in Figure 12.4 are also shown in this figure. The set index selects the one location in cache where all values in memory with an ending address of 0x824 are stored. The data index selects the word/halfword/byte in the cache line, in this case the second word in the cache line. The tag field is the portion of the address that is compared to the cache-tag value found in the directory store. In this example there are one million possible locations in main memory for every one location in cache memory. Only one of the possible one million values in the main memory can exist in the cache memory at any given time. The comparison of the tag with the cache-tag determines whether the requested data is in cache or represents another of the million locations in main memory with an ending address of 0x824.

During a cache line fill the cache controller may forward the loading data to the core at the same time it is copying it to cache; this is known as *data streaming*. Streaming allows a processor to continue execution while the cache controller fills the remaining words in the cache line.

If valid data exists in this cache line but represents another address block in main memory, the entire cache line is evicted and replaced by the cache line containing the requested address. This process of removing an existing cache line as part of servicing a cache miss is known as *eviction*—returning the contents of a cache line to main memory from the cache to make room for new data that needs to be loaded in cache.

A direct-mapped cache is a simple solution, but there is a design cost inherent in having a single location available to store a value from main memory. Direct-mapped caches are subject to high levels of *thrashing*—a software battle for the same location in cache memory. The result of thrashing is the repeated loading and eviction of a cache line. The loading and eviction result from program elements being placed in main memory at addresses that map to the same cache line in cache memory.

Figure 12.6 takes Figure 12.5 and overlays a simple, contrived software procedure to demonstrate thrashing. The procedure calls two routines repeatedly in a `do while` loop. Each routine has the same set index address; that is, the routines are found at addresses in physical memory that map to the same location in cache memory. The first time through the loop, routine A is placed in the cache as it executes. When the procedure calls routine B, it evicts routine A a cache line at a time as it is loaded into cache and executed. On the second time through the loop, routine A replaces routine B, and then routine B replaces routine A. Repeated cache misses result in continuous eviction of the routine that not running. This is cache thrashing.

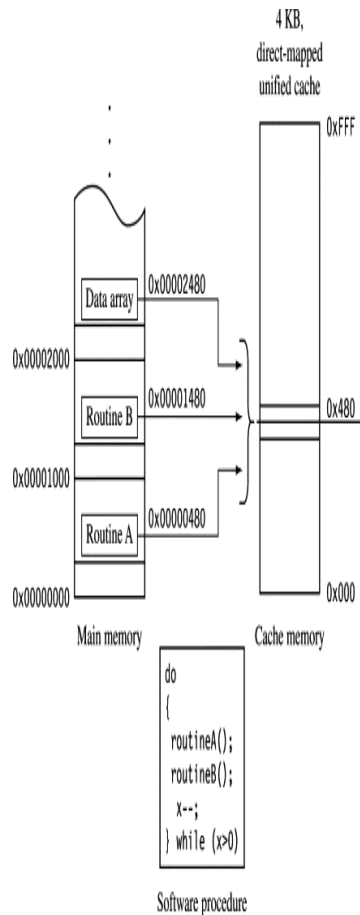


Figure 12.6. Thrashing: two functions replacing each other in a direct-mapped cache.

[Read full chapter](#)

URL: <https://www.sciencedirect.com/science/article/pii/B9781558608740500137>

Management of Cache Contents

Bruce Jacob, ... David T. Wang, in [Memory Systems](#), 2008

Dynamic Decision

The cache/not-cache decision for a particular item can change over time and is typically made at run time in response to application behavior, for example, in the same manner as adaptive training branch predictors [Yeh & Patt 1991] or adaptive filters [Haykin 2002]. One of the earliest schemes using hardware predictors to dynamically partition data is McFarling's

modification to direct-mapped caches that dynamically detects scenarios in which memory requests competing for the same cache set exhibit different degrees of locality. In such scenarios, the scheme chooses not to cache the item exhibiting poor locality. It is also possible to have a dynamic partition with software-managed memories. For instance, Kandemir's dynamic management of scratch-pad memories [Kandemir et al. 2001] changes the contents of the scratch-pad over time.

Dynamic data-prefetching heuristics include correlation and content-based schemes [Rechstschaffen 1983, Pomerene et al. 1989, Korner 1990, Tait & Duchamp 1991, Griffioen & Appleton 1994, Charney & Reeves 1995, Alexander & Kedem 1996]. Instruction-prefetching heuristics include all forms of branch predictors [Smith 1981b, Lee & Smith 1984, Yeh & Patt 1991, Yeh et al. 1993, Lee et al. 1997a].

Dynamic locality optimizations include reordering memory transactions, for instance, at the memory-controller level (see, for example, Cuppu et al. [1999], Rixner et al. 2000, Cuppu and Jacob [2001], Briggs et al. [2002], Hur and Lin [2004], and Chapter 13, “*DRAM Memory Controller*”).

[Read full chapter](#)

URL: <https://www.sciencedirect.com/science/article/pii/B9780123797513500059>

Dataflow Processing

Zivojin Sustran, ... Veljko Milutinovic, in [Advances in Computers](#), 2015

7.1 Basic Design of the Solution

The basic functioning model of this solution is splitting the cache into two parts, one handling data exhibiting temporal and the other handling data exhibiting spatial locality (as depicted in Fig. 21). Each part has a different organization and data handling, in style with data stored in it.

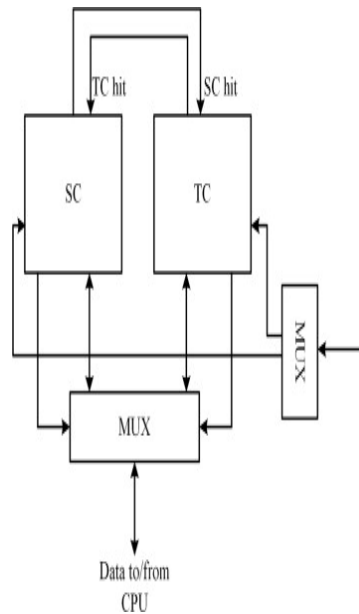


Figure 21. Organization of the STS cache.
Legend: MM—main memory; SC—spatial part; TC—temporal part; MUX—multiplexer. *Description:* the novel cache organization—STS cache consists of two parts: the temporal and the spatial parts. Temporal part is organized as a small and fast direct-mapped cache with a one-word block size. Spatial part is a larger cache and has the usual block size. *Explanation:* the optimal cache structure (from the locality-related cache organization point of view) implies that the cache is split into a “temporal” part (with a smaller and faster cache) and a “spatial” part. Spatial data usually includes a temporal component which is symbolically indicated by including a larger block for the spatial part. *Implication:* correct dimensioning of the cache hierarchy levels is of crucial importance for the performance of the STS approach.

The spatial part is a larger cache and fetches data using usual block size. This part has latencies usual for similarly sized traditional caches.

The temporal part is organized as a small and fast DM cache and works with one-word-sized blocks. Smaller size and direct mapping bring lower access latencies.

To enable marking data locality (spatial or temporal), each memory word is

augmented with one bit tag. A single tag can be used for an entire spatial block. A compiler should set those tags to an initial value and they will be changed at profile and run time. A default locality can be assumed, though it will later be clear that the default locality type should be set to spatial. During the run time, as the data is fetched, evicted, and later refetched, tags are updated and the system begins to function properly.

Created in this way, such a system can also be used with software compiled without STS support.

Processor sends a memory reference request simultaneously to both STS cache modules. When one STS part (temporal or spatial) detects a hit, a signal is sent to the other part to prevent its request to the main memory.

A memory fetch is initiated when requested data is not found in either of the STS modules. The data block length that is fetched from the memory depends upon its current locality tag. This tag determines whether the block is to be sent to the spatial or to the temporal STS part. The addressed STS part receives the data and responds to the CPU, while the other STS part cancels the request received from the CPU.

When a write-back to memory is performed, the cache also sends the information if the data has been retagged. For this purpose, the bus is widened by 1 bit to allow sending this new type of locality.

[Read full chapter](#)

URL: <https://www.sciencedirect.com/science/article/pii/S0065245814000060>

Hardware/Software Co-Synthesis with Memory Hierarchies

Yanbing Li, ... Fellow, IEEE, in [Readings in Hardware/Software Co-Design](#), 2002

Assumption 1

Only one-level cache is modeled and tasks are well-contained in the level-1 cache (each task's program size and data size are no bigger than the instruction and data cache size, respectively). This may not be a reasonable assumption in a general-purpose system, but it is plausible for many embedded systems. The kernels of time-critical operations are frequently small enough to fit into a modest-sized cache. Even when a task is too large to be contained in a level-1 cache, it can be specified at a finer granularity to satisfy this assumption.

Assumption 2

The caches are direct-mapped. We make this assumption to simplify the analysis, because in a direct-mapped cache, each data block is loaded into a deterministic location in the cache.

Assumption 3

The cache sizes are powers of two. This is usually true for most direct-mapped caches.

Assumption 4

A task's program is allocated a continuous region of the main memory and is, therefore, mapped into a continuous region of the cache. A task's data can be scattered in several regions of the main memory.

Due to the first assumption, when a task executes on a processor, if not preempted by other tasks, the only cache misses are *compulsory misses* [30]. A compulsory miss refers to the cache miss that happens when the first access to a block is not in the cache, so the block must be brought into the cache. As opposed to *capacity misses* (which occur when the cache cannot contain all the blocks needed during execution of a program and some blocks are discarded and later retrieved) and *conflict misses* (which occur when too many blocks map to one cache set, they are also called interference misses), a nice feature about compulsory misses is that their number does not change with cache size [30].

We now analyze the cache performance in two situations that the model must handle in order for it to be useful in co-synthesis: multiple tasks on a fixed-size cache, and changing the cache size.

[Read full chapter](#)

URL: <https://www.sciencedirect.com/science/article/pii/B9781558607026500235>

Logical Organization

Bruce Jacob, ... David T. Wang, in [Memory Systems](#), 2008

2.6.1 A Horizontal-Exclusive Organization: Victim Caches, Assist Caches

As an example of an exclusive partition on a horizontal level, consider Jouppi's *victim cache* [1990], a small cache with a high degree of associativity that sits next to another, larger cache that has limited set associativity. The addition of a victim cache to a larger main cache allows the main cache to approach the miss rate of a cache with higher associativity. For example, Jouppi's experiments show that a direct-mapped cache with a small fully associative victim cache can approach the miss rate of a two-way set associative cache.

The two caches in a main-cache-victim-cache pair exhibit an exclusive organization: a block is found in either the main cache or in the victim cache, but not in both. In theory, the exclusive organization gives rise to a non-trivial mapping problem of deciding which cache should hold a particular datum, but the mechanism is actually very straightforward as well as effective. The victim cache's contents are defined by the replacement policy of the main cache; that is, the only data items found within the victim cache are those that have been thrown out of the main cache at some point in the past. When a cache probe, which looks in both caches, finds the requested block in the victim cache, that block is swapped with the block it replaces in the main cache.

This swapping ensures the exclusive relationship. One of many variations on this theme is the HP-7200 *assist cache* [Chan et al. 1996], whose design was motivated by the processor's automatic stride-based prefetch engine: if too aggressive, the engine would prefetch data that would victimize soon-to-be-used cache entries. In the assist cache, the incoming prefetch block is loaded not into the main cache, but instead into the highly associative cache and is then promoted into the main cache only if it exhibits temporal locality by being referenced again soon. Blocks exhibiting only spatial locality are not moved to the main cache and are replaced to main memory in a FIFO fashion. These two management heuristics are compared later in Chapter 3 (see “Jouppi's Victim Cache, HP-7200 Assist Cache” in Section 3.1).

By adding a fully associative victim cache, a direct-mapped cache can approach the performance of a two-way set-associative cache, but because the victim cache is probed in parallel with the main cache, this performance boost comes at the expense of performing multiple tag comparisons (more than just two), which can potentially dissipate more dynamic power than a simple two-way set-associative cache. However, the victim cache organization uses fewer transistors than an equivalently performing set-associative cache, and leakage is related to the total number of transistors in the design (scales roughly with die area). Leakage power in caches is quickly outpacing dynamic power, so the victim cache is likely to become even more important in the future.

As we will see in later sections, an interesting point on the effectiveness of the victim cache is that there have been numerous studies of sophisticated dynamic heuristics that try to partition references into multiple use-based streams (e.g., those that exhibit locality and so should be cached, and those that exhibit streaming behavior and so should

go to a different cache or no cache at all). The victim cache, a simpler organization embodying a far simpler heuristic, tends to keep up with the other schemes to the point of outperforming them in many instances [Rivers et al. 1998].

A cache organization (and management heuristic) similar to the victim cache is implemented in the virtual memory code of many operating systems. A virtual memory system maintains a pool of physical pages, similar to the I/O system's buffer cache (see discussion of BSD below). The primary distinction between the two caches is that the I/O system is invoked directly by an application and thus observes all user activity to a particular set of pages, whereas the virtual memory system is transparent—invoked only indirectly when translations are needed—and thus sees almost no user-level activity to a given set of pages. Any access to the disk system is handled through a system call: the user application explicitly invokes a software routine within the operating system to read or write the disk system. In the virtual memory system, a user application accesses the physical memory directly without any assistance from the operating system, except in the case of TLB misses, protection violations, or page faults. In such a scenario, the operating system must be very clever about inferring application intent because, unlike the I/O system, the virtual memory system has no way of maintaining page-access statistics to determine each page's suitability for replacement (e.g., how recently and/or frequently has each page been referenced).

Solutions to this problem include the *clock* algorithm and other similar algorithms [Corbato 1968, Bensoussan et al. 1972, Carr & Hennessy 1981], in which pages are maintained in an LRU fashion based upon hardware providing a *use* bit for each page, a per-page indicator set by hardware whenever its corresponding page has been accessed. The clock algorithm periodically walks the list of pages and marks them ready for replacement if they have not been used recently. Modern processor

architectures do not typically support a *use* bit because the per-page reference information has traditionally been communicated to the operating system by the hardware modifying the page table directly (e.g., Bensoussan et al. [1972]), which is currently considered an extravagant overhead. However, modern architectures do support protection bits, and so the clock algorithm is often approximated in modern operating systems by tagging a page as non-readable and setting an appropriate bit when the operating system's page-protection-violation handler is invoked upon the page's first reference. If, within a period of time, the page has not been accessed, the handler is not invoked, and the corresponding bit does not get set. In the MIPS R2000 port of Mach, this mechanism is implemented through a set of queues: a page in the system is found on one of several queues based upon how recently it has been referenced on each period (execution of the clock algorithm), all unused pages are migrated one hop toward the *available* queue, and all referenced pages are moved onto the *in-use* queue. Thus, a physical page has several periods to prove its usefulness by being referenced before its time expires and it becomes available for replacement. This mechanism provides the same second-chance behavior to data that the victim cache does: blocks that are identified as unused or tagged to be replaced are not thrown out immediately, but instead move through a series of victim caches (queues) before being thrown out.

[Read full chapter](#)

URL: <https://www.sciencedirect.com/science/article/pii/B9780123797513500047>

Knights Landing architecture

Jim Jeffers, ... Avinash Sodani, in [Intel Xeon Phi Processor High Performance Programming \(Second Edition\)](#), 2016

Cache Mode

In cache mode, MCDRAM is configured as a cache for the DDR memory. The MCDRAM cache is completely hardware managed, requiring no software enablement. Legacy, unmodified software can use this mode and benefit from the high bandwidth provided by MCDRAM. The benefits will, of course, depend on the hit rate observed by the software.

MCDRAM cache is a direct-mapped cache with 64-byte cache lines. The tags are stored in the MCDRAM, as part of extra bits available in each cache line. These extra bits are read at the same time as the cache line, thereby allowing tag and data to be read on the same access, and hence, enabling the determination of a hit or miss in cache without needing a separate read access. Since MCDRAM size is much larger than traditional caches, in most cases there should be no significant change in conflict misses due to the direct-mapped policy (see Chapter 6 for further discussion on direct-mapped policy).

In this mode, all requests first go to the MCDRAM for a cache look-up and then, in case of a miss, a request is sent to DDR. The memory accesses will never bypass the MCDRAM and go to the DDR. On a miss, the data is read from the DDR and sent to the MCDRAM, to fill the cache, and the requesting tile, simultaneously.

MCDRAM cache is a *memory-side cache*, as opposed to *CPU-side caches* such as an L1, L2, or last level caches, in that a memory-side cache is closer to memory in terms of its properties as compared to CPU-side caches on the cores or tiles. It acts more like a high-bandwidth buffer sitting on the way to memory, exhibiting memory semantics, instead of a cache sitting closer to the cores. Unlike caches on the cores or tiles, MCDRAM cache does not need to be snooped by external transactions since any access to memory first goes through MCDRAM cache. An “uncacheable” memory type that does not allocate in core cache will allocate in MCDRAM cache,

since MCDRAM cache is but a part of memory, just higher bandwidth.

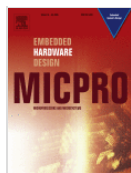
MCDRAM cache is made inclusive of all modified lines in the L2 caches. In other words, if a cache line resides in a modified state in an L2 cache, it must also be in MCDRAM. This ensures that when a modified line is written back from an L2 cache, there is no need to query the MCDRAM cache first before writing the line into it. There is no risk of overwriting a different line, since with the modified-inclusive property, the line being written back is guaranteed to be present in the MCDRAM cache. To maintain this modified-inclusive property, before a line is evicted from MCDRAM, a snoop is sent to check if a *modified* copy of that line exists in L2 cache and, if so, downgrade it from *modified* to *shared* by forcing a writeback of the modified line. The line does not get evicted from the cache.

As mentioned before, the MCDRAM cache mode is completely transparent to software and will work out of the box on unmodified code. The benefits derived from the MCDRAM cache will depend on the hit rate seen by software. The cache works well for many applications, but for applications that do not show good hit rates in the MCDRAM, the other two memory modes provide more control for applications to directly manage their use of MCDRAM (also see Chapter 3).

[Read full chapter](#)

URL: <https://www.sciencedirect.com/science/article/pii/B9780128091944000041>

Recommended publications:



[Microproc](#)
[and](#)
[Microsystem](#)
Journal



Intel Xeon
Phi Proces
High
Performar
Programn
(Second
Edition)

Book • 2016



High-
Performar
Embedde
Computin
(Second
Edition)

Book • 2014




Journal of
Parallel
and
Distribute
Computin

Journal

Browse
Journals &
Books



Copyright © 2020 Elsevier B.V. or its licensors or contributors.  RELX™
ScienceDirect® is a registered trademark of Elsevier B.V.