Cache Design

Prof. Naga Kandasamy
ECE Department, Drexel University

The following notes are derived from: D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 2018.

A cache is a *small, fast, on-chip memory* that stores instructions and data that are likely to be accessed repeatedly by the processor. Caches are typically organized in a hierarchy with multiple levels of memories such that as the distance from the processor increases, the size of the memories and the access time both increase. Caches are designed to exploit spatial and temporal locality exhibited by programs.

- *Temporal locality*: If an item is referenced in memory, it will be referenced soon. For example, most programs contain loops and so instructions and data are likely to be referenced repeatedly.
- *Spatial locality*: If an item is referenced in memory, items whose addresses are close by will be referenced soon. For example, instructions are normally accessed sequentially, and sequential accesses to elements of an array also have high degree of spatial locality.

Terminology

Let us first define some general terms applicable to caches.

- *Cache block or line*: The minimum unit of information (in bytes) that can be either present or not present in the cache. Cache block sizes typically range from 4 to 128 bytes.
- *Hit rate*: The fraction of memory accesses found in a cache. The miss rate, which is (1 - hit rate), is the fraction of memory accesses not found in the cache.
- *Hit time*: The time required to access the cache, which includes the time needed to determine whether the access is a hit or a miss. The hit time typically ranges from 1 to 4 CPU clock cycles.
- *Miss penalty*: The time required to fetch a block into the cache from main memory, including the time to access the block in main memory, transmit it to the cache, place it in the cache, and pass the block to the CPU. The miss penalty ranges from 8 to 32 CPU clock cycles.

Accessing a Cache

Restrictions on where a block is placed in the cache create three categories of cache organization.

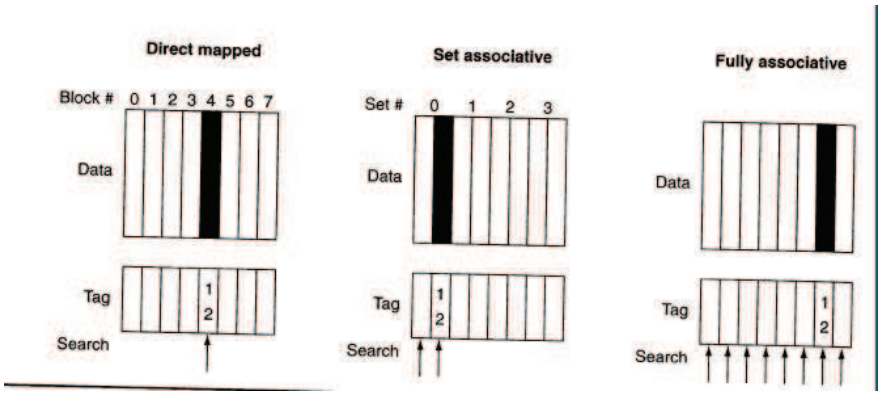

Figure 1: The location of a memory block whose address is 12 in a cache with 8 blocks using different block-placement strategies.

- If a block has only one place it can appear in the cache, the cache is said to be *direct mapped*. The mapping is as follows:

  address of the block in cache =
  (address of block in memory) modulo (number of blocks in the cache)

- If a block can be placed in a restricted set of places in the cache, the cache is said to be *set associative*. A set is a group of two or more blocks in the cache. A block is first mapped onto a set, and then the block can be placed anywhere within the set. The mapping is as follows:

  set containing block in cache =
  (address of block in memory) modulo (number of sets in the cache)

  If there are n blocks in a set, the cache placement is called *n-way set associative*.

- If a block can be placed anywhere in the cache, the cache is said to be *fully associative*.

Figure 1 shows where block 12 from main memory can be placed in a cache according to the block-placement policy. In direct-mapped placement, there is only one cache block where memory block 12 can be found, and that block is given by (12 modulo 8) = 4. In a two-way set-associative cache, there are four sets, and memory block 12 must be placed in set (12 modulo 4) = 0; the memory block can be placed in either element of the set. In fully associative, block 12 from memory can be placed in any of the 8 blocks of the cache.

Multiple memory blocks may map to the same cache location. For example, in the direct-mapped cache in Figure 1, memory block 20 maps to the same cache location as block 12, that is (12 modulo 8) = (20 modulo 8) = 4. So, how do we know if the data in cache block 4 belongs to memory block 12 or block 20? We answer this question by adding a *tag* to each cache block. The tag contains the upper portion of the address, corresponding to the bits that are not used as an index into the cache. Now, assuming that memory has 32 blocks, block 12 (01100_2) and block 20 (10100_2) will map to cache location 100_2.[1] The upper 2 bits of blocks 12 and 20 will serve as tags

[1] If the number of blocks in the cache is a power of 2, then modulo can be computed simply by using the low-order $\log 2$ (number of blocks in the cache) bits of the address.

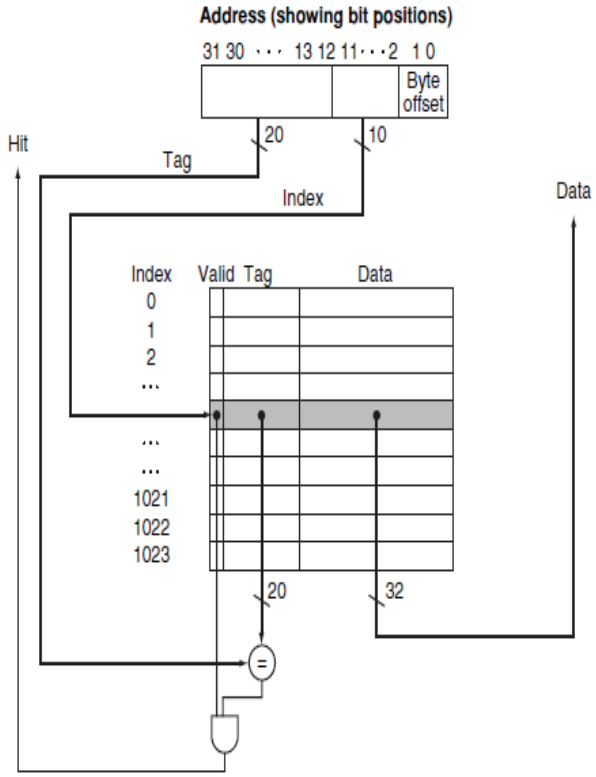

Figure 2: Structure of a direct mapped cache with 1-word cache blocks.

to distinguish between the two blocks. Finally, we need to add a *valid* bit to indicate whether an cache entry contains a valid address or not.

Figure 2 shows a 4 KB direct-mapped cache holding 1024 words. We assume 32-bit addresses. The tag from the cache is compared against the upper portion of the address to determine whether the entry in the cache corresponds to the requested address. Because the cache has 2^{10} (or 1024) words and a block size of one word (4 bytes), 10 bits are used to index the cache, leaving $32-10-2 = 20$ bits to be compared against the tag. If the tag and upper 20 bits of the address are equal and the valid bit is on, then the request hits in the cache, and the word is supplied to the processor. Otherwise, a miss occurs.

Finally, we also need a way to recognize that a cache block does not have valid information. For instance, when a processor starts up, the cache does not have good data, and the tag fields will be meaningless. Even after executing many instructions, some of the cache entries may still be empty. Thus, we need to know that the tag should be ignored for such entries. The most common method is to add a *valid bit* to indicate whether an entry contains a valid address. If the bit is not set, there cannot be a match for this block.

Mapping a Memory Address to a Multiword Cache Block

The previous section considered cache structures where the cache block is a single word—not useful in exploiting spatial locality. Now, we consider how a main memory address is mapped to a cache with multiword blocks.

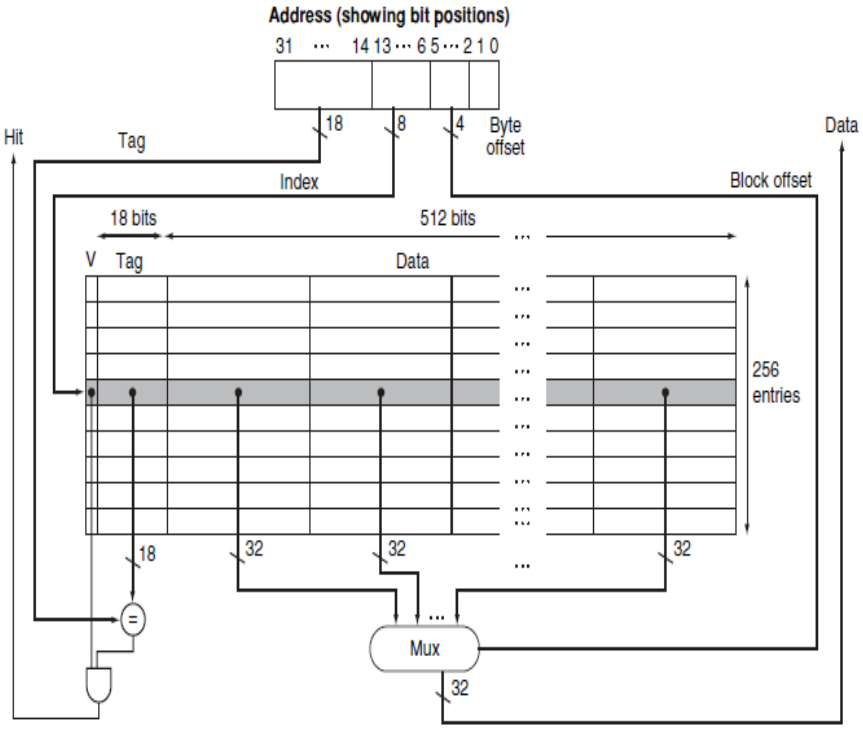

Figure 3: Structure of a 16 KB direct mapped cache with 16-word cache blocks.

Example: Consider a cache with 64 blocks and a block size of 16 bytes. To what block number does byte address 1200 map?

Consider a direct-mapped cache. We know that the mapping is as follows:

$$\begin{aligned}&\text{address of the block in cache} =\\ &(\text{address of block in memory}) \quad \text{modulo} \quad (\text{number of blocks in the cache})\end{aligned}$$

where the address of the block in memory is

$$\frac{\text{Byte address}}{\text{Bytes per block}}.$$

Notice that this block address is the block containing all addresses between

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor \times \text{Bytes per block}$$

and

$$\left\lfloor \frac{\text{Byte address}}{\text{Bytes per block}} \right\rfloor \times \text{Bytes per block} + (\text{Bytes per block} - 1)$$

Thus, with 16 bytes per block, byte address 1200 maps to memory block $\left\lfloor \frac{1200}{16} \right\rfloor = 75$, which maps to cache block number (75 modulo 64) = 11. In fact, this block maps all addresses between 1200 and 1215.

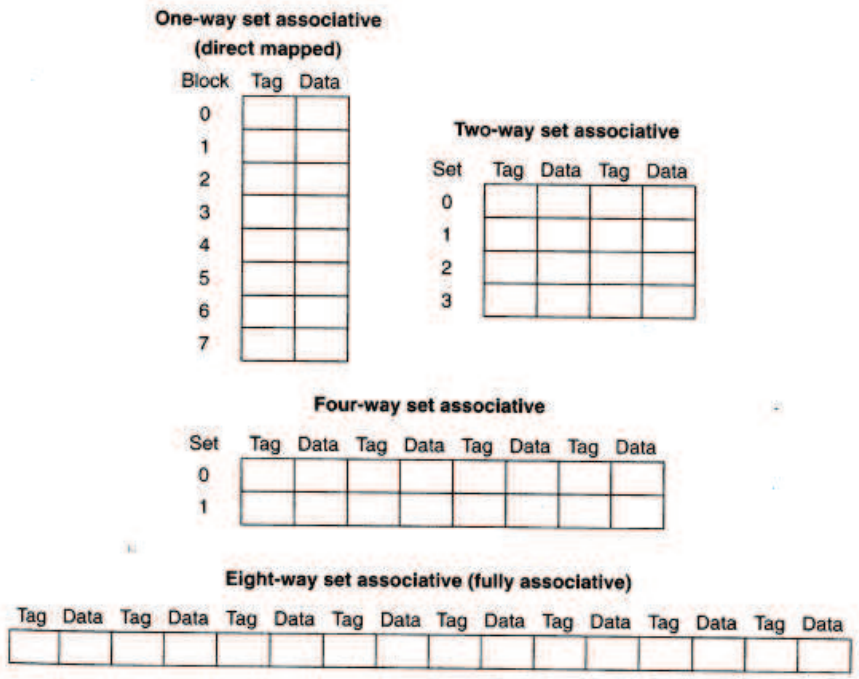

Figure 4: An eight-block cache configures as direct mapped, two-way set associative, four-way set associative, and fully associative.

Figure 3 shows the structure of a 16 KB direct-mapped cache containing 256 blocks with 16 words per block. The tag field is 18 bits wide and the index field is 8 bits wide, while a 4-bit field (bits 5–2) is used to index the block and select the word from the block using a 16-to-1 multiplexor. A read request to this cache is processed as follows:

1. Send the address to the cache. The address comes either from the PC (for an instruction) or from the ALU (for data).
2. If the cache signals hit, the requested word is available on the data lines. Since there are 16 words in the desired block, we need to select the right one.
3. If the cache signals miss, we send the address to the main memory. When the memory returns with the data, we write it into the cache and then read it to fulfill the request.

Misses and Associativity in Caches

A set associative cache aims to reduce cache misses by a more flexible placement of blocks. We can think of every block placement strategy as a variation on set associativity. Figure 4 shows the 8-block cache configured as direct mapped, two-way set associative, four-way set associative, and fully associative. The total size of the cache in blocks is equal to the number of sets times the associativity. Thus, for a fixed cache size, increasing the associativity decreases the number of sets, while increasing the number of elements per set. With eight blocks, an eight-way set associative cache is the same as a fully associative cache.

Example: Assume there are three small caches: fully associative, two-way set associative, and direct mapped, each comprising four one-word blocks. We will find the number of misses for each

Block address	Cache block
0	(0 modulo 4) = 0
6	(6 modulo 4) = 2
8	(8 modulo 4) = 0

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

Figure 5: Operation of a direct-mapped cache containing four one-word blocks for a sequence of memory addresses

Block address	Cache set
0	(0 modulo 2) = 0
6	(6 modulo 2) = 0
8	(8 modulo 2) = 0

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Figure 6: Operation of a two-way set associative cache for a sequence of memory addresses

cache organization given the following sequence of main memory addresses: 0, 8, 0, 6, 8. Figure 5 shows how the main memory addresses map to cache blocks in the case of a direct-mapped cache, and the state of the cache after all the memory references have been processed. The direct-mapped cache generates five misses for the five accesses.

Figure 6 shows how the main memory addresses map to sets in the two-way associative cache and its operation. The cache has two sets with indices 0 and 1, with two elements per set. Because we have a choice of which entry in a set to replace on a miss, we use the least recently used (LRU) replacement rule, that is, we replace the block that was used furthest in the past. So, when block 6 is referenced, it replaces block 8, since block 8 has been less recently referenced than block 0. The two-way set associative cache has four misses and one hit.

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

Figure 7: Operation of a fully associative cache for a sequence of memory addresses

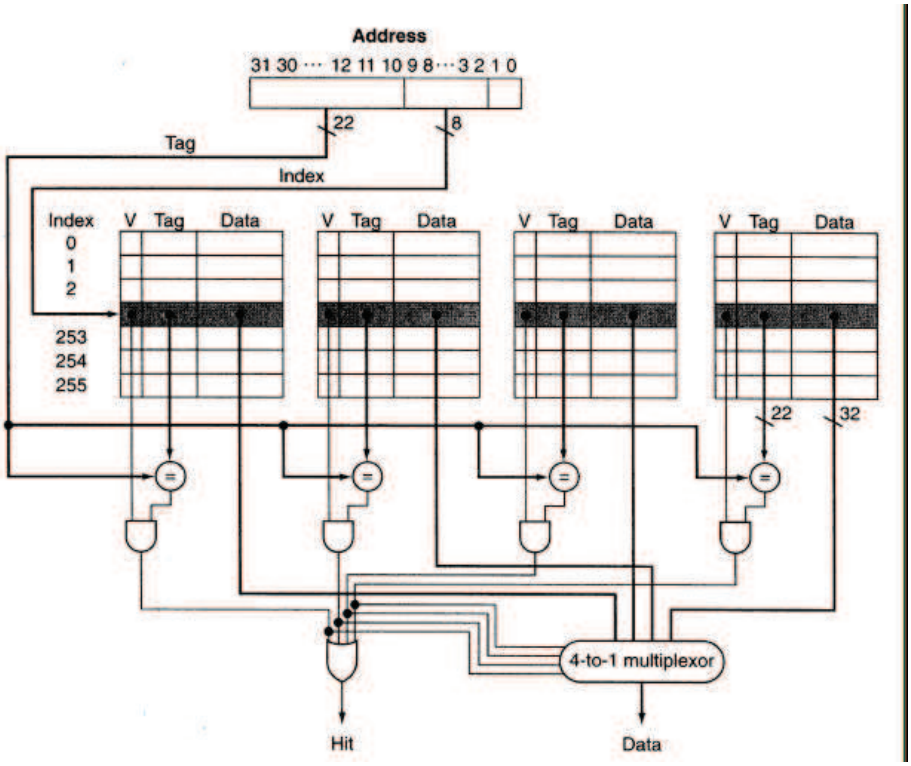

Figure 8: The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor. The comparators determine which element of the selected set (if any) matches the tag. The output of the comparators is used to select the data from one of the four blocks of the indexed set, using a multiplexor with a decoded select signal.

Now, let us consider the general problem of finding a block in a set-associative cache. Figure 8 shows how a 32-bit main memory address provided by the processor is decomposed for a four-way set associative cache with 1KB of data and 1-word blocks. The index value is used to select the set containing the address of interest, and *the tags of all blocks in the set* must be checked to see if any of the tags match the block address from the processor. Because speed is important, all tags in the selected set are searched in parallel, since a sequential search would make the hit time of a set-associative cache too slow. Four comparators are needed to perform the search in parallel, together with a 4-to-1 multiplexor to choose among the four potential members of the selected set. The cache access consists of indexing the appropriate set and then searching the tags of the set.

The costs of an associative cache are the extra comparators and any delay imposed by having to do the compare and select from among the elements of the set. Increasing associativity requires more comparators, and more tag bits per cache block.

The choice among direct-mapped, set-associative, or fully associative caches depends on the cost of a miss versus the cost of implementing associativity, both in time and extra hardware.

Figure 7 shows the operation of a fully associative cache with four cache blocks in a single set. Any memory block can be stored in any cache block. The fully associative cache has the best performance with three misses and two hits.

To summarize, one measure of the benefits of different cache organizations is miss rate: the number of accesses that miss in the cache divided by the number of accesses. Misses fall into three categories:

- *Compulsory*: The very first access to a block cannot be in the cache, so the block must be brought into the cache. Compulsory misses are those that occur even if you had an infinite sized cache.
- *Capacity*: If the cache cannot contain all the blocks needed during execution of a program, capacity misses (in addition to compulsory misses) will occur because of blocks being discarded and later retrieved.
- *Conflict*: If the placement strategy is not fully associative, conflict misses (in addition to compulsory and capacity misses) will occur because a block may be discarded and later retrieved if multiple blocks map to its set and accesses to the different blocks are intermingled.

Block Replacement Policy

When a miss occurs in direct-mapped cache, the requested memory block can go in exactly one position, and the block currently occupying that position in the cache must be replaced. In an associative cache, however, we have a choice of where to place the requested block, and hence a choice of which block to replace. In a fully associative cache, all blocks are candidates for replacement. In a set-associative cache, we must choose among the blocks in the selected set.

There are three primary replacement strategies in set-associative or fully associative caches:

- *Random*: Candidate blocks are randomly selected.
- *Least recently used* (LRU): The block replaced is the one that has been unused for the longest time. LRU replacement is implemented by keeping track of when each element in a set was used relative to other elements in the set. For a two-way set-associative cache, tracking when the two elements were used can be implemented by keeping a single bit in each set and setting the bit to indicate an element whenever that element is referenced. In practice, LRU is too costly to implement for higher set associativities. Even for four-way set associativity, LRU is often approximated—for example, by keeping track of which of a pair of blocks is LRU (which requires 1 bit), and then tracking which block in each pair is LRU (which requires 1 bit per pair).
- *First in, first out*: Since LRU can be complicated to calculate, this approximates LRU by determining the oldest block rather than the least recently used.

Handling Writes in Caches

There are two basic options for handling writes to cache memory. In the case of a write hit in the cache, we can follow one of two policies:

- *Write-through*. The information is written to both the block in the cache and the corresponding block in main memory.
- *Write-back*. The information is written only to the block in the cache. The modified block, indicated by a *dirty bit*, is written to main memory only when it is replaced.

The other key aspect of writes is what occurs on a write miss. Since the data are not needed on a write, there are two options on a write miss:

- *Write allocate*: The block is allocated on a write miss, followed by the write hit actions above. That is, the block is fetched from memory and then the appropriate portion of the block is overwritten.
- *No-write allocate*: An alternative strategy is to update the portion of the block in memory but not put it in the cache. The motivation is that sometimes programs write entire blocks of data, such as when the operating system zeros a page of memory. In such cases, the fetch associated with the initial write miss may be unnecessary.

As an example of the impact of the above allocation policies, consider a fully associative write-back cache in which the cache entries start off as empty. Consider the following sequence of five memory operations:

```
Write Mem[100];
Write Mem[100];
Read Mem[200];
Write Mem[200];
Write Mem[100];
```

For no-write allocate, the address 100 is not in the cache, and there is no allocation on write, so the first two writes will result in misses. Address 200 is also not in the cache, so the read is also a miss. The subsequent write to address 200 is a hit. The last write to 100 is still a miss. The result for no-write allocate is four misses and one hit.

For write allocate, the first accesses to memory addresses 100 and 200 are misses, and the rest are hits since 100 and 200 are both found in the cache. Thus, the result for write allocate is two misses and three hits.

Finally, with a write-through scheme, every write causes the data to be written to main memory. These writes will take a long time, likely at least 100 processor clock cycles, and could slow down the processor considerably. For example, suppose 10% of the instructions are stores. If the CPI without cache misses was 1.0, spending 100 extra cycles on every write would lead to a CPI of $1.0 + 100 \times 0.1 = 11$, reducing performance by more than a factor of 10. One solution to this problem is to use a *write buffer*. A write buffer stores the data while it is waiting to be written to memory. After writing the data into the cache and into the write buffer, the processor can continue execution. When a write to main memory completes, the entry in the write buffer is freed. If the write buffer is full when the processor reaches a write, the processor must stall until there is an

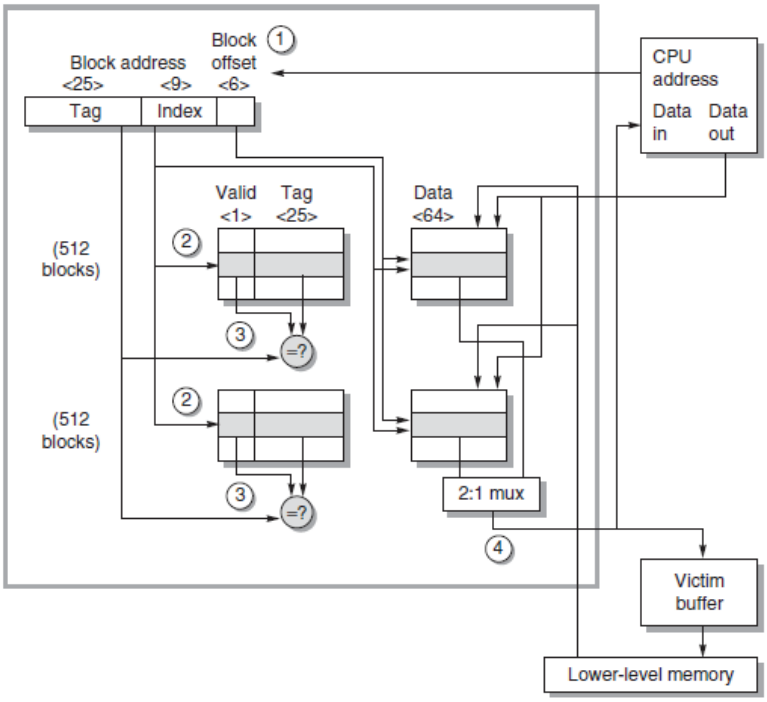

Figure 9: The organization of the data cache in the Opteron microprocessor.

empty position in the write buffer. Of course, if the rate at which the memory can complete writes is less than the rate at which the processor is generating writes, no amount of buffering can help, because writes are being generated faster than the memory system can accept them.

Example: The Opteron Data Cache

Figure 9 shows the organization of the data cache in the AMD Opteron microprocessor. The cache contains 65,536 (64K) bytes of data in 64-byte blocks with two-way set associative placement, LRU replacement, write-back, and write allocate on a write miss.

- *Step* 1: The 40-bit physical address coming into the cache is divided into two fields: the 34-bit block address and the 6-bit block offset (64 = 26 and 34 + 6 = 40). The block address is further divided into an address tag and cache index. The number of index bits is calculated as follows:

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} = \frac{65,536}{64 \times 2} = 512 = 2^9.$$

  The index is 9 bits wide and the tag is $34 - 9 = 25$ bits wide.

- *Step* 2: The 9-bit index selects among 512 sets. We index into the appropriate set within the cache and read the tags.

- *Step* 3: The tags read from the set are compared to the tag portion of the physical address form the processor. To be sure that the tag contains valid information, the valid bit must be set; else the results of the comparison are ignored.

- *Step* 4: Assuming that one tag does match, we signal the processor to load the proper data from the cache by using the winning input from the 2:1 Mux. The Opteron allows two clock cycles for these four steps; so the instructions in the following two clock cycles would wait if they tried to use the result of the load.

On a read miss, the cache sends a signal to the processor telling it the data are not yet available, and 64 bytes are read from the next level of the hierarchy. The latency is 7 clock cycles to the first 8 bytes of the block, and then 2 clock cycles per 8 bytes for the rest of the block. Since the data cache is set associative, there is a choice on which block to replace. Opteron uses LRU, which selects the block that was referenced longest ago, so every access must update the LRU bit. Replacing a block means updating the data, the address tag, the valid bit, and the LRU bit.

The Opteron uses a write-back cache; so, the old data block could have been modified, and hence it cannot simply be discarded. The Opteron keeps one dirty bit per block to record if the block was written. If the "victim" was modified, its data and address are sent to the victim buffer. (This structure is similar to a write buffer in other computers.) The Opteron has space for eight victim blocks. In parallel with other cache actions, it writes victim blocks to the next level of the hierarchy. If the victim buffer is full, the cache must wait.

A write miss is very similar to a read miss, since the Opteron allocates a block on a read or a write miss.

Quantifying Cache Performance

The average memory access time (AMAT) of a system with a cache hierarchy is given by:

$$\text{AMAT} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}.$$

The components of average access time can be measured either in absolute time on a hit or in the number of clock cycles that the processor waits for the memory—such as a miss penalty of 150 to 200 clock cycles.

If we assume that memory accesses dominate other reasons for pipeline stalls, then the CPU time can be calculated as

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory stall clock cycles}) \times \text{Clock cycle time}$$

Consider an in-order execution computer. Assume that the cache miss penalty is 200 clock cycles, and all instructions normally take 1.0 clock cycles (ignoring memory stalls). Assume that the average miss rate is 2%, there is an average of 1.5 memory references per instruction, and the average number of cache misses per 1000 instructions is 30. We can now calculate the impact on performance when behavior of the cache is included, using both misses per instruction as well as the miss rate.

The CPU time is given by

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$

The performance including cache misses is

$$\begin{aligned}\text{CPU time}_{\text{with cache}} &= \text{IC} \times (1.0 + 30/1000 \times 200) \times \text{Clock cycle time} \\ &= \text{IC} \times 7.00 \times \text{Clock cycle time}\end{aligned}$$

The CPU performance can also be calculated using the miss rate:

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

$$\begin{aligned}\text{CPU time}_{\text{with cache}} &= \text{IC} \times (1.0 + 1.5 \times .02 \times 200) \times \text{Clock cycle time} \\ &= \text{IC} \times 7.00 \times \text{Clock cycle time}\end{aligned}$$

The clock cycle time and instruction count are the same, with or without a cache. Thus, CPU time increases sevenfold, with CPI from 1.00 for a "perfect cache" to 7.00 with a cache that can miss. Without any memory hierarchy at all the CPI would increase again to $1.0 + 200 \times 1.5 = 301$—a factor of more than 40 times longer than a system with a cache.

Basic Cache Optimizations

We know that the average memory access time is given by:

$$\text{AMAT} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}.$$

A number of cache optimizations can be considered for reducing the miss rate and the miss penalty.

- *Larger block size to reduce miss rate*: The simplest way to reduce miss rate is to increase the block size. Larger block sizes will reduce also compulsory misses. This reduction occurs because the principle of locality has two components: temporal locality and spatial locality. Larger blocks take advantage of spatial locality.

  Larger blocks increase the miss penalty. Since they reduce the number of blocks in the cache, larger blocks may increase conflict misses and even capacity misses if the cache is small. There is also no benefit to reducing miss rate if it increases the average memory access time. The increase in miss penalty may outweigh the decrease in miss rate.

- *Larger caches to reduce miss rate*: This reduces capacity misses at the expense of potentially longer hit time and higher cost and power.

- *Higher associativity to reduce miss rate*: Caches with higher associativities reduce the miss rate. Greater associativity can come at the cost of increased hit time. Hence, the pressure of a fast processor clock cycle encourages simple cache designs, but the increasing miss penalty rewards associativity.

- *Multilevel caches to reduce miss penalty*: The performance gap between processors and memory leads to this question: Should I make the cache faster to keep pace with the speed of processors, or make the cache larger to overcome the widening gap between the processor and main memory? One answer is, do both. Adding another level of cache between the original cache and memory simplifies the decision. The first-level cache can be small enough to match the clock cycle time of the fast processor. Yet, the second-level cache can be large enough to capture many accesses that would go to main memory, thereby lessening the effective miss penalty.

  The average memory access time for a two-level cache can be calculated as follows. Let subscripts L1 and L2 refer to a first-level and a second-level cache, respectively, then

  $$\text{AMAT} = \text{Hit time}_{\text{L1}} + \text{Miss rate}_{\text{L1}} \times \text{Miss penalty}_{\text{L1}},$$

  and

  $$\text{Miss penalty}_{\text{L1}} = \text{Hit time}_{\text{L2}} + \text{Miss rate}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}}.$$

  In this formula, the L2 miss rate is measured on the leftovers from the L1 cache. These terms are adopted here for a two-level cache system:

  - *Local miss rate*: This rate is simply the number of misses in a cache divided by the total number of memory accesses to this cache. For the first-level cache it is equal to $\text{Miss rate}_{\text{L1}}$, and for the second-level cache it is $\text{Miss rate}_{\text{L2}}$.
  - *Global miss rate*: The number of misses in the cache divided by the total number of memory accesses generated by the processor. Using the terms above, the global miss rate for the first-level cache is still just $\text{Miss rate}_{\text{L1}}$, but for the second-level cache it is $\text{Miss rate}_{\text{L1}} \times \text{Miss rate}_{\text{L2}}$.

  The global miss rate indicates what fraction of memory accesses leave the processor and go all the way to memory.

  Suppose that in 1000 memory references there are 40 misses in the first-level cache and 20 misses in the second-level cache. Then the various miss rates at these caches are as follows: $\text{Miss rate}_{\text{L1}} = 40/1000 = 4\%$; $\text{Miss rate}_{\text{L2}} = 20/40 = 50\%$; and the global miss rate of the second level cache is $20/1000 = 2\%$. If the miss penalty from the L2 cache to memory is 200 clock cycles, the hit time of the L2 cache is 10 clock cycles, the hit time of L1 is 1 clock cycle, and there are 1.5 memory references per instruction, then

  $$\begin{aligned}\text{AMAT} &= \text{Hit time}_{\text{L1}} + \text{Miss rate}_{\text{L1}} \times (\text{Hit time}_{\text{L2}} + \text{Miss rate}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}}) \\ &= 1 + 4\% \times (10 + 50\% \times 200) \\ &= 1 + 4\% \times 110 \\ &= 5.4 \text{ clock cycles}\end{aligned}$$

  To calculate the average stall cycles per instruction, we calculate how many misses we get per instruction. Since we have 1000 memory references and 1.5 references per instruction,

we have a total of $1000/1.5 = 667$ instructions. To get the number of misses per 1000 instructions, we have $40 \times 1.5 = 60$ L1 misses and $20 \times 1.5 = 30$ L2 misses.

$$\begin{aligned}\text{Avg. memory stalls/instruction} &= \text{Misses/instruction}_{\text{L1}} \times \text{Hit time}_{\text{L2}} + \\ &\quad \text{Misses/instruction}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}} \\ &= 60/1000 \times 10 + 30/1000 \times 200 \\ &= 6.6 \text{ clock cycles}\end{aligned}$$

- *Prioritizing read misses over writes to reduce miss penalty*: This optimization serves reads before writes have been completed. However, doing this introduces some consistency problems. Assume a direct-mapped, write-through cache that maps 512 and 1024 to the same block, and a four-word write buffer that is not checked on a read miss. Consider the following code snippet:

```
SW R3, 512(R0)  ; M[512] = R3  (cache index 0)
LW R1, 1024(R0) ; R1 = M[1024] (cache index 0)
LW R2, 512(R0)  ; R2 = M[512]  (cache index 0)
```

  The data in R3 is placed into the write buffer after the store. The following load uses the same cache index and is therefore a miss. The second load instruction tries to put the value in location 512 into register R2; this also results in a miss. If the write buffer hasn't completed writing to location 512 in memory, the read of location 512 will put the old, wrong value into the cache block, and then into R2.

  The simplest solution for the above problem is for the read miss to wait until the write buffer is empty. The alternative is to check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue.

  The cost of writes in a write-back cache can also be reduced. Suppose a read miss will replace a dirty memory block. Instead of writing the dirty block to memory, and then reading memory, we could copy the dirty block to a buffer, then read memory, and then write memory. This way the processor read, for which the processor is probably waiting, will finish sooner. Similar to the previous situation, if a read miss occurs, the processor can either stall until the buffer is empty or check the addresses of the words in the buffer for conflicts.

- *Code optimizations to reduce miss rate*: This technique reduces miss rates without any hardware changes.

  – *Loop interchange*: Some programs have nested loops that access data in memory in non-sequential order. Simply exchanging the nesting of the loops can make the code access the data in the order in which they are stored. Assuming the arrays do not fit in the cache, this technique reduces misses by improving spatial locality; reordering maximizes use of data in a cache block before they are discarded. For example, if x is a two-dimensional array of size $[5000, 100]$ allocated so that $x[i, j]$ and $x[i, j + 1]$ are adjacent (an order called row major, since the array is laid out by rows), then the two pieces of code below show how the accesses can be optimized:

```
/* Before and after */
```

```
for (j = 0; j < 100; j = j+1)
    for (i = 0; i < 5000; i = i+1)
        x[i][j] = 2 * x[i][j];

for (i = 0; i < 5000; i = i+1)
    for (j = 0; j < 100; j = j+1)
        x[i][j] = 2 * x[i][j];
```

– *Tile blocking*: Instead of operating on entire rows or columns of an array, blocked algorithms operate on sub-matrices or blocks. Consider the code example below, which performs matrix multiplication of two $N \times N$ matrices y and z:

```
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1){
        r = 0;
        for (k = 0; k < N; k = k + 1)
            r = r + y[i][k]*z[k][j];
        x[i][j] = r;
    }
```

The two inner loops read all N-by-N elements of z, read the same N elements in a row of y repeatedly, and write one row of N elements of x. Our goal is to maximize accesses to the data loaded into the cache before the data are replaced. The original code is changed to compute on a sub-matrix of size B by B, where B is the block size. Two inner loops now compute in steps of size B rather than the full length of x and z. Assume x is initialized to zero.

```
for (jj = 0; jj < N; jj = jj+B)
    for (kk = 0; kk < N; kk = kk+B)
        for (i = 0; i < N; i = i+1)
            for (j = jj; j < min(jj+B,N); j = j+1){
                r = 0;
                for (k = kk; k < min(kk+B,N); k = k + 1)
                    r = r + y[i][k]*z[k][j];
                x[i][j] = x[i][j] + r;
}
```