Caches

- A cache is a small, fast array of memory placed between the processor core and main memory that stores portions of recently referenced main memory.
- The processor uses cache memory instead of main memory whenever possible to increase system performance.
- The goal of a cache is to reduce the memory access bottleneck imposed on the processor core by slow memory.
- Often used with a cache is a write buffer—a very small first-in-first-out (FIFO) memory placed between the processor core and main memory.
- The purpose of a write buffer is to free the processor core and cache memory from the slow write time associated with writing to main memory.
- Both the cache and write buffer have additional control hardware that automatically handles the movement of code and data between the processor and main memory.
- The main drawback of cache memory is the difficulty of determining the execution time of a program.
  - Since cache memory only represents a very small portion of main memory, the cache fills quickly during program execution.
  - Once full, the cache controller frequently evicts existing code or data from cache memory to make more room for the new code or data. This eviction process tends to occur randomly, leaving some data in cache and removing others. Thus, at any given instant in time, a value may or may not be stored in cache memory.
  - Because data may or may not be present in cache at any given point in time, the execution time of a routine may vary slightly from run to run due to the difference between the time it takes to use data immediately out of cache memory and the time it takes to load a cache line from main memory.
- The repeated use of the same code or data in memory, or those very near, is the reason a cache improves performance.

- By loading the referenced code or data into faster memory when first accessed, each subsequent access will be much faster.
- It is the repeated access to the faster memory that improves performance.
- The cache makes use of this repeated local reference in both time and space. If the reference is in time, it is called temporal locality. If it is by address proximity, then it is called spatial locality.

The memory hierarchy and cache memory

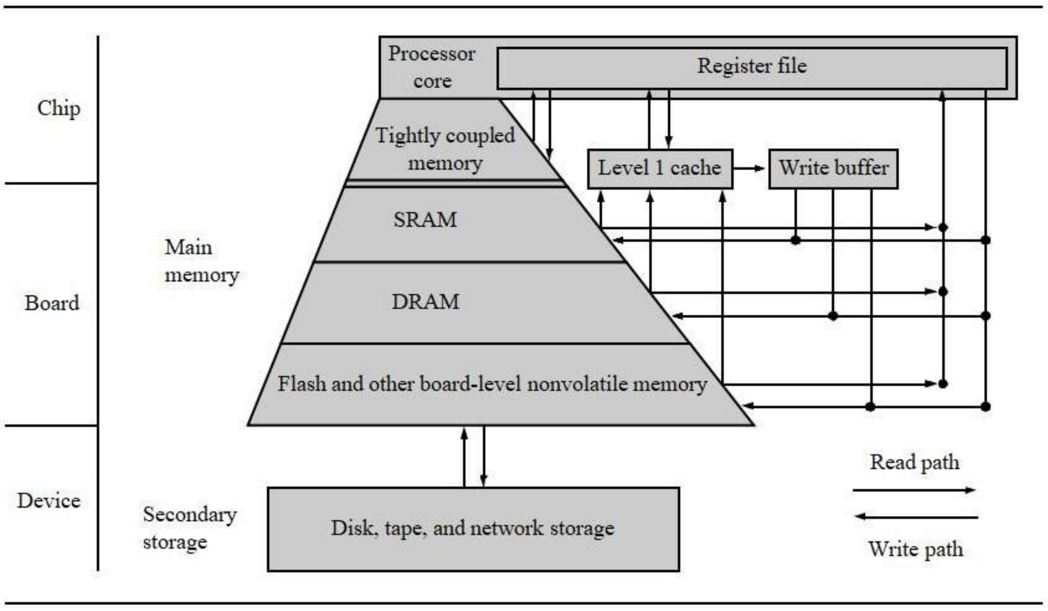

Figure 12.1 Memory hierarchy.

- The innermost level of the hierarchy is at the processor core. This memory is known as a register file. These registers are integral to the processor core and provide the fastest possible memory access in the system.
- At the primary level, memory components are connected to the processor core through dedicated on-chip interfaces. It is at this level we find tightly coupled memory (TCM) and level 1 cache.
- Also at the primary level is main memory. It includes volatile components like SRAM and DRAM, and nonvolatile components like flash memory. The purpose of main memory is to hold programs while they are running on a system.
- The next level is secondary storage—large, slow, relatively inexpensive mass storage devices such as disk drives or removable memory. Also included in

this level is data derived from peripheral devices, which are characterized by their extremely long access times. Secondary memory is used to store unused portions of very large programs that do not fit in main memory and programs that are not currently executing.

- Memory hierarchy depends as much on architectural design as on the technology surrounding it. For example, TCM and SRAM are of the same technology yet differ in architectural placement: TCM is located on the chip, while SRAM is located on a board.
- A cache may be incorporated between any level in the hierarchy where there is a significant access time difference between memory components. A cache memory system takes information stored in a lower level of the hierarchy and temporarily moves it to a higher level.
- Figure 12.1 includes a level 1 (L1) cache and write buffer. The L1 cache is an array of high-speed, on-chip memory that temporarily holds code and data from a slower level. A cache holds this information to decrease the time required to access both instructions and data.
- The write buffer is a very small FIFO buffer that supports writes to main memory from the cache.
- Not shown in the figure is a level 2 (L2) cache. An L2 cache is located between the L1 cache and slower memory. The L1 and L2 caches are also known as the primary and secondary caches.

Relationship that a cache has between the processor core and main memory.

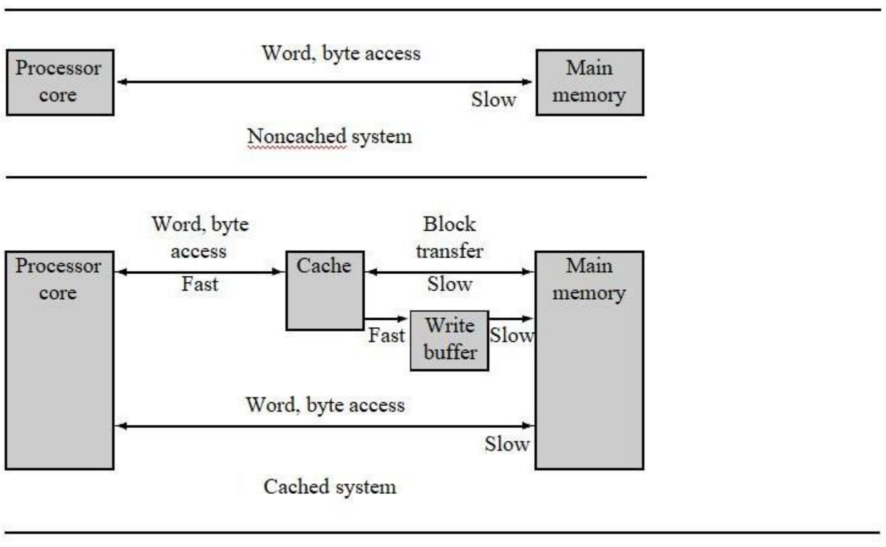

Figure 12.2 Relationship that a cache has between the processor core and main memory.

- The upper half of the figure shows a block diagram of a system without a cache. Main memory is accessed directly by the processor core using the datatypes supported by the processor core.
- The lower half of the diagram shows a system with a cache.
  - ➢ The cache memory is much faster than main memory and thus responds quickly to data requests by the core.
  - ➢ The cache's relationship with main memory involves the transfer of small blocks of data between the slower main memory to the faster cache memory. These blocks of data are known as cache lines.
  - ➢ The write buffer acts as a temporary buffer that frees available space in the cache memory.
  - ➢ The cache transfers a cache line to the write buffer at high speed and then the write buffer drains it to main memory at slow speed.

<u>Caches and Memory Management units</u>

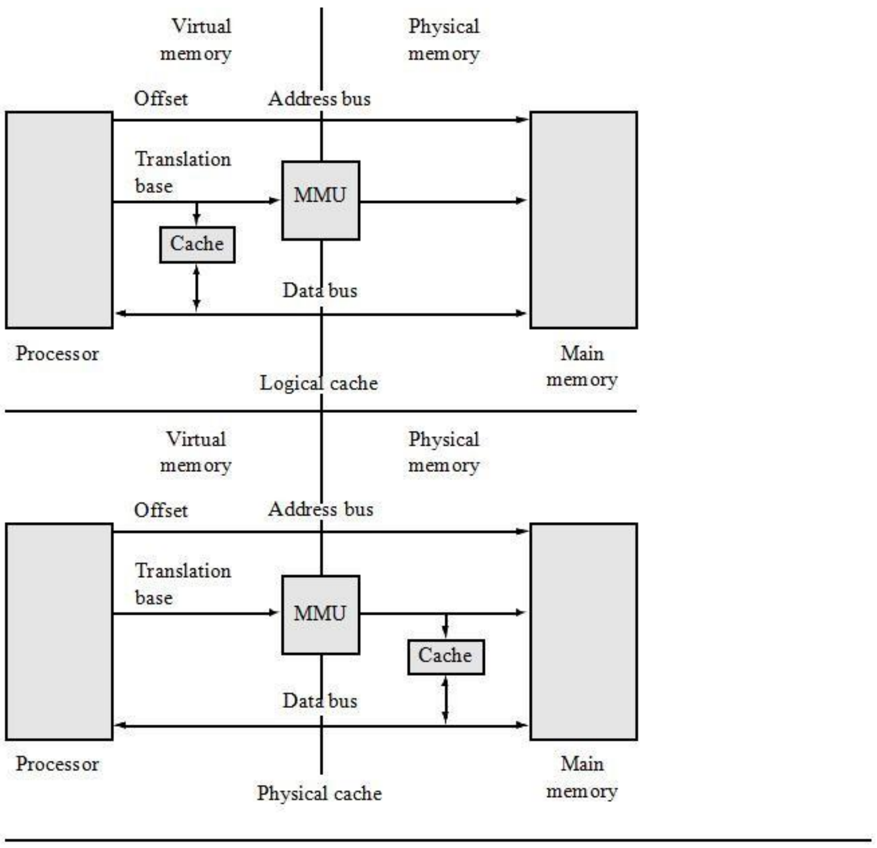

Figure 12.3 Logical and physical caches.

- If a cached core supports virtual memory, it can be located between the core and the memory management unit (MMU), or between the MMU and physical memory.
- Placement of the cache before or after the MMU determines the addressing realm the cache operates in and how a programmer views the cache memory system. Figure 12.3 shows the difference between the two caches.
- A logical cache stores data in a virtual address space.
  - A logical cache is located between the processor and the MMU.
  - The processor can access data from a logical cache directly without going through the MMU.
  - A logical cache is also known as a virtual cache.
  - ARM cached cores with an MMU use logical caches for processor families ARM7 through ARM10, including the Intel StrongARM and Intel XScale processors.
- A physical cache stores memory using physical addresses.
  - A physical cache is located between the MMU and main memory.

- For the processor to access memory, the MMU must first translate the virtual address to a physical address before the cache memory can provide data to the core.
- The ARM11 processor family uses a physical cache.

Cache Architecture

- ARM uses two bus architectures in its cached cores, the Von Neumann and the Harvard. A different cache design is used to support the two architectures.
  - In processor cores using the Von Neumann architecture, there is a single cache used for instruction and data. This type of cache is known as a unified cache. A unified cache memory contains both instruction and data values.
  - The Harvard architecture has separate instruction and data buses to improve overall system performance, but supporting the two buses requires two caches. In processor cores using the Harvard architecture, there are two caches: an instruction cache (I-cache) and a data cache (D-cache). This type of cache is known as a split cache. In a split cache, instructions are stored in the instruction cache and data values are stored in the data cache. We introduce the basic architecture of caches by showing a unified cache in Figure 12.4.
- The two main elements of a cache are the cache controller and the cache memory.
  - The cache memory is a dedicated memory array accessed in units called cache lines.
  - The cache controller uses different portions of the address issued by the processor during a memory request to select parts of cache memory.

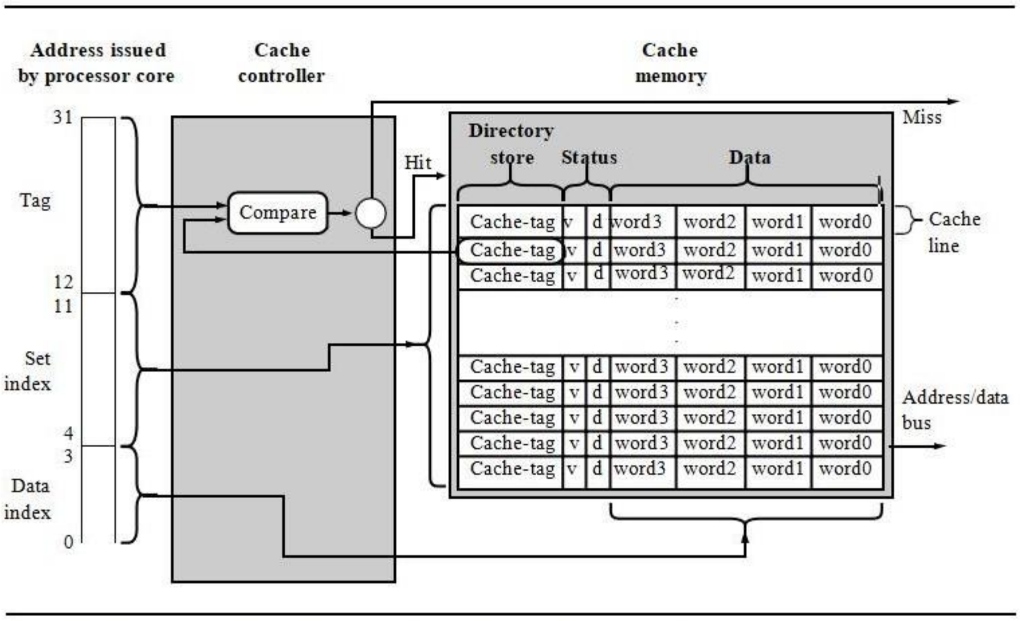

Figure 12.4 A 4 KB cache consisting of 256 cache lines of four 32-bit words.

Basic Architecture of a cache Memory

- A simple cache memory is shown on the right side of Figure 12.4.
- It has three main parts: a directory store, a data section, and status information. All three parts of the cache memory are present for each cache line.
- The cache must know where the information stored in a cache line originates from in main memory. It uses a directory store to hold the address identifying where the cache line was copied from main memory. The directory entry is known as a cache-tag.
- A cache memory must also store the data read from main memory. This information is held in the data section.
- The size of a cache is defined as the actual code or data the cache can store from main memory.
- There are also status bits in cache memory to maintain state information. Two common status bits are the valid bit and dirty bit.
  - A valid bit marks a cache line as active, meaning it contains live data originally taken from main memory and is currently available to the processor core on demand.
  - A dirty bit defines whether or not a cache line contains data that is different from the value it represents in main memory

Basic Operation of a cache controller

- The cache controller is hardware that copies code or data from main memory to cache memory automatically.
- The cache controller intercepts read and write memory requests before passing them on to the memory controller.
- It processes a request by dividing the address of the request into three fields, the tag field, the set index field, and the data index field. The three bit fields are shown in Figure 12.4.
  - First, the controller uses the set index portion of the address to locate the cache line within the cache memory that might hold the requested code or data. This cache line contains the cache-tag and status bits, which the controller uses to determine the actual data stored there.
  - The controller then checks the valid bit to determine if the cache line is active, and compares the cache-tag to the tag field of the requested address. If both the status check and comparison succeed, it is a cache hit. If either the status check or comparison fails, it is a cache miss.
  - On a cache miss, the controller copies an entire cache line from main memory to cache memory and provides the requested code or data to the processor. The copying of a cache line from main memory to cache memory is known as a cache line fill.
  - On a cache hit, the controller supplies the code or data directly from cache memory to the processor. To do this it moves to the next step, which is to use the data index field of the address request to select the actual code or data in the cache line and provide it to the processor.

The relationship between cache and Main Memory

I. 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.

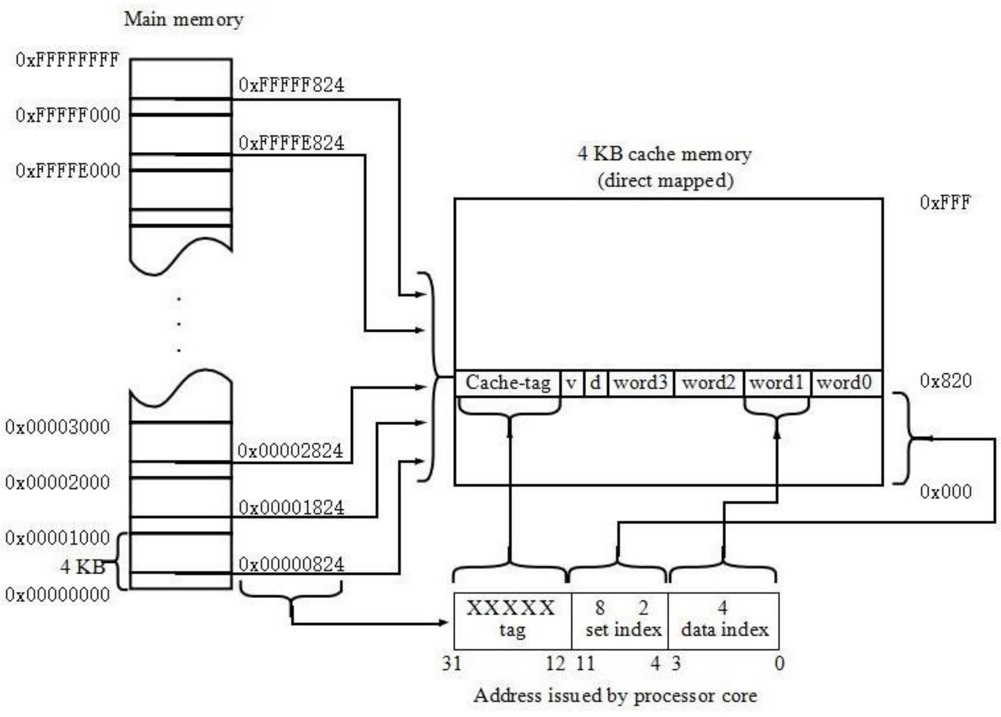

Figure 12.5 How main memory maps to a direct-mapped cache.

- 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.

Cache Thrashing

- 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.

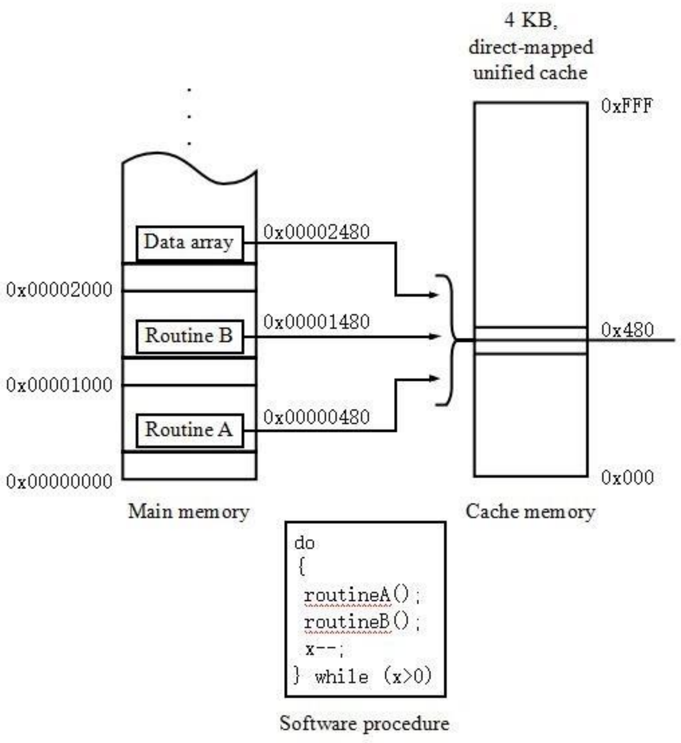

Figure 12.6 Thrashing: two functions replacing each other in a direct-mapped cache.

- 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.

II. Set- Associativity

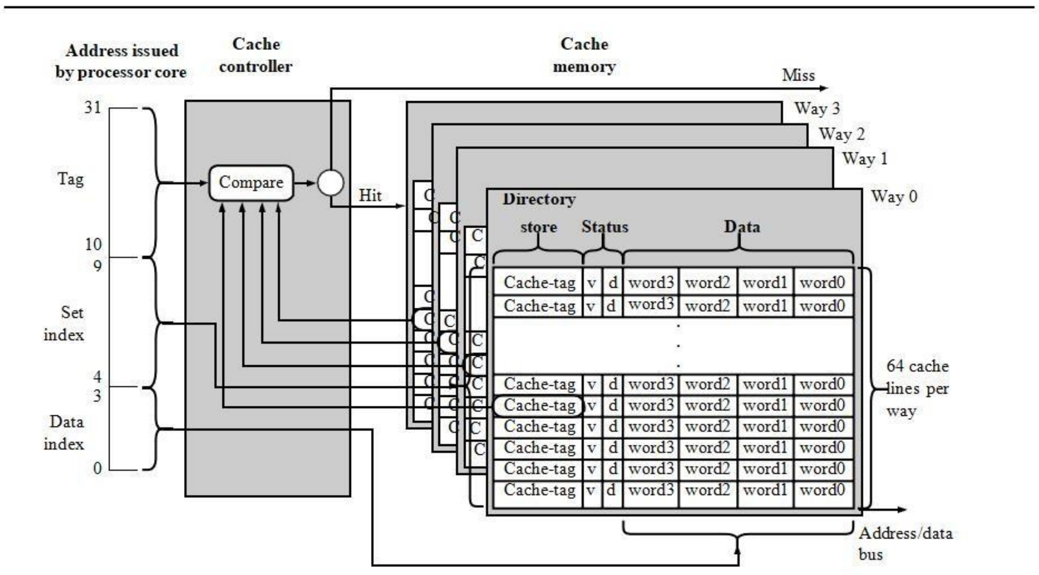

Figure 12.7 A 4 KB, four-way set associative cache. The cache has 256 total cache lines, which are separated into four ways, each containing 64 cache lines. The cache line contains four words.

-
- Set- Associative caches reduce the frequency of thrashing (see Figure 12.7).
- This structural design feature is a change that divides the cache memory into smaller equal units, called ways.
- Figure 12.7 is still a four KB cache; however, the set index now addresses more than one cache line—it points to one cache line in each way.
- Instead of one way of 256 lines, the cache has four ways of 64 lines. The four cache lines with the same set index are said to be in the same set, which is the origin of the name “set index.”
- A data or code block from main memory can be allocated to any of the four ways in a set without affecting program behavior the data or code blocks from a specific location in main memory can be stored in any cache line that is a member of a set.

- The placement of values within a set is exclusive to prevent the same code or data block from simultaneously occupying two cache lines in a set.
- The mapping of main memory to a cache changes in a four-way set associative cache. Figure 12.8 shows the differences. Any single location in main memory now maps to four different locations in the cache.

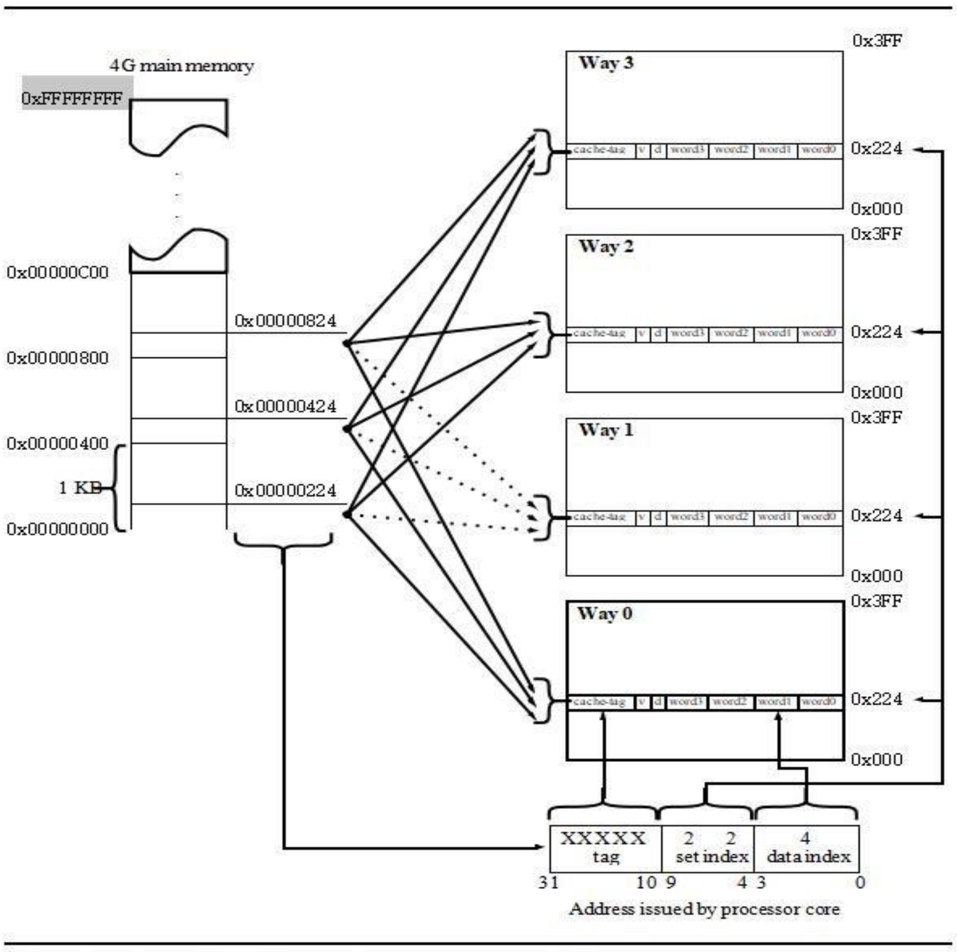

Figure 12.8 Main memory mapping to a four-way set associative cache.

- Although Figures 12.5 and 12.8 both illustrate 4 KB caches, here are some differences
  - The bit field for the tag is now two bits larger, and the set index bit field is two bits smaller. This means four million main memory addresses now map to one set of four cache lines, instead of one million addresses mapping to one location.
  - The size of the area of main memory that maps to cache is now 1 KB instead of 4 KB. This means that the likelihood of mapping cache line data blocks to the same set is now four times higher. This is offset by the fact that a cache line is one fourth less likely to be evicted.

III. Fully Associative Cache

- As the associativity of a cache controller goes up, the probability of thrashing goes down.
- The goal of fully associative cache would be to maximize the set associativity of a cache by designing it so any main memory location maps to any cache line.
- However, as the associativity increases, so does the complexity of the hardware that supports it.
- One method used by hardware designers to increase the set associativity of a cache includes a content addressable memory (CAM).
- A CAM uses a set of comparators to compare the input tag address with a cache-tag stored in each valid cache line.
- A CAM works in the opposite way a RAM works. Where a RAM produces data when given an address value, a CAM produces an address if a given data value exists in the memory.
- Using a CAM allows many more cache-tags to be compared simultaneously, thereby increasing the number of cache lines that can be included in a set.
- Figure 12.9 shows a block diagram of an ARM940T cache.
- The cache controller uses the address tag as the input to the CAM and the output selects the way containing the valid cache line.
- The tag portion of the requested address is used as an input to the four CAMs that simultaneously compare the input tag with all cache-tags stored in the 64 ways. If there is a match, cache data is provided by the cache memory. If no match occurs, a miss signal is generated by the memory controller.
- The controller enables one of four CAMs using the set index bits. The indexed CAM then selects a cache line in cache memory and the data index portion of the core address selects the requested word, halfword, or byte within the cache line.

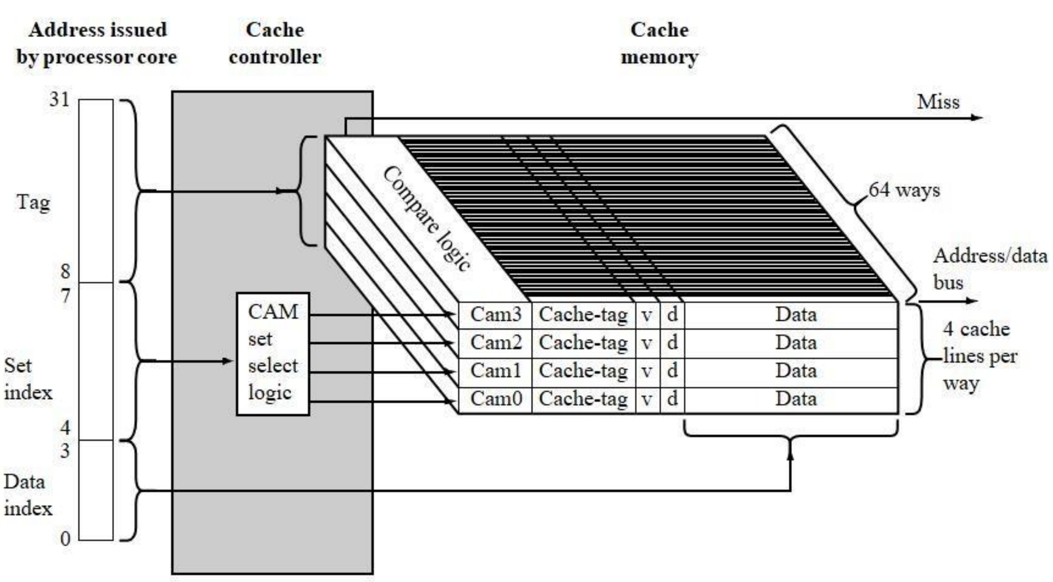

Figure 12.9 ARM940T—4 KB 64-way set associative D-cache using a CAM.

<u>Write Buffers</u>

- A write buffer is a very small, fast FIFO memory buffer that temporarily holds data that the processor would normally write to main memory.
- In a system without a write buffer, the processor writes directly to main memory. In a system with a write buffer, data is written at high speed to the FIFO and then emptied to slower main memory.
- The write buffer reduces the processor time taken to write small blocks of sequential data to main memory.
- The FIFO memory of the write buffer is at the same level in the memory hierarchy as the L1 cache.
- The efficiency of the write buffer depends on the ratio of main memory writes to the number of instructions executed.
- A write buffer also improves cache performance; the improvement occurs during cache line evictions. If the cache controller evicts a dirty cache line, it writes the cache line to the write buffer instead of main memory. Thus the new cache line data will be available sooner, and the processor can continue operating from cache memory.
- Data written to the write buffer is not available for reading until it has exited the write buffer to main memory.

Measuring Cache Efficiency

Hit Rate : It is the number of cache hits divided by the total number of memory requests over a given time interval.

$$\text{Hit Rate} = \frac{\text{Cache Hits}}{\text{Memory Requests}} * 100$$

Miss Rate : It is the total cache misses divided by the total number of memory requests expressed as a percentage over a time interval.

$$\text{Miss Rate} = \frac{\text{Cache Misses}}{\text{Memory Requests}} * 100$$

Hit Time : It is the time it takes to access a memory location in the cache.

Miss Penalty: It is the time it takes to load a cache line from main memory into cache.

Cache Policy

There are three policies that determine the operation of a cache: the write policy, the replacement policy, and the allocation policy.

- ➢ The cache write policy determines where data is stored during processor write operations.
- ➢ The replacement policy selects the cache line in a set that is used for the next line fill during a cache miss.
- ➢ The allocation policy determines when the cache controller allocates a cache line.

Write Policy

When the processor core writes to memory, the cache controller has two alternatives for its write policy.

1. Writethrough
2. Writeback or Copyback

1. Writethrough

- When the cache controller uses a writethrough policy, it writes to both cache and main memory when there is a cache hit on write, ensuring that the cache and main memory stay coherent at all times.

- Under this policy, the cache controller performs a write to main memory for each write to cache memory.
- Because of the write to main memory, a writethrough policy is slower than a writeback policy.

2. Writeback

- When a cache controller uses a writeback policy, it writes to valid cache data memory and not to main memory.
- Consequently, valid cache lines and main memory may contain different data. The cache line holds the most recent data, and main memory contains older data, which has not been updated.
- Caches configured as writeback caches must use one or more of the dirty bits in the cache line status information block.
- When a cache controller in writeback writes a value to cache memory, it sets the dirty bit true. If the core accesses the cache line at a later time, it knows by the state of the dirty bit that the cache line contains data not in main memory. If the cache controller evicts a dirty cache line, it is automatically written out to main memory. The controller does this to prevent the loss of vital information held in cache memory and not in main memory.
- One performance advantage a writeback cache has over a writethrough cache is in the frequent use of temporary local variables by a subroutine. These variables are transient in nature and never really need to be written to main memory.

Cache Line Replacement Policies

- On a cache miss, the cache controller must select a cache line from the available set in cache memory to store the new information from main memory.
- The cache line selected for replacement is known as a victim.
- If the victim contains valid, dirty data, the controller must write the dirty data from the cache memory to main memory before it copies new data into the victim cache line.
- The process of selecting and replacing a victim cache line is known as eviction.
- The strategy implemented in a cache controller to select the next victim is called its replacement policy.

- ARM cached cores support the following replacement policies - pseudorandom, round-robin or Least Recently Used (LRU).
  - **Round-robin or cyclic replacement** simply selects the next cache line in a set to replace. The selection algorithm uses a sequential, incrementing victim counter that increments each time the cache controller allocates a cache line. When the victim counter reaches a maximum value, it is reset to a defined base value.
  - **Pseudorandom replacement** randomly selects the next cache line in a set to replace. The selection algorithm uses a nonsequential incrementing victim counter. In a pseudorandom replacement algorithm the controller increments the victim counter by randomly selecting an increment value and adding this value to the victim counter. When the victim counter reaches a maximum value, it is reset to a defined base value.
  - **Least Recently Used (LRU**) : This policy keeps track of cache line use and selects the cache line that has been unused for the longest time as the next victim.

Allocation policy on a cache Miss

- There are two strategies ARM caches may use to allocate a cache line after a the occurrence of a cache miss - read-allocate, and read-write-allocate.
- A **read allocate** on cache miss policy allocates a cache line only during a read from main memory.
  - If the victim cache line contains valid data, then it is written to main memory before the cache line is filled with new data.
  - Under this strategy, a write of new data to memory does not update the contents of the cache memory unless a cache line was allocated on a previous read from main memory.
  - If the cache line contains valid data, then a write updates the cache and may update main memory if the cache write policy is writethrough. If the data is not in cache, the controller writes to main memory only.
- A **read-write allocate** on cache miss policy allocates a cache line for either a read or write to memory.
  - Any load or store operation made to main memory, which is not in cache memory, allocates a cache line.

- On memory reads the controller uses a read-allocate policy.
- On a write, the controller also allocates a cache line.
- If the victim cache line contains valid data, then it is first written back to main memory before the cache controller fills the victim cache line with new data from main memory.
- If the cache line is not valid, it simply does a cache line fill. After the cache line is filled from main memory, the controller writes the data to the corresponding data location within the cache line. The cached core also updates main memory if it is a writethrough cache.