

Chapter 10

Memory System Design

This chapter discusses the memory system design. We discuss the memory basics, how a memory module is organised. The hierarchical of memory, which is one of the most important aspects of a high performance computer system today, is introduced. The high-speed memory, the cache memory design is explored. The operating system services, which provide logical memory space, have a strong implication on the memory system design. The relationship between processors and operating systems are discussed. We conclude the chapter with a discussion of the memory technology, its rapid changes and the future of technology.

Memory basics

There are many types of memory in a computer system. The major type of semiconductor memory is random-access-memory (RAM). We will discuss the memory technology topic in the later section. A memory module consists of an array of memory cells. A memory cell can store one bit of information. To read or write a memory cell, it must be selected (addressed) and the control signal (read/write) is asserted. The data can be read off or written into the cell via the data line.

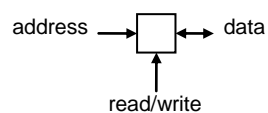


Figure 10.1 a memory cell

A memory module is built on an array of memory cells. The most widely used type of RAM is *dynamic* RAM (DRAM). A DRAM is made with cells that store data as charge on capacitors. It requires periodic refreshing of the cell's content. Fig. 10.2 shows a typical organisation of a 16 Mbit DRAM (4M × 4 bits). The memory array is organised as four arrays of 2048 × 2048 elements. The address lines supply the address $A_0 \dots A_{11}$. They are fed into row and column decoders ($2^{11} = 2048$). The row and column addresses are multiplexed to reduce the number of pin of the memory package. The row-address-strobe (RAS) and the column-address-strobe (CAS) signals provide the control to the memory chip. The circuits on the chip included refreshing logic and input/output (I/O) lines interface to the external bus. A number of these typical chips are used to build up a larger memory for a computer system.

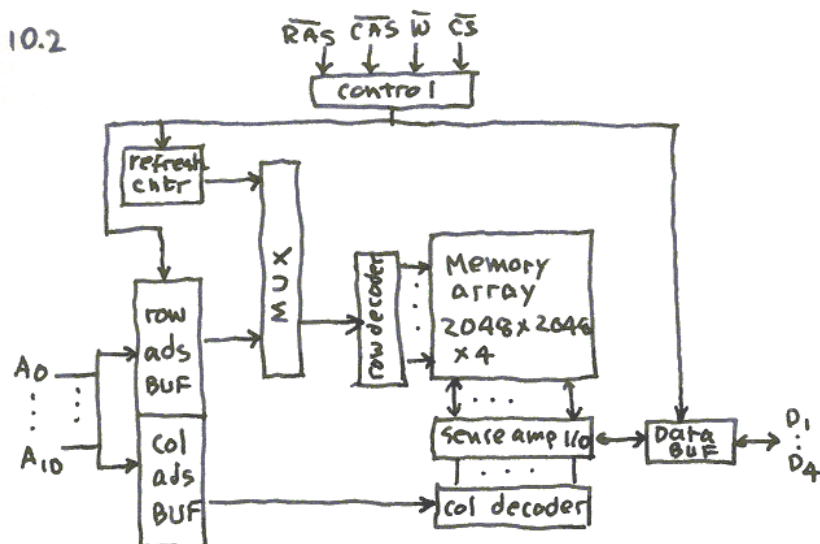


Figure 10.2 a typical 16 Mbit DRAM (4M × 4 bits)

Memory hierarchy

There are many types of memory in a computer system. The range spanned the memory in the control unit, the processor, the high-speed memory (the cache), the main memory, and finally in the secondary storage (the disk cache). We discuss each of them in turn. Fig.10.3 shows a typical memory hierarchy.

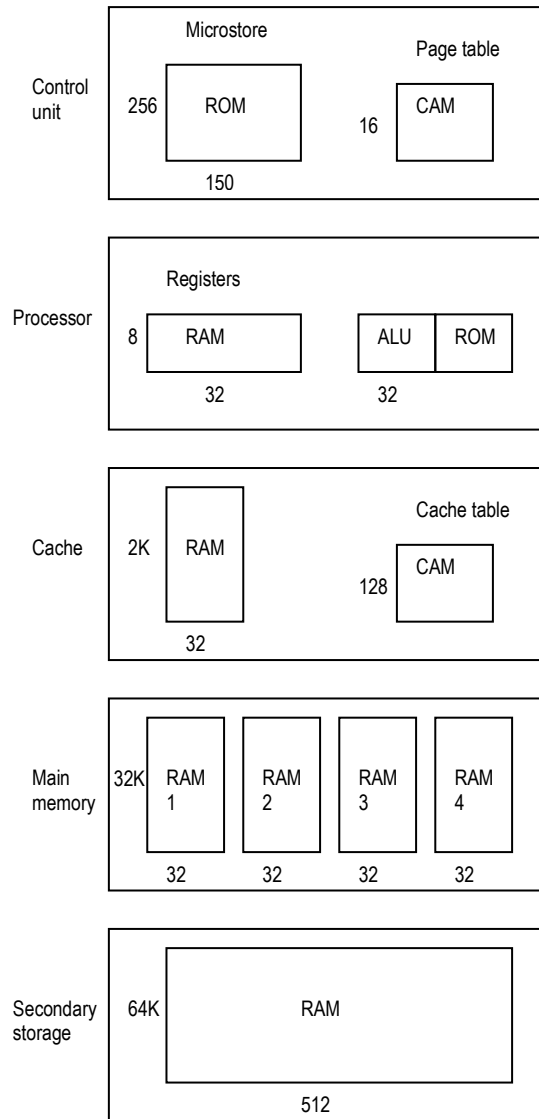


Figure 10.3 a memory hierarchy in a computer system

The control unit contains a micromemory storing the microprogram. It is a read-only-memory (ROM) (sometimes it may be a writable micromemory, in that case, it is a high-speed RAM). The page table, which is used to map virtual address (explained in the later section), contains the content-addressable-memory (CAM). A CAM is a special kind of memory. Rather than retrieving the content by the address lines, the content of a CAM is retrieved by the "association" of the pattern of data. The query such as "Is there any number 105 stored in this memory?" demonstrates the principle of the CAM. A CAM is generally more expensive than a RAM.

Within a processor there are registers. The registers have very fast access time. They are used to store temporary values during computation. The ALU has a ROM storing many constant values used in the floating-point calculations such as rounding operations.

A cache memory is a high-speed memory connecting a processor to a main memory. Because the speed of a processor is much faster than the speed of a main memory (at least by a factor of 10). The cache memory is matched to the processor speed. It is much smaller than the main memory, but using a scheme of storing the recently used data and instructions enables a cache memory to behave as the larger main memory. The cache table is a CAM. Its use will be explained in the later section.

The main memory is usually organised as one large unit. It can also be organised as a number of parallel memory units. Each unit can be addressed independently. The figure shows a four-way parallel main memory with 32-bit words. This improves the memory bandwidth.

The next level in the hierarchy is the secondary storage, usually it is the magnetic disk. The secondary storage has an access time much slower than the main memory (a disk is at least 10,000 times slower than a RAM). Using the principle of a cache memory, the disk cache, a large RAM, is an intermediate backup to the main memory. The goal is to buffer large amount of data to and from the main memory and the secondary storage. This arrangement improves the performance and reduces the system cost. The figure shows 64K of 512-bit words.

The cost of a memory varies with its speed. A multi-level memory hierarchy provides a large amount of memory that is not expensive. If the hierarchy is properly matched to the patterns of addresses generated by programs run on the system, its effective speed can match the processor speed. The address patterns

generated by programs are an important factor in the design of a memory system and will be discussed further in the later section. Figure 10.4 shows a typical parameters of memory hierarchy.

Level	1	2	3	4
Name	Registers	Cache	Main memory	Disk
Typical size	< 1 KB	< 4 MB	< 4 GB	> 1 GB
Technology	CMOS BiCMOS	On-chip or off- chip CMOS SRAM	CMOS DRAM	Magnetic disk
Access time (ns)	2–5	3–10	80–400	5,000,000
Bandwidth (MB/sec)	4000–32000	800–5000	400–2000	4–32
Managed by	Compiler	Hardware	OS	OS/User
Backed by	Cache	Main memory	Disk	Tape

Figure 10.4 a typical memory hierarchy (from [HEN96] p.41)

Interleaved memory

The constraint of a von Neumann architecture is that a single memory module of conventional design can access no more than one word during each cycle of the memory clock. There are several ways to increase memory bandwidth (bits/bytes per second).

- reduce cycle time
- increase word size
- concurrent access (banking, interleaving)

Reduction of the cycle time can be achieved by using a faster memory, which is more expensive. Increasing the word size will increase the number of connection between memory and processor. In term of the number of pin on the package, increasing the number of wires implies increasing the number of pin. The last alternative, *interleaved memory*, is achieved by arranging multiple memory arrays into parallel units, each of which can be accessed independently. All units are addressed at the same time, hence the consecutive locations can be accessed with zero delay after the first data is available.

Cache

A cache memory is a high-speed memory connecting a processor to a main memory. It is much smaller than the main memory but it is faster. Its advantage is that the average memory access time is nearly equal to the speed of the fastest memory, whereas the average unit cost of the memory system approaches the cost of the cheapest memory.

A cache memory stores the most frequently used values from the memory. This is possible because of the principle of locality of references [DEN68]. The patterns of addresses generated by programs run on a system exhibit two kinds of locality:

1. *Temporal*, the values that have been accessed will be accessed again in the near future.
2. *Spatial* (array, vector, code segment), the values near the location of the recently accessed will be accessed in the near future.

Temporal locality

Cache memory stores the most frequently used values from the memory. The access pattern has "temporal locality", the locations in the memory may be spatially separated but the cache memory stores them together when they are accessed.

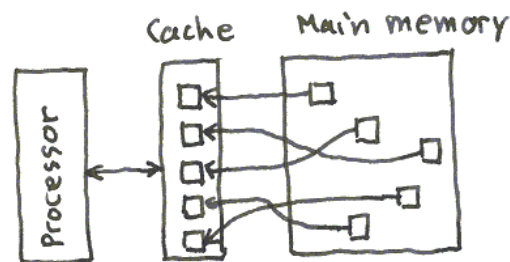


Figure 10.5 cache memory stores values that "temporally related"

Working set of an application is the amount of memory the application requires in a period of time. It varies with time.

Cache performance

When the processor attempts to fetch a word from main memory, the word is supplied from the cache if it is resident there. If not the word is supplied from main memory, but the cache is also loaded with a block of words including the one requested. Thus, subsequent accesses to the same word can be handled at the cache speed.

Let h , the cache *hit ratio*, be the fraction of the references to main memory that can be found in the cache.

$$T_e = hT_c + (1-h)T_m$$

where T_e the effective cycle time, T_c the cycle time of cache memory, T_m the cycle time of main memory. The speed up due to cache is

$$S_c = T_m / T_e$$

Let express the speed up in term of hit ratio

$$S_c = \frac{T_m}{T_e} = \frac{T_m}{hT_c + (1-h)T_m} = \frac{1}{1+h\left(\frac{T_c}{T_m}-1\right)}$$

since $T_c/T_m < 1$ we can write

$$S_c = \frac{1}{1-\left(1-\frac{T_c}{T_m}\right)h}$$

if $h = k/k+1$ then

$$S_c = \frac{1}{1+\frac{k}{k+1}\left(\frac{T_c}{T_m}-1\right)} = \frac{1}{\frac{k}{k+1}\frac{T_c}{T_m} + \frac{1}{k+1}} = \frac{(k+1)T_m}{T_m + kT_c} < k+1$$

Thus, if $h = 1/2$ we can not achieve a speed up of more than 2. The figure shows maximum S_c versus h .

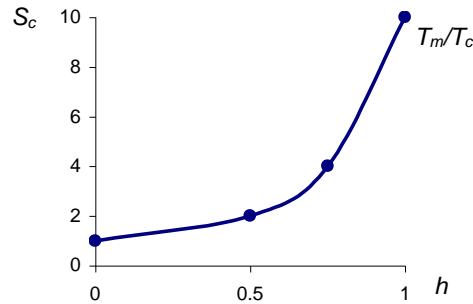


Figure 10.6 maximum possible cache speed up versus hit ratio

Example We want to compare two size of cache memory, 512 K bytes and 1 M bytes. Let the average hit ratio of 512 K bytes cache be 0.93 and the average hit ratio of 1 M bytes cache be 0.97. Let $T_c/T_m = 0.12$.

$$S_c^{512} = 1 / (1 - 0.88 \times 0.93) = 5.5$$

$$S_c^{1M} = 1 / (1 - 0.88 \times 0.97) = 6.85$$

Thus, adding 512 K bytes to the cache achieved a system speed up improvement of

$$\frac{S_c^{1M} - S_c^{512}}{S_c^{512}} = 0.24$$

This 24 percent improvement could be a good return on relatively small investment.

Example We want to assess how much a cache contribute to the performance of a system. Assume a cache miss penalty is 50 clock cycles, all instructions take 2 clock cycles (not take into account the caches miss), the miss rate is 2%. There is an average of 1.33 memory references per instruction. What is the impact on performance when taking cache behaviour into consideration?

We start with the definition of performance

Performance = how fast a processor finishes its job

execution time = number of instruction used \times cycle per instruction \times cycle time

Taking the behaviour of memory into account

$$\begin{aligned} \text{execution time} &= (\text{CPU execution cycle} + \text{memory stall cycle}) \times \text{cycle time} \\ \text{memory stall cycle} &= \text{number of instruction used} \times \text{memory reference per} \\ &\quad \text{instruction} \times \text{miss rate} \times \text{miss penalty} \end{aligned}$$

Therefore

$$\begin{aligned} \text{execution time} &= \text{number of instruction used} \times (\text{cycle per instruction} + \\ &\quad \text{memory reference per instruction} \times \text{miss rate} \times \text{miss} \\ &\quad \text{penalty}) \times \text{cycle time} \\ &= \text{n.o.i.} \times (2 + (1.33 \times 2\% \times 50)) \times \text{cycle time} \\ &= \text{n.o.i.} \times 3.33 \times \text{cycle time} \end{aligned}$$

Therefore the execution time increases with CPI from 2 (no cache miss) to 3.33 with a cache that can miss. Without a cache at all, CPI would be $2 + 50 \times 1.33 = 68.5$, a factor of 30 times longer!

Cache organisation

A cache memory stored some part of the main memory. The main memory can be viewed as "blocks". A cache stored a number of these blocks, which are indexed by part of the address bit. The size of the block varies. For any size larger than one, the lower address bits are used as "offset" to indicate the required word within a block. The relationship between the size of the block, the size of the cache, the organisation of the cache and the hit ratio are complex. The larger cache size has a higher hit ratio. The cache contains the memory for storing the address, called "tag", and the memory for storing the blocks.

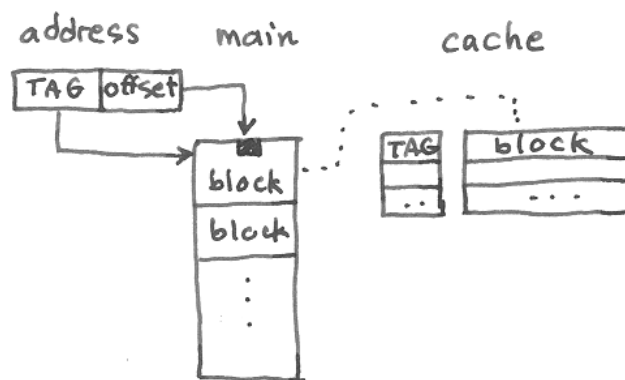


Figure 10.7 the relationship between main memory and cache

The activities in a cache consist of the request for reading and writing. In response to those requests, there are these possible events: load hit, load miss, store hit, and store miss. A cache is classified according to its organisation. We will discuss three major organisations: fully associative cache, direct map cache, and set associative cache. Mostly they differ in the way they respond to the request:

1. Where can a block be placed in the cache?
2. How is a block found if it is in the cache?
3. Which block should be replaced on a miss?
4. What happens on a write?

Fully associative

In a fully associative cache, the tag memory is made of CAM, therefore a block can be placed in any slot in the cache. The search for address matching is done with all tags in parallel using the retrieval by association. If a conventional memory (RAM) is used, it will require scanning every address. If there is no ordering among the content, scanning will take $O(n)$. If there is ordering, using binary search will take $O(n \log n)$. An associative memory takes just $O(1)$ to find the required content.

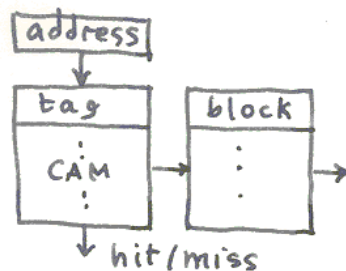


Figure 10.8 a fully associative cache

However, the CAM is expensive and the fully associative cache is used mainly in places where the speed is important and the small cache size is appropriate. To increase the cache size without increasing the tag size (the tag is made of CAM), the block size (made of RAM) can be increased.

Direct map

A direct map cache uses RAM instead of CAM to store tags. The lower bits of address are used to index the block. Therefore a block has a unique place in the cache. The addresses that have the similar higher bits will be placed in the same slot. This reduces the effectiveness of the cache, as some addresses will collide. Due to its simplicity, a direct map cache can be implemented with efficiency (it can be very fast).

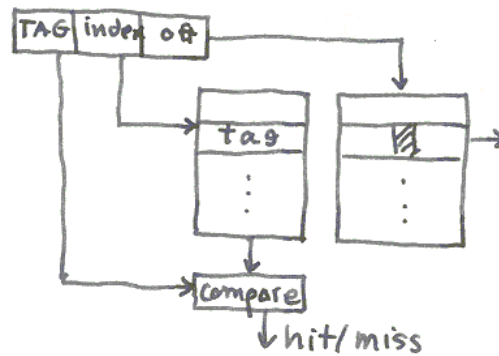


Figure 10.9 a direct map cache

Example to understand how a cache stores the tags, let us follow a simple exercise. Assume a cache has 8 slots with a block size 1. A sequence of memory requests is issued by the processor. Suppose the address sequence is 0, 1, 3, 6, 9, 13, 14, 3, 18, 19, 13. The following diagram shows the state of tags.

address			Tag _{t+3}	Tag _{t+7}	Tag _{t+10}
0	8	16	0	0	0
1	9	17	0	1	1
2	10	18			2
3	11	19	0	0*	2
4	12	20			
5	13	21		1	1*
6	14	22	0	1	1
7	15	23			

address sequence: 0,1,3,6 9,13,14,3* 18,19,13*

Figure 10.10 the state of tags after the sequence of accesses

The tag is calculated by $(\text{address} / 8)$ and the index to the slot is $(\text{address} \bmod 8)$. The address marked with * are the cache hits. There are two hits in this example.

Set associative

To improve the performance of a direct map cache, the set associative cache uses a number of direct map caches in parallel, called a set, k . The search of the matching address is done on k tags in parallel. The mechanism requires only k set of comparators and a multiplexor to select data from k sets. This improves the hit ratio as it reduces the chance of collision which different addresses map to the same slot. Having a set of caches introduces another consideration, which set to be replaced when a miss occurs? This question is settled with cache replacement policy.

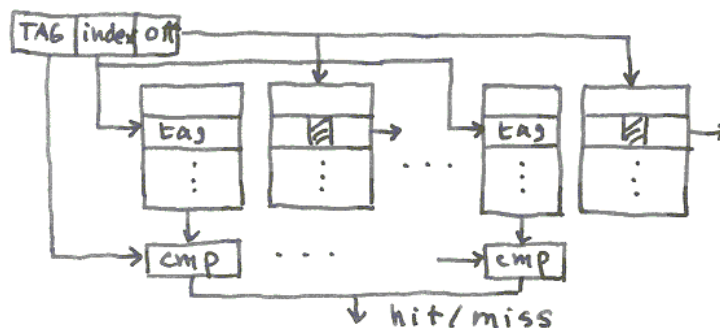


Figure 10.11 a set associative cache

Replacement policy

On a load miss, the value must be read from the main memory and also the whole block must be updated into the cache. There is no choice which block to be replaced in the direct map cache. The selection is done in hardware using the lower address bits to index the slot. For the fully associative cache and the set associative cache there are two major policies:

1. *Random* replacement, the block to be replaced is randomly selected so that it is uniformly distributed in the cache.
2. *Least-Recently-Used (LRU)*, this policy replaces the block that has been unused for the longest time.

The random replacement policy is easy to implement in hardware, requiring only a random number generator. The LRU policy must keep track of the number of access to each block hence it is more complicate. Nearly all caches in commercial production use LRU.

Write policy

Reads dominate cache accesses. All instruction accesses are reads. For data accesses, writes constitute 7% of the overall memory traffic and about 25% of data cache traffic. Reads can be made fast. A block can be read at the same time as the tag is read and compared. If it is a miss the value read can be ignored.

It is not the case for writes. Because the tag checking cannot be done in parallel with the memory access (write cannot be undone), writes normally take longer than reads. On a write request, there are two options:

1. *Write through*, the information is written to both the block in the cache and to the block in the main memory.
2. *Write back*, the information is written only the cache. The modified cache block will be written to main memory only when that block is replaced.

There is a bit associated with each block in cache called the *dirty bit*. This bit is set when the content of that block is modified. When the block is going to be replaced, the dirty bit is examined, if it is clean, it is not necessary to write this block to the main memory. This reduces the frequency of writing back to the main memory.

Address Trace

To measure the performance of a cache memory, a miss ratio is measured from an address trace. Some problems are presented:

1. the workload on the trace may not be representative
2. the initialisation transient may grossly affect the evaluation
3. the trace may be too short to obtain accurate measurement.

The length of the trace is important for accuracy. Also the concern about initialisation of cache (should or should not take into account for cache miss). If cache size is 1,000 blocks, assuming 1 byte per block, and the miss ratio is 1%, for a miss to occur once for every block requires the trace length of 100,000 addresses just to initialise the cache. An empirical formula for the trace length is:

$$\text{Trace length} = \text{cache size}^{1.5}$$

This formula implies that for each quadrupling of cache size, the trace length increases by a factor of 8. A typical simulation run to collect the address trace covers hundreds of milliseconds at most.

Improving cache performance

The performance gap between processors and memory is increasing every year. During 1980-1986 the processor performance increased 35% per year and 1987-2000 it increased 55% per year while the memory performance increased during 1980-2000 only 7% per year. In the year 2000, the processor is about 500 times faster than memory. The cache is becoming more and more important to bridge this gap. The average access time of a memory system is.

$$T_{\text{average}} = T_{\text{hit}} + \text{miss rate} \times \text{miss penalty}$$

All three factors T_{hit} , miss rate, and miss penalty are considered in improving cache performance.

To reduce miss rate the larger block size can be used. The larger block size takes advantage of spatial locality. However, increasing block size will increase the miss penalty. Another way to reduce miss rate is to increase associativity. A rule of thumb for cache is that a direct map cache of size N has about the same miss rate as a 2-way set associative cache of size $N/2$. An 8-way set associative is as effective in reducing misses as fully associative cache of the same size. Using two separate caches for instruction and data instead of a unified cache can be beneficial. A processor relies on prefetching instructions to fully use the pipeline. Having a separate instruction cache reduces the interference from data access.

The miss penalty can be reduced by using a second-level cache. In designing a cache, it is always a question of whether to make the cache larger or to make it faster. By adding another level of cache between the first cache and the main memory, the first cache can be fast while the second cache can be large. The second cache will reduce the number of traffic to the main memory. It is, in essence, reducing the miss penalty.

The hit time can be reduced in two ways. First, a simpler and smaller hardware is faster, therefore the simple organisation of cache such as a direct map cache can be faster than a complicate cache. Second, use an on-chip cache. Because the advancement of microelectronics, the number of logic on a chip has been

increasing. It has been possible to include a cache on the same die as a processor. The on-chip cache is much faster than an off-chip cache due to shorter paths and smaller delay in signal paths. However the size of an on-chip cache is limited.

For more detailed discussion of high performance cache design, there are numerous research papers on the subject, for example [TAE98] [TAE98a]. The thesis describes the design of a cache controller for a high performance processor. Many other organisations of cache have been introduced to cope with the changing of the workload that emphasis multimedia applications [CHI00].

Example The on-chip cache of Intel 486 is a 4-way set associative, the line size is 16 bytes, total cache size 8Kbytes. To simplify the hardware, it used pseudo-LRU having 3 bits: r_0 , r_1 , r_2 . The replacement policy works as follows. Let the set be s_0 , s_1 , s_2 , and s_3 . The most recently access sets the bit using these rules:

if the set is s_0 or s_1 set bit r_0
 if the set is s_0 set bit r_1
 if the set is s_2 set bit r_2

Then, the replacement follows these rules:

if $r_0 r_1 = 00$ replace s_0
 if $r_0 r_1 = 11$ replace s_1
 if $r_0 r_2 = 10$ replace s_2
 if $r_0 r_2 = 11$ replace s_3

Virtual Memory

An operating system provides a multi-task capability. It performs three types of scheduling: accepting a number of processes, switching between processes, and handling I/O requests. One important function of an operating system is *memory management*. Many programs, including the operating system itself resided in main memory. The allocation and reclamation of memory for these programs must be done dynamically by the operating system.

The process in memory contains instructions and data. When a process is swapped in and out, the addresses of these instructions and data will be changed. A process is not likely to be loaded into the same place each time it is swapped. The concept of logical address and physical address is used to solve this problem. A *logical address* is expressed as a location relative to the beginning of the program. A *physical address* is an actual location in main memory. A part of

processor hardware is designed to support the mapping between logical to physical addresses.

Paging

To facilitate memory allocation, main memory is divided into small fixed-size blocks, called *pages*. A number of pages are allocated to each process on demand. To translate a logical address to a physical address, the operating system maintains a *page table* for each process. The page table shows the location for each page of the process. A processor uses the page table to produce a physical address.

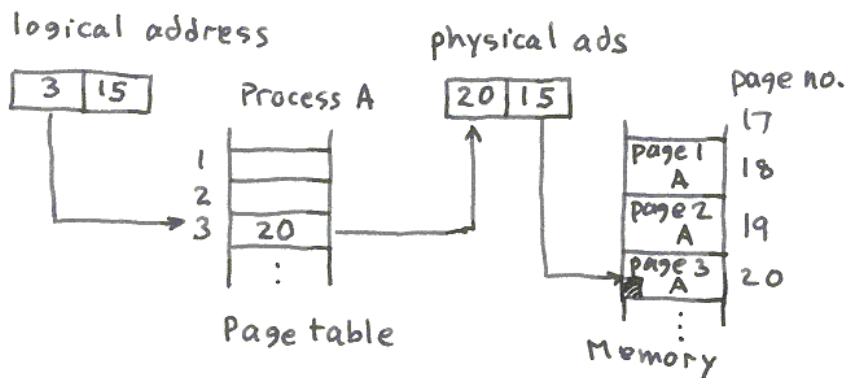


Figure 10.12 translation of logical to physical address

With *demand paging* it is possible for a process to be larger than main memory. A process executes in main memory referred to as *real memory*. A user perceives a much larger memory, the size of disk, referred to as *virtual memory*.

Parameter	First-level cache	Virtual memory
Page size	16–128 bytes	4096–65,536 bytes
Hit time	1–2 clocks	40–100 clocks
Miss penalty	8–100 clocks	700,000–6,000,000 clocks
(Access time)	(6–60 clocks)	(500,000–4,000,000 clocks)
(Transfer time)	(2–40 clocks)	(200,000–2,000,000 clocks)
Miss rate	0.5–10%	0.00001–0.001%
Data memory size	0.016–1MB	16–8192 MB

Figure 10.13 Typical ranges of parameters for caches and virtual memory (from [HEN96] p.441)

Figure 10.13 shows the difference between first-level caches and virtual memory. Comparing the two, the difference in magnitude is about $10\times$ to $100,000\times$. A cache miss is $8\text{--}50\times$ costly as a cache hit but a page fault is $6,000\text{--}20,000\times$ as costly as a page hit. A cache miss latency is in the range of hundred of clocks but a page fault latency is in range of millions of clocks. A processor can execute a fair number of instructions during this time.

Address translation

The address translation requires accessing the page table. The page table resides in main memory. If it is large, accessing a page table may cause page fault. The worst case requires two memory accesses to get the data. A special cache, a *translation lookaside buffer* (TLB), caches page table entries, hence improving the speed of address translation. The TLB functions the same way as an ordinary cache and contains page table entries that have been most recently used. The principle of locality of references also applied to TLB, if the accesses have locality, then the address translation of the accesses must also have locality.

The operation of paging and TLB is as follows [FUR87].

Request access to a page

```
CPU checks TLB
if the page table entry is not in TLB then
    access page table
    if page is not in memory then generate Page Fault
    else update TLB
CPU generates physical address
```

Page Fault

```
OS call routines to read the page from disk
CPU activates I/O hardware
If memory is full then perform Page Replacement
page table update
restart request to access a page
```

The virtual memory mechanism (TLB) must interact with the cache system. When a memory access occurs the TLB is looked up, if the matching page is present, the real (physical) address is generated. If TLB does not contain the page, the entry is accessed from the page table. Once the real address is

generated the first-level cache is consulted, if the cache is a hit, the cache supplies the word. If it is a miss, the word is retrieved from main memory.

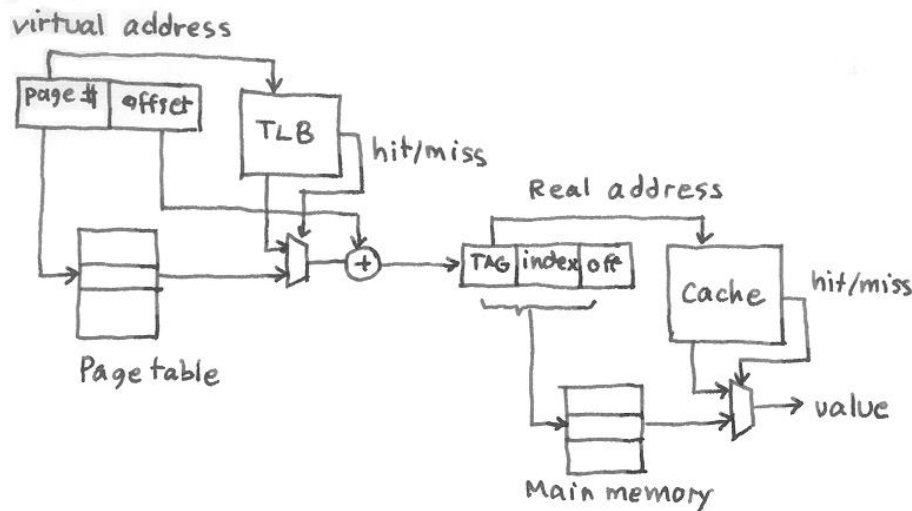


Figure 10.14 Translation lookaside buffer and cache operation

Address translation can easily be on the critical path and hence determining the clock cycle of a processor. The TLB is usually smaller and faster than a first-level cache. The TLB access is sometimes pipelined to improve its speed.

TLB size	32–8192 bytes
Block size	4–8 bytes (1 page table entry)
Hit time	1 clock
Miss penalty	10–30 clocks
Miss rate	0.1%–2%

Figure 10.15 Typical parameters for a translation lookaside buffer

Page Replacement

A page fault is different from a cache miss as the page fault is very costly and the latency is very long. During that time, a processor can perform a significant amount of processing. This processing power can be used to reduce the miss rate. Another difference is that virtual memory supports a multi-task environment. The behaviour of tasks swapping in and out of main memory are

very dynamic compared to the behaviour of a cache miss that occurs in a single thread of execution over a shorter period of time.

When a page fault occurs, a new page must be loaded from the secondary memory, usually a disk. The *trashing* behaviour describes the situation where there are excessively high traffic between main memory and the disk. Trashing causes significant impairment of system performance. The replacement policy is similar to that of a cache, the LRU. However, the question arises on how many pages to be allocated to a process? The answer is to allocate as many pages as the process needs at a given time. This requires the notion of a *working set*, the footprint of a program execution over a short period of time.

How to find working set

$W(t, w)$ the working set at time t for window w . It is the set of pages referenced in the last w seconds at time t . The following steps allocate pages according to the working set of a program:

1. when page fault, add a new page to the working set
2. from set of pages not referenced. within w , delete the LRU page otherwise let the set grow
3. if two or more pages not referenced within w , delete two LRU pages, w is measured by process time.

Page-fault frequency method

This method is based on the observation that a high page fault signals the change in phase to a new working set

1. select a threshold Z
2. when page fault, estimate frequency. $f = 1/(t_1 - t_0)$
3. $f > Z$ assume change phase, add page to the working set
4. $f < Z$ assume stable. add new page, remove the old page (LRU)
5. $f < Z$ over some period, assume stable and dead pages, reduce working set and delete unreferenced. page (LRU)

How to allocate the number of pages to a program

The optimal is to get least page-fault rate. In a multi-task environment there are many processes competing for resources. The allocation must consider all processes. Let $R_i(x_i)$ be fault rate of process i with x_i be the memory allocation. The optimality can be achieved in the following sense:

$$\frac{d}{dx} R_i(x_i) = \frac{d}{dx} R_j(x_j)$$

The fault rates for each process for their respective allocation are equal. This implies that an optimum allocation depends on *fault rate derivatives*. Figure 10.16 compare the working set method and the page-fault frequency method.

	working set	page-fault frequency
assumption	immediate future will be like the recent past	transient between two program phases signaled by higher than normal fault rate
implementation	difficult because of sliding window	observable quantities use hardware logging

Figure 10.16 comparison of the working set method and the page-fault frequency method

Memory technology

Memory system design is becoming increasingly important as a computer system performance improved. There are three major reasons for this. First, the new generation of microelectronics technology improves speed and reduces power consumption. However, the data transfer rate on a circuit board is independent of technology scaling and remains at about 1 ns.

Second, the use of parallelism in processor design demands higher data transfer rate but data parallelism cannot be exploited because the number of chip-to-chip connections (the packaging) is limited.

Third, as memory is larger, the time for address decoding is at least logarithmic increased, hence the memory becomes slower. As a result, while processor performance improves exponentially according to Moore's law, memory system performance does not.

History

Before 1960s, computer memory systems consisted of cathode-ray storage tubes, ferrite cores, and thin magnetic films [ECK97]. The first semiconductor memory was a six-transistor SRAM (static random access memory) cell, which is now

used mostly for cache and battery-backup memory. The one-transistor dynamic memory was invented in 1968 [DEN68a] [DEN84]. The DRAM became very successful, because of its low bit cost and high density. The DRAM has dominated the computer main memory market.

DRAM operation

A DRAM is characterised by its access time and its cycle time. An *access* time is the time between the request (the address is presented) and the time when the data is ready. A *cycle* time is the time between two consecutive memory operations. This is longer than the access time due to the electrical characteristics of the memory circuits.

The memory cell consisted of storage capacitor and selection transistor. Its binary state is represented by the amount of charge it holds. The storage capacitor is implemented as a MOS (Metal Oxide Semiconductor) capacitor. The memory array composed of cross-point array of memory cells. Its operational scheme is as follows.

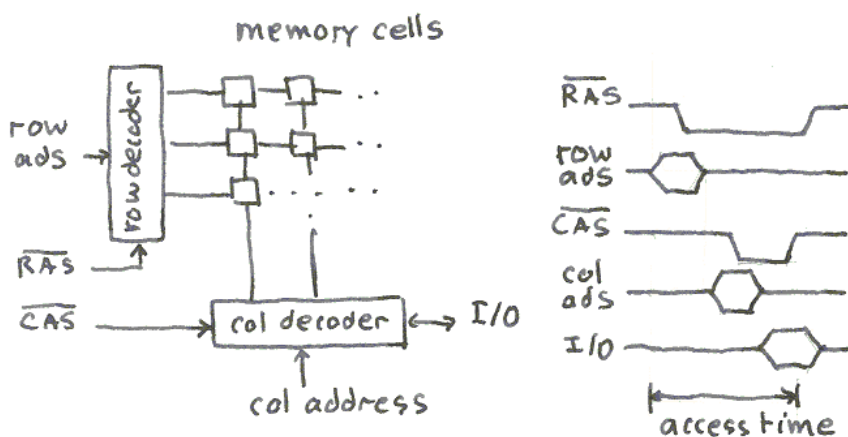


Figure 10.17 read operation of DRAM

For read operation, the RAS (row address strobe) latches the row address and decoding. The CAS (column address strobe) decodes column address and multiplexes data. The row and column address inputs are usually multiplexed to minimize the number of pins in the DRAM package. Since the read operation is

destructive, the data must be written back to the memory cells in parallel with the column access. This causes delay in RAS cycle time. The array then must be precharged for the next memory access operation.

The write operation is similar, only difference is the I/O bit is forced into a new logic state. In other words, writing to a DRAM cell is similar to writing back to the cell with a new logic state.

The limitation in performance of DRAM involves two aspects: the latency and cycle time in the row access and the data rate in the column access. The performance in row access is caused by the resistance capacitance time constant in both charging and discharging the dynamic memory cell and array. The cycle time in column access determines the data rate. Widening the width for the chip-to-chip connections increases the cost as the number of I/O drivers and packing pins increases.

High-speed DRAM development

The research and development in late 1980s results in chips access time in the low 20 ns range and a column access time in the low 10 ns range, using CMOS (Complementary Metal Oxide Semiconductor) technology [LUN89] [TAK90].

For faster data rate, EDO (extended data out) DRAMs improve the data rate for column access by adding an extra pipeline stage in the output buffer. A typical 4-M bit EDO DRAM with $\times 8$ I/Os operating at 33 MHz column access cycle can provide a peak data rate of 266 Mbytes/second per chip.

In the 16-Mbit generation, SDRAM (synchronous DRAM) employed a high-speed synchronous interface. The data rate was improved using pipelining the data path or prefetching data bits over wider data lines. The performance was also improved by interleaving multiple banks of memory arrays (normally two) on the same chip. The peak data rate for a 16-Mbit SDRAM with $\times 16$ I/O operating at 66 MHz is 1.1 Gbps (133 Mbytes/second) per chip. JEDEC (the Joint Electron Device Engineering Council) has standardized both EDO DRAM and SDRAM.

For faster random access, the performance near SRAM level is achieved by integrating a small amount of SRAM as cache or using multiple banks of DRAM. These designs are: EDRAM (enhanced DRAM), CDRAM (cached DRAM), and MDRAM (multibank DRAM).

Rambus DRAM [KUS92] uses a packet-type memory interface to realise a peak data transfer rate of 500 Mbps per data I/O pin (4 Gbps, or 500 Mbytes per chip) with a 250 MHz clock. The improved version, called Concurrent Rambus, realises a peak data rate of 600 Mbps per data I/O pin.

There are many types of memory that designed for specific applications, for example VRAM, WRAM, and 3DRAM. Video RAM (VRAM) realised concurrent dual port access by multiplexing the data in the DRAM array. The internal data rate, with 4,096 full transfer, operating at 10 MHz, is 41 Gbps (5.1 Gbytes/second). It is aimed for video application using video screen refresh data. Window RAM (WRAM) improves graphics operations in a GUI environment. It simplifies VRAM design and it has additional functions such as *bitblt* (bit-block-transfer) and block write for graphics applications. 3DRAM has been especially designed for processing 3D graphics applications. The read-modify-write operation, which occurs frequently in 3D graphics can be achieved with one write operation.

DRAM Trend

The performance of a computer system can be associated with memory system bandwidth and memory system capacity [KAT97].

$$\begin{aligned} \text{performance} &= k_1 (\text{memory system bandwidth}) \\ &= k_1 (\text{DRAM data rate}) N_{\text{dram}} / N_{\text{bank}} \end{aligned}$$

$$\begin{aligned} \text{performance} &= k_2 (\text{memory system capacity}) \\ &= k_2 (\text{DRAM density}) N_{\text{dram}} \end{aligned}$$

where N_{dram} is the number of DRAMs in the system, N_{bank} is the number of banks sharing the same data bus, and k_1 and k_2 are coefficients. By dividing the two equations.

$$\text{DRAM data rate} = N_{\text{bank}} (k_2/k_1) (\text{DRAM density})$$

This equation implies that the DRAM needs a higher data rate as its density increases. The coefficient $N_{\text{bank}}(k_2/k_1)$ is called *full frequency* [PRZ96], it depends on the application. The empirical numbers are 100 Hz for PC graphics, 10 to 20 Hz for PC main memory, and less than 10 Hz for servers and workstations. The driving forces of DRAM changes are low-end and graphics

applications, where the data rate requirement per DRAM density is higher. The transition to high-speed DRAM occurred not in the high end of the market but in the low end. The high-speed memory DRAM could provide smaller memory granularity for a given bandwidth requirement.

There are three major candidates for the next generation of high-speed DRAMs.

1. SDRAM-DDR (SDRAM double data rate), which uses a synchronous RAS and CAS similar to SDRAM. The data transfer is at both edges of the clock. A 16-Mbit SDRAM-DDR with $\times 16$ I/O operating at 100 MHz clock (200 MHz data rate) can provide 3.2 Gbps.
2. Rambus DRAM, the data rate can achieved a peak of 13 Gbps per chip due to 400 MHz clock (800 MHz data rate) and 16-bit bus width.
3. SLDRAM (Rambus IEEE standard 1596.4) [GIL97], It builds on the features of SDRAM-DDR by adding an address/control packet protocol. The internal DRAM address and control paths are decoupled from the data interface to achieve higher bandwidth.

Beyond high-speed DRAM lies the merging between logic and memory, using the processing at the memory cell. Putting RAM and processor together can achieved a very high bandwidth, such as one proposed by Patterson [PAT97].

References

- [CHI00] Chiueh, T., and Pradhan, P., "Cache memory design for Internet processors, IEEE Micro, Jan./Feb. 2000.
- [DEN68] Denning, P., "The working set model for program behavior", Communications of the ACM, May 1968.
- [DEN68a] Dennard, R., Field Effect Transistor Memory, US Patent 3387286, 1968.
- [DEN84] Dennard, R., "Evolution of the MOSFET dynamic RAM – A personal view", IEEE Electron Devices, no, 11, Nov. 1984, pp.364-369.
- [ECK97] Eckert, J., "A survey of digital computer memory systems", Proc. IEEE, vol. 85, no. 1, Jan. 1997, pp.184-197. (Originally in Proc. Inst. Radio Engineers, vol. 41, Oct. 1953, pp. 1393).
- [FUR87] Furht, B., and Milutinovic, V., "A survey of microprocessor architectures for memory management", Computer, March 1987.
- [GIL97] Gillingham, P. and Vogley, B., "SLDRAM: High-performance Open-Standard Memory", IEEE Micro, Nov./Dec. 1997, pp. 29-39.
- [HEN96] Hennessy, J., and Patterson, D., Computer architecture: A quantitative approach, 2 ed., Morgan Kaufmann, 1996.

- [KAT97] Katayama, Y., "Trends in semiconductor memories", IEEE Micro, Nov./Dec. 1997, pp.10-17.
- [KUS92] Kushiyama, N. et al., "500 Mbytes/s Data rate 512 Kbits \times 9 DRAM using Novel I/O Interface", Dig. Tech. Papers, 1992, Symp. VLSI Circuit, June 1992, pp.66-67. <http://www.rambus.com>
- [LUN89] Lu, N. et al., "A 22-ns 1-Mbit CMOS high speed DRAM with address multiplexing", IEEE J. Solid-state circuits, vol. 24, Oct. 1989, pp.1196-1205.
- [PAT97] Patterson, D. et al., "A case for intelligent RAM", IEEE Micro, Mar./Apr. 1997, pp.34-44.
- [PRZ96] Przybylski, S. et al., "SDRAMs ready to enter PC mainstream", Microprocessor report, vol. 10, no. 6, May 1996.
- [TAK90] Takai, Y. et al., "A 23-ns 1-Mbit BiCMOS DRAM", IEEE J. Solid-state circuits, vol. 25, Oct. 1990, 1102-1111.
- [TAE98] Taechashong, Primas, A VLSI design of a load/store unit for a RISC processor, Master of Engineering thesis, Department of computer engineering, Chulalongkorn University, 1998. (in Thai)
- [TAE98a] Taechashong, P. and Chongstitvatana, P., "A VLSI design of a load/store unit for a RISC microprocessor", Proc. of The Second Annual National Symposium on Computational Science and Engineering, pp. 244-248, Bangkok, March 25-27, 1998.

