DEPARTMENT
OF
ELECTRONICS & COMMUNICATION ENGINEERING

EMBEDDED SYSTEM DESIGN

(Theory Notes)

Autonomous Course

Prepared by

Prof.Tejashree S

Module – 3 Contents
Memory Hierarchy, Bus and Cache: Memory Hierarchy Technology, Virtual Memory Technology, Backplane Bus Systems, Cache Memory Organizations

Dayananda Sagar College of Engineering
Shavige Malleshwara Hills, Kumaraswamy Layout,
Banashankari, Bangalore-560078, Karnataka
Tel : +91 80 26662226 26661104 Extn : 2731 Fax : +90 80 2666 0789
Web - http://www.dayanandasagar.edu Email : hod-ece@dayanandasagar.edu
(An Autonomous Institute Affiliated to VTU, Approved by AICTE & ISO 9001:2008 Certified)
(Accredited by NBA, National Assessment & Accreditation Council (NAAC) with 'A' grade)

MODULE-3

MEMORY HIERARCHY TECHNOLOGY

Storage devices such as Registers, caches, main memory, disk devices, and tape units are often organized as a hierarchy as depicted in fig below.

The memory technology and storage organization at each level are characterized by five parameters:

(i) The access time (t_i).
(ii) Memory size (s_i).
(iii) Cost per byte (c_i).
(iv) Transfer bandwidth (b_i) and
(v) Unit of transfer (x_i).

→ THE ACCESS TIME (t_i):

It Refers to the round-trip time from the CPU to the i^{th} level memory.

→ MEMORY SIZE (s_i):

It is the number of bytes or words in level i.

→ COST PER BYTE (c_i):

The cost of the i^{th} level memory is estimated by the product $c_i s_i$.

→ TRANSFER BANDWIDTH (b_i):

The bandwidth b_i refers to the rate at which information is transferred between adjacent levels.

→ UNIT OF TRANSFER (x_i):

The Grain size for data transfer between level i & $i+1$.

Memory devices at lower levels are faster to access, smaller in size and more expensive per byte, having a higher bandwidth and using smaller unit of transfer as compared with those at a higher level.

In other words, we have $t_{i-1} < t_i$, $S_{i-1} < S_i$, $C_{i-1} \geq C_i$, $b_{i-1} > b_i$ and $x_{i-1} < x_i$ for $i = 1, 2, 3$ & 4, in hierarchy where $i = 0$ corresponds to the CPU register level.

The cache is at level 1.

Main memory at level 2.

The disk at level 3 and

Tape unit at level 4.

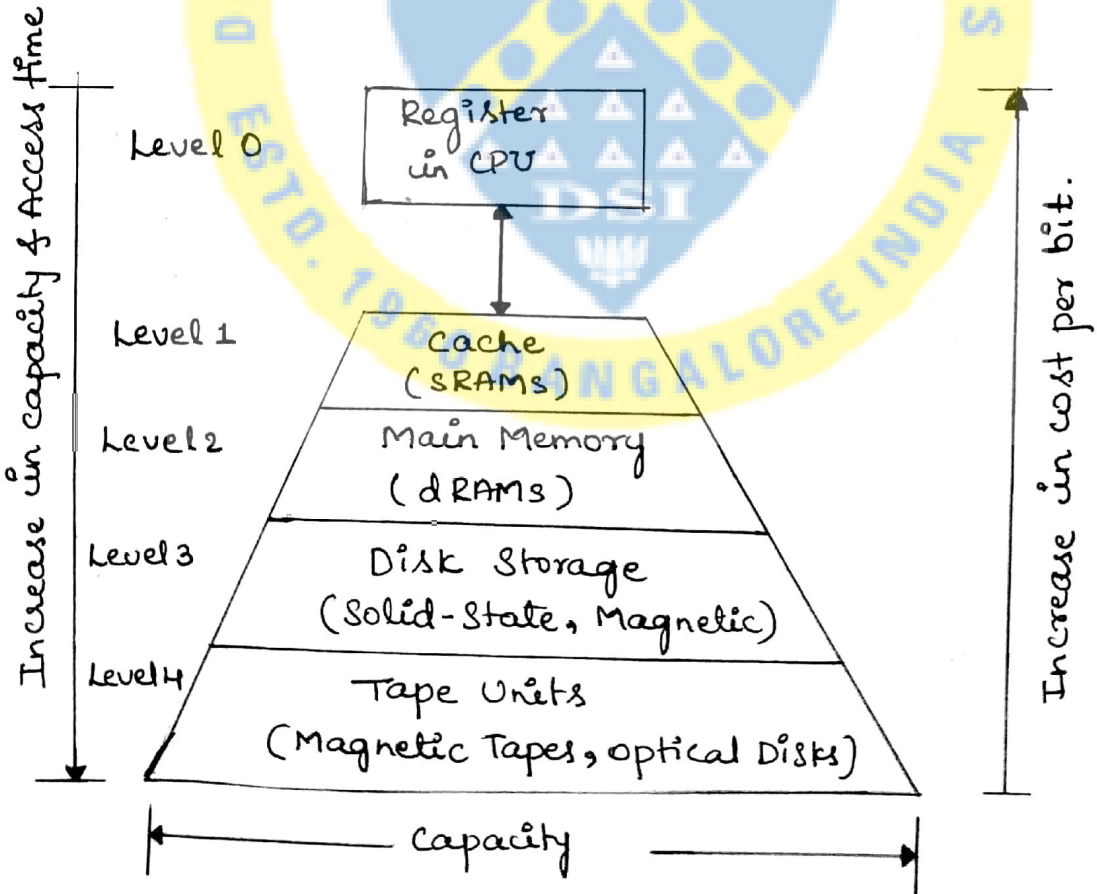


Figure : A four-level memory hierarchy with increasing capacity and decreasing speed and cost from low to high levels.

REGISTERS AND CACHES:

→ The register and the cache are parts of the processor complex built either on the processor chip or on the processor board.

→ Register assignment is often made by the compiler.

→ Register transfer operations are directly controlled by the Processor after instructions are decoded.

→ Register transfer is conducted at processor speed, usually in one clock cycle.

Therefore, many designers would not consider register a level of memory. It is listed here for comparison purposes.

→ The cache is controlled by the Memory Management Unit (MMU) and is programmer-transparent.

→ The cache can also be implemented at one or more levels depending on the speed and application requirements.

MAIN MEMORY:

The main memory is sometimes called the primary memory of a computer system. It is usually larger than the cache and often implemented by the most cost-effective RAM chips, such as 4Mbit DRAMs used in 1991 and the 64-Mbit in 1995.

→ The main memory is managed by a MMU in cooperation with operating system.

→ Memory can be extended by adding more memory boards to the system.

→ Main memory it-self is divided into sublevels using different memory technologies.

DISK DRIVES AND TAPE UNITS:

- Disk drives and tape units are handled by the OS with limited user intervention.
- The disk storage is considered the highest level of on-line memory.
- It holds the system programs such as the OS and compilers and some users programs and their data sets.
- The magnetic tape units are off-line memory for use as backup storage.
- They hold copies of present and past user programs and processed results and files.

A typical workstation computer has the cache and main memory on a processor board and hard disk in an attached disk drive. In order to access the magnetic tape units, user intervention is needed.

PERIPHERAL TECHNOLOGY:

Beside disk drives and tape units, peripheral devices include Printers, plotters, monitors, graphics display, optical scanners etc. Some I/O devices are tied to special-purpose or multimedia applications.

The technology of peripherals device has improved rapidly in recent years. For example, we used dot-matrix printers in past. Now as laser printers becomes so popular, in-house Publishing becomes a reality.

Table below presents representative values of memory parameters for a typical 32-bit mainframe computer built in 1993.

Memory Level characteristics	Level 0 CPU Registers	Level 1 Cache	Level 2 Main Memory	Level 3 Disk Storage	Level 4 Tape Storage
Device technology	ECL	256K-bit SRAM	4M-bit DRAM	1-Gbyte magnetic disk unit	5-Gbyte magnetic tape unit
Access time, t_i	10ns	25-40ns	60-100ns	12-20ms	2-20 min (Search time)
Capacity, s_i (in bytes)	512 bytes	128K bytes	512M bytes	60-228 Gbytes	512Gbytes-2Tbytes.
Cost c_i (in cents/KB)	18,000	72	5.6	0.23	0.01
Bandwidth, b_i (in MB/s)	400-800	250-400	80-133	3-5	0.18-0.23
Unit of transfer, x_i	4-8 bytes Per word	32 bytes Per block	0.5-1 Kbytes Per page	5-512 Kbytes Per file.	Back up storage.
Allocation management	compiler assignment	Hardware control	operating System	operating System/User	operating System/User.

Table: Memory characteristics of a typical Mainframe computer in 1993

INCLUSION, COHERENCE AND LOCALITY:

Information stored in a memory hierarchy ($M_1, M_2, \ldots M_n$) satisfies three important properties: inclusion, coherence and locality as illustrated in Fig.

→ We consider cache memory the innermost level M1, which directly communicates with the CPU registers.

→ The outermost level Mn contains all the information words stored

→ The collection of all addressable words in Mn forms the virtual address space of a computer.

→ Program locality is characterized below as the foundation for using a memory hierarchy effectively.

INCLUSION PROPERTY:

→ The Inclusion property is stated as $M_1 \subset M_2 \subset M_3 \ldots \subset M_n$.

→ The set inclusion relationship implies that all information items are originally stored in the outermost level M_n.

→ During the processing, subsets of M_n are copied into M_{n-1}. Similarly, subsets of M_{n-1} are copied into M_{n-2} and so on.

→ In other words, if an information word is found in M_i, then copies of the same word can be also found in all upper levels $M_{i+1}, M_{i+2}, \ldots, M_n$.

→ However, a word stored in M_{i+1} may not be found in M_i.

→ A word miss in M_i implies that it is also missing from all lower levels $M_{i-1}, M_{i-2}, \ldots M_1$.

→ The highest level is the backup storage, where everything can be found.

- Information transfer between the CPU and cache is in terms of words (4 or 8 bytes).

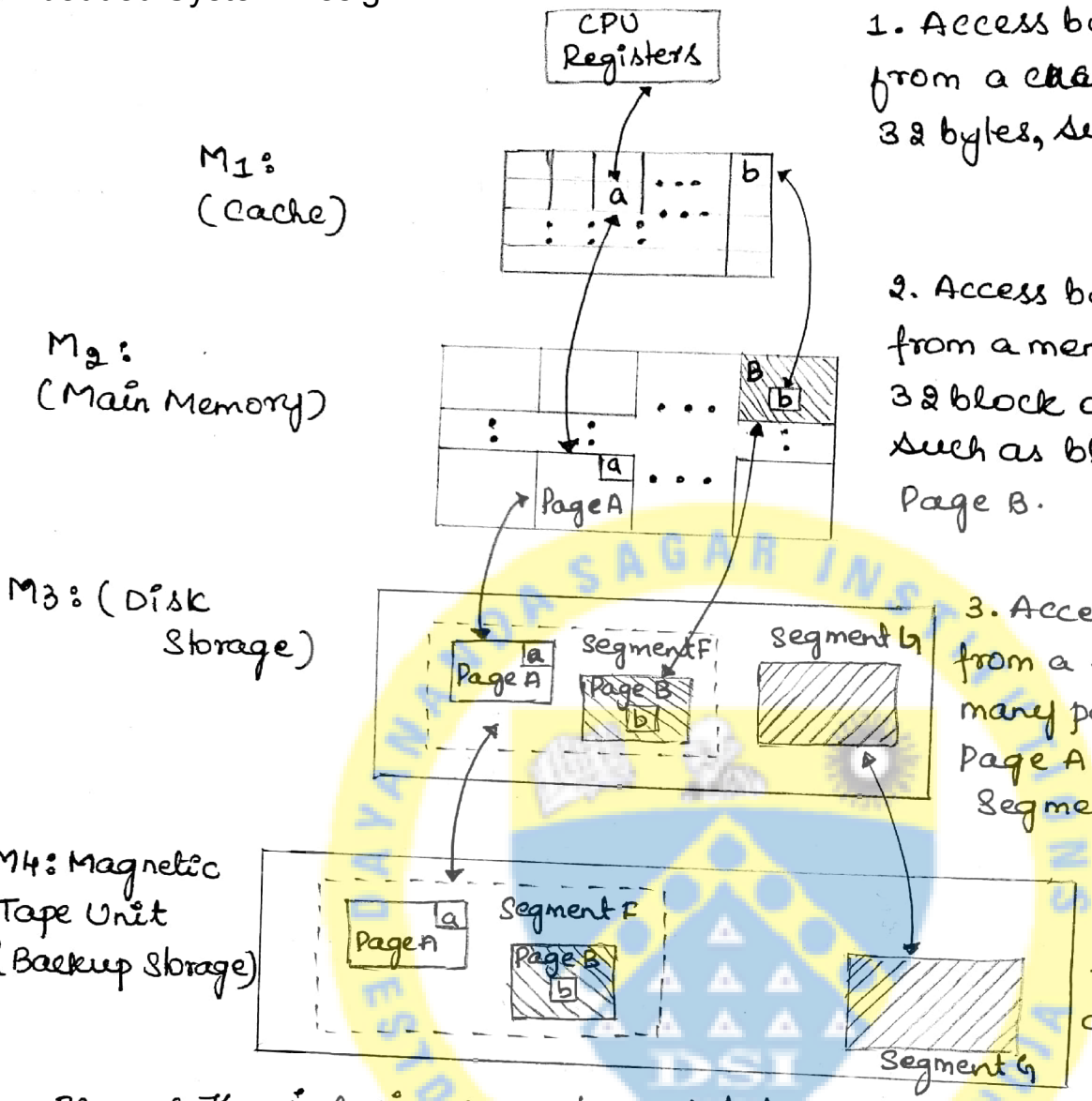
→ The cache (M_1) is divided into cache blocks, also called cache lines.

- Each block is typically 32 bytes (8 words).
- Blocks (such as 'a' & 'b' in fig) are the units of data transfer between the cache and main memory.

→ The main memory (M_2) is divided into pages, say, 4Kbytes eac

- Each page contains 128 blocks for the example in fig below.
- Pages are the units of information transferred between disk and main memory.
- Scattered pages are organized as a segment in the disk memory, for example segment F contains page A, page B & other pages.

  The size of a segment varies depending on the user's need.
- Data transfer between the disk and the tape unit is handled at the file level, such as segments F and G illustrated in fig below.



1. Access by word (4 bytes) from a cache block of 32 bytes, such as block a

2. Access by block (32 bytes) from a memory page of 32 block or 1 KBytes, such as block b from Page B.

3. Access by page (1 KByte) from a file consisting of many pages, such as Page A & Page B in Segment F.

4. Segment transfer with different number of pages.

Figure: The inclusion property and data transfer between adjacent levels of a memory hierarchy.

COHERENCE PROPERTY:

- It requires that copies of the same information item at successive memory levels be consistent.
- If a word is modified in cache, copies of that word must be updated immediately or eventually at all higher levels.
- The Hierarchy should be maintained such that frequently used information is often found in the lower levels in order to ~~maintain~~ minimize the effective access time of the memory Hierarchy.

There are two strategies for maintaining the coherence in a Memory Hierarchy.

1. WRITE-THROUGH (WT) - Demands immediate update in M_{i+1} if a word is modified in M_i, for $i = 1, 2, \ldots n-1$.

2. WRITE-BACK (WB) - It delays the update in M_{i+1} until the word being modified in M_i is replaced or removed from M_i.

LOCALITY OF REFERENCES:

The Memory Hierarchy works on the principle of Locality, which means that references to memory are localized in sense of time or temporal sense as well as in Spatial sense or in terms of Space.

Hennessy and Patterson (1990) have pointed out a 90-10 rule which states that a typical program may spend 90% of its execution time on only 10% of the code such as the innermost loop of a nested looping operation.

There are three dimensions of the locality property:

- Temporal
- Spatial and
- Sequential.

During the lifetime of a software process, a number of pages are used dynamically. These references of pages vary from time to time, however they follow certain access patterns as shown in figure below These memory references patterns are caused by the locality Properties.

1. TEMPORAL LOCALITY:

Recently referenced item (instructions or data) are likely to be referenced again in the near future This is often caused by special program constructs such as iterative loops, process stacks, temporary variables or subroutines.

Once the loop is entered or subroutine is called, a small code segment will be referenced repeatedly many times. Thus temporal locality tends to cluster the access in the recently used areas.

2. SPATIAL LOCALITY:

This refers to the tendency for a process to access items whose addresses are near one another. Example: operations on tables or arrays involve accesses of a certain clustered area in the address space. program segments such as routines and macros, tends to be stored in the same neighborhood of the memory space.

3. SEQUENTIAL LOCALITY:

In typical programs, the execution of instructions follows a sequential order unless branch instruction create out-of-order executions.

The ratio of in-order execution to out-of-order execution is roughly 5 to 1 in ordinary programs. Besides, the access of a large data array also follows a sequential order.

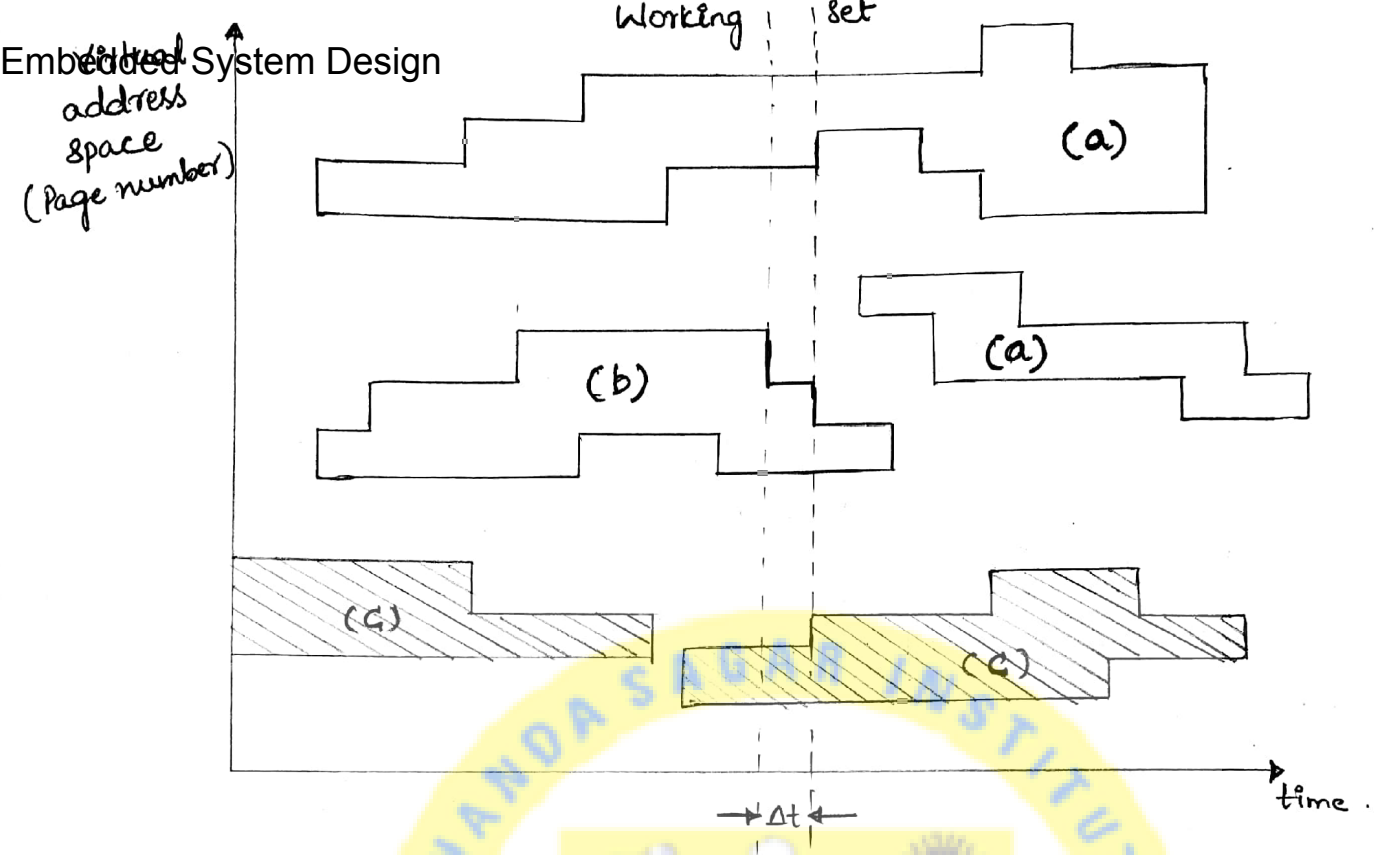


Figure A: Memory reference patterns in typical program trace experiments, where regions (a), (b) and (c) are generated with the execution of three software processes.

MEMORY DESIGN IMPLICATIONS:

→ The sequentiality in program behavior also contributes to the spatial locality because sequentially coded instructions and array elements are often stored in adjacent locations.

→ Each type of locality affects the design of memory hierarchy.

→ The temporal locality leads to the popularity of the Least recently used (LRU) replacement algorithm.

→ The spatial locality assists in determing the size of unit data transfer between adjacent memory levels.

→ Temporal locality helps determine the size of memory at successive levels.

→ The sequential locality affects the determination of grain size for optimal scheduling.

The principle of localities will guide to design cache, main memory and even virtual memory organization.

THE WORKING SETS:

Figure A shows the memory references patterns of 3 programs. As a function of time, the virtual address space is clustered into regions due to the locality of references.

The subset of addresses referenced within a given time window $(t, t+\Delta t)$ is called the working set.

During the execution of a program, the working set changes slowly and maintains a certain degree of continuity as shown in figure A.

This implies that the working set is accumulated at the innermost level such as cache memory. This will reduce the memory access time with a higher hit-ratio at the lowest memory level.

The time window Δt is a critical parameter set by the OS kernel which affects the size of working set & thus the desired cache size.

MEMORY CAPACITY PLANNING:

The performance of a memory hierarchy is determined by the effective access time Teff to any level in hierarchy.
It depends on the hit ratio and access frequencies at successive levels.

HIT RATIOS: Hit ratio is difined for any two adjacent levels of memory hierarchy.
When an information item is found in M_i, we call a hit, else a miss.
Consider memory levels M_i & M_{i-1} in hierarchy. The hit ratio h_i at M_i is the probability that an information will be found in M_i
The miss ratio at M_i is defined as $1-h_i$.

The hit ratios at successive levels are a function of memory capacities, management policies & program behavior.

Successive hit ratios are independent random variables with values between 0 and 1.

Assume $h_0 = 0$ and $h_n = 1$, which means the CPU always accesses M_1 first and the access to the outermost memory M_n is always a hit.

The access frequency to M_i is defined as

$$f_i = (1-h_1)(1-h_2)\cdots(1-h_{i-1})h_i.$$

This is indeed the probability of successfully accessing M_i when there are $i-1$ misses at the lower levels and a hit at M_i.

$$\sum_{i=1}^{n} f_i = 1 \quad \& \quad f_1 = h_1$$

Due to the locality property, the access frequencies decrease very rapidly from low to high levels; that is

$$f_1 >> f_2 >> f_3 >> \cdots >> f_n$$

This implies that the inner levels of memory are accessed more often than the outer levels.

EFFECTIVE ACCESS TIME:

We wish to achieve a hit ratio at M_1. Every time a miss occurs, a penalty must be paid to access the next higher level of memory. The misses have been called block misses in the cache & page faults in the main memory because blocks & pages are the units of transfer between these levels.

The time penalty for a page fault is much longer than that for a block miss due to the fact that $t_1 < t_2 < t_3$.

A cache miss is 2 to 4 times as costly as a cache hit but a Page fault is 1000 to 10000 times as costly as a page hit.

Using the access frequencies f_i for $i = 1, 2, \ldots n$, we can formally define the effective access time.

$$T_{eff} = \sum_{i=1}^{n} f_i \cdot t_i$$

$$= h_1 t_1 + (1-h_1) h_2 t_2 + (1-h_1)(1-h_2) h_3 t_3 + \cdots +$$

$$(1-h_1)(1-h_2) \cdots (1-h_{n-1}) t_n.$$

HIERARCHY OPTIMIZATION:

The total cost of a memory hierarchy is estimated as

$$C_{total} = \sum_{i=1}^{n} c_i s_i.$$

This implies that the cost is distributed over n levels. Since $c_1 > c_2 > c_3 > \ldots c_n$, we have to choose $s_1 < s_2 < s_3 < \cdots s_n$.

Optimal design of a memory hierarchy should result in T_{eff} close to t_1 of M_1 and a total cost close to the c_n of M_n. But difficult to achieve due to the tradeoffs among n levels.

$$T_{eff} = \sum_{i=1}^{n} f_i \cdot t_i$$

$$s_i > 0, \; t_i > 0 \quad \text{for } i = 1, 2, \ldots n$$

$$C_{total} = \sum_{i=1}^{n} c_i s_i < C_0$$

The unit cost c_i, capacity s_i at each level M_i depend on speed t_i

∴ Above optimization involves tradeoffs among t_i, c_i, s_i & f_i or h_i at all levels $i = 1, 2, \ldots n$.

The following example shows such tradeoff design.

Example:

Consider the design of a 3-level memory hierarchy with following specifications for memory characteristics.

Memory level	Access time	capacity	cost/kbyte
cache	$t_1 = 25ns$	$S_1 = 512\,KB$	$C_1 = \$1.25$
Main memory	t_2 = unknown	$S_2 = 32\,MB$	$C_2 = \$0.2$
Disk array	$t_3 = 4ms$	S_3 = unknown	$C_3 = \$0.0002$.

The design goal is to achieve an effective memory-access time $t = 10.04\,\mu s$ with a cache hit ratio $h_1 = 0.98$ and a hit ratio $h_2 = 0.9$ in the main memory. The memory hierarchy cost is calculated as

$$C = C_1 S_1 + C_2 S_2 + C_3 S_3 < 15,000$$

Thus maximum capacity of disk is obtained as $S_3 = 39.8$ Gbytes without exceeding the budget.

Now choose the access time (t_2) of the RAM to build the main memory. The effective memory-access time is calculated as

$$t = h_1 t_1 + (1-h_1) h_2 t_2 + (1-h_1)(1-h_2) h_3 t_3 \leq 10.04.$$

Substituting all parameters, we have

$$t_2 = 903ns.$$

If the main memory is doubled to 64Mbytes at the expense of reducing the disk capacity under same budget. This doesn't change cache hit ratio. but it may increase the hit ratio in the main memory if a proper page replacement algorithm is used.

Also, the effective memory-access time will be enhanced.

VIRTUAL MEMORY TECHNOLOGY:

In this we study two models of Virtual memory.

1. Translation mechanisms.
2. Page replacement policies for memory management.

VIRTUAL MEMORY-MODELS:

The main memory is physical memory in which multiple running programs may reside. But limited size of memory cannot load all programs simultaneously.

→ The Virtual memory concept was introduced to alleviate this Problem.

→ The idea is to expand the use of the physical memory among many programs with the help of auxiliary memory such as disk arrays.

→ only active programs become residents of the physical memory at a time.

→ Inactive programs are stored on disk.

→ All the programs can be loaded in & out of the memory with the coordination of operating system.

→ Without Virtual memory, it would have been impossible to develop the multiprogrammed or time-sharing computer system.

ADDRESS SPACES:

→ Each word in the physical memory is identified by a unique Physical address.

→ All memory words in the main memory forms a physical addressspace

→ Virtual addresses are generated by the processor during compile time.

→ The virtual addresses must be translated into physical addresses at run time.

translation tables and mapping functions are used

The address translation & memory management are affected by the Virtual memory model

→ The use of Virtual memory facilitates sharing of the main memory by many users on a dynamic basis.

→ Software probability and allows user to execute programs requiring much more memory than the physical memory.

→ only currently executable programs are brought into main memory. This permits the relocation of code and data makes it possible to implement protection in OS kernel, and allows high-level optimization of memory allocation and management.

ADDRESS MAPPING:

- Let V be the set of virtual address generated by a program running on a processor.
- Let M be the set of physical address allocated to run this program.
- A virtual memory system demands an automatic mechanism to implement the following mapping.

$$f_t : V \longrightarrow M \cup \{\emptyset\}$$

This mapping is a time function which varies from time to time because the physical memory is dynamically allocated and deallocated.

Consider any virtual address $v \in V$. The mapping f_t is formally defined as

$$f_t(v) = \begin{cases} m, & \text{if } m \in M \text{ has been allocated to store the data identified by virtual address } v \\ \emptyset & \text{if data } v \text{ is missing in } M. \end{cases}$$

- In other words, the mapping $f_t(v)$ uniquely translates the virtual address v into a physical address m if there is a memory hit in M.
- When there is a memory miss, the value returned, $f_t(v) = \emptyset$, signals that the referenced item has not brought into the main memory yet.
- The efficiency of the address translation process affects the performance of the virtual memory.

Two virtual memodel are.

1. PRIVATE VIRTUAL MEMORY:

→ The first model uses a private virtual memory space associated with each processor. (fig B)

→ Each private virtual space is divided into pages.

→ Virtual pages from different virtual spaces are mapped into same physical memory shared by all processors.

→ Advantage of using private virtual memory include the use of a small processor address space, protection on each page and use of private memory map, which require no locking.

→ Disadvantage lies in the synonym problem, in which different virtual addresses in different virtual spaces point to the same physical page.

→ The same virtual address in different virtual spaces may point to different pages in the main memory.

2. SHARED VIRTUAL MEMORY:

→ This model combines all the virtual address space into a single globally shared virtual space. (figBbelow)

→ Each processor is given a portion of the shared virtual memory to declare their addresses.

→ Different processor is given a may use disjoint spaces. Some areas of virtual space can be also shared by multiple processors.

→ Advantage in using shared virtual memory include the fact that all addresses are unique.

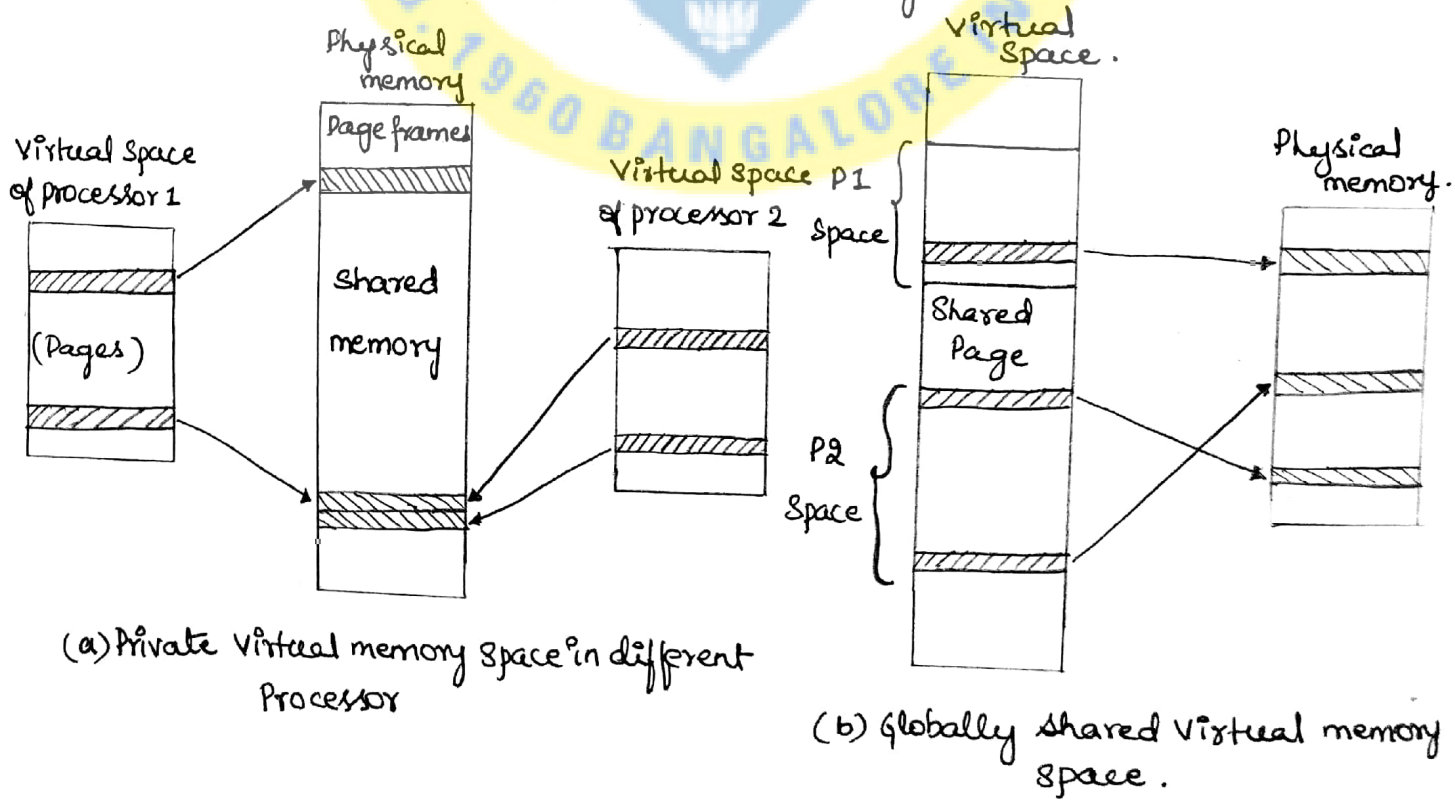
→ Each processor must be allowed to generate addresses larger than 32 bits, such as 46 bits for a 64Tbyte address space.

→ Synonyms are not allowed in a globally shared virtual memory.

→ The page table must allow shared accesses. mutual exclusion is needed to enforce protected access.

→ Segmentation is built on top of the paging system to confine each process to its own address space.

→ Address translation process is longer.



(a) Private Virtual memory Space in different Processor

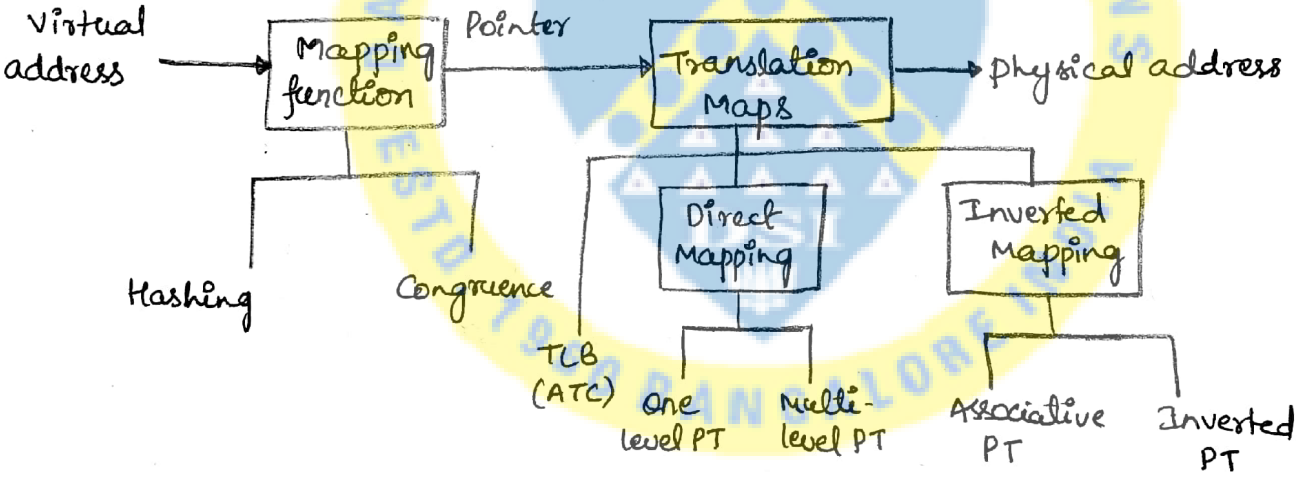
(b) Globally shared Virtual memory Space.

Figure 8 : Two Virtual memory models for multiprocessor system.

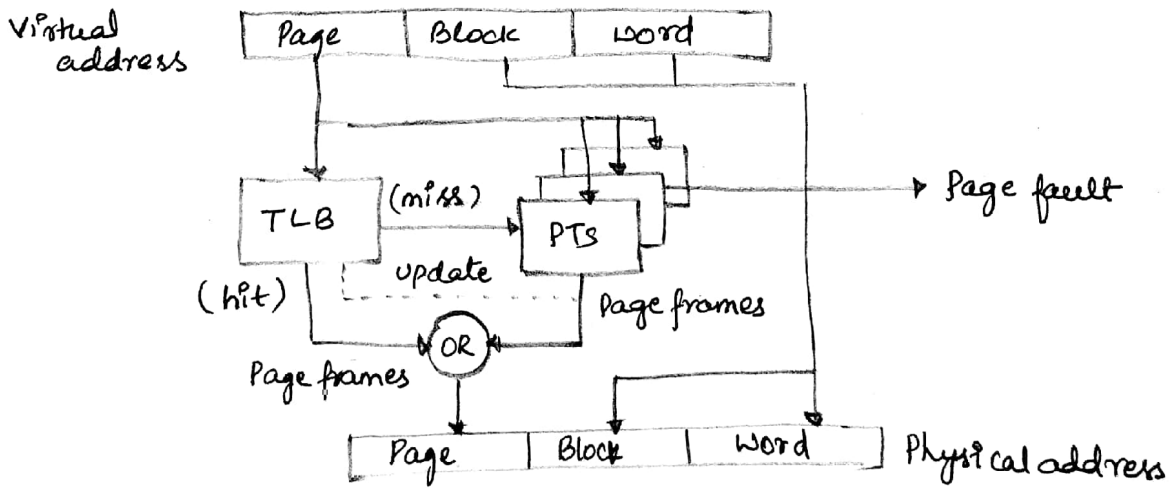
Both the virtual memory & physical memory are partitioned into fixed-length pages as shown in figure A.
The purpose of memory allocation is to allocate pages of virtual memory to the page frames of the physical memory.

ADDRESS TRANSLATION MECHANISM:

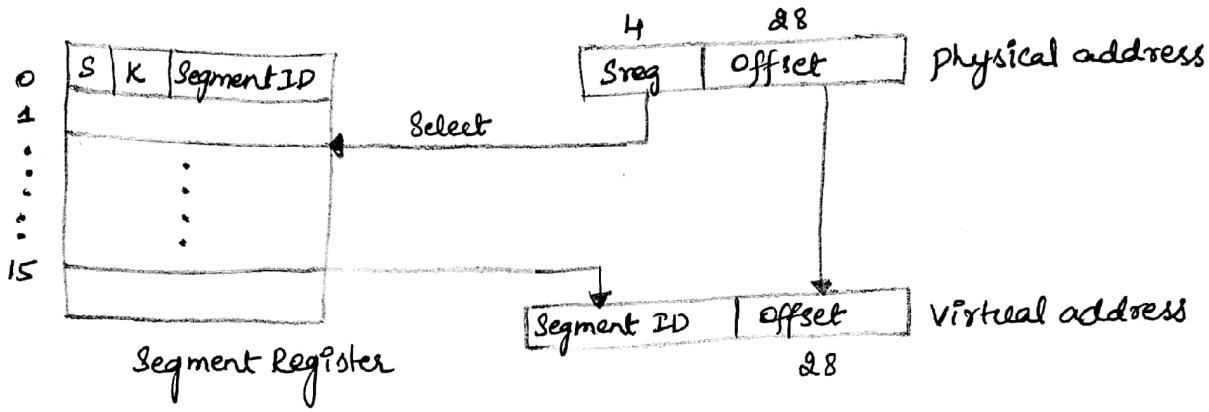
The process demands the translation of virtual address into Physical address.
The translation demands the use of translation maps which can be implemented in various ways.
Various schemes for virtual address translation are summerized below in figure C(a).



(a) Virtual address translation schemes (PT = Page table)



(b) Use of a TLB and PTs for address translation



(c) Inverted address mapping.

Figure C : Address translation mechanism using a TLB and various forms of page tables.

→ Translation maps are stored in the cache, in associative memory or in main memory.

→ To access these maps, a mapping function is applied to the virtual address.

→ This function generates a pointer to the desired translation map.

→ This mapping can be implemented with a hashing or congruence function.

→ Hashing is a simple computer technique for converting a long page number into a short one with fewer bits.

→ Hashing function should randomise the virtual page number & Produce a unique hashed number to be used as the pointer.

→ The congruence function provides hasing into linked list.

TRANSLATION LOOKASIDE BUFFER:

→ Translation maps appear in the form of a translation lookaside buffer (TLB) and page tables (PTs).

→ Based on the principle of locality in memory references, a particular working set of pages is referenced within a given context/time window.

→ The TLB is a high-speed lookup table which stores the most recently or likely referenced pages entries

→ Each page table entry consists of essentially a pair (virtual page no/page frame no)
It is hoped that pages belonging to the same working set will be directly translated using the TLB entries.

→ The use of TLB & PTs for address translation is shown in fig 6(b)

Virtual address is divided into three fields:

→ The leftmost field holds the virtual page number, the middle field identifies the cache block number and the rightmost field is the word address within the block.

→ The first step of the translation is to use the virtual page no as a key to search through the TLB for a match.

→ The TLB can be implemented with a special associative memory or use part of the cache memory.

→ In case of a match (a hit) in TLB, the page frame number is retrieved from the matched page entry. The cache block & word address are copied directly.

→ If the match cannot be found (a miss) in TLB, a hashed pointer is used to identify one of the page tables where the desired Page frame number can be retrieved.

PAGED MEMORY:

→ Paging is a technique for partitioning both the Physical & virtual memory into fixed-size pages.

→ Exchange of information between them is conducted at page level.

→ Page tables are used to map between pages and page frames.

→ These tables are implemented in the main memory.

→ Many user processes are created dynamically, so the number of PTs maintained in main memory can be very large.

→ Page table entries (PTEs) are similar to TLB entries.

→ If the demanded page cannot be found in PT, a page fault is declared.

→ A page fault implies that the referenced page is not resident in the main memory.

→ When a page fault occurs, the running process is suspended.

→ A context switch is made to another ready-to-run process while the missing page is transferred from the disk or tape unit to physical memory.

→ This direct page mapping can be extended with multiple level of page tables.

→ Multiple memory :-

→ Multilevel paging takes a longer time to generate the desired Physical address because multiple memory references are needed to access a sequence of page tables.

SEGMENTED MEMORY :

→ A Large number of pages can be shared by segmenting the virtual address space among multiple user programs.

→ A segment of scattered pages is formed logically in virtual memory space.

→ In a segmented memory system, user programs can be logically structured as segments.

→ Segments can invoke each other. Unlike pages, segments can have variable lengths.

→ Segments provide logical structures of programs and data in virtual address space. paging facilitates the management of Physical memory.

In a paged system, all page addresses form a linear address space within the virtual space.

→ The segmented memory is arranged as a two-dimensional address space.

→ Each virtual address in this space has a prefix field called the segment number and a postfix field called the offset within the segment.

→ Offset address within each segment form one-dimension of the contiguous addresses.

→ The segment numbers, not necessarily contiguous to each other form the second dimension of the address space.

PAGE SEGMENTS:

→ Concepts of paging & segmentation can be combined to implement a type of virtual memory with paged segments.

→ In each segment, the addresses are divided into fixed-size pages.

→ Each virtual address is thus divided into three fields.

- Upper field is the segment number
- middle one is the page number.
- Lower one is the offset within each page.

→ paged segment offer the advantage of both paged memory and segmented memory.

For users, programs files can be logically structured.
For OS, the virtual memory can be systematically managed with fixed-size pages within each segment.

→ Tradeoff do exist among the size of segment field, page field & the offset field. This sets limits on the number of pag. segments that can be declared by user, the segment size and page size.

INVERTED PAGING :

→ The direct paging works well with a small virtual address space such as 32 bits.

→ The virtual address space is very large in modern computers.

→ A large virtual address space demands either large PTs or multilevel direct paging which will slow down the address translation process and thus lower the performance.

→ Direct mapping, address translation maps can also be implemented with inverted mapping (figure c (c)).

→ The inverted page table is created for each page frame that has been allocated to users.

→ Any virtual page number can be paired with a given physical page number.

→ Inverted page table can be accessed either by an associative search or by the use of a hashing hashing function.

→ In inverted PTE, only virtual pages are that are currently resident in physical memory are included. This provides a significant reduction in the size of the page tables.

→ The generation of a long virtual address from a short physical address is done with the help of segment registers as shown in fig c.(c)

→ The leading 4 bits (denoted Sreg) of a 32 bit address name a segment register.

→ The register provides a segment id that replaces the 4 bit Sreg to form a long virtual address.

→ Either associative page tables or inverted page tables can be used to implement inverted mapping.

→ The inverted page table can also be assisted with the use of a TLB. An inverted PT avoids the use of large page table or sequence of page table.

Given a virtual address to be translated, the hardware searches the inverted PT for that address and if it is found, uses the table index of the matching entry as the address of the desired page frame.

→ A hashing table is used to search through the inverted PT.

→ The size of an inverted PT is governed by the size of the Physical space while that a traditional PTs is determined by the size of the virtual space.

→ Because of limited physical space, no multiple levels are needed for the inverted page table.

MEMORY REPLACEMENT POLICIES:

→ Memory management policies include the allocation & deallocat-ion of memory pages to active processes and the replacement of memory pages.

→ Page replacement refers to the process in which a resident page in a main memory is replaced by a new page transferred from the disk.

→ Since the number of available page frames is much smaller than the number of pages, the frames will eventually be fully occupied. In order to accomodate a new page, one of the resident pages must be replaced.

Different policies have beensuggested for page replacement, which are specified below.

→ The goal of a page replacement policy is to minimize the number of possible pages faults so that the effective memory-access time can be reduced.

→ The effectiveness depends on program behaviours & memory traffic patterns. A good policy should match the program locality property, & it is affected by page size & the no of available frames.

PAGE TRACES: To analyze the performance of a paging memory system, page traces experiments are performed.

→ A page trace is a sequence of page frame numbers (PFNs) generated during the execution of a given program.

→ Each PFN corresponds to the prefix portion of a physical memory address.

→ By tracing the successive PFNs in a page trace against the resident page numbers in the page frames, one can determine the occurrence of page hits or of page faults.

→ When all the pages frames are taken, a certain replacement policy must be applied to swap the pages.

→ A page trace experiment can be performed to determine the hit ratio of the paging memory system.

→ A similar idea can also be applied to perform block traces on cache behaviour.

Consider a page trace $P(n) = r(1)r(2)\ldots r(n)$ consisting of n PFNs requested in discrete time 1 to n, where $r(t)$ is the PFN requested at time t.

We define two reference distance between the repeated occurrences of the same page in $P(n)$.

The forward distance $f_t(x)$ for page x is the number of time slots from time t to the first repeated reference of page x in future

$$f_t(x) = \begin{cases} k, & \text{if } k \text{ is the smallest integer such that } r(t+k) = r(t) = x \text{ in } P(n) \\ \infty, & \text{if } x \text{ does not reappear in } P(n) \text{ beyond time } t \end{cases}$$

Similarly we define a backward distance $b_t(x)$ as the number of time slots from time t to the most recent reference of page x in the past:

$$b_t(x) = \begin{cases} k, & \text{if } k \text{ is the smallest integer such that } r(t-k) = r(t) = x \text{ in } P(n) \\ \infty, & \text{if } x \text{ never appeared in } P(n) \text{ in the past.} \end{cases}$$

Let R(t) be the resident set of all pages residing in main memory at time t. Let q(t) be the page to be replaced from R(t) when a Page fault occurs at time t.

PAGE REPLACEMENT POLICIES:

The following page replacement policies are specified in a demand Paging memory system for a page fault at time t.

1. LEAST RECENTLY USED (LRU):

This policy replaces the pages in R(t) which has the longest backward distance.

$$q(t) = y, \quad \text{if } b_t(y) = \max_{x \in R(t)} \{b_t(x)\}$$

2. OPTIMAL (OPT) ALGORITHM:

This policy replaces the pages in R(t) with longest forward distance.

$$q(t) = y, \quad \text{iff } f_t(y) = \max_{x \in R(t)} \{f_t(x)\}$$

3. FIRST-IN-FIRST-OUT (FIFO):

This policy replaces the page in R(t) which has been in memory for the longest time.

4. LEAST FREQUENTLY USED (LFU):

This policy replaces the page in R(t) which has been least refrenced in the past.

5. CIRCULAR FIFO :

This policy joins all the page frame entries into a circular FIFO queue using a pointer to indicate the front to the queue. An allocation bit is associated with each page frame. This bit is set upon initial allocation of a page to the frame.

When page fault occurs, the queue is circularly scanned from Pointer position. The pointer skips & resets the allocated page frame and replaces the very first unallocated page frame. When all frames allocated, the front of the queue is replaced, as in the FIFO policy.

6. RANDOM REPLACEMENT:

This is a trivial algorithm which chooses any page for replacement randomly.

EXAMPLE : PAGE TRACING EXPERIMENTS AND INTERPRETATION OF RESULTS.

Consider a paged virtual memory system with a two-level hierarchy: main memory M1 and disk memory M2. For clarity of illustration, assume a page size of four words. The number of page frames in M_1 is 3, labeled a, b and c, and the number of pages in M_2 is 10, identified by 0, 1, 2, ...9. The i^{th} page in M_2 consist of word address $4i$ to $4i+3$ for all $i = 0, 1, 2, ...9$.

A certain program generates the following sequence of word addresses which are grouped together if they belong to the same page. The sequence of page numbers so formed is the page trace:

Word trace : 0,1,2,3, 4,5,6,7, 8,16,17, 9,10,11, 12,28,29,30, 8,9,10, 4,5, 12,4,5

Page trace : 0 1 2 4 2 3 7 2 1 3 1

Page tracing experiments are described below for 3 page replacements Policies : LRU, OPT and FIFO, respectively. The successive pages loaded in the page frames (PFs) from the traces entries. Initially all PFs are empty

	PF	0	1	2	4	2	3	7	2	1	3	1	Hit ratio
LRU	a	0	0	0	4	4	4	7	7	7	3	3	3/11
	b		1	1	1	1	3	3	3	1	1	1	
	c			2	2	2	2	2	2	2	2	2	
	Faults	*	*	*	*		*	*		*	*		
OPT	a	0	0	0	4	4	3	7	7	7	3	3	4/11
	b		1	1	1	1	1	1	1	1	1	1	
	c			2	2	2	2	2	2	2	2	2	
	Faults	*	*	*	*		*	*			*		
FIFO	a	0	0	0	4	4	4	4	2	2	2	2	2/11
	b		1	1	1	1	3	3	3	1	1	1	
	c			2	2	2	2	7	7	7	3	3	
	Faults	*	*	*	*		*	*	*	*	*		

The above results indicate the superiority of the OPT policy over the others.

However, the OPT cannot be implemented in practice. The LRU policy performs better than the FIFO due to the locality of references. From these results, we realise that the LRU is generally better than the FIFO. However, exceptions still exist due to the dependence on program behavior.

RELATIVE PERFORMANCE:

The performance of a page replacement algorithm depends on the Page trace encountered.

→ The best policy is the OPT algorithm. However, the OPT replacement is not realizable because no one can predict the future page demand in a program.

→ The LRU algorithm is a popular policy and often results in a high hit ratio. The FIFO and random policies may perform badly because of violation of the program locality.

→ The circular FIFO policy attempts to approximate the LRU with a simple circular queue implementation.

→ The LFU policy may perform between the LRU & the FIFO policies.

→ There is no fixed superiority of any policy over the other because of the dependence on program behavior and run-time status of the pages frames.

→ In general, the page fault rate is a monotonic decreasing function of the size of the resident set R(t) at time t because more resident pages result in a higher hit ratio in the main memory.

BLOCK REPLACEMENT POLICIES:

The relationship between the cache block frames and cache blocks is similar to that between page frames and page on a disk.

∴ Those page replacement policies can be modified for block replacement when cache miss occurs.

The cache memory is often associatively searched, while the main memory is randomly addressed.

→ Due to the difference between page allocation in main memory and block allocation in the cache, the cache hit ratio and memory page hit ratio are affected by the replacement Policies differently.

→ Cache traces are often needed to evaluate the cache Performance.

Synchronization signals are used to coordinate exchange of the address, command, capability status, and data during a bus transaction. The *address handshake lines* and the *data handshake lines* are used by both master and slaves. The *bus tenure line* is used to coordinate transfer of bus control.

The *arbitration bus lines* carry a number which signifies the precedence of competitors during the bus arbitration process. *Synchronization* and *condition lines* are used to coordinate handshaking and special conditions.

The *central arbitration lines* are used by a central arbiter in case central bus control is desired. The *miscellaneous lines* are needed to indicate geographical addresses (slot addresses) and to initialize the buses during system reset or after live insertion of a card. The total number of bus lines ranges from 91, 127, 199, and 343 for 32-, 64-, 128-, and 256-bit Futurebus+ widths, respectively. Additional lines may be needed to provide utility, clock, and power connections.

■

Technology/Architecture Independence The Futurebus+ standard achieves *technology independence* through basing the protocols on fundamental protocol and physics principles and optimizing them for maximum communication efficiency rather than a particular generation or type of processor. Timing and handshake protocols are governed by operational constraints rather than limitations of technology such as device delays and capture windows.

The notion of *broadcast* facilitates the ability of slow, fast, old, or new modules to be attached to the same Futurebus+ segment. The standard specification may be implemented with any logic family (TTL, BTL, CMOS, ECL, GaAs), provided that physical implementation meets the Futurebus+ signaling requirements.

Architecture independence provides a flexible general-purpose solution to cache consistency within which other cache protocols will operate compatibly while at the same time providing an elegant unification with the message-passing protocols used in a multicomputer environment. Large systems can be built from smaller subsystems in a hierarchical manner using the Futurebus+.

Examples of this approach include the choice of split or interlocked transactions and a fully recursive, hierarchical cacheing paradigm. The Futurebus+ has been designed with possible VLSI implementation of the bus interface in mind. The architecture independence increases the application flexibility of a multiprocessor system built around the new bus standard.

5.2 Cache Memory Organizations

This section deals with physical address caches, virtual address caches, cache implementation using direct, fully associative, set-associative, and sector mapping. Finally, cache performance issues are analyzed based on some reported trace results. Multicache coherence protocols will be studied in Chapter 7.

5.2.1 Cache Addressing Models

Most multiprocessor systems use private caches associated with different processors as depicted in Fig. 5.7. Caches can be addressed using either a physical address or a virtual address. This leads to the two different cache design models presented below.

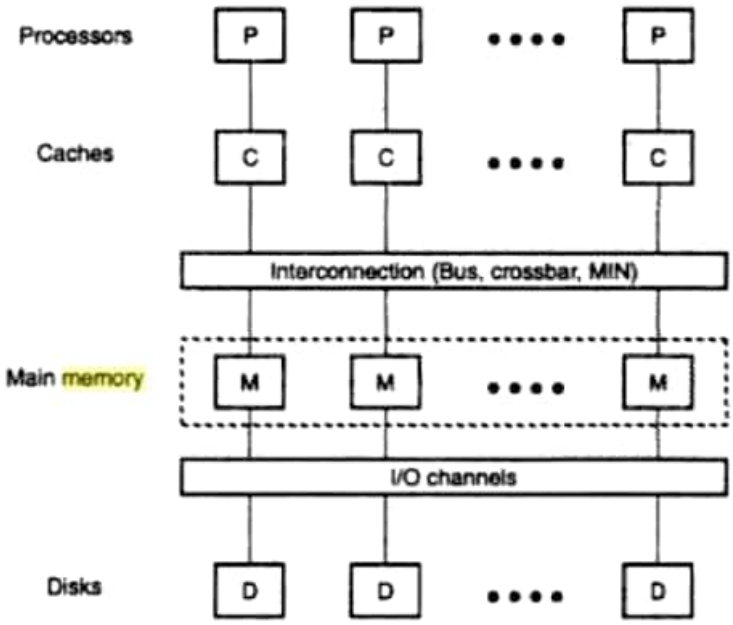


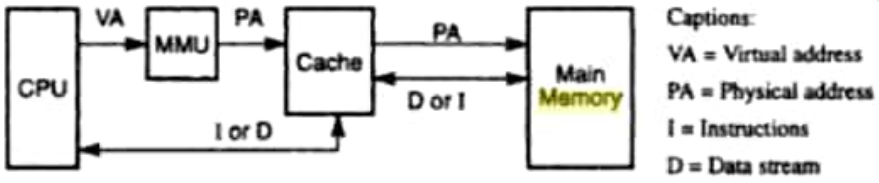
Figure 5.7 A memory hierarchy for a shared-memory multiprocessor.

Physical Address Caches When a cache is accessed with a physical memory address, it is called a *physical address cache*. Physical address cache models are illustrated in Fig. 5.8. In Fig. 5.8a, the model is based on the experience of using a unified cache as in the VAX 8600 and the Intel i486.

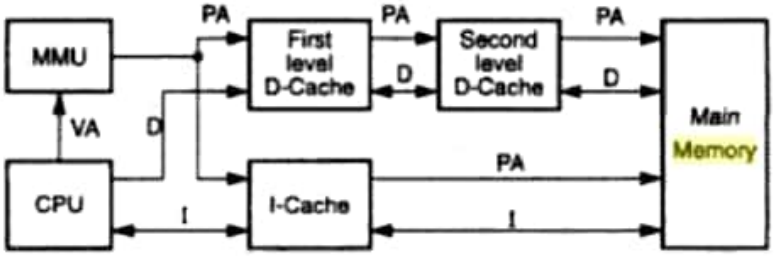
In this case, the cache is indexed and tagged with the physical address. Cache lookup must occur after address translation in the TLB or MMU. No aliasing is allowed so that the address is always uniquely translated without confusion.

A *cache hit* occurs when the addressed data/instruction is found in the cache. Otherwise, a *cache miss* occurs. After a miss, the cache is loaded with the data from the memory. Since a whole cache block is loaded at one time, unwanted data may also be loaded. Locality of references will find most of the loaded data useful in subsequent instruction cycles.

Data is written through the main memory immediately via a *write-through* (WT) cache, or delayed until block replacement by using a *write-back* (WB) cache. A WT cache requires more bus or network cycles to access the main memory, while a WB cache allows the CPU to continue without waiting for the memory to cycle.



(a) A unified cache accessed by physical address



(b) Split caches accessed by physical address in the Silicon Graphics workstation

Figure 5.8 Physical address models for unified and split caches.

Example 5.2 Cache design in a Silicon Graphics workstation

Figure 5.8b demonstrates the split cache design using the MIPS R3000 CPU in the Silicon Graphics 4-D Series workstation. Both data cache and instruction cache are accessed with a physical address issued from the on-chip MMU. A two-level data cache is implemented in this design.

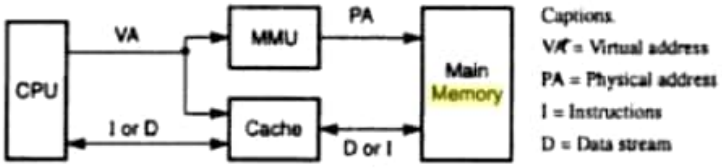
The first level uses 64 Kbytes of WT D-cache. The second level uses 256 Kbytes of WB D-cache. The single-level I-cache is 64 Kbytes. By the inclusion property, the first-level cache is always a subset of the second-level cache. Most manufacturers put the first-level caches on the processor chip and leave the second-level caches as options on the processor board. Hardware consistency needs to be enforced between the two cache levels.

■

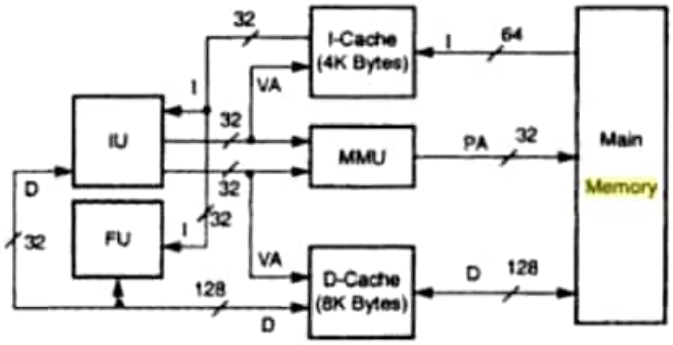
The major advantages of physical address caches include no need to perform cache flushing, no aliasing problems, and thus fewer cache bugs in the OS kernels. The shortcoming is the slowdown in accessing the cache until the MMU/TLB finishes translating the address. This motivates the use of a virtual address cache. Integration of the MMU and caches on the same VLSI chip can alleviate some of these problems.

Most conventional system designs use a physical address cache because of its simplicity and because it requires little intervention from the OS kernel. When physical address caches are used in a UNIX environment, no flushing of data caches is needed if bus watching is provided to monitor the system bus for DMA requests from I/O devices or from other CPUs. Otherwise, the cache must be flushed for every I/O without proper bus watching.

Virtual Address Caches When a cache is indexed or tagged with a virtual address as shown in Fig. 5.9, it is called a *virtual address cache*. In this model, both cache and MMU translation or validation are done in parallel. The physical address generated by the MMU can be saved in tags for later write-back but is not used during the cache lookup operations. The virtual address cache is motivated with its enhanced efficiency to access the cache faster, overlapping with the MMU translation as exemplified below.



(a) A unified cache accessed by virtual address



(b) A split cache accessed by virtual address as in the Intel i860 processor

Figure 5.9 Virtual address models for unified and split caches. (Courtesy of Intel Corporation, 1989)

Example 5.3 The virtual addressed split cache design in Intel i860

Figure 5.9b shows the virtual address design in the Intel i860 using split caches for data and instructions. Instructions are 32 bits wide. Virtual addresses generated by the integer unit (IU) are 32 bits wide, and so are the physical addresses generated by the MMU. The data cache is 8 Kbytes with a block size of 32 bytes. A two-way, set-associative cache organization (Section 5.2.3) is implemented with 128 sets in the D-cache and 64 sets in the I-cache.

■

The Aliasing Problem The major problem associated with a virtual address cache is *aliasing*, when different logically addressed data have the same index/tag in the cache. Multiple processes may use the same range of virtual addresses. This aliasing problem may create confusion if two or more processes access the same physical cache location.

One way to solve the aliasing problem is to flush the entire cache whenever aliasing occurs.

Large amounts of *flushing* may result in a poor cache performance, with a low hit ratio and too much time wasted in flushing. When a virtual address cache is used with UNIX, flushing is needed after each context switching. Before I/O writes or after I/O reads, the cache must be flushed. Furthermore, aliasing between the UNIX kernel and user data is a serious problem. All of these problems will introduce additional system overhead.

Flushing the cache will not overcome the aliasing problem completely when using a shared memory with mapped files and copy-on-write as in UNIX systems. These UNIX operations may not benefit from virtual caches. In each entry/exit to or from the UNIX kernel, the cache must be flushed upon every system call and interrupt.

The virtual space must be divided between kernel and user modes by tagging kernel and user data separately. This implies that a virtual address cache may lead to a lower performance unless most processes are compute-bound.

With a frequently flushed cache, the debugging of programs is almost impossible to perform. Two commonly used methods to get around the virtual address cache problems are to apply special tagging with a *process key* or with a *physical address*. For example, the SUN 3/200 Series has used a virtual address, write-back cache with the capability of being noncacheable. Three-bit keys are used in the cache to distinguish among eight simultaneous contexts.

The flushing can be done at the page, segment, or context level. In this case, context switching does not need to flush the cache but needs to change the current process key. Thus cached shared memory must be shared on fixed-size boundaries. Other memory can be shared with noncacheability. Flushing and special tagging may be traded for performance/cost reasons.

5.2.2 Direct Mapping and Associative Caches

The transfer of information from main memory to cache memory is conducted in units of cache blocks or cache lines. Four block placement schemes are presented below. Each placement scheme has its own merits and shortcomings. The ultimate performance depends on the cache-access patterns, cache organization, and management policy used.

Blocks in caches are called *block frames* in order to distinguish them from the corresponding *blocks* in main memory. Block frames are denoted as $\overline{B_i}$ for $i = 0, 1, 2, \ldots, m$. Blocks are denoted as B_j for $j = 0, 1, 2, \ldots, n$. Various mappings can be defined from set $\{B_j\}$ to set $\{\overline{B_i}\}$. It is also assumed that $n >> m$, $n = 2^s$, and $m = 2^r$.

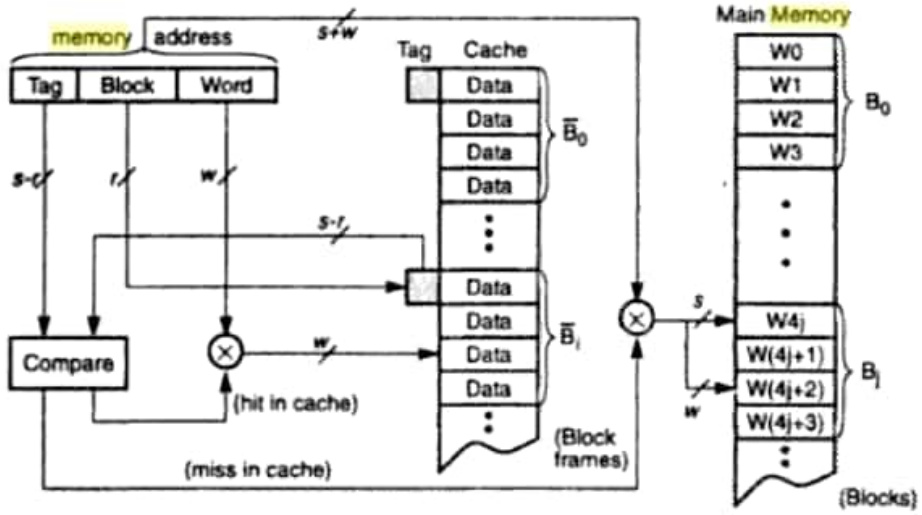
Each block (or block frame) is assumed to have b words, where $b = 2^w$. Thus the cache consists of $m \cdot b = 2^{r+w}$ words. The main memory has $n \cdot b = 2^{s+w}$ words addressed by $(s + w)$ bits. When the block frames are divided into $v = 2^t$ *sets*, $k = m/v = 2^{r-t}$ blocks are in each set.

Direct-Mapping Cache This cache organization is based on a direct mapping of $n/m = 2^{s-r}$ memory blocks, separated by equal distances, to one block frame in the cache. The placement is defined below using a modulo-m function. Block B_j is mapped

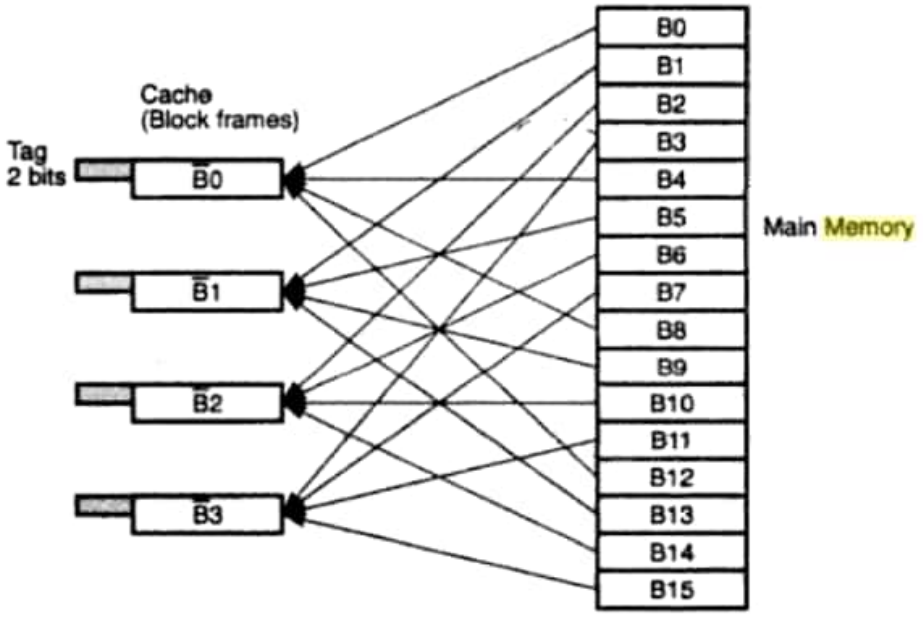
to block frame $\overline{B_i}$:

$$B_j \rightarrow \overline{B_i}, \qquad \text{if } i = j \text{ (modulo } m) \tag{5.1}$$

There is a *unique* block frame $\overline{B_i}$ that each B_j can load into. There is no way to implement a block replacement policy. This direct mapping is very rigid but is the simplest cache organization to implement. Direct mapping is illustrated in Fig. 5.10a for the case where each block contains four words ($w = 2$ bits).



(a) The cache/memory addressing



(b) Block B_j can be mapped to block frame $\bar{B}_i$ if $i = j$ (modulo 4)

Figure 5.10 Direct-mapping cache organization and a mapping example.

The memory address is divided into three fields: The lower w bits specify the *word offset* within each block. The upper s bits specify the *block address* in main memory, while the leftmost $(s - r)$ bits specify the *tag* to be matched. The *block* field (r bits) is used to implement the (modulo-m) placement, where $m = 2^r$. Once the block $\overline{B_i}$ is uniquely identified by this field, the tag associated with the addressed block is compared with the tag in the memory address.

A cache hit occurs when the two tags match. Otherwise a cache miss occurs. In case of a cache hit, the word offset is used to identify the desired data word within the addressed block. When a miss occurs, the entire memory address ($s + w$ bits) is used to access the main memory. The first s bits locate the addressed block, and the lower w bits locate the word within the block.

Example 5.4 Direct-mapping cache design and block mapping

An example mapping is given in Fig. 5.10b, where $n = 16$ blocks are mapped to $m = 4$ block frames, with four possible sources mapping into one destination using modulo-4 mapping. The direct-mapping cache has been implemented in the IBM System/370 Model 158 and in the VAX/8800.

■

Cache Design Parameters In practice, the two parameters n and m differ by at least two to three orders of magnitude. A typical cache block has 32 bytes corresponding to eight 32-bit words. Thus $w = 3$ bits if the machine is word-addressable. If the machine is byte-addressable, one may even consider $w = 5$ bits.

Consider a cache with 64 Kbytes. This implies $m = 2^{11} = 2048$ block frames with $r = 11$ bits. Consider a main memory with 32 Mbytes. Thus $n = 2^{20}$ blocks with $s = 20$ bits, and the memory address needs $s + w = 20 + 3 = 23$ bits for word addressing and 25 bits for byte addressing. In this case, $2^{s-r} = 2^9 = 512$ blocks are possible candidates to be mapped into a single block frame in a direct-mapping cache.

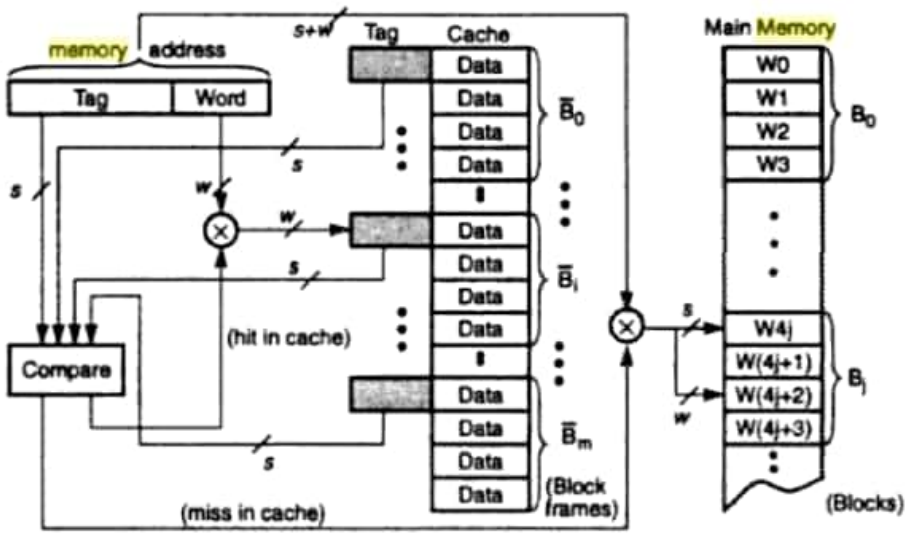
Advantages of a direct-mapping cache include simplicity in hardware, no associative search needed, no page replacement algorithm needed, and thus lower cost and higher speed.

However, the rigid mapping may result in a poorer hit ratio than with the associative mappings to be introduced next. The scheme also prohibits parallel virtual address translation. The hit ratio may drop sharply if many addressed blocks have to map into the same block frame. For this reason, direct-mapped caches tend to use a larger cache size with more block frames to avoid the contention.

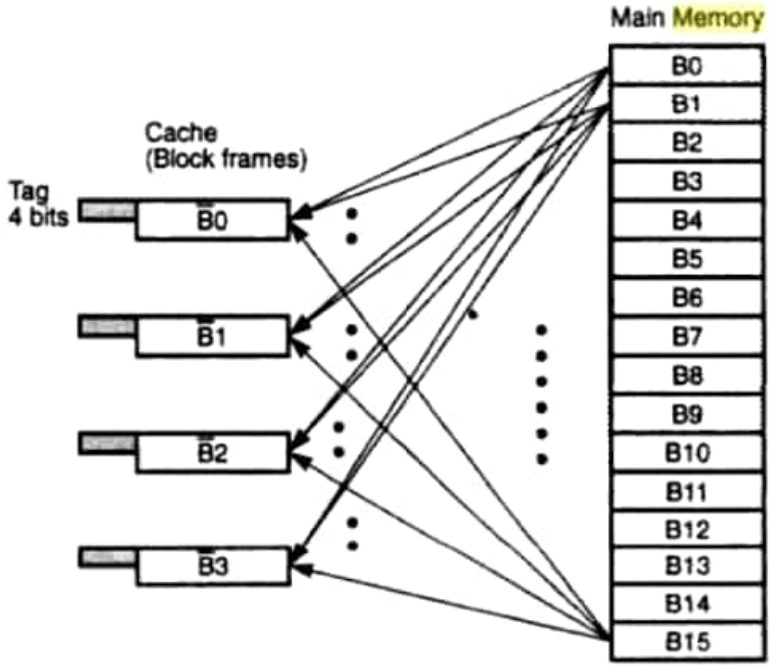
Fully Associative Cache Unlike direct mapping, this cache organization offers the most flexibility in mapping cache blocks. As illustrated in Fig. 5.11a, each block in main memory can be placed in any of the available block frames. Because of this flexibility, an s-bit tag is needed in each cache block. As $s > r$, this represents a significant increase in tag length.

The name *fully associative cache* is derived from the fact that an m-way associative search requires the tag to be compared with all block tags in the cache. This scheme

offers the greatest flexibility in implementing block replacement policies for a higher hit ratio.



(a) Associative search with all block tags



(b) Every block is mapped to any of the four block frames identified by the tag

Figure 5.11 Fully associative cache organization and a mapping example.

The m-way comparison of all tags is very time-consuming if the tags are compared

sequentially using RAMs. Thus an *associative memory* (content-addressable memory, CAM) is needed to achieve a parallel comparison with all tags simultaneously. This demands a higher implementation cost for the cache. Therefore, a fully associative cache has been implemented only in moderate size, such as those used in microprocessor-based computer systems.

Figure 5.11b shows a four-way mapping example using a fully associative search. The tag is 4 bits long because 16 possible cache blocks can be destined for the same block frame. The major advantage of using full associativity is to allow the implementation of a better block replacement policy with reduced block contention. The major drawback lies in the expensive search process requiring a higher hardware cost.

5.2.3 Set-Associative and Sector Caches

Set-associative caches are the most popular cache designs built into commercial computers. Sector mapping caches offer a design alternative to set-associative caches. These two types of cache design are described below.

Set-Associative Cache This design offers a compromise between the two extreme cache designs based on direct mapping and full associativity. If properly designed, this cache may offer the best performance-cost ratio. Most high-performance computer systems are based on this approach. The idea is illustrated in Fig. 5.12.

In a *k-way associative* cache, the m cache block frames are divided into $v = m/k$ sets, with k blocks per set. Each set is identified by a d-bit *set number*, where $2^d = v$. The cache block tags are now reduced to $s - d$ bits. In practice, the *set size* k, or *associativity*, is chosen as 2, 4, 8, 16, or 64, depending on a tradeoff among block size w, cache size m, and other performance/cost factors.

Fully associative mapping can be visualized as having a single set (i.e., $v = 1$) or an m-way associativity. In a k-way associative search, the *tag* needs to be compared only with the k tags within the identified set, as shown in Fig. 5.12a. Since k is rather small in practice, the k-way associative search is much more economical than the full associativity.

In general, a block B_j can be mapped into any one of the available frames $\overline{B_f}$ in a set S_i defined below. The matched tag identifies the current block which resides in the frame.

$$B_j \rightarrow \overline{B_f} \in S_i \qquad \text{if} \quad j(\text{modulo } v) = i \tag{5.2}$$

Design Tradeoffs The set size (associativity) k and the number of sets v are inversely related by

$$m = v \times k \tag{5.3}$$

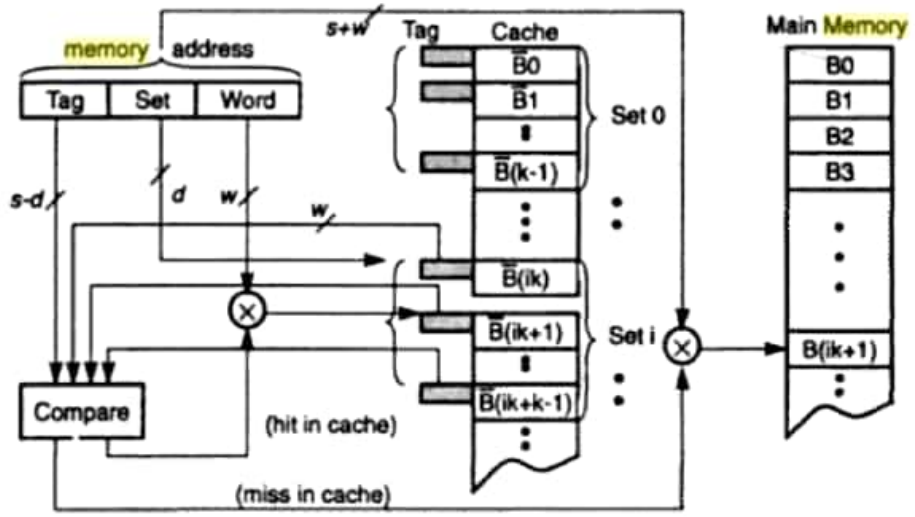
For a fixed cache size there exists a tradeoff between the set size and the number of sets.

The advantages of the set-associative cache include the following:

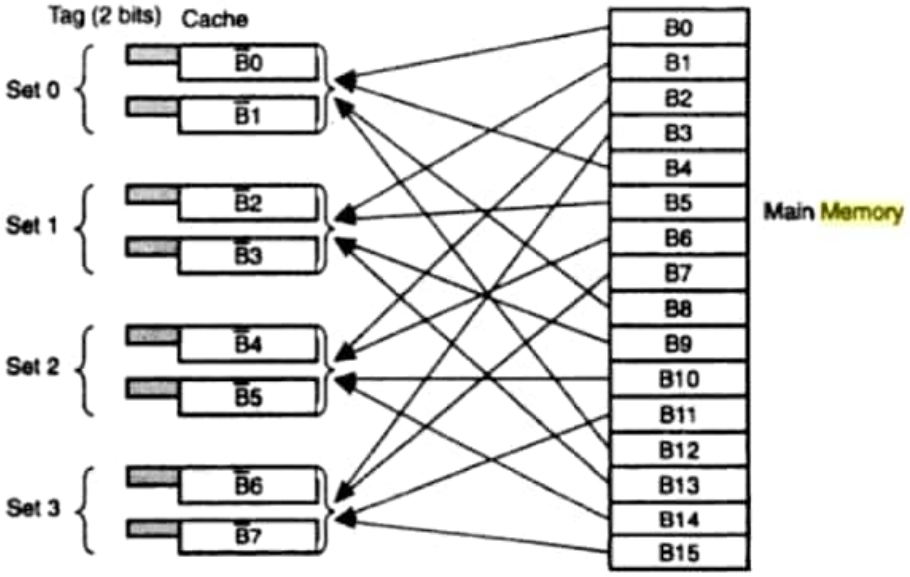
First, the block replacement algorithm needs to consider only a few blocks in the same set. Thus the replacement policy can be more economically implemented with limited choices, as compared with the fully associative cache.

Second, the k-way associative search is easier to implement, as mentioned earlier.

Third, many design tradeoffs can be considered (Eq. 5.3) to yield a higher hit ratio in the cache. The cache operation is often used together with TLB.



(a) A k-way associative search within each set of k cache blocks



(b) Mapping cache blocks in a two-way associative cache with four sets

Figure 5.12 Set-associative cache organization and a two-way associative mapping example.

Example 5.5 Set-associative cache design and block mapping

An example is shown in Fig. 5.12b for the mapping of $n = 16$ blocks from main memory into a two-way associative cache ($k = 2$) with $v = 4$ sets over $m = 8$ block frames. For the i860 example in Fig. 5.9b, both the D-cache and I-cache are two-way associative ($k = 2$). There are 128 sets in the D-cache and 64 sets in the I-cache, with 256 and 128 block frames, respectively. Table 5.1 presents other example systems using set-associative caches. Note that the associativity ranges from 2 to 16 in these systems.

Table 5.1 Representative Set-Associative Cache Organizations

System	Cache Size, m	Set Size, k	No. of Sets, v
DEC VAX 11/780	1024	2	512
Amdahl 470/V6	512	2	256
Intel i860 D-cache	256	2	128
Honeywell 66/60	512	4	128
Amdahl 470/V7	2048	4	512
IBM 370/168	1024	8	128
IBM 3033	1024	16	64
Motorola 88110 I-cache	256	2	128

■

Sector Mapping Cache This block placement scheme is generalized from the above schemes. The idea is to partition both the cache and main memory into fixed-size *sectors*. Then a fully associative search is applied. That is, each sector can be placed in any of the available sector frames.

The memory requests are destined for blocks, not for sectors. This can be filtered out by comparing the sector tag in the memory address with all sector tags using a fully associative search. If a matched sector frame is found (a cache hit), the block field is used to locate the desired block within the sector frame.

If a cache miss occurs, only the missing block is fetched from the main memory and brought into a congruent block frame in an available sector. That is, the ith block in a sector must be placed into the ith block frame in a destined sector frame. A *valid bit* is attached to each block frame to indicate whether the block is *valid* or *invalid*.

When the contents of a block frame are replaced, the remaining block frames in the same sector are marked invalid. Only the most recently replaced block frame in a sector is marked valid for reference. However, multiple valid bits can be used to record other block states. The sector mapping just described can be modified to yield other designs, depending on the block replacement policy being implemented.

Compared with fully associative or set-associative caches, the sector mapping cache offers the advantages of being flexible to implement various block replacement algorithms and being economical to perform a fully associative search across a limited number of

sector tags.

The sector partitioning offers more freedom in grouping cache lines at both ends of the mapping. Making design choice between set-associative and sector mapping caches requires more trace and simulation evidence.

Example 5.6 Sector mapping cache design

Figure 5.13 shows an example of sector mapping with a sector size of four blocks. Note that each sector can be mapped to any of the sector frames with full associativity at the sector level.

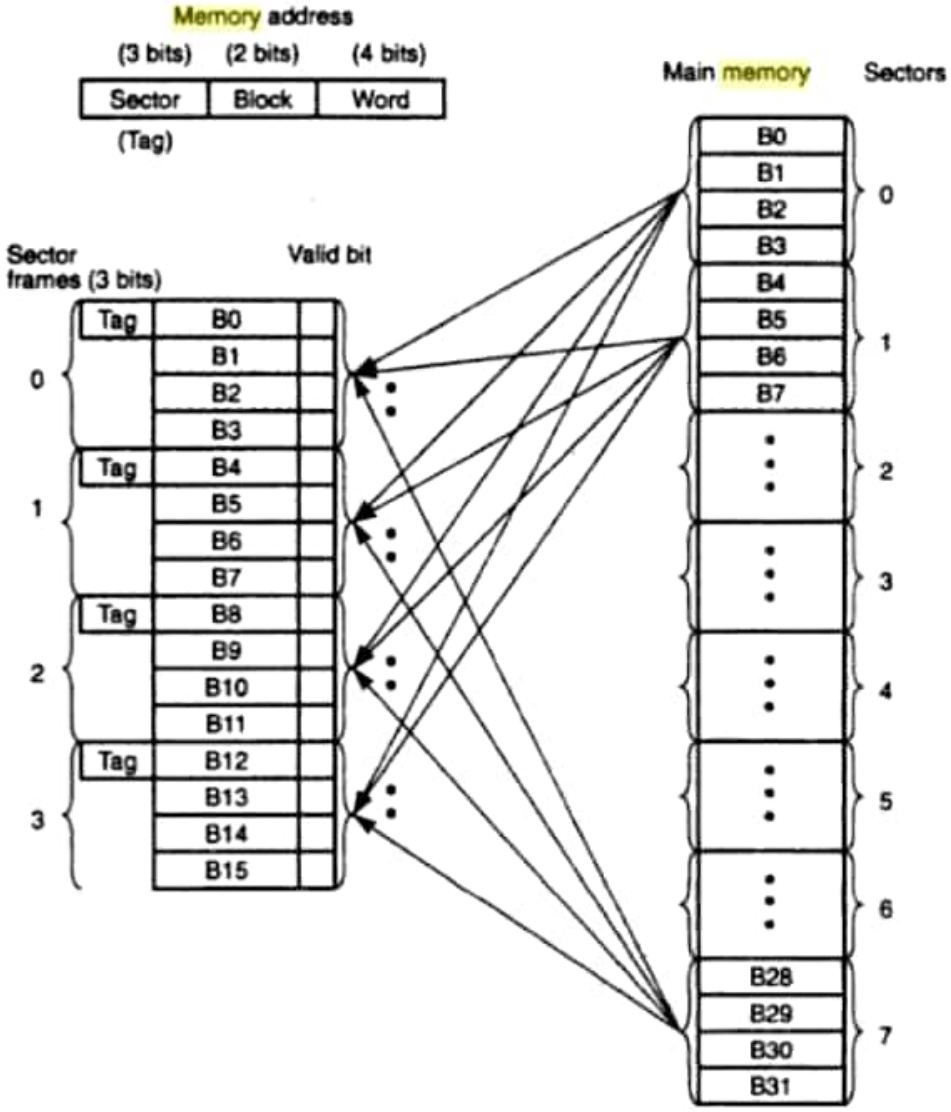


Figure 5.13 A four-way sector mapping cache organization.

This scheme was first implemented in the IBM System/360 Model 85. In the

Model 85, there are 16 sectors, each having 16 blocks. Each block has 64 bytes, giving a total of 1024 bytes in each sector and a total cache capacity of 16 Kbytes using a LRU block replacement policy.

■

5.2.4 Cache Performance Issues

The performance of a cache design concerns two related aspects: the *cycle count* and the *hit ratio*. The cycle count refers to the number of basic machine cycles needed for cache access, update, and coherence control. The hit ratio determines how effectively the cache can reduce the overall memory-access time. Tradeoffs do exist between these two aspects. Key factors affecting cache speed and hit ratio are discussed below.

Program trace-driven simulation and *analytical modeling* are two complementary approaches to studying cache performance. Both have to be applied together in order to provide a credible performance assessment.

Simulation studies present snapshots of program behavior and cache responses but they suffer from having a microscopic perspective.

Analytical models may deviate from reality under simplification. However, they provide some macroscopic and intuitive insight into the underlying processes.

Agreement between results generated from the two approaches allows one to draw a more credible conclusion. However, the generalization of any conclusion is limited by the finite-sized address traces and by the assumptions about address trace patterns. Simulation results can be used to verify the theoretical results, and analytical formulation can guide simulation experiments on a wider range of parameters.

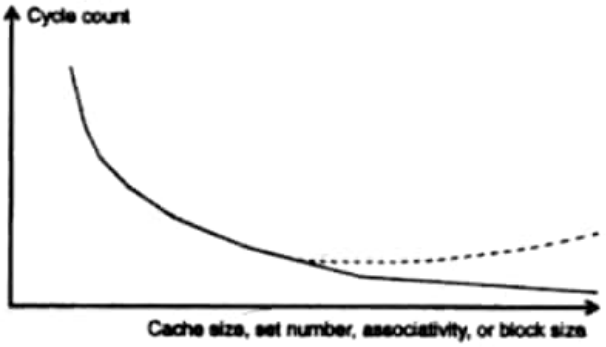
Cycle Counts The cache speed is affected by the underlying static or dynamic RAM technology, the cache organization, and the cache hit ratios. The total cycle count should be predicated with appropriate cache hit ratios. This will affect various cache design decisions, as already seen in previous sections.

The cycle counts are not credible unless detailed simulation of all aspects of a memory hierarchy is performed. The write-through or write-back policies also affect the cycle count. Cache size, block size, set number, and associativity all affect the cycle count as illustrated in Fig. 5.14.

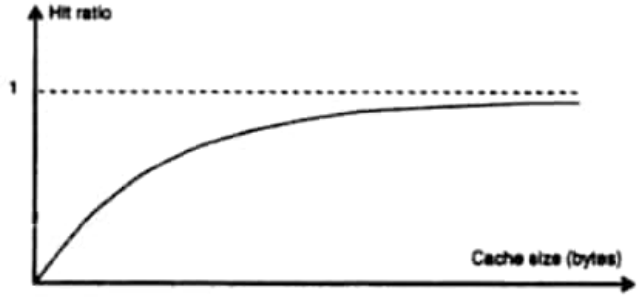
The cycle count is directly related to the hit ratio, which decreases almost linearly with increasing values of the above cache parameters. But the decreasing trend becomes flat and after a certain point turns into an increasing trend (the dashed line in Fig. 5.14a). This is caused primarily by the effect of the block size on the hit ratio, which will be discussed below.

Hit Ratios The cache hit ratio is affected by the cache size and by the block size in different ways. These effects are illustrated in Figs. 5.14b and 5.14c, respectively. Generally, the hit ratio increases with respect to increasing cache size (Fig. 5.14b).

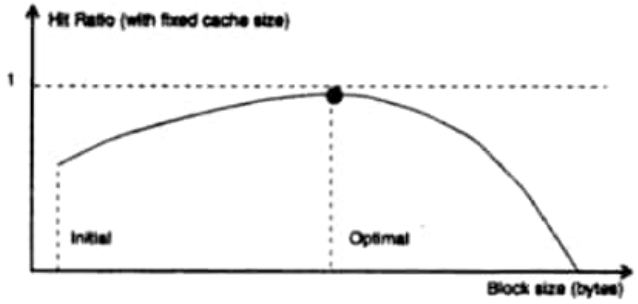
When the cache size approaches infinity, a 100% hit ratio should be expected. However, this will never happen because the cache size is always bounded by a limited budget. The initial cache loading and changes in locality also prevent such an ideal



(a) The total cycle count for cache access (Courtesy of S. A. Przybylski; reprinted with permission from *Cache and Memory Hierarchy Design*, Morgan Kaufmann Publishers, 1990)



(b) Hit ratio versus cache size



(c) Hit ratio versus block size

Figure 5.14 Cache performance versus design parameters used.

performance. The curves in Fig. 5.14b can be approximated by $1 - C^{-0.5}$, where C is the total cache size.

Effect of Block Size With a fixed cache size, cache performance is rather sensitive to block size. Figure 5.14c illustrates the rise and fall of the hit ratio as the cache block varies from small to large. Initially, we assume a block size (such as 32 bytes per block). This block size is determined mainly by the temporal locality in typical programs.

As the block size increases, the hit ratio improves because of spatial locality in referencing larger instruction/data blocks. The increase reaches its peak at a certain *optimum block size*. After this point, the hit ratio decreases with increasing block size. This is caused by the mismatch between program behavior and block size.

As a matter of fact, as the block size becomes very large, many words fetched into the cache may never be used. Also, the temporal locality effects are gradually lost with larger block size. Finally, the hit ratio approaches zero when the block size equals the entire cache size.

For a bus-based system, Smith (1987) has determined that the optimum block size should be chosen to minimize the effective memory-access time. This optimum size depends on the ratio of the access latency and the bus cycle time (data transfer rate). He has identified design targets for the hit ratio, bus traffic, and average delay per reference based on an empirical model derived from a wide variety of benchmark simulations.

Effects of Set Number In a set-associative cache, the effects of set number are obvious. For a fixed cache capacity, the hit ratio may decrease as the number of sets increases. As the set number increases from 32 to 64, 128, and 256, the decrease in the hit ratio is rather small based on Smith's 1982 report. When the set number increases to 512 and beyond, the hit ratio decreases faster. Also, the tradeoffs between block size and set number should not be ignored (Eq. 5.3).

Other Performance Factors In a performance-directed design, tradeoffs exist among the cache size, set number, block size, and memory speed. Independent blocks, fetch sizes, and fetch strategies also affect the performance in various ways.

Multilevel cache hierarchies offer options for expanding the cache effects. Very often, a write-through policy is used in the first-level cache, and a write-back policy in the second-level cache. As in the memory hierarchy, an optimal cache hierarchy design must match special program behavior in the target application domain.

The distribution of cache references for instruction, loads, and writes of data will affect the hierarchy design. Based on some previous program traces, 63% instruction fetches, 25% loads, and 12% writes were reported. This affects the decision to split the instruction cache from the data cache.

Of course, the optimal hierarchy design must be based on the access time and memory technology used. Pipelined access to the cache is also very much desired to enhance the performance.