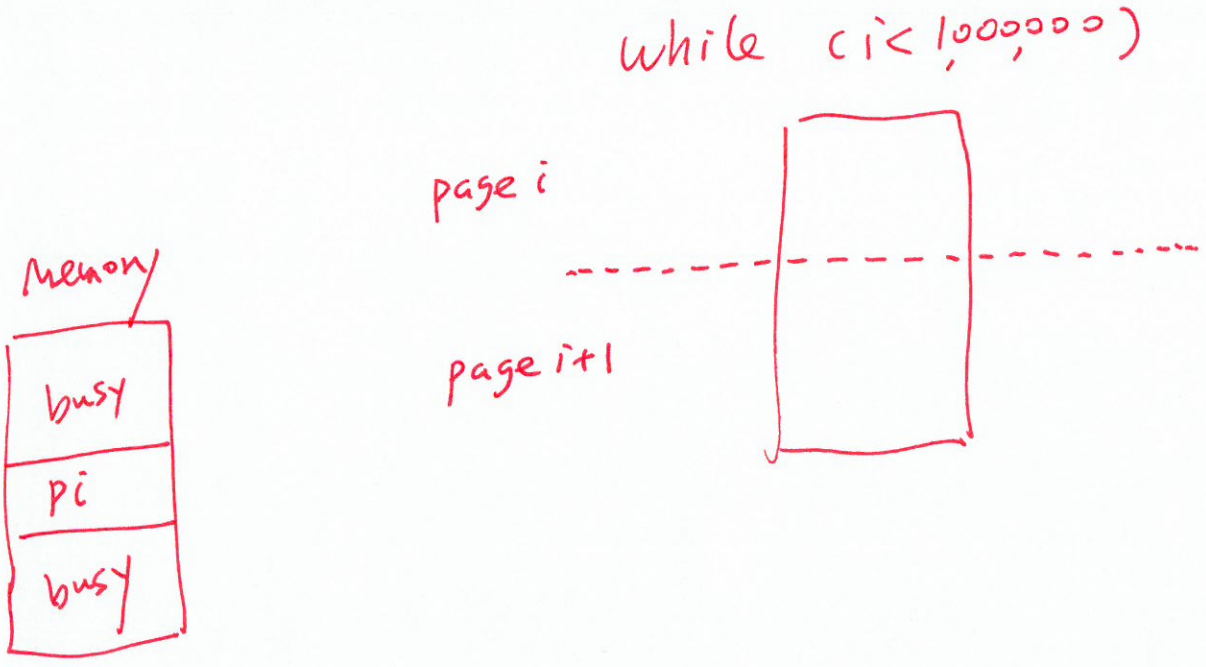
- How does the computer fetch an instruction?
  - 1. Start processing instruction
  - 2. Generate data address
  - 3. Compute page number
  - 4. If page is in memory
  - then
  - get data and finish instruction
  - advance to the next instruction
  - return to step 1
  - else
  - generate page interrupt
  - call page interrupt handler

- **Algorithm: Page Interrupt Handler**
  - 1. If there is no free page frame
  - then
  - select page to be swapped out using
  - a page removal algorithm
  - update job's Page Map Table
  - if content of page had been changed
  - then write page to disk
  - 2. Use page number (step 3 of the previous
  - algorithm to get disk address where
  - page is stored (the File Manager, to
  - be discussed later, uses the page
  - number to get the disk address)
  - 3. Read page into memory
  - 4. Update job's Page Map Table
  - 5. Update Memory Map Table
  - 6. Restart interrupted instruction

FIFO
least used
1st update
lazy write is also possible.
2nd update, for the incoming page's job.

- Although demand paging is a solution to inefficient memory utilization, it does not solve all the problems
- **Thrashing**: if a large amount of page swapping is performed, the system efficiency is affected.
- **Page fault**: a failure to find a page in memory.



CSCI 460—Operating Systems

Lecture 4

Memory Management–memory hierarchy

Textbook: Operating Systems
by William Stallings

1. The Memory Hierarchy

- In the more recent decades, computer memory is not arranged in a linear fashion.
- The design constraints on memory rest on:
  - 1. `Capacity.`
  - 2. `Speed (access time).`
  - 3. `Cost (unit cost).`
- Their relationship
  - Faster Speed (access time) → Greater Cost.
  - Greater Capacity → Smaller Cost.
  - From these two, we have: Greater Capacity → Slower Speed.
  - So you can't have Greater Capacity, Small Cost and Fast Speed at the same time!

$A \to B \iff \sim B \to \sim A$

or, smaller cost → slower speed

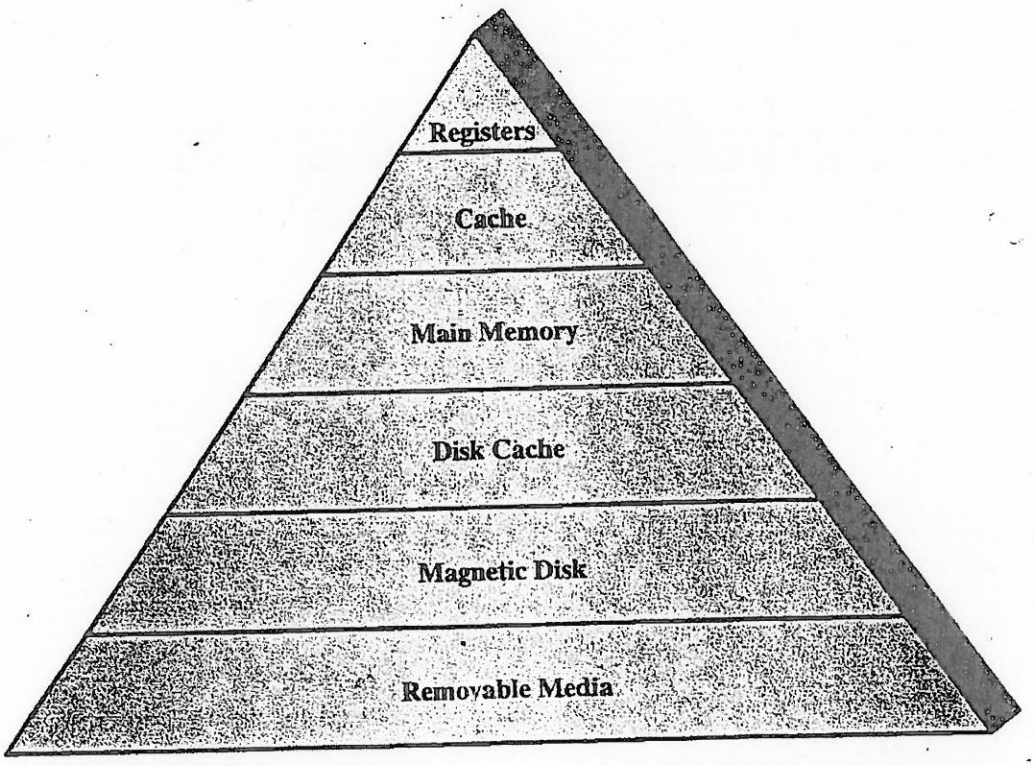
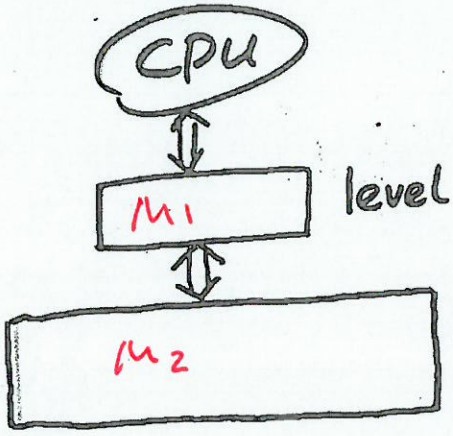


Figure 1.14 The Memory Hierarchy

2. Memory Hierarchy (cont.)

- If we look from top to bottom at Figure 1.14 (in Stallings), the following can be observed.
  - Cost is decreasing.
  - Memory capacity is increasing.
  - Speed is slowing down.
  - Frequency of access of memory by processor is decreasing.
- Why?
  - **Locality of Reference.**
  - Locality of reference is not only valid in OS. It is the basis for compiler optimization, computer architecture and database management (and recently in the Internet browsing).
- Thanks to the semiconductor industry (for building different kinds of storage media for us)!



level 1 memory : 1,000 words
0.1 μs access time

level 2 memory : 100,000 words
1 μs access time

T_1 — access time for M_1
T_2 — access time for M_2
T — access time for the whole system
H — Hit ratio, the probability that you can find an item in M_1.

$$T = H \cdot T_1 + (1-H) \cdot (T_1 + T_2)$$
$$= \cancel{HT_1} + T_1 + T_2 - \cancel{HT_1} - HT_2$$
$$= T_1 + T_2 - HT_2$$
$$= T_1 + (1-H) \cdot T_2$$

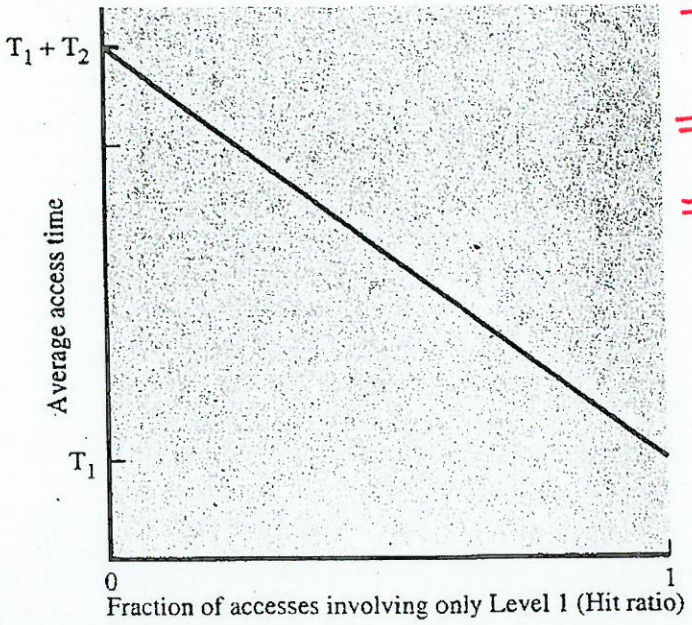


Figure 1.15 Performance of a Simple Two-Level Memory

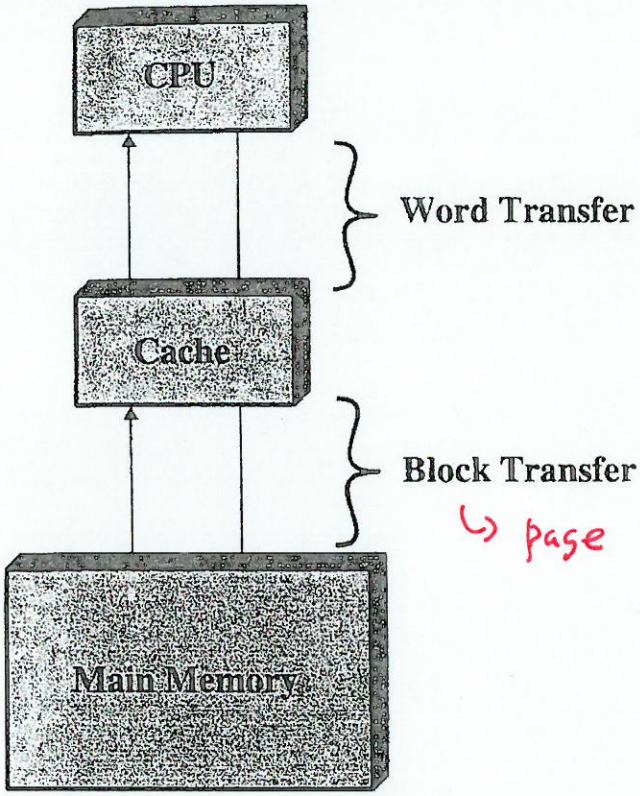
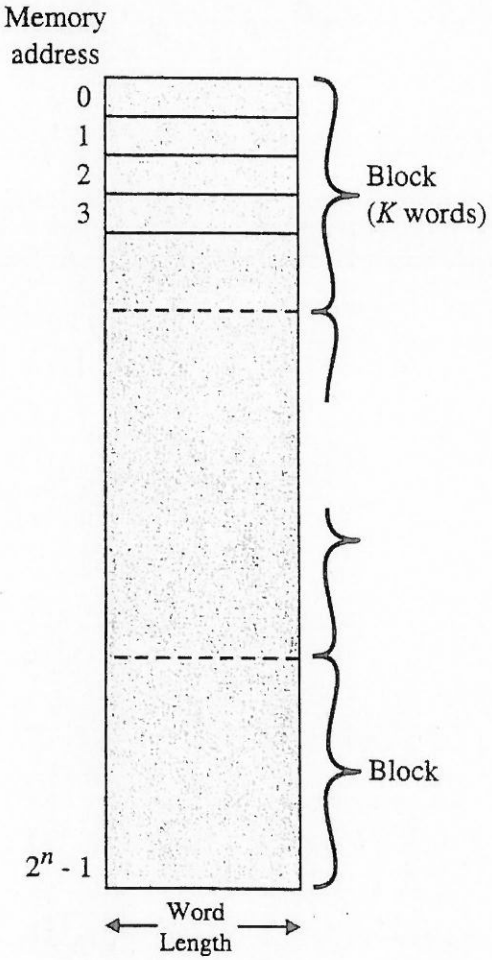
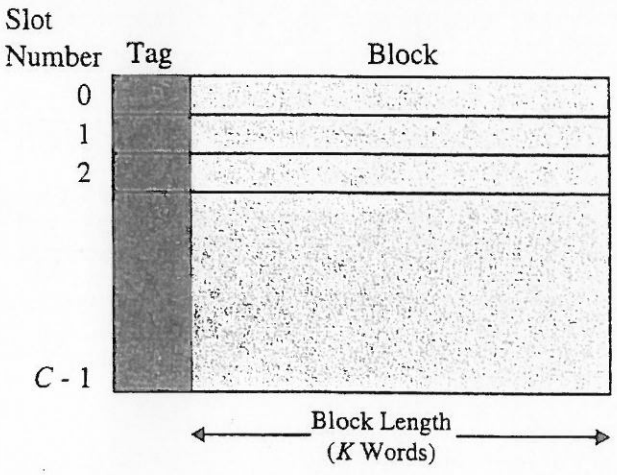


Figure 1.16 Cache and Main Memory



(a) Main memory



(b) Cache

$C << M = \frac{2^n}{K}$

$=$

# of blocks in main memory

Figure 1.17 Cache/Main-Memory Structure

3. Cache Memory

- **Motivation.**
  - On all instruction cycles, the processor access memory at least once: to fetch the instruction, to fetch operands and/or store the results. *Think of executing an assemble instruction*: `ADD C, A, B` ($C \leftarrow A + B$).
  - In general memory access speed cannot match the processor speed. So it makes sense to exploit the principle of locality by building a small, fast memory between the processor and main memory.
- This fast memory, almost *invisible* from OS, is **cache**.
- The objective of cache memory is to speed up the memory so that it is almost as fast as the speed of processor and at the same time it provides a memory size which is large enough (for most jobs).
- Let's us look at the structure of a cache/memory system.

RA — Read Address

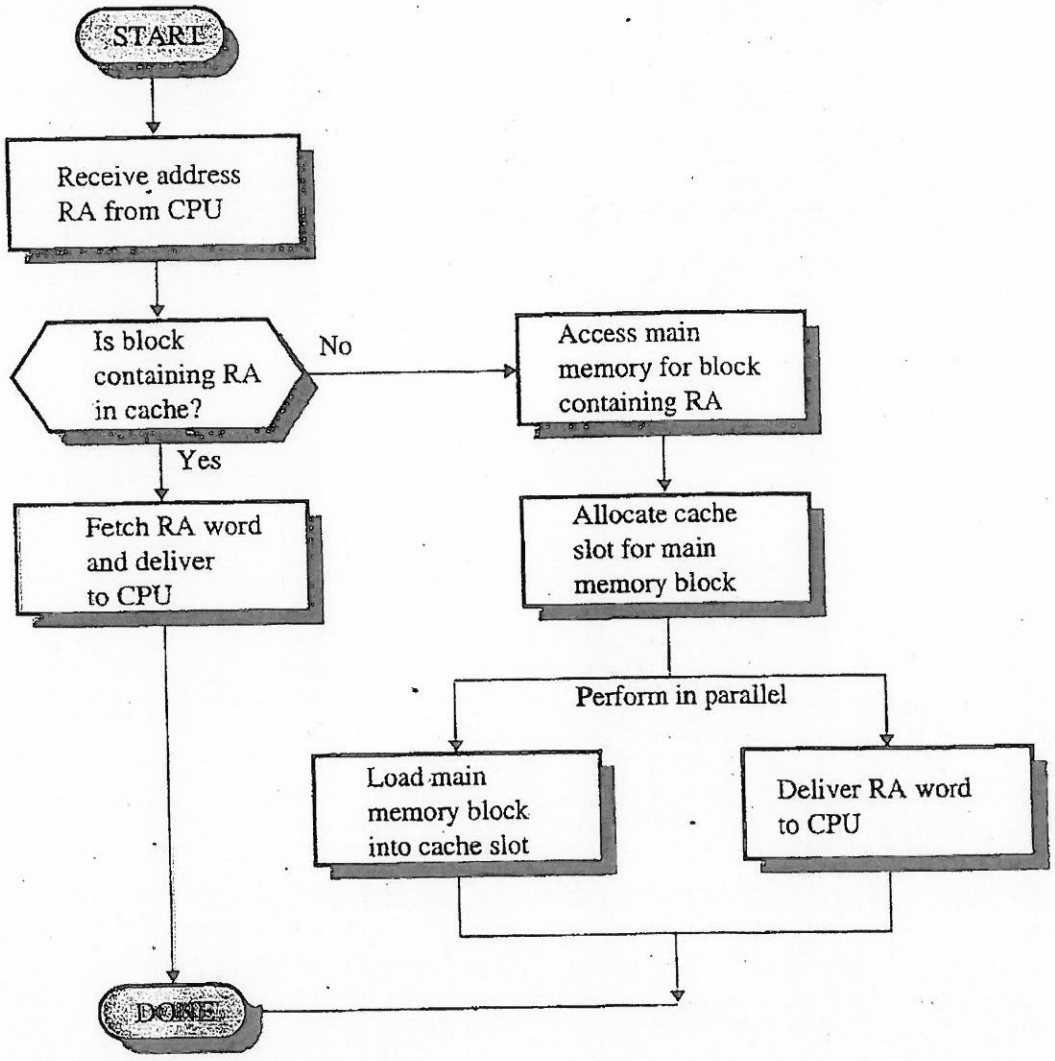


Figure 1.18 Cache Read Operations

4. Cache Memory (cont.)

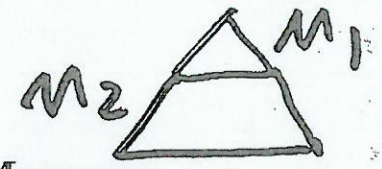
- Let's look at Figure 1.18. What problems can you see with this example?

  - Sth must be moved out of the cache to load the block containing RA into the cache.
  - write/wait?

- Cache design is beyond this course. But the following issues must be considered in general.
  - 1. Cache size.
  - 2. Block size. Suitable size of block will ensure that the hit ratio is high.
  - 3. Mapping function. When a block is read into the cache, the 1st question is to decide where we should put it. (2 hints: (A) When one block is read in, another one should be moved out, so we should minimize the probability that a moved-out block will be referenced again in the near future. (B) The more flexible the mapping function, the more time it takes to search the cache to find a block.)

    hash function without chaining

  - 4. The replacement policy. (Can you think of some?) FIFO
  - 5. Write policy. If the contents of a block in the cache are changed, we should write it back to the main memory before replacing it. So when should this write operation takes place?



5 Performance Analysis of Two-level Memory

- Assume that we have two levels of memory, M_1, M_2 (M_1 is smaller, but faster.) Let's first look at the average system access time T_s.

$$\begin{aligned} T_s &= H \times T_1 + (1 - H) \times (T_1 + T_2) \\ &= T_1 + (1 - H) \times T_2 \end{aligned} \tag{1.1}$$

where

T_s = average (system) access time
T_1 = access time of M1 (e.g., cache, disk cache)
T_2 = access time of M2 (e.g., main memory, disk)
H = hit ratio (fraction of time reference is found in M1)

- Let $\frac{T_1}{T_s}$ be the *access efficiency*, we have

$$\frac{T_1}{T_s} = \frac{1}{1 + (1 - H)\frac{T_2}{T_1}}$$

→ constant.

We want this ratio to be close to 1.

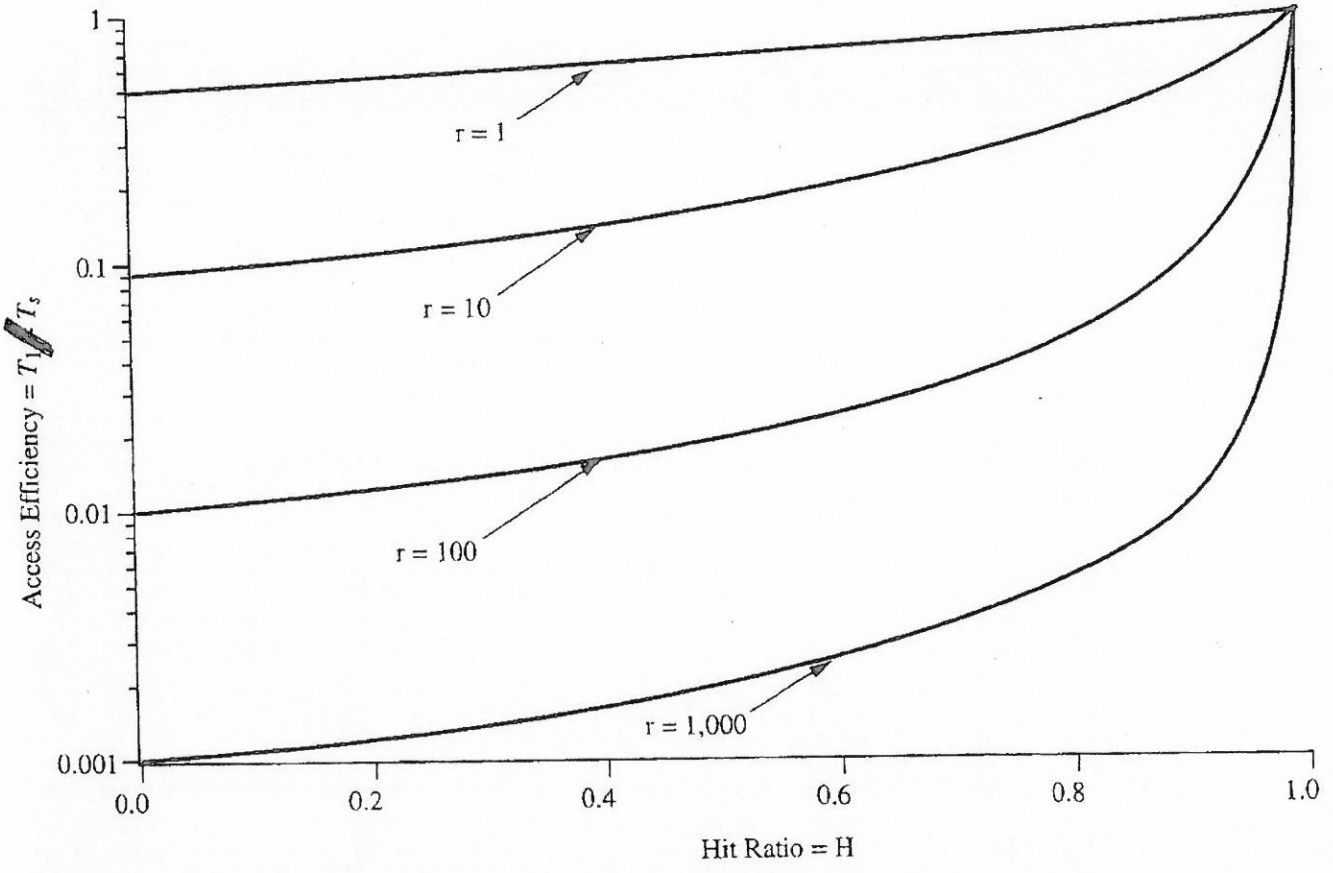


Figure 1.23 Access Efficiency as a Function of Hit Ratio ($r = T_2/T_1$)

- Let's now look at the average cost per bit for the two-level memory, C_s.

$$C_s = \frac{C_1 S_1 + C_2 S_2}{S_1 + S_2} \tag{1.2}$$

where

C_s = average cost per bit for the combined two-level memory
C_1 = average cost per bit of upper-level memory M1
C_2 = average cost per bit of lower-level memory M2
S_1 = size of M1
S_2 = size of M2

- To make C_s roughly the same as C_2. We should make $S_1 << S_2$. ($C_1 >> C_2$ due to the hardware cost, which we can do very little to change it.) Notice that → constant

$$\frac{C_s}{C_2} = \frac{\frac{C_1}{C_2} + \frac{S_2}{S_1}}{1 + \frac{S_2}{S_1}}.$$

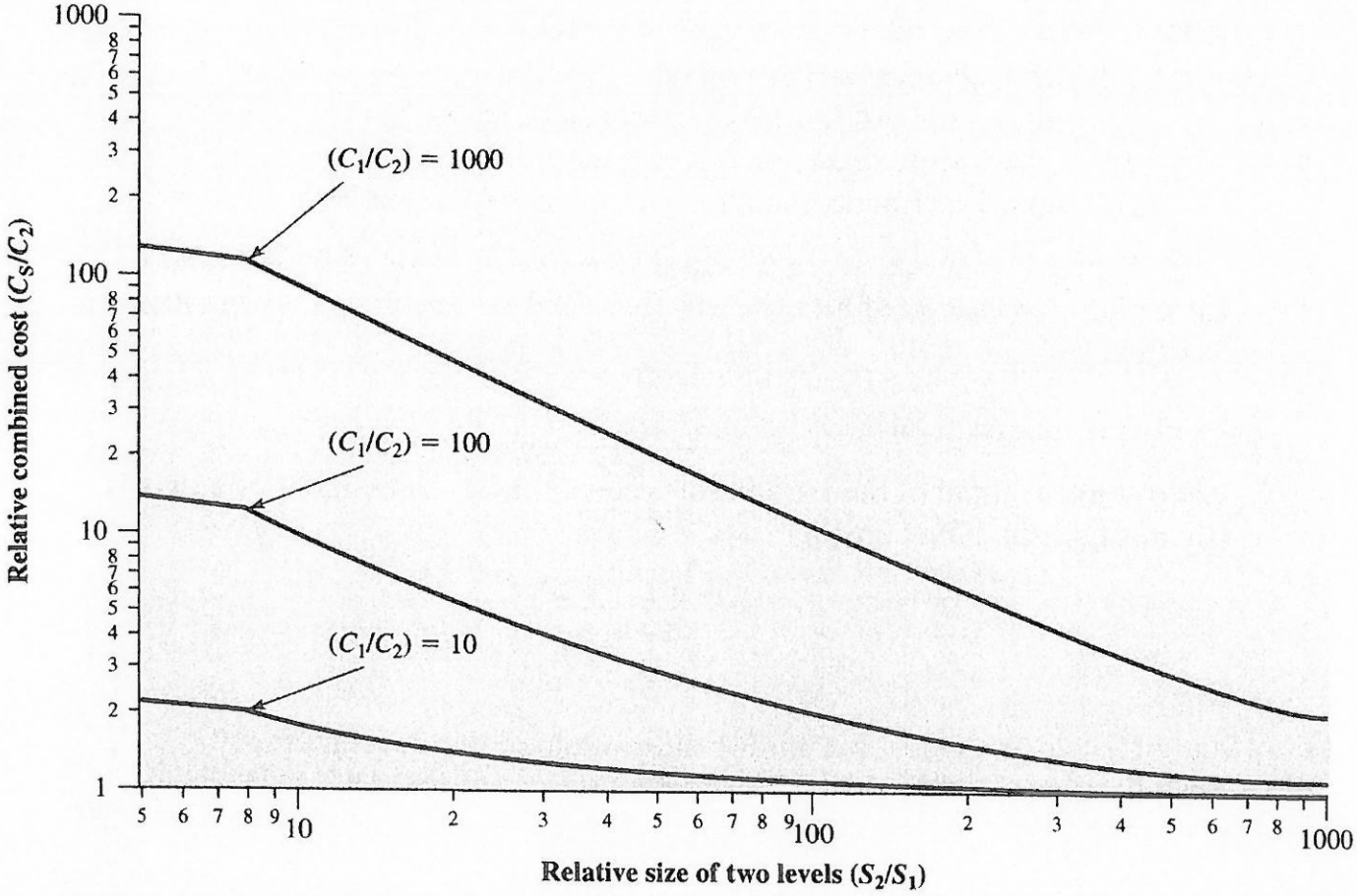


Figure 1.22 Relationship of Average Memory Cost to Relative Memory Size for a Two-Level Memory

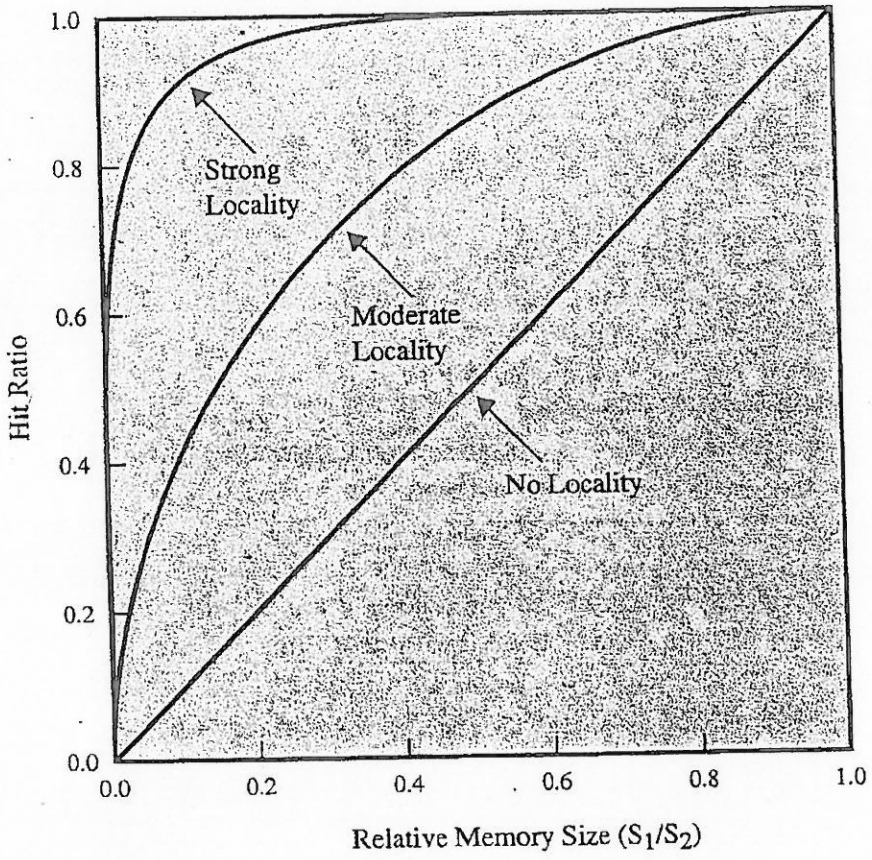


Figure 1.24 Hit Ratio as a Function of Relative Memory Size

In practice,

(1) Cache size: 1K ~ 128K

(2) Hit ratio > 0.75 almost all the time