



Memory Subsystems

The memory hierarchy, cache memories, DRAM main memory and virtual memory — how a processor gets the illusion of a memory that is both large and fast

What this handbook covers

- | | |
|-------------------------------------|---------------------------------|
| 1 The memory hierarchy and locality | 3 Main memory: DRAM |
| 2 Cache memory | 4 Virtual memory |
| 5 The memory system in practice | Appendices: problems, simulator |

Dr. Bikram Paul

School of Computing and Electrical Engineering (SCEE)
Indian Institute of Technology Mandi
bikram@iitmandi.ac.in

Jan–June 2026

Contents

1	The Memory Hierarchy and Locality	1
1.1	Memory: the programmer's view and the ideal	1
1.2	Memory technologies	2
1.2.1	DRAM	2
1.2.2	SRAM	2
1.2.3	Bank organization and access	2
1.2.4	The practical limitation	3
1.3	The memory hierarchy	3
1.4	Locality of reference	4
1.4.1	Locality in everyday life	4
1.4.2	Memory locality	4
1.4.3	Exploiting locality	5
1.5	Caching in a pipelined design	5
1.6	Hierarchical latency analysis	6
1.7	Beyond SRAM and DRAM: other memory technologies	7
	Exercises	8
2	Cache Memory	9
2.1	What is a cache?	9
2.1.1	Caching basics	9
2.1.2	Cache abstraction and metrics	9
2.2	Blocks and addressing	10
2.3	Placement: direct-mapped, set-associative, fully associative	10
2.3.1	Direct-mapped cache	10
2.3.2	Set associativity	11
2.3.3	Trade-offs of associativity	11
2.4	Replacement policies	12
2.4.1	Implementing LRU	12
2.4.2	Approximations of LRU	12
2.4.3	LRU versus random	13
2.4.4	Optimal replacement (Belady's OPT)	13

2.4.5	Cache versus page replacement	14
2.5	The tag-store entry and write policies	14
2.5.1	When do we write modified data to the next level?	14
2.5.2	Do we allocate a block on a write miss?	14
2.5.3	Sectored (sub-blocked) caches	15
2.6	Instruction versus data caches; multi-level caches	15
2.7	Write buffers and inclusion policies	16
2.8	Cache parameters versus miss rate	17
2.9	Classification of cache misses (the 3 Cs)	18
2.9.1	Reducing cache misses	18
2.9.2	Improving cache performance	18
2.10	Cache-aware programming	19
	Exercises	20
3	Main Memory: DRAM	21
3.1	The main memory system	21
3.2	Major trends affecting main memory	21
3.2.1	The memory capacity gap	22
3.2.2	The DRAM scaling problem and reliability	22
3.2.3	Memory interference	22
3.3	Fundamental concepts and organization	23
3.3.1	Page-mode DRAM and the row buffer	23
3.3.2	A 128M × 8-bit DRAM chip	24
3.3.3	Rank and module	24
3.3.4	Multiple DIMMs and channels	24
3.3.5	Transferring a cache block	24
3.4	Latency of basic DRAM operation	25
3.5	Multiple banks (interleaving) and channels	25
3.6	Address mapping	26
3.6.1	Bank-mapping randomization	27
3.6.2	Address mapping with multiple channels	27
3.6.3	Interaction of virtual-to-physical mapping	27
3.7	The memory controller	28
	Exercises	29
4	Virtual Memory	30

4.1	Programmers' view and the abstraction	30
4.1.1	A system with physical memory only	30
4.2	Virtual memory: the basic mechanism	31
4.2.1	Virtual pages and physical frames	31
4.2.2	Physical memory as a cache	31
4.2.3	Supporting virtual memory	32
4.3	Page faults	32
4.4	Address translation	32
4.4.1	The page-table entry	33
4.5	Page replacement	34
4.6	Page-size trade-offs	34
4.7	Protection	35
4.8	The three major issues	35
4.8.1	Issue I: page-table size	36
4.8.2	Issue II: speed of translation — the TLB	37
4.8.3	Issue III: address translation and caching	38
4.9	Virtual memory in action	39
4.9.1	Thrashing and the working set	39
	Exercises	40
5	The Memory System in Practice	41
5.1	The journey of a load	41
5.2	Non-blocking caches and memory-level parallelism	42
5.3	Victim caches	42
5.4	Prefetching	43
5.5	Caches in multi-core systems: coherence	44
5.5.1	A snooping MSI protocol	44
5.5.2	False sharing	45
5.5.3	Directory-based coherence	45
5.5.4	Coherence with I/O and accelerators	46
5.6	Memory consistency in one paragraph	46
5.7	The energy of data movement	46
	Exercises	47
A	Solved Problems	48

B A Trace-Driven Cache Simulator	51
C Quick Review Questions	53
D Formula Sheet	55

The Memory Hierarchy and Locality

Units 2 and 3 assumed a “magic” memory that answers in one cycle. Real memories are either fast and small or large and slow — never both. This chapter explains why, by looking inside SRAM and DRAM, and shows how a **memory hierarchy** exploiting **locality of reference** gives programs the illusion of a memory that is simultaneously large, fast and cheap. A simple recursive latency model quantifies how well the illusion works.

1.1 Memory: the programmer’s view and the ideal

To the programmer, memory is a large, flat array of bytes: load and store name an address and the hardware does the rest. Two abstractions are involved:

- **Virtual memory** — the programmer sees a large virtual address space (e.g. 2^{39} bytes in RV64 Sv39); the system maps it onto a smaller physical memory and disk (Chapter 4).
- **Physical memory** — the actual storage, organized as a hierarchy of caches, main memory (DRAM) and storage.

The **ideal memory** would have zero access time (latency), infinite capacity, zero cost and infinite bandwidth (to support multiple accesses in parallel). The requirements conflict:

- **Bigger is slower:** a bigger memory takes longer to determine the location (longer decode, longer wires).
- **Faster is more expensive:** memory technology (SRAM vs. DRAM vs. disk) trades cost per bit against speed.
- **Higher bandwidth is more expensive:** more banks, more ports, higher-frequency or wider interfaces.

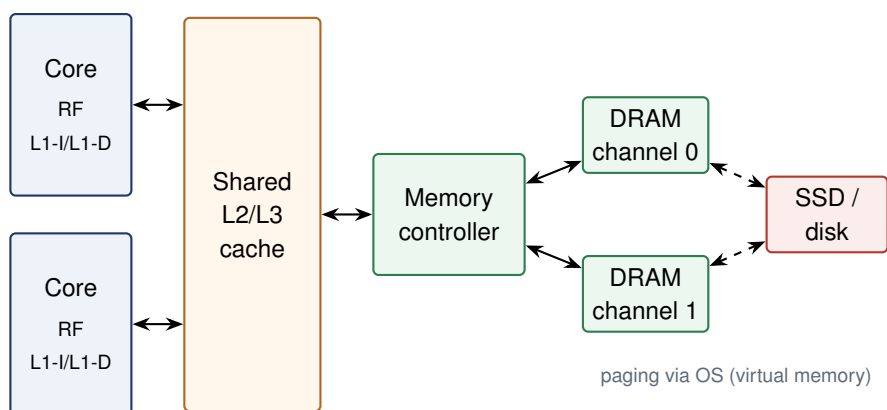


Figure 1.1: Memory in a modern system: private L1 caches, a shared last-level cache, memory controllers with several DRAM channels, and storage.

1.2 Memory technologies

1.2.1 DRAM

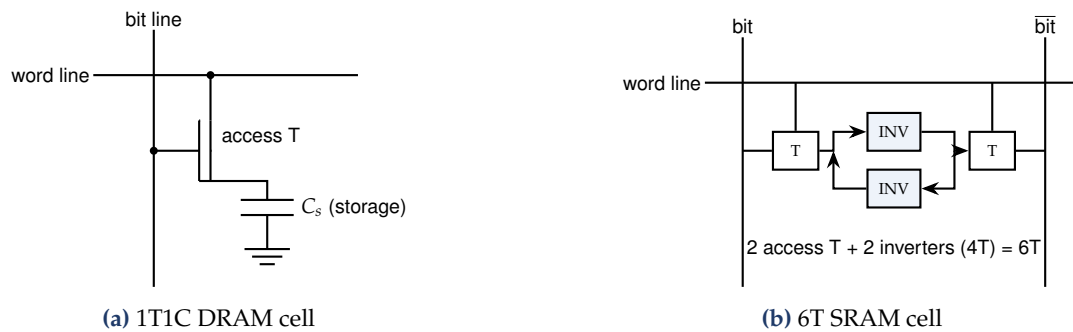


Figure 1.2: The two on-chip memory technologies.

A DRAM cell stores a bit as *charge on a capacitor* (charged = 1, discharged = 0), accessed through one transistor. Properties:

- The capacitor leaks through the RC path: DRAM cells lose charge over time and must be **refreshed** (read and rewritten), typically every 64 ms.
- Reading is **destructive**: the tiny cell charge is shared with the bit line and detected by a sense amplifier, which then restores it.
- Dense (1 transistor + 1 capacitor per bit), cheap per bit, but slow; DRAM is built in a special process that is hard to merge with logic.

1.2.2 SRAM

An SRAM cell stores a bit in two cross-coupled inverters (a feedback loop) plus two access transistors: 6 transistors per bit. The value is retained as long as power is supplied; no refresh. It is fast and built in the same process as logic, but about $6\times$ larger per bit.

1.2.3 Bank organization and access

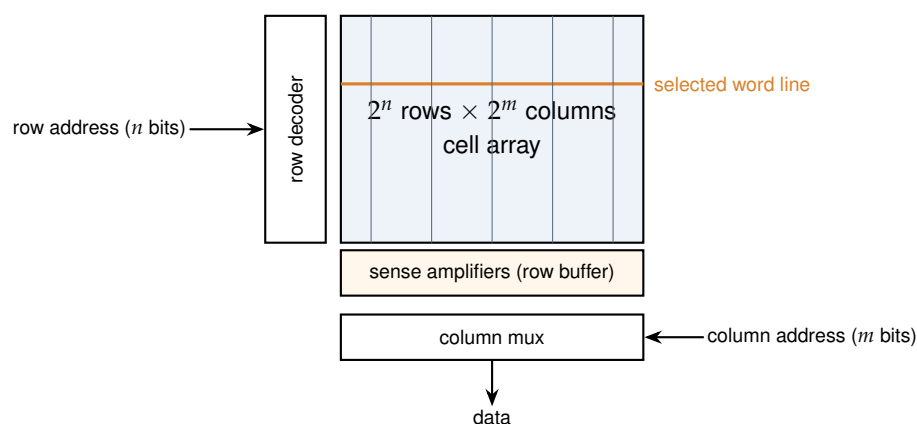


Figure 1.3: A memory bank: the row address selects a word line; all cells of the row drive their bit lines; sense amplifiers detect the values; the column address selects the requested bits.

A bank is a 2-D array of cells. An access proceeds in two steps:

1. **Decode the row address** and drive the word line: *every* cell in the row connects to its bit line.
2. **Sense** the bit lines (in DRAM: amplify and latch the whole row into the row buffer; in SRAM: differential sensing of bit/ $\overline{\text{bit}}$), then the **column address** selects the requested bits.

In DRAM the row stays latched in the row buffer, so a later access to the same row needs only the column step (a *row-buffer hit*) — the basis of DRAM page mode (Chapter 3).

Table 1.1: DRAM versus SRAM.

	DRAM	SRAM
Cell	1 transistor + 1 capacitor	6 transistors
Density / cost per bit	high / low	low / high ($\approx 6\text{--}10\times$)
Access time	slower ($\approx 10\text{--}50$ ns)	faster ($< 1\text{--}5$ ns)
Refresh	needed (every 64 ms)	not needed
Read	destructive (restore needed)	non-destructive
Process	special DRAM process	same as logic
Use	main memory	register files, caches, scratchpads

Worked Example 1.1: Cost of DRAM refresh

A DRAM chip with 8 banks of 65 536 rows must refresh every row once per 64 ms. DDR4 issues a REF command every $t_{\text{REFI}} = 7.8 \mu\text{s}$ ($64 \text{ ms} / 8192$); each refresh blocks the rank for $t_{\text{RFC}} \approx 350$ ns (for an 8 Gb chip). Fraction of time unavailable = $350 \text{ ns} / 7.8 \mu\text{s} = 4.5\%$. For 32 Gb chips, $t_{\text{RFC}} \approx 550 \text{ ns} \text{--} 1 \mu\text{s}$, i.e. 7–13%: refresh overhead grows with density, one facet of the DRAM scaling problem.

1.2.4 The practical limitation

Bigger is slower		
SRAM	512 bytes	sub-nanosecond
SRAM	KByte–MByte	$\sim$ nanoseconds
DRAM	Gigabytes	~ 50 nanoseconds
Hard disk	Terabytes	~ 10 milliseconds
Faster is more expensive (dollars and chip area)		
SRAM	$< \$10$ per megabyte	
DRAM	$< \$1$ per megabyte	
Hard disk	$< \$1$ per gigabyte	

These sample values scale with time. Other technologies have their place as well: flash memory, phase-change memory (PCM), MRAM, RRAM.

1.3 The memory hierarchy

We want both fast and large, but we cannot achieve both with a single level of memory. **Idea:** have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor) and ensure most of the data the processor needs is kept in the fast(er) level(s).

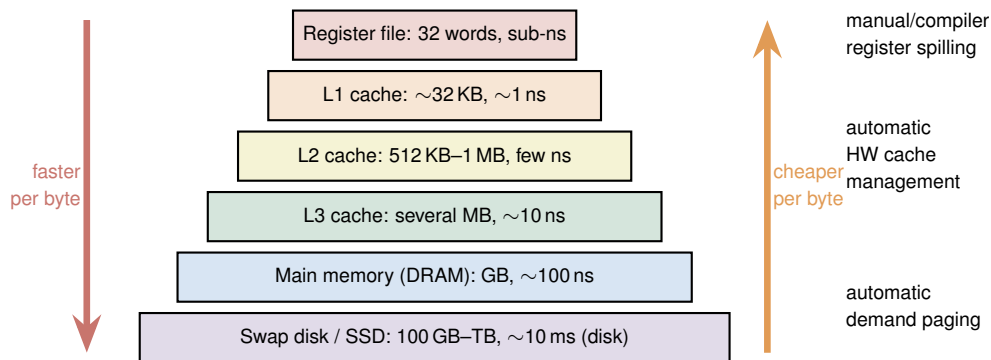


Figure 1.4: The modern memory hierarchy. With good locality of reference, memory appears as fast as the top level and as large as the bottom level.

Fundamental trade-off: fast memory is small, large memory is slow. The hierarchy moves what you use to the top (fast, small) and backs up everything at the bottom (big but slow). Its four figures of merit are latency, cost, size and bandwidth.

1.4 Locality of reference

1.4.1 Locality in everyday life

One's recent past is a very good predictor of one's near future.

- **Temporal locality:** if you just did something, it is very likely that you will do the same thing again soon — since you are here today, there is a good chance you will be here again and again regularly.
- **Spatial locality:** if you did something, it is very likely you will do something similar/related (in space) — every time I find you in this room, you are probably sitting close to the same people.

1.4.2 Memory locality

A “typical” program has a lot of locality in memory references; typical programs are composed of “loops”.

- **Temporal:** a program tends to reference the same memory location many times, and all within a small window of time.
- **Spatial:** a program tends to reference a cluster of memory locations at a time. The most notable examples: (1) instruction memory references, (2) array/data-structure references.

Worked Example 1.2: Locality in a simple loop

```
int sum = 0;
for (int i = 0; i < 1024; i++)
    sum += a[i];
```

Instructions: the 4–5 loop instructions are fetched 1024 times — strong temporal locality — and are contiguous — spatial locality. **Data:** `sum` and `i` are accessed every iteration (temporal, usually in registers); `a[i]` is accessed sequentially (spatial). With 64-byte cache blocks and

4-byte integers, each block brings 16 elements: only 1 in 16 accesses to a misses (hit rate 93.75%), even though no element of a is ever reused.

Traversing a 1024×1024 row-major matrix *column by column* destroys spatial locality: consecutive accesses are 4 KB apart, each touching a different block, so (if the cache cannot hold 1024 blocks for the full column sweep) nearly every access misses. Loop interchange restores the 1-in-16 miss ratio.

1.4.3 Exploiting locality

Temporal locality. Idea: store recently accessed data in an automatically managed fast memory (called a **cache**). Anticipation: the data will be accessed again soon. This is what Maurice Wilkes had in mind (“Slave memories and dynamic storage allocation,” IEEE Trans. on Electronic Computers, 1965): “The use is discussed of a fast core memory of, say 32 000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.”

Spatial locality. Idea: store addresses adjacent to the recently accessed one in the automatically managed fast memory. Logically divide memory into equal-size **blocks**; fetch the accessed block in its entirety into the cache. Anticipation: nearby data will be accessed soon. This is what IBM 360/85 implemented (16 KB cache with 64-byte blocks; Liptay, “Structural aspects of the System/360 Model 85 II: the cache,” IBM Systems Journal, 1968).

The bookshelf analogy. Book in your hand; desk; bookshelf; boxes at home; boxes in storage. Recently used books tend to stay on the desk (computer-architecture books, books for the classes you are currently taking) — until the desk gets full. Adjacent books on the shelf are needed around the same time — if you have organized/categorized your books well on the shelf.

1.5 Caching in a pipelined design

The cache needs to be tightly integrated into the pipeline: ideally, an access completes in one cycle so that dependent operations do not stall. A high-frequency pipeline cannot make the cache large, but we want a large cache *and* a pipelined design. **Idea:** a cache hierarchy — a small, fast Level 1 cache in the pipeline, backed by a larger Level 2 cache, backed by main memory.

Manual versus automatic management.

- **Manual:** the programmer manages data movement across levels — too painful for programmers on substantial programs (“core” vs “drum” memory in the 1950s); still done in some embedded processors (on-chip scratchpad SRAM in lieu of a cache) and in GPUs (shared memory).
- **Automatic:** hardware manages data movement across levels, transparently to the programmer. The programmer’s life is easier: the average programmer does not need to know about it — you don’t need to know how big the cache is and how it works to write a “correct” program. (What if you want a “fast” program?)

1.6 Hierarchical latency analysis

For a given memory-hierarchy level i , it has a technology-intrinsic access time t_i . The *perceived* access time T_i is longer than t_i . Except for the outermost level, when looking for a given address there is

- a chance (hit rate h_i) that you “hit” and the access time is t_i ;
- a chance (miss rate m_i) that you “miss” and the access time is $t_i + T_{i+1}$;
- $h_i + m_i = 1$.

Thus

$$T_i = h_i t_i + m_i (t_i + T_{i+1}) = t_i + m_i T_{i+1}. \quad (1.1)$$

Here h_i and m_i are the hit and miss rates of just the references that missed at L_{i-1} (*local* rates).

Design considerations. The goal is to achieve the desired T_1 within the allowed cost; $T_i \approx t_i$ is desirable.

- **Keep m_i low:** increasing capacity C_i lowers m_i , but beware of increasing t_i ; lower m_i by smarter management (replacement: anticipate what you don’t need; prefetching: anticipate what you will need).
- **Keep T_{i+1} low:** faster lower hierarchies, but beware of increasing cost; introduce intermediate hierarchies as a compromise.

Worked Example 1.3: Practical example: 90 nm Pentium 4, 3.6 GHz (the lecture’s numbers)

L1 D-cache: $C_1 = 16$ KB, $t_1 = 4$ cycles (int) / 9 cycles (fp). L2 D-cache: $C_2 = 1024$ KB, $t_2 = 18$ cycles (int) / 18 (fp). Main memory: $t_3 \approx 50$ ns ≈ 180 cycles. Using Equation (1.1) with integer latencies, $T_3 = t_3 = 180$:

m_1	m_2	$T_2 = t_2 + m_2 T_3$	$T_1 = t_1 + m_1 T_2$
0.10	0.10	$18 + 18 = 36$	$4 + 3.6 = 7.6$
0.01	0.01	$18 + 1.8 = 19.8$	$4 + 0.198 = 4.2$
0.05	0.01	19.8	$4 + 0.99 = 5.0$
0.01	0.50	$18 + 90 = 108$	$4 + 1.08 = 5.08$

Notice: the best-case latency is not 1 cycle (it is 4); the worst-case access latencies are 500+ cycles (TLB miss + L2 miss + DRAM row conflict); and a poor L2 ($m_2 = 0.5$) costs about as much as a $5\times$ worse L1 when the L1 is good.

Worked Example 1.4: Effect on CPI

A pipeline has base CPI 1.1; 35% of instructions are loads/stores; the L1 hit latency (4 cycles) is hidden by the pipeline except when a miss occurs. Using the second row ($m_1 = m_2 = 0.01$): the average *miss penalty* beyond L1 is $m_1 T_2 = 0.198$ cycles per access, so $\text{CPI} = 1.1 + 0.35 \times 0.198 = 1.17$. With the first row ($m_1 = m_2 = 0.1$): $1.1 + 0.35 \times 3.6 = 2.36$ — the same processor runs $2\times$ slower. Memory-hierarchy behaviour can matter more than the entire pipeline design of Unit 3.

Advanced Insight: The memory wall

Between 1980 and 2005, processor performance improved $\approx 50\%$ /year while DRAM latency improved $\approx 7\%$ /year; a DRAM access went from a few cycles to several hundred. Wulf and McKee (1995) called this the *memory wall*: eventually performance is limited entirely by memory. The response has been deeper hierarchies (L3, L4/eDRAM, HBM), much larger caches (the last-level cache occupies about half of many server dies; AMD's 3D V-Cache stacks 64 MB of SRAM on top of a CCD), prefetching, out-of-order execution with hundreds of in-flight loads to overlap misses, and processing-in-memory research that moves computation to the data.

1.7 Beyond SRAM and DRAM: other memory technologies

Table 1.2: Characteristics of memory and storage technologies (representative orders of magnitude).

Technology	Read latency	Write latency	Endurance (writes)	Volatile?
SRAM	< 1–5 ns	< 1–5 ns	unlimited	yes
DRAM	~ 15 –50 ns	~ 15 –50 ns	unlimited	yes (refresh)
NAND flash (SSD)	~ 25 –100 μ s	~ 200 μ s–1 ms	10^3 – 10^5	no
Phase-change memory (PCM)	~ 50 –100 ns	~ 150 –1000 ns	10^6 – 10^8	no
STT-MRAM	~ 5 –20 ns	~ 10 –30 ns	$> 10^{12}$	no
RRAM	~ 10 –100 ns	~ 50 –100 ns	10^6 – 10^{10}	no
Hard disk	~ 5 –10 ms	~ 5 –10 ms	unlimited	no

Non-volatile memories (NVMs) such as PCM, STT-MRAM and RRAM store data as a material state (crystalline vs. amorphous, magnetic orientation, filament resistance) rather than charge, so they need no refresh and can scale differently from DRAM. Their drawbacks are slower and more energetic writes and limited endurance, so they are used as an additional hierarchy level (e.g. a large, slower memory tier behind a DRAM cache), as embedded non-volatile memory in microcontrollers (MRAM replacing embedded flash), or in research on in-memory computing, where an array of resistive cells computes a matrix–vector product in place.

Worked Example 1.5: Hybrid DRAM + PCM main memory

A system has 16 GB of DRAM (60 ns) acting as a cache for 256 GB of PCM (read 150 ns, write 600 ns). For a workload whose DRAM-cache hit rate is 95% and in which 30% of PCM accesses are write-backs: average PCM access = $0.7 \times 150 + 0.3 \times 600 = 285$ ns, so the average main-memory latency is $60 + 0.05 \times 285 = 74$ ns — only about 24% worse than an all-DRAM system, with $16\times$ the capacity. With the endurance of 10^7 writes per cell and perfect wear levelling, writing 1 GB/s to 256 GB lasts $256 \times 10^9 \times 10^7 / 10^9$ s ≈ 81 years.

Chapter summary

- Ideal memory (fast, large, cheap, high bandwidth) is impossible; bigger is slower and faster is more expensive.
- DRAM: 1T1C, dense, destructive read, refresh; SRAM: 6T, fast, no refresh.
- A hierarchy of progressively larger, slower levels works because programs exhibit

temporal and spatial locality; caches exploit both automatically.

- Recursive latency: $T_i = t_i + m_i T_{i+1}$; keep m_i low without inflating t_i , and keep T_{i+1} low.

Exercises

E1.1 Compute T_1 for $t_1 = 2$, $t_2 = 12$, $t_3 = 200$ cycles, $m_1 = 0.05$, $m_2 = 0.2$.

Answer: $T_2 = 12 + 40 = 52$; $T_1 = 2 + 2.6 = 4.6$ cycles.

E1.2 An L3 with $t = 40$ cycles and local miss rate 0.3 is inserted between the L2 above and memory. New T_1 ?

Answer: $T_3 = 40 + 60 = 100$; $T_2 = 12 + 20 = 32$; $T_1 = 2 + 1.6 = 3.6$.

E1.3 How many transistors does a 32 KB SRAM data array need? How many capacitors and transistors a 1 Gb DRAM?

Answer: $32 \times 1024 \times 8 \times 6 \approx 1.57\text{M}$; 2^{30} of each.

E1.4 Explain why a 2D array traversal order changes the miss rate but not the number of instructions. Estimate both miss rates for 512×512 doubles, 64-byte blocks, a 32 KB cache.

Answer: Row order: 1/8 misses; column order: each column sweep touches 512 blocks (32 KB) — just the cache size, so conflicts make nearly every access miss.

E1.5 A DRAM refresh blocks a rank for 350 ns every $7.8 \mu\text{s}$. What is the probability that a random access must wait for refresh, and the average added latency?

Answer: 4.5%; average wait $0.045 \times 175 = 7.9$ ns.

2

Cache Memory

A cache is a small, fast memory that automatically keeps copies of recently used blocks of main memory. This chapter develops caches from first principles: how an address is split into tag, index and offset; direct-mapped, set-associative and fully associative organizations; replacement policies from LRU to Belady's optimum; write policies; sectorized, split and multi-level caches; and the classification and reduction of misses. Every concept is exercised on concrete address traces.

2.1 What is a cache?

- Generically, any structure that “memoizes” frequently used results to avoid repeating the long-latency operations required to reproduce the results from scratch (e.g. a web cache).
- Most commonly in the processor context: an automatically managed memory structure, based on SRAM, that memoizes in fast storage the frequently or recently accessed DRAM memory locations to avoid repeatedly paying the DRAM access latency.

2.1.1 Caching basics

- **Block (line):** the unit of storage in the cache. Memory is logically divided into cache blocks that map to locations in the cache.
- When data is referenced: **hit** — if in the cache, use the cached data instead of accessing memory; **miss** — if not, bring the block into the cache (maybe kicking something else out to do it).
- Important cache design decisions: **placement** — where and how to place/find a block in the cache; **replacement** — what data to remove to make room; **granularity of management** — large, small, uniform blocks; **write policy** — what do we do about writes; **instructions/data** — do we treat them separately?

2.1.2 Cache abstraction and metrics

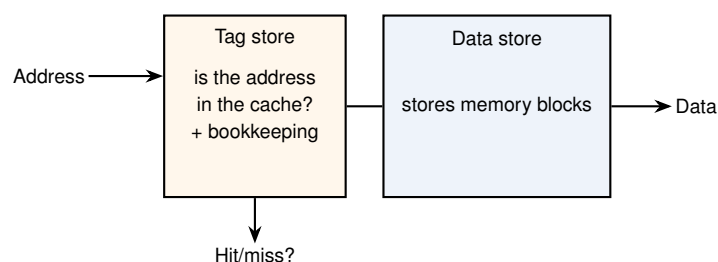


Figure 2.1: Cache abstraction: a tag store and a data store.

Definition 2.1: Cache metrics

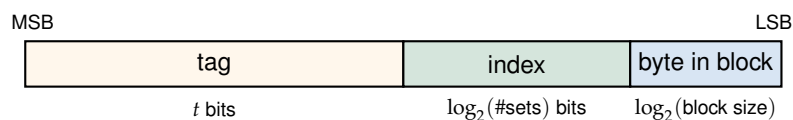
Cache hit rate = $(\# \text{ hits}) / (\# \text{ hits} + \# \text{ misses}) = (\# \text{ hits}) / (\# \text{ accesses})$. Average memory access time (AMAT) = $(\text{hit rate} \times \text{hit latency}) + (\text{miss rate} \times \text{miss latency})$, where the miss latency includes the hit-check time; equivalently

$$\text{AMAT} = t_{\text{hit}} + \text{miss rate} \times \text{miss penalty}.$$

Aside: can reducing AMAT reduce performance? Yes, in principle: if a policy lowers the miss rate but evicts blocks whose re-fetch is more expensive, or serializes misses that would otherwise overlap, execution time can increase. AMAT is a good first-order metric, not the final word.

2.2 Blocks and addressing

Memory is divided into fixed-size blocks; each block maps to a location in the cache, determined by the **index** bits in the address, used to index into the tag and data stores.



For a cache of capacity C bytes, block size B bytes and associativity A (blocks per set):

$$\begin{aligned} \# \text{blocks} &= C/B, & \# \text{sets} &= C/(B \cdot A), \\ \text{offset} &= \log_2 B, & \text{index} &= \log_2 \frac{C}{BA}, & \text{tag} &= n_{\text{addr}} - \text{index} - \text{offset}. \end{aligned}$$

Worked Example 2.1: Address breakdown for a 32 KB, 4-way cache

$C = 32 \text{ KB}$, $B = 64 \text{ B}$, $A = 4$, 32-bit physical addresses. Blocks = 512, sets = 128. Offset = 6 bits, index = 7 bits, tag = $32 - 7 - 6 = 19$ bits.

Address $0x1234ABCD = 0001\ 0010\ 0011\ 0100\ 1010\ 1011\ 1100\ 1101_2$: offset = low 6 bits = $001101_2 = 13$; index = bits [12:6] = $0101111_2 = 47$ ($0x2F$); tag = bits [31:13] = $0x091A5$. So the access looks in set 47, compares four tags with $0x091A5$, and on a hit returns byte 13 of the matching block.

Tag-store size: each of 512 lines needs 19 tag bits + valid + dirty = 21 bits, and each set needs replacement state (5 bits for true LRU of 4 ways): $512 \times 21 + 128 \times 5 = 11\ 392$ bits $\approx 1.39 \text{ KB}$ — 4.3% overhead on the 32 KB data store.

2.3 Placement: direct-mapped, set-associative, fully associative

2.3.1 Direct-mapped cache

Assume byte-addressable memory: 256 bytes, 8-byte blocks $\Rightarrow$ 32 blocks. Assume the cache has 64 bytes, 8 blocks. **Direct-mapped:** a block can go to only one location. Address = tag (2 bits) | index (3 bits) | byte-in-block (3 bits).

Addresses with the same index contend for the same location and cause **conflict misses**.

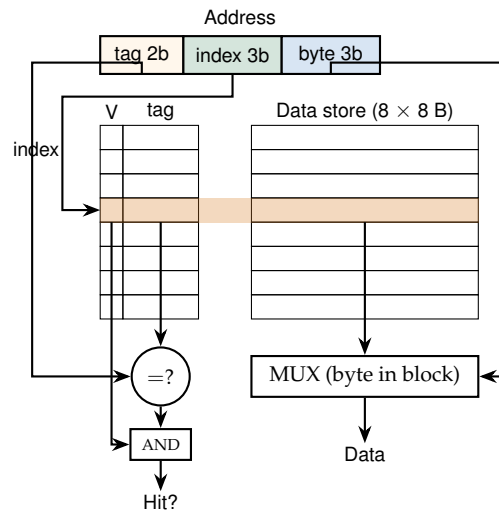


Figure 2.2: Direct-mapped cache: the index selects one entry; the stored tag is compared with the address tag; valid AND match = hit.

Worked Example 2.2: Conflict in a direct-mapped cache

Addresses 0 and 64 (00 000 000 and 01 000 000) have the same index (000) but different tags. Interleaved accesses A, B, A, B, ... (A = 0, B = 64) miss *every time*: 0% hit rate, although the cache has 7 other empty entries.

2.3.2 Set associativity

Addresses 0 and 64 always conflict in a direct-mapped cache. **Instead of having one column of 8, have 2 columns of 4 blocks:** a 2-way set-associative cache. Each set holds 2 blocks; the index (now 2 bits) selects a set; both tags are compared in parallel; a multiplexer selects the matching way. A multiplexer selects the matching way.

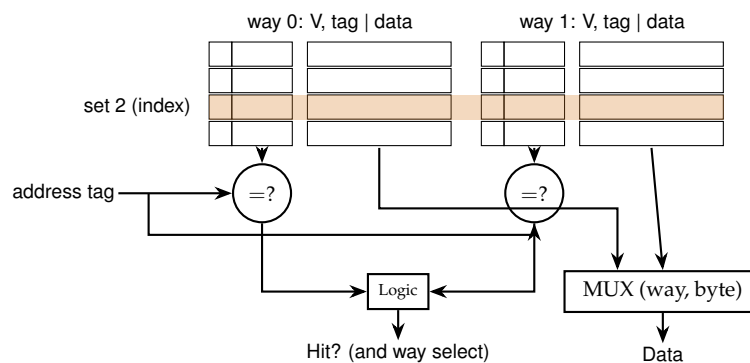


Figure 2.3: 2-way set-associative cache with 4 sets: two tag comparators work in parallel.

Higher associativity: a 4-way cache has 4 comparators and a wider data MUX — likelier to hit but slower. **Full associativity:** a block can be placed in any location in the cache: no index; every tag is compared (a content-addressable memory, CAM).

2.3.3 Trade-offs of associativity

Degree of associativity: how many blocks can map to the same index (or set)?

- **Higher associativity:** higher hit rate; slower cache access time (hit latency and data-access latency); more expensive hardware (more comparators).
- **Diminishing returns** from higher associativity.
- Smaller associativity: lower cost, lower hit latency — especially important for L1 caches.
- Is a power-of-2 associativity required? No (e.g. 12-way and 20-way L3 caches exist).

Worked Example 2.3: Three organizations on the same trace

Cache: 64 bytes, 16-byte blocks (4 blocks). Byte-address trace: 0x00, 0x40, 0x00, 0x48, 0x10, 0x20, 0x40, 0x80, 0x00, 0x10, i.e. block numbers 0, 4, 0, 4, 1, 2, 4, 8, 0, 1.

Block	0	4	0	4	1	2	4	8	0	1	Hits
Direct-mapped (4 sets)	M	M	M	M	M	M	H	M	M	H	2
2-way, LRU (2 sets)	M	M	H	H	M	M	H	M	M	H	4
Fully assoc., LRU	M	M	H	H	M	M	H	M	M	M	3
Fully assoc., OPT	M	M	H	H	M	M	H	M	H	H	4

Direct-mapped: blocks 0, 4 and 8 all map to set 0 and evict each other. **2-way:** set 0 holds two of {0, 2, 4, 8}; block 1 lives alone in set 1. **Fully associative LRU** does *worse* than 2-way here: when 8 arrives, LRU evicts 0 (needed next), then 0 evicts 1 (needed next). Higher associativity usually but not always helps; it is the replacement policy that went wrong.

2.4 Replacement policies

Which block in the set to replace on a cache miss? Any invalid block first. If all are valid, consult the replacement policy: random; FIFO; least recently used (how to implement?); not most recently used; least frequently used; least costly to re-fetch (why would memory accesses have different cost?); hybrid replacement policies; optimal replacement policy?

2.4.1 Implementing LRU

Idea: evict the least recently accessed block. Problem: we need to keep track of the access ordering of blocks.

- 2-way set-associative cache: one bit per set (which way was used last) suffices.
- 4-way: there are $4! = 24$ possible orderings of the 4 blocks; encoding the LRU order needs $\lceil \log_2 24 \rceil = 5$ bits per set; the logic to determine the LRU victim and update the order on each access is non-trivial.
- In general, $\lceil \log_2(A!) \rceil$ bits: 8-way needs 16 bits, 16-way 45 bits per set.

2.4.2 Approximations of LRU

Most modern processors do not implement “true LRU” (also called “perfect LRU”) in highly-associative caches. Why? True LRU is complex, and LRU is an approximation to predict locality anyway (i.e. not the best possible cache-management policy). Examples:

- **Not MRU** (not most recently used): track only the MRU block; evict any other.

- **Hierarchical LRU:** divide the 4-way set into two 2-way “groups”; track the MRU group and the MRU way in each group. On replacement, select a not-MRU block in one of the not-MRU groups. (This is the tree pseudo-LRU, $A - 1$ bits per set.)
- **Victim/next-victim replacement:** only keep track of the victim and the next victim.

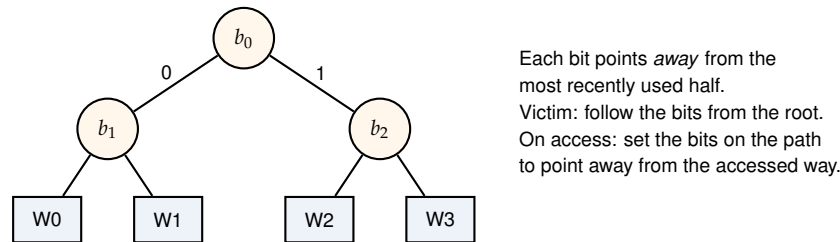


Figure 2.4: Tree pseudo-LRU (hierarchical LRU) for a 4-way set: 3 bits instead of 5.

Worked Example 2.4: Tree pseudo-LRU in action

Bits start $b_0b_1b_2 = 000$ (victim path: $b_0 = 0 \rightarrow \text{left}$, $b_1 = 0 \rightarrow W0$). Access sequence of ways after the set fills: W0, W2, W1, W3.

- Access W0: path root $\rightarrow$ left $\rightarrow$ W0; set $b_0 = 1$ (point right), $b_1 = 1$ (point to W1).
- Access W2: $b_0 = 0$ (point left), $b_2 = 1$ (point to W3).
- Access W1: $b_0 = 1$, $b_1 = 0$ (point to W0).
- Access W3: $b_0 = 0$, $b_2 = 0$ (point to W2).

Now $b_0b_1b_2 = 000$: victim = left, then W0. True LRU order (oldest first) is W0, W2, W1, W3: true LRU also picks W0. But after the fill sequence W0, W1, W2, W3 followed by one more access to W0, pseudo-LRU picks W2 whereas the true LRU block is W1 — the approximation is cheap and usually close, not exact.

2.4.3 LRU versus random

Example: a 4-way cache with cyclic references to A, B, C, D, E (all mapping to the same set): 0% hit rate with the LRU policy — each block is evicted just before it is needed again. **Set thrashing:** when the “program working set” in a set is larger than the set associativity. The random replacement policy is better when thrashing occurs: with random eviction, each access hits with a substantial probability. In practice the result depends on the workload; the average hit rates of LRU and random are similar. Best of both worlds: a hybrid of LRU and random — how to choose between the two? **Set sampling:** dedicate a few sets to each policy and let the winner govern the rest (the idea behind dynamic insertion policies such as DIP).

2.4.4 Optimal replacement (Belady’s OPT)

Replace the block that is going to be referenced furthest in the future by the program (Belady, “A study of replacement algorithms for a virtual-storage computer,” IBM Systems Journal, 1966). How do we implement this? We cannot — the future is unknown — but we can *simulate* it on a recorded trace to obtain an upper bound.

Worked Example 2.5: OPT on the cyclic trace

Trace A B C D E repeated four times (20 accesses), one 4-way set.

Policy	Hits	Pattern
LRU	0/20	every access misses
OPT	12/20	MMMM HHHM HHHM HHHM HHH

OPT keeps three of the five blocks resident and sacrifices one slot to the block whose next use is furthest away; after warm-up it misses once per four accesses.

Is OPT optimal for minimizing *miss rate*? Yes. Is it optimal for minimizing *execution time*? No: cache-miss latency/cost varies from block to block. Two reasons: remote versus local caches (or DRAM banks with open/closed rows) and miss overlapping (several misses serviced in parallel cost less than isolated ones).

2.4.5 Cache versus page replacement

Physical memory (DRAM) is a cache for disk, usually managed by system software via the virtual memory subsystem. Page replacement is similar to cache replacement; the page table is the “tag store” for the physical-memory data store. What is different? The required speed of access to the cache versus physical memory; the number of blocks in a cache versus physical memory; the “tolerable” amount of time to find a replacement candidate (disk versus memory access latency); and the role of hardware versus software (Chapter 4).

2.5 The tag-store entry and write policies

A tag-store entry contains: the **valid** bit, the **tag**, the **replacement-policy** bits, and (for write-back caches) the **dirty** bit.

2.5.1 When do we write modified data to the next level?

- **Write-through:** at the time the write happens.
 - + Simpler; all levels are up to date. *Consistency:* simpler cache coherence because there is no need to check lower-level caches.
 - More bandwidth-intensive; no coalescing of writes.
- **Write-back:** when the block is evicted.
 - + Can consolidate multiple writes to the same block before eviction; potentially saves bandwidth between cache levels and saves energy.
 - Needs a bit in the tag store indicating the block is “dirty/modified”.

2.5.2 Do we allocate a block on a write miss?

- **Allocate on write miss** (write-allocate): [+] can consolidate writes instead of writing each of them individually to the next level; [+] simpler, because write misses can be treated the same way as read misses; [–] requires (?) transfer of the whole cache block.
- **No-allocate:** [+] conserves cache space if the locality of writes is low (potentially better

cache hit rate).

Usual pairings: write-back + write-allocate (most caches); write-through + no-allocate (often with a write buffer).

Worked Example 2.6: Write traffic: write-through versus write-back

A core performs 10^9 stores/s of 8 bytes each. **Write-through**: 8 GB/s of write traffic to L2, regardless of locality. **Write-back**, 64-byte blocks: suppose each dirty block receives on average 6 stores before eviction and 2% of stores miss and allocate. Evicted dirty blocks per second = $10^9 / 6 = 1.67 \times 10^8$, each writing 64 B: 10.7 GB/s?! That is *more* — because with 8-byte stores and only 6 writes per 64-byte block, a write-back moves 64 bytes to save 48. With 32 stores per block (e.g. memset), $10^9 / 32 \times 64 = 2$ GB/s: a $4\times$ saving. Write-back wins when writes have spatial and temporal locality; write-through wins (and simplifies coherence) when they are sparse.

What if the processor writes an entire block over a small amount of time? Is there any need to bring the block into the cache from memory in the first place? Ditto for a *portion* of the block (e.g. 4 bytes out of 64).

2.5.3 Sected (sub-blocked) caches

Idea: divide a block into subblocks (or sectors), with separate valid and dirty bits for each sector. When is this useful? (Think writes...)



- ++ No need to transfer the entire cache block into the cache (a write simply validates and updates a subblock).
- ++ More freedom in transferring subblocks into the cache (a cache block does not need to be in the cache fully). How many subblocks do you transfer on a read?
- More complex design; may not exploit spatial locality fully when used for reads.

2.6 Instruction versus data caches; multi-level caches

Separate or unified?

- **Unified**: [+] dynamic sharing of cache space: no over-provisioning that might happen with static partitioning (i.e. split I and D caches); [–] instructions and data can thrash each other (no guaranteed space for either); [–] I and D are accessed in different places in the pipeline: where do we place the unified cache for fast access?
- **First-level caches are almost always split**, mainly for the last reason above. Second and higher levels are almost always unified.

Multi-level caching in a pipelined design.

- **First-level caches** (instruction and data): decisions very much affected by cycle time; small, lower associativity; tag store and data store accessed in *parallel*.

- **Second-level caches:** decisions need to balance hit rate and access latency; usually large and highly associative; latency not as important; tag store and data store accessed *serially* (saves energy: only the matching way is read).
- **Serial vs. parallel access of levels:** serial — the second-level cache is accessed only if the first level misses. The second level does not see the same accesses as the first: the first level acts as a filter (filters some temporal and spatial locality), so management policies are therefore different.

Worked Example 2.7: Local and global miss rates

1000 memory references: 40 miss in L1, 20 of those also miss in L2. L1 miss rate = 4%. L2 *local* miss rate = $20/40 = 50\%$; L2 *global* miss rate = $20/1000 = 2\%$. With $t_{L1} = 1$, $t_{L2} = 10$, memory 100 cycles: $AMAT = 1 + 0.04 \times (10 + 0.5 \times 100) = 1 + 0.04 \times 60 = 3.4$ cycles. A 50% local miss rate looks terrible but the L2 still cuts AMAT from $1 + 0.04 \times 100 = 5.0$ to 3.4 cycles.

2.7 Write buffers and inclusion policies

Write buffers. A write-through cache would stall on every store if it waited for the next level. A small **write buffer** (4–16 entries) holds pending writes so the processor can continue; writes to the same block are merged (*write combining*). A write-back cache uses a similar buffer for evicted dirty blocks so that the incoming block can be read first. Loads must check the write buffer (or wait for it to drain) to avoid reading stale data from the next level.

Worked Example 2.8: Sizing a write buffer

Stores arrive at 0.35 per cycle on average in bursts; the L2 accepts one write every 4 cycles (0.25 per cycle). Over a burst of 40 cycles at 0.6 stores/cycle, 24 stores arrive while 10 drain: the buffer must hold 14 entries to avoid stalls; with 8-entry merging of stores to the same block (say 3 stores per block on average), 14 stores occupy only about 5 entries. On average ($0.35 > 0.25$) a non-merging buffer would eventually fill: merging is what makes write-through with a small buffer workable.

Inclusion policies for multi-level caches.

- **Inclusive:** every block in L1 is also in L2. Coherence snoops need to check only the L2 (it is a superset), but the effective capacity is that of L2 alone, and evicting a block from L2 forces its eviction from L1 (“back-invalidation”).
- **Exclusive:** a block is in L1 or L2, never both (L2 acts as a victim cache for L1). Effective capacity = $L1 + L2$, at the cost of moving blocks on every L1 eviction.
- **Non-inclusive (NINE):** no guarantee either way; the common choice in recent designs.

Worked Example 2.9: Capacity under different inclusion policies

Per core: 32 KB L1D, 32 KB L1I and 1 MB L2; 8 cores share an 8 MB L3. Inclusive L3: unique capacity = 8 MB (all L2 contents are duplicated in L3, and 8 MB of L2 in total equals the whole L3 — an inclusive L3 is barely useful here). Exclusive/victim L3: $8 \times (1.06) + 8 \approx 16.5$ MB of distinct data. This arithmetic is why server processors that grew their private L2 caches to 1–2 MB moved from inclusive to non-inclusive last-level caches.

2.8 Cache parameters versus miss rate

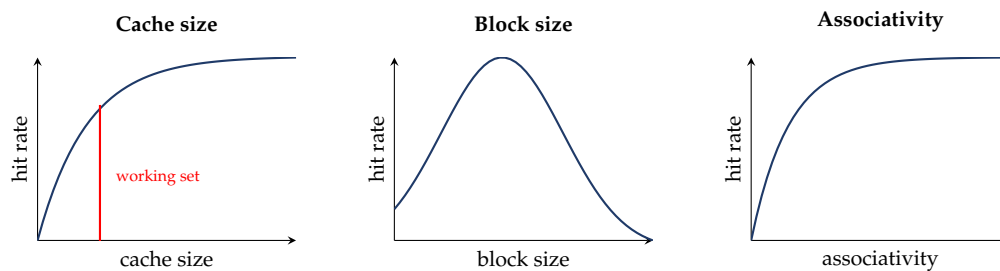


Figure 2.5: Qualitative effect of cache parameters on hit rate (after the lecture's plots).

Cache size — total data (not including tag) capacity. A bigger cache can exploit temporal locality better — but is not ALWAYS better. Too large a cache adversely affects hit and miss latency (smaller is faster $\Rightarrow$ bigger is slower; access time may degrade the critical path). Too small a cache does not exploit temporal locality well: useful data replaced often. **Working set:** the whole set of data the executing application references within a time interval; hit rate jumps once the cache holds the working set.

Block size — the data associated with an address tag (not necessarily the unit of transfer between hierarchies; sub-blocking: a block divided into multiple pieces each with a V bit, which can improve “write” performance). Too small blocks: do not exploit spatial locality well; larger tag overhead. Too large blocks: too few total blocks $\Rightarrow$ less temporal locality exploitation; waste of cache space and bandwidth/energy if spatial locality is not high.

Large blocks: critical word first and subblocking. Large cache blocks can take a long time to fill into the cache: fill the cache line *critical word first* and restart the cache access before the complete fill. Large cache blocks can waste bus bandwidth: divide a block into subblocks with separate valid bits.

Associativity — larger associativity: lower miss rate, less variation among programs, diminishing returns, higher hit latency. Smaller associativity: lower cost, lower hit latency (especially important for L1).

Worked Example 2.10: Choosing a block size with AMAT

A 16 KB cache, hit time 1 cycle; memory latency 40 cycles plus 1 cycle per 8 bytes transferred. Measured miss rates: 16 B: 4.0%; 32 B: 2.8%; 64 B: 2.2%; 128 B: 2.0%; 256 B: 2.3%.

Block	Miss penalty	Miss rate	AMAT
16	$40 + 2 = 42$	4.0%	2.68
32	$40 + 4 = 44$	2.8%	2.23
64	$40 + 8 = 48$	2.2%	2.06
128	$40 + 16 = 56$	2.0%	2.12
256	$40 + 32 = 72$	2.3%	2.66

64-byte blocks minimize AMAT: the growing transfer time eventually outweighs the falling

miss rate — and at 256 B the miss rate itself rises (too few blocks). This is why 64 B is the near-universal block size today.

2.9 Classification of cache misses (the 3 Cs)

- **Compulsory miss:** the first reference to an address (block) always results in a miss; subsequent references should hit unless the cache block is displaced for the reasons below.
- **Capacity miss:** the cache is too small to hold everything needed; defined as the misses that would occur even in a fully associative cache (with optimal replacement) of the same capacity.
- **Conflict miss:** defined as any miss that is neither a compulsory nor a capacity miss.

Worked Example 2.11: 3C classification of the earlier trace

For the 10-access trace (5 distinct blocks): compulsory misses = 5. Fully associative with OPT: 6 misses $\Rightarrow$ capacity = $6 - 5 = 1$. Direct-mapped: 8 misses $\Rightarrow$ conflict = $8 - 5 - 1 = 2$. 2-way LRU: 6 misses $\Rightarrow$ conflict = 0.

2.9.1 Reducing cache misses

- **Compulsory:** caching cannot help; *prefetching* can.
- **Conflict:** more associativity; other ways to get more associativity without making the cache associative: *victim cache*, *hashing* (of the index), software hints.
- **Capacity:** utilize cache space better (keep blocks that will be referenced); software management: divide the working set such that each “phase” fits in the cache (*tiling/blocking*).

2.9.2 Improving cache performance

Remember $AMAT = (\text{hit-rate} \times \text{hit-latency}) + (\text{miss-rate} \times \text{miss-latency})$. We can:

- **Reduce miss rate** (caveat: reducing the miss rate can reduce performance if more costly-to-refetch blocks are evicted): more associativity; alternatives/enhancements to associativity (victim caches, hashing, pseudo-associativity, skewed associativity); better replacement/insertion policies; software approaches.
- **Reduce miss latency/cost:** multi-level caches; critical word first; subblocking/sectoring; better replacement/insertion policies; non-blocking caches (multiple cache misses in parallel); multiple accesses per cycle; software approaches.
- **Reduce hit latency/cost.**

Worked Example 2.12: Tiling matrix multiplication

$C = A \times B$ for 512×512 doubles (2 MB per matrix) with a 32 KB L1. The naive ijk loop streams a whole column of B (512 blocks, one per row) for every element of C : almost every access to B misses ($\approx 512^3$ misses). Tiling into 32×32 sub-blocks (three tiles = 3×8 KB = 24 KB fit in L1): a B tile is loaded once (128 blocks) and each of its elements is then used 32 times, and every 64-byte block holds 8 doubles, so misses on B fall from about one per access

to about one per $32 \times 8 = 256$ accesses. The instruction count is unchanged; the run time typically improves 3–10 $\times$. Capacity misses are a *software* problem as much as a hardware one.

Advanced Insight: Beyond LRU: insertion policies and prefetching

Modern last-level caches do not simply insert new blocks at the MRU position. DIP inserts most new blocks at the LRU position (so streaming data cannot flush the cache) and uses set dueling to decide; RRIP (re-reference interval prediction) keeps 2-bit “predicted re-reference” values per block and is used in recent Intel LLCs. Hardware prefetchers (next-line, stride, stream, and spatial-pattern prefetchers) remove most compulsory misses for regular access patterns. A modern core’s L1 data cache is typically 32–64 KB, 8–12-way, 4–5 cycles; L2 0.5–2 MB, 12–16 cycles; LLC 2–4 MB per core, 40–60 cycles.

2.10 Cache-aware programming

The hardware manages the cache, but the programmer controls the access pattern. Three classic transformations:

Loop interchange. C stores arrays row-major. Traversing a 2-D array column-wise touches a new block on every access; interchanging the loops makes the inner loop sequential.

```
for (j = 0; j < N; j++)           // poor: stride N*8 bytes
    for (i = 0; i < N; i++) s += a[i][j];
for (i = 0; i < N; i++)           // good: stride 8 bytes (1 miss per 8 doubles)
    for (j = 0; j < N; j++) s += a[i][j];
```

Structure layout. Group fields that are used together; separate hot and cold fields; prefer “structure of arrays” when a loop touches one field of many objects.

Worked Example 2.13: Array of structures versus structure of arrays

struct particle { double x, y, z, vx, vy, vz, mass, charge; } is 64 bytes. A loop that updates only x for 10^6 particles touches 10^6 blocks (64 MB of traffic for 8 MB of useful data): 12.5% of each block is used. Storing x in its own array (structure of arrays) touches $10^6 \times 8/64 = 125\,000$ blocks: 8 $\times$ less traffic, and the loop vectorizes.

Blocking (tiling). Restructure loops so that a working set that fits in the cache is reused before moving on, as in the matrix-multiply example above.

Listing 2.1: Tiled matrix multiplication (tile size T chosen so 3 tiles fit in the cache).

```
for (ii = 0; ii < N; ii += T)
    for (kk = 0; kk < N; kk += T)
        for (jj = 0; jj < N; jj += T)
            for (i = ii; i < ii + T; i++)
                for (k = kk; k < kk + T; k++) {
                    double r = A[i][k];
                    for (j = jj; j < jj + T; j++)
```

```
C[i][j] += r * B[k][j];
}
```

Worked Example 2.14: Choosing the tile size

For a 32 KB L1 and doubles: three $T \times T$ tiles need $3 \times 8T^2 \leq 32768$ bytes $\Rightarrow T \leq 36.9$; choose $T = 32$ (24 KB, leaving room for other data and reducing conflict misses). For a 1 MB L2 tile, $T \leq 209$, e.g. $T = 128$ or 192. High-performance libraries (BLAS) tile for every level of the hierarchy and for registers.

Chapter summary

- Address = tag | index | offset; sets = $C/(BA)$.
- Direct-mapped (fast, conflicts), set-associative (compromise), fully associative (no conflicts, expensive).
- Replacement: LRU ($\lceil \log_2 A \rceil$ bits), pseudo-LRU ($A - 1$ bits), random, Belady's OPT as the bound; LRU thrashes on cyclic patterns larger than the set.
- Writes: write-through vs write-back (dirty bit); write-allocate vs no-allocate; sectoring.
- Split L1, unified L2+; local vs global miss rates; $AMAT = t_{hit} + m \times \text{penalty}$.
- 3 Cs: compulsory, capacity, conflict; prefetching, larger/associative caches, victim caches and tiling attack them.

Exercises

E2.1 A 64 KB, 8-way cache with 64-byte blocks and 40-bit physical addresses: compute the offset, index and tag widths and the tag-store size (V, D, true LRU).

Answer: Sets = 128: offset 6, index 7, tag 27; per line 29 bits; LRU 16 bits/set: $1024 \times 29 + 128 \times 16 = 31744$ bits ≈ 3.9 KB.

E2.2 Simulate a 2-way, 4-set cache with 16-byte blocks and LRU on byte addresses 0, 32, 64, 0, 128, 32, 192, 0. Count hits.

Answer: Blocks 0,2,4,0,8,2,12,0; sets 0,2,0,0,0,2,0,0: hits on the 4th (0), 6th (2): 2 hits; the 8th (0) misses (evicted by 12).

E2.3 Compute AMAT for L1 (1 cycle, 5% miss), L2 (12 cycles, 30% local miss), memory 150 cycles. What L2 local miss rate gives $AMAT = 2.5$?

Answer: $1 + 0.05(12 + 45) = 3.85$; $1 + 0.05(12 + 150m) = 2.5 \Rightarrow m = 0.12$.

E2.4 Show that for a cyclic reference pattern over $A + 1$ blocks in one A -way set, LRU yields 0% hits while random replacement yields a non-zero hit rate.

E2.5 For the write-traffic example, at how many stores per dirty block does write-back break even with write-through?

Answer: 8 stores of 8 bytes per 64-byte block.

Main Memory: DRAM

Everything that misses in the last-level cache goes to main memory, built from DRAM. Main memory is a shared, critical resource whose capacity, bandwidth, latency, energy and reliability all constrain the whole system. This chapter surveys the trends that make DRAM a first-class design problem, dissects the DRAM subsystem from channel to cell, works out the latency of basic DRAM operations with real timing parameters, and studies how banks, channels and address mapping provide parallelism.

3.1 The main memory system

Main memory is a critical component of all computing systems: server, mobile, embedded, desktop, sensor. The main memory system must scale (in size, technology, efficiency, cost and management algorithms) to maintain performance growth and technology-scaling benefits.

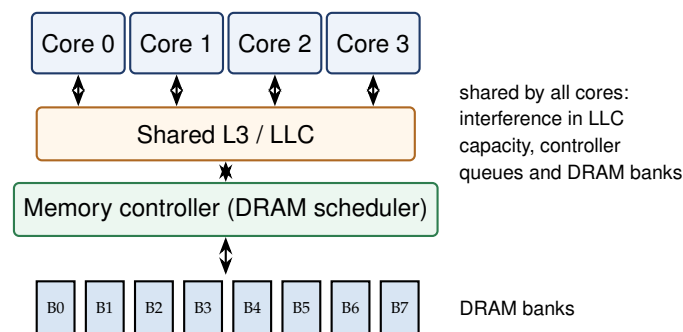


Figure 3.1: Memory system: a shared-resource view.

State of the main memory system. Recent technology, architecture and application trends lead to new requirements and exacerbate old ones. DRAM and memory controllers, as we know them today, are unlikely to satisfy all requirements on their own. Emerging non-volatile memory technologies (e.g. phase-change memory) enable new opportunities such as merging memory and storage. The main memory system needs to be rethought as a whole.

3.2 Major trends affecting main memory

- **Need for capacity, bandwidth and quality of service is increasing:** multi-core (more cores and agents), data-intensive applications, consolidation (cloud computing, GPUs, mobile, heterogeneity).
- **Main-memory energy/power is a key system design concern:** in servers a large fraction of system energy (often quoted as 40–50%) is spent in the off-chip memory hierarchy, and

DRAM consumes power even when idle (periodic refresh).

- **DRAM technology scaling is slowing:** scaling has historically provided higher capacity, lower cost per bit and lower energy per bit, but each new generation is harder.

3.2.1 The memory capacity gap

Core count has doubled roughly every 2 years, while DRAM DIMM capacity doubles roughly every 3 years (Lim et al., ISCA 2009). Consequently, memory capacity per core falls over time, and the trend is worse still for memory bandwidth per core, because the number of pins per socket grows slowly.

Worked Example 3.1: Capacity per core over a decade

Start: 16 cores with 64 GB (4 GB/core). After 10 years: cores $\times 2^{10/2} = 32 \times$ (512 cores); DIMM capacity $\times 2^{10/3} \approx 10.1 \times$ (645 GB with the same number of DIMMs). Capacity per core = $645/512 \approx 1.26$ GB — a $3.2 \times$ reduction.

3.2.2 The DRAM scaling problem and reliability

DRAM stores charge in a capacitor (charge-based memory). The capacitor must be large enough for reliable sensing, and the access transistor must leak little enough for adequate retention time. As cells shrink, both requirements become harder to meet: retention times vary more from cell to cell, cells become more sensitive to their neighbours' activity, and manufacturing variation grows. Field studies of large data-centre fleets (e.g. Schroeder et al., SIGMETRICS 2009; Meza et al., DSN 2015) found that memory errors are far more common than once assumed, that a small fraction of modules account for most errors, and that error rates depend on chip density and vintage.

The engineering responses are:

- **Error-correcting codes (ECC):** server DIMMs add 8 check bits to every 64 data bits (a 72-bit channel) implementing SEC-DED (single-error correction, double-error detection); DDR5 chips additionally include on-die ECC. Chipkill-style codes tolerate the failure of an entire chip.
- **Refresh management:** refreshing more often where needed, and targeted refresh of rows that experience heavy neighbouring activity, as specified in recent DRAM standards.
- **Testing and retirement:** memory controllers and operating systems track correctable errors and retire faulty pages.

Worked Example 3.2: SEC-DED overhead

A Hamming code that corrects one error in k data bits needs r check bits with $2^r \geq k + r + 1$; for $k = 64$, $r = 7$ ($2^7 = 128 \geq 72$). One extra overall-parity bit adds double-error detection: 8 bits, i.e. the (72,64) code. Storage overhead = $8/64 = 12.5\%$ — exactly one extra $\times 8$ chip per rank of eight.

3.2.3 Memory interference

In a multi-core system, applications running on different cores share the memory controller, channels and banks. One application's row-buffer-friendly streaming accesses can delay another application's requests (the controller prefers row hits), causing unfair slowdowns and unpre-

dictable performance. Fair and QoS-aware memory schedulers, bank and channel partitioning, and source throttling are architectural answers.

3.3 Fundamental concepts and organization



Figure 3.2: The DRAM subsystem hierarchy.

- **Channel:** an independent command/address and data bus driven by a memory controller (typically 64 data bits, 72 with ECC).
- **DIMM** (dual in-line memory module): a circuit board with DRAM chips on both sides.
- **Rank:** a set of chips that respond together to a command and together supply the 64-bit data bus (e.g. eight $\times$ 8 chips); a DIMM has one or two ranks (front and back).
- **Chip:** contains several **banks** (8 in DDR3, 16 in DDR4, 32 in DDR5) that can operate concurrently.
- **Bank:** a 2-D array of cells with a **row buffer** (sense amplifiers).

3.3.1 Page-mode DRAM and the row buffer

A DRAM bank is a 2-D array of cells: rows $\times$ columns. A “DRAM row” is also called a “DRAM page”; the sense amplifiers are also called the “row buffer”. Each address is a $\langle$ row, column $\rangle$ pair. Access to a “closed row”:

1. the **ACTIVATE** command opens the row (places it into the row buffer);
2. a **READ/WRITE** command reads/writes a column in the row buffer;
3. the **PRECHARGE** command closes the row and prepares the bank for the next access.

Access to an “open row”: no need for an ACTIVATE command.

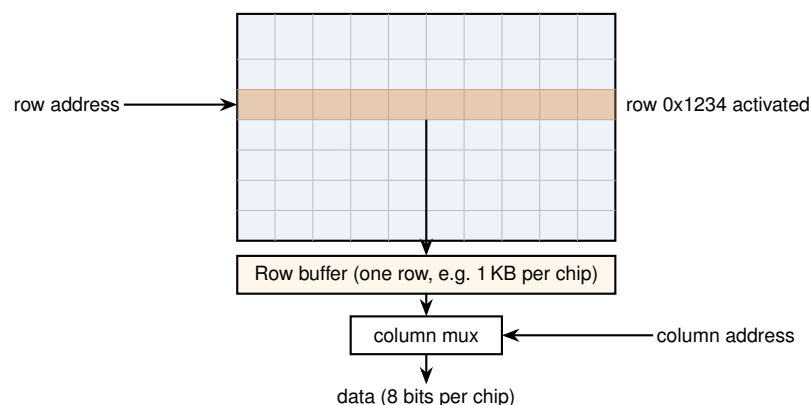


Figure 3.3: DRAM bank operation: **ACTIVATE** copies a whole row into the row buffer; **READ** selects columns from it; **PRECHARGE** writes the row back and closes it.

Worked Example 3.3: Row hits, row misses and row conflicts

A bank receives requests to (row, column): (0, 0), (0, 1), (0, 85), (1, 0).

- (0, 0): bank closed $\Rightarrow$ ACTIVATE row 0, READ col 0 — *row miss (empty)*.
- (0, 1), (0, 85): row 0 is open $\Rightarrow$ READ only — *row hits*.
- (1, 0): a different row is open $\Rightarrow$ PRECHARGE, ACTIVATE row 1, READ — *row conflict*, the slowest case.

3.3.2 A $128\text{M} \times 8\text{-bit}$ DRAM chip

A 1 Gb DRAM chip organized as $128\text{M} \times 8$ has 8 banks; each bank holds $2^{14} = 16\,384$ rows $\times$ $2^{10} = 1024$ columns $\times$ 8 bits. Check: $8 \times 2^{14} \times 2^{10} \times 8 = 2^{30}$ bits. One row of one chip is 1024×8 bits = 1 KB.

3.3.3 Rank and module

Eight such $\times 8$ chips side by side form a **rank** that supplies a 64-bit data bus. Every chip receives the same command and address; chip k supplies data bits $\langle 8k + 7 : 8k \rangle$. The rank's capacity is $8 \times 1\text{Gb} = 1\text{GB}$ and its effective row size is $8 \times 1\text{KB} = 8\text{KB}$. A 64-bit-wide DIMM with one rank is a single-rank module; a DIMM with chips on both sides usually has two ranks that share the bus and are selected by chip-select (CS) signals.

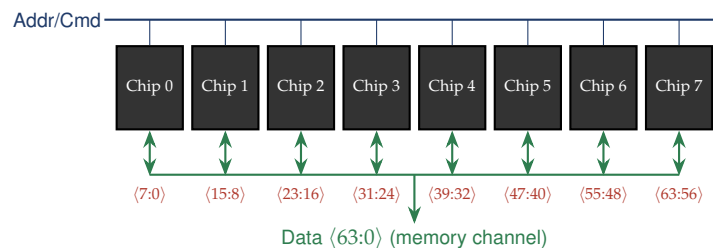


Figure 3.4: A rank of eight $\times 8$ chips sharing command/address and together providing a 64-bit data bus.

3.3.4 Multiple DIMMs and channels

Several DIMMs can share one channel (more capacity, but more electrical load and therefore lower bus frequency). Multiple **channels** provide more bandwidth: either 2 independent channels with 2 memory controllers, or 2 dependent/lock-step channels with 1 memory controller and a wide interface.

3.3.5 Transferring a cache block

A 64-byte cache block in physical memory maps to one row and 8 consecutive 8-byte columns of a rank (in the address mapping of Section 3.6). After ACTIVATE, one READ command returns a **burst** of 8 transfers of 64 bits: in each transfer every chip delivers 8 bits. A 64 B cache block takes 8 I/O transfers; during the process, 8 columns are read sequentially.

3.4 Latency of basic DRAM operation

Components of a memory read's latency:

1. CPU → controller transfer time;
2. controller latency: queuing and scheduling delay at the controller; the access is converted to basic commands;
3. controller → DRAM transfer time;
4. DRAM bank latency: simple CAS (column address strobe) if the row is "open", OR RAS (row address strobe) + CAS if the array is precharged, OR PRE + RAS + CAS (worst case);
5. DRAM → controller transfer time: bus latency (BL);
6. controller → CPU transfer time.

Definition 3.1: Key DRAM timing parameters

t_{RCD} : ACTIVATE to READ delay (row to column). t_{CL} (CAS latency): READ to first data. t_{RP} : PRECHARGE time. t_{RAS} : minimum time a row must stay open. t_{BURST} : time to transfer one burst (BL8 = 4 clock cycles on a DDR bus).

Worked Example 3.4: DDR4-3200 latencies and bandwidth

DDR4-3200 CL22-22-22: bus clock 1600 MHz ($t_{\text{CK}} = 0.625$ ns), data rate 3200 MT/s. $t_{\text{CL}} = t_{\text{RCD}} = t_{\text{RP}} = 22 \times 0.625 = 13.75$ ns; a BL8 burst takes $4 \times 0.625 = 2.5$ ns.

Case	Commands	DRAM latency to last data
Row hit	READ	$13.75 + 2.5 = 16.25$ ns
Row closed (empty)	ACT, READ	$13.75 + 13.75 + 2.5 = 30.0$ ns
Row conflict	PRE, ACT, READ	$3 \times 13.75 + 2.5 = 43.75$ ns

Peak channel bandwidth = $3200 \times 10^6 \times 8 \text{ B} = 25.6 \text{ GB/s}$. Adding controller queuing, on-chip interconnect and cache-miss handling, a typical end-to-end load-to-use latency from DRAM is 70–100 ns — 250–400 cycles at 3.5 GHz. Note how little DRAM latency has improved: the CAS latency of DDR (2000) was also around 15 ns.

Worked Example 3.5: Energy per bit of a DRAM access

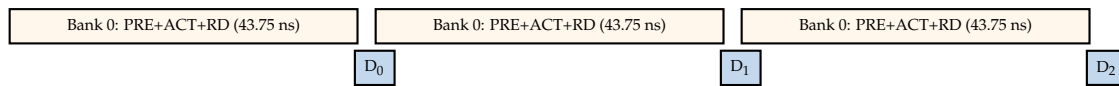
A rough DDR4 energy budget for reading one 64-byte block from a closed row: ACTIVATE + PRECHARGE ≈ 2 nJ (opening an 8 KB row of which only 64 B are used), the READ burst on the chips ≈ 1 nJ, and I/O signalling over the channel ≈ 1.5 nJ: ≈ 4.5 nJ per block, or ≈ 9 pJ/bit. A row hit avoids the ACT/PRE energy: ≈ 2.5 nJ. Activation energy wasted on unused row bits is one reason row-buffer locality matters for power as well as for latency, and why low-power DRAMs (LPDDR) and HBM use smaller rows and shorter, lower-capacitance interfaces.

3.5 Multiple banks (interleaving) and channels

- **Multiple banks:** enable concurrent DRAM accesses. Bits in the address determine which bank an address resides in.

- **Multiple independent channels** serve the same purpose, but they are even better because they have separate data buses: increased bus bandwidth.
- Enabling more concurrency requires reducing **bank conflicts** and **channel conflicts**.
- How to select/randomize bank/channel indices in the address? Lower-order bits have more entropy; randomizing hash functions (XOR of different address bits).

No overlap (all accesses to one bank, different rows)



Overlapped (accesses to banks 0–3)

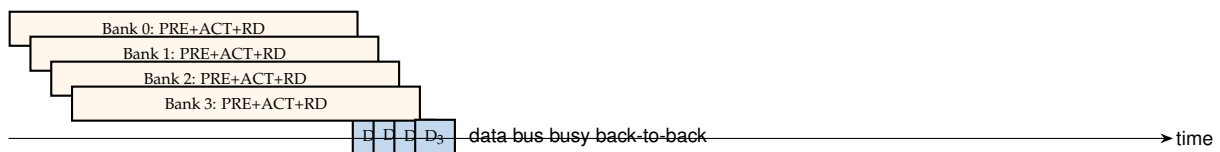


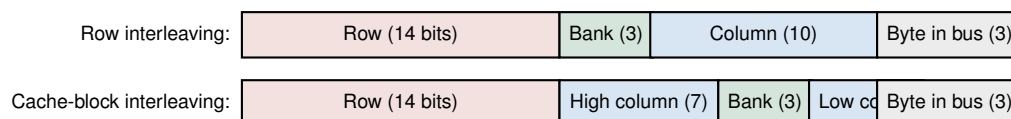
Figure 3.5: Bank interleaving: with accesses spread over banks, the long PRE/ACT/RD latencies overlap and the data bus delivers bursts back to back.

Worked Example 3.6: Bandwidth with and without bank parallelism

Sixteen 64 B requests, each a row conflict (43.75 ns, burst 2.5 ns). **One bank:** $16 \times 43.75 = 700 \text{ ns} \Rightarrow 1024 \text{ B}/700 \text{ ns} = 1.46 \text{ GB/s}$ (5.7% of peak). **Spread evenly over 16 banks:** latencies overlap; the limit becomes the data bus ($16 \times 2.5 = 40 \text{ ns}$) plus one latency: $\approx 43.75 + 15 \times 2.5 = 81 \text{ ns} \Rightarrow 12.6 \text{ GB/s}$. (Real DRAMs also limit the activation rate, e.g. at most 4 ACTs per $t_{\text{FAW}} \approx 20\text{--}30 \text{ ns}$, so the achievable figure is somewhat lower.) **Row hits:** 16 hits to one open row stream at the bus limit: $13.75 + 16 \times 2.5 = 53.75 \text{ ns} \Rightarrow 19 \text{ GB/s}$. Bank-level parallelism and row-buffer locality are the two levers of DRAM performance.

3.6 Address mapping

The memory controller maps each physical address to (channel, rank, bank, row, column). Consider a single channel with one rank: 8 banks, 2^{14} rows per bank, 2^{10} columns per row, 8 bytes per column (64-bit bus), i.e. 1 GB and 30 address bits.



- **Row interleaving:** consecutive rows of memory in consecutive banks. Accesses to consecutive cache blocks are serviced in a pipelined manner *within* one open row (row hits).
- **Cache-block interleaving:** consecutive cache-block addresses in consecutive banks; 64-byte cache blocks. Accesses to consecutive cache blocks can be serviced *in parallel* by different banks.

Worked Example 3.7: Decoding a physical address

Physical address $0x12345678$ (29 significant bits).

- **Row interleaving:** byte-in-bus = 0; column = bits [12:3] = 719; bank = bits [15:13] = 2; row = bits [29:16] = $0x1234 = 4660$.
- **Cache-block interleaving:** low column = bits [5:3] = 7; bank = bits [8:6] = 1; high column = bits [15:9] = 43; row = 4660.

A sequential sweep through memory changes the bank every 8 KB with row interleaving (long runs of row hits in one bank) but every 64 B with cache-block interleaving (all 8 banks busy in parallel).

3.6.1 Bank-mapping randomization

The DRAM controller can randomize the address mapping to banks so that bank conflicts are less likely: e.g. bank index = (3 bank bits) XOR (3 low-order row bits). Two addresses that differ only in the row — which would otherwise always conflict in the same bank — are then spread over different banks.

Worked Example 3.8: XOR bank hashing

For $0x12345678$ with row interleaving: bank bits = 2 = 010_2 , low row bits = $4660 \bmod 8 = 4 = 100_2$; hashed bank = $010 \oplus 100 = 110_2 = 6$. An address with the same bank bits but row 4661 maps to bank $010 \oplus 101 = 7$: the pair no longer conflicts. A program striding through memory with a stride equal to the row-interleaving period (64 KB here) would hit one bank only without hashing, and all 8 banks with it.

3.6.2 Address mapping with multiple channels

With 2 channels, one address bit selects the channel. Placing it just above the byte-in-bus bits (or the cache-block offset) spreads consecutive blocks across channels and doubles streaming bandwidth; placing it high in the address keeps each application's data in one channel (useful for partitioning). Where to place the channel bits involves the same trade-off as bank bits: parallelism versus locality.

3.6.3 Interaction of virtual-to-physical mapping

The operating system influences *which* DRAM bank or channel a virtual page maps to, because it chooses the physical page frame. With 4 KB pages, bits [11:0] are the page offset (fixed by the program); bits above are chosen by the OS. With row interleaving above, the bank bits [15:13] lie above the page offset: the OS can control the bank of every page (*page colouring*), e.g. to give each application its own banks and reduce interference. With cache-block interleaving, the bank bits [8:6] lie inside the page offset: every page is spread across all banks, and the OS has no control.

Advanced Insight: High-bandwidth memory and processing in memory

HBM stacks 4–16 DRAM dies on a base logic die connected by through-silicon vias, placed next to the processor on a silicon interposer: each stack offers a 1024-bit interface and 0.8–1.2 TB/s (HBM3/3E), versus 25–50 GB/s for a DDR channel. GPUs and AI accelerators rely on it. Because data movement dominates energy (a 64-bit DRAM access costs on the order

of 1–2 nJ, hundreds of times a 64-bit add), research and early products place simple compute units in or near the DRAM banks (processing-in-memory), performing bulk operations without moving the data across the channel.

3.7 The memory controller

The memory controller turns cache-block requests into DRAM commands while obeying dozens of timing constraints. It contains read and write request queues, a scheduler, the refresh logic and power management.

Scheduling. FCFS (first come, first served) is simple but ignores row-buffer state. **FR-FCFS** (first ready, first come first served) prioritizes requests that hit in an open row, then older requests; it maximizes row hits and throughput and is the baseline in practice. Writes are buffered and drained in batches, because switching the bus between reads and writes costs turnaround time (t_{WTR}).

Row-buffer management policy.

- **Open-page:** keep the row open after an access, betting that the next access hits it. Good for workloads with row locality; a miss to another row pays the full conflict latency.
- **Closed-page:** precharge immediately after each access, so the next access is always “row empty”. Good for random access patterns and many-core interleaved streams.
- Adaptive policies predict per bank which is better.

Worked Example 3.9: Open-page or closed-page?

Using the DDR4-3200 numbers (hit 16.25 ns, empty 30 ns, conflict 43.75 ns): with open-page, average latency = $h \times 16.25 + (1 - h) \times 43.75$; with closed-page it is always 30 ns (the precharge is hidden after each access). Break-even: $16.25h + 43.75(1 - h) = 30 \Rightarrow h = 0.5$. Row-hit rates above 50% favour open-page (typical for single-threaded streaming code); below 50% (typical when many cores interleave requests to the same banks) closed-page wins.

Power management. Idle ranks are put into power-down or self-refresh modes; the controller trades wake-up latency against background power, which can be a large fraction of DRAM power in lightly loaded servers.

Chapter summary

- Main memory faces growing capacity and bandwidth demand, energy constraints, slowing technology scaling and rising reliability challenges (handled with ECC, refresh management and page retirement).
- Hierarchy: channel $\rightarrow$ DIMM $\rightarrow$ rank $\rightarrow$ chip $\rightarrow$ bank $\rightarrow$ row/column; a rank of eight $\times$ 8 chips forms a 64-bit bus; a 64 B block is one 8-beat burst.
- Commands: ACTIVATE, READ/WRITE, PRECHARGE. Row hit $\approx t_{CL}$, empty $\approx t_{RCD} + t_{CL}$, conflict $\approx t_{RP} + t_{RCD} + t_{CL}$ (plus burst).

- Bank and channel parallelism overlap latencies; address mapping (row vs cache-block interleaving, XOR hashing, channel bits) decides how much parallelism and locality a stream gets.

Exercises

E3.1 A DDR5-4800 device has CL40, tRCD 39, tRP 39 (bus clock 2400 MHz). Compute row-hit, row-empty and row-conflict latencies (BL16 burst = 8 clocks).

Answer: $t_{CK} = 0.4167$ ns: hit $(40 + 8) \times 0.4167 = 20.0$ ns; empty $(39 + 40 + 8) \times 0.4167 = 36.3$ ns; conflict 52.5 ns.

E3.2 How many $\times 4$ chips form a 64-bit rank? With ECC?

Answer: 16; 18.

E3.3 Decode `0x0ABCDEF0` with both interleaving schemes of this chapter.

Answer: Row interleaving: row 2748 (`0xABC`), bank 6, column 990; block interleaving: low col 6, bank 3, high col 111, row 2748.

E3.4 A program streams through a 1 MB array. Compare the number of row activations under row interleaving and cache-block interleaving (8 KB rows).

Answer: Both need 128 row activations (1 MB / 8 KB per bank-row), but with row interleaving they occur one bank at a time, whereas with block interleaving consecutive blocks rotate over the 8 banks, so 8 rows are open and worked on in parallel.

E3.5 Why does the OS lose the ability to colour pages by bank when the bank bits are inside the page offset? What alternative mapping restores it while keeping some block-level parallelism?

4

Virtual Memory

Every program believes it owns a huge, private, contiguous memory that starts at address zero. None of that is true: physical memory is smaller, shared by many processes, and fragmented. **Virtual memory** reconciles the two views with one level of indirection — the page table — implemented jointly by hardware (address translation, TLBs, protection checks) and system software (page tables, page-fault handling, replacement). This chapter builds virtual memory from the basic mechanism to multi-level page tables, TLBs and the interaction with caches, with worked translations throughout.

4.1 Programmers' view and the abstraction

- **Programmer's view:** the programmer sees virtual memory and can assume the memory is “infinite”.
- **Reality:** physical memory size is much smaller than what the programmer assumes.
- **The system** (system software + hardware, cooperatively) maps virtual memory addresses to physical memory; it automatically manages the physical memory space transparently to the programmer.

Benefits of automatic management.

- + The programmer does not need to know the physical size of memory nor manage it: a small physical memory can appear as a huge one to the programmer, and life is easier for the programmer.
- More complex system software and architecture.

A classic example of the programmer/(micro)architect trade-off.

4.1.1 A system with physical memory only

Examples: most Cray machines, early PCs, nearly all embedded systems. The CPU's load or store addresses are used directly to access memory.

Difficulties of direct physical addressing.

- The programmer needs to manage the physical memory space: inconvenient and hard, and harder when you have multiple processes.
- It is difficult to support code and data relocation (where does a program go when loaded?).

- It is difficult to support multiple processes: protection and isolation between multiple processes; sharing of physical memory space.
- It is difficult to support data/code sharing across processes.

4.2 Virtual memory: the basic mechanism

Idea: give the programmer the illusion of a large address space while having a small physical memory, so that the programmer does not worry about managing physical memory. The programmer can assume he/she has an “infinite” amount of physical memory. Hardware and software cooperatively and automatically manage the physical memory space to provide the illusion; the illusion is maintained for each independent process.

Basic mechanism: indirection (in addressing). The address generated by each instruction in a program is a **virtual address**, i.e. not the physical address used to address main memory (called a “linear address” in x86). An **address translation** mechanism maps this address to a **physical address** (called a “real address” in x86). The translation mechanism can be implemented in hardware and software together.

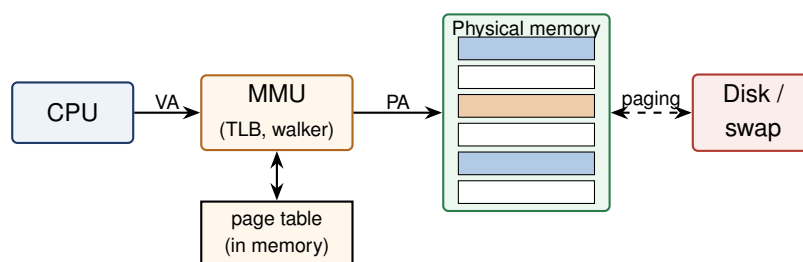


Figure 4.1: A page-based virtual-memory system: the MMU translates every virtual address using the process’s page table; pages not in memory live on disk.

4.2.1 Virtual pages and physical frames

The virtual address space is divided into **pages**; the physical address space is divided into **frames**. A virtual page is mapped to a physical frame, assuming the page is in memory, or to a location in disk otherwise. If an accessed virtual page is not in memory but on disk, the virtual memory system brings the page into a physical frame and adjusts the mapping — this is called **demand paging**. The **page table** is the table that stores the mapping of virtual pages to physical frames.

4.2.2 Physical memory as a cache

In other words, physical memory is a cache for pages stored on disk. In fact, it is a fully associative cache in modern systems (a virtual page can be mapped to any physical frame). Similar caching issues exist as we have covered earlier: **placement** (where and how to place/find a page in the cache?), **replacement** (what page to remove to make room?), **granularity of management** (large, small, uniform pages?), **write policy** (what do we do about writes? Write back!).

Table 4.1: Cache terminology versus virtual-memory terminology.

Cache	Virtual memory
block / line	page
cache miss	page fault
tag store	page table (+ TLB)
miss penalty ~100 cycles	page-fault penalty ~millions of cycles (disk)
managed by hardware	managed by OS software (with hardware assistance)

4.2.3 Supporting virtual memory

Virtual memory requires both hardware and software support. The page table is in memory and can be cached in special hardware structures called **translation lookaside buffers (TLBs)**. The hardware component is called the **MMU** (memory management unit): it includes the page-table base register(s), TLBs and page walkers. It is the job of the **software** to leverage the MMU to populate page tables, decide what to replace in physical memory, change the page table register on a context switch (to use the running thread's page table), and handle page faults and ensure correct mapping.

4.3 Page faults

A **page fault** is a “miss” in physical memory: the page table entry indicates that the virtual page is not in memory. Handled by an exception handler (the OS):

1. the processor signals the controller (the OS takes over);
2. the OS selects a victim page; if it is dirty, it writes it back to disk;
3. the OS reads the requested page from disk into the freed frame (DMA; meanwhile other processes run);
4. the OS updates the page table entry;
5. the faulting instruction is restarted (precise exceptions are essential), now hitting.

Worked Example 4.1: How rare must page faults be?

Memory access time 100 ns; page-fault service time 8 ms (disk). With page-fault probability p per access,

$$\text{EAT} = (1 - p) \times 100 \text{ ns} + p \times 8\,000\,000 \text{ ns} \approx 100 + 8 \times 10^6 p \text{ ns}.$$

For $p = 10^{-6}$: EAT = 108 ns (8% slowdown). To keep the slowdown below 10%: $8 \times 10^6 p < 10 \Rightarrow p < 1.25 \times 10^{-6}$ — fewer than one fault per 800 000 accesses. With an SSD (100 μ s service) the threshold relaxes to $p < 10^{-4}$. Paging only works because of locality.

4.4 Address translation

The page table is per process: each process has its own virtual address space, i.e. the full address space for each program. This simplifies memory allocation, sharing, linking and loading.

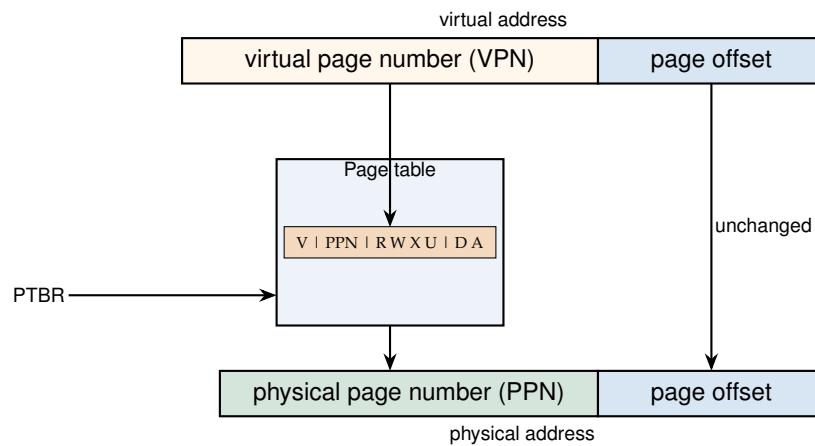


Figure 4.2: Address translation: the VPN indexes the page table (located by the page-table base register); the PTE supplies the PPN; the page offset passes through unchanged.

Worked Example 4.2: Translating an address

32-bit virtual addresses, 4 KB pages ($p = 12$ offset bits, 20-bit VPN), 30-bit physical addresses (18-bit PPN). Virtual address `0x00403ABC`: VPN = `0x00403`, offset = `0xABC`. Suppose $\text{PTE}[0x403] = \{V = 1, \text{PPN} = 0x1F2A, R = 1, W = 1\}$.

$$\text{PA} = (0x1F2A \ll 12) | 0xABC = 0x01F2AABC.$$

A store to `0x00403ABC` also checks $W = 1$ and sets the PTE's dirty bit. If $V = 0$, a page fault is raised and the OS loads the page.

Page-table size: 2^{20} entries $\times$ 4 bytes = 4 MB per process — the first of the “three major issues” below.

4.4.1 The page-table entry



Figure 4.3: RISC-V Sv32 page-table entry (34-bit physical addresses: 22-bit PPN).

- **V** valid (the page is in memory and the entry is meaningful);
- **R, W, X** read, write, execute permissions ($R = W = X = 0$ means “pointer to the next level of the page table”);
- **U** accessible in user mode; **G** global mapping (all address spaces);
- **A** accessed (reference bit, used by replacement), **D** dirty (page must be written back before eviction);
- **RSW** reserved for supervisor software; **PPN** physical page number.

x86 PTEs carry the same information: present, read/write, user/supervisor, accessed, dirty, plus cache-control bits.

4.5 Page replacement

Page replacement is similar to cache replacement, but with very different numbers: millions of cycles per miss allow software to run a replacement algorithm; there are millions of frames, so true LRU is impractical; the hardware provides only the reference (A) and dirty (D) bits.

Page replacement algorithms. FIFO; LRU (expensive to implement exactly); Belady's OPT (unrealizable, used as a bound); and **CLOCK** (second chance), an approximation of LRU used in most operating systems.

Definition 4.1: CLOCK algorithm

Frames are arranged in a circle with a "hand". Each frame has a reference bit, set by hardware on every access to the page. On a page fault, the hand advances: if the current frame's reference bit is 1, clear it (second chance) and move on; if it is 0, evict that page, load the new page there with its bit set to 1, and advance the hand.

Worked Example 4.3: FIFO, LRU and CLOCK on a reference string

Reference string 1 2 3 4 1 2 5 1 2 3 4 5.

Policy	3 frames	4 frames
FIFO	9 faults	10 faults
LRU	10 faults	8 faults
CLOCK	9 faults	—

FIFO with 4 frames incurs *more* faults than with 3: **Belady's anomaly**. LRU (a "stack algorithm") can never do this.

CLOCK trace with 3 frames (frame contents, reference bits, hand position after the access): after 1, 2, 3: frames [1,2,3], bits [1,1,1]. Reference 4: the hand sweeps clearing all bits, returns to frame 0 and evicts 1: [4,2,3], bits [1,0,0]. Reference 1 evicts 2 → [4,1,3]; 2 evicts 3 → [4,1,2]; 5 sweeps and evicts 4 → [5,1,2]; 1 and 2 hit (bits set); 3 evicts 1 after clearing 5 → [5,3,2]; 4 evicts 2 → [5,3,4]; 5 hits. Total 9 faults.

4.6 Page-size trade-offs

What is the granularity of management of physical memory? Large versus small pages. The trade-offs have analogies to large versus small cache blocks, and there are many of them:

- size of the page table (tag store);
- reach of the translation lookaside buffer;
- transfer size from disk to memory (waste of bandwidth?);
- waste of space within a page (internal fragmentation);
- waste of space within the entire physical memory (external fragmentation);
- granularity of access protection.

Worked Example 4.4: 4 KB versus 2 MB pages

48-bit virtual address space, 8-byte PTEs, a process using 1 GB.

	4 KB pages	2 MB pages
Pages to map 1 GB	262 144	512
Leaf PTE storage	2 MB	4 KB
Reach of a 64-entry TLB	256 KB	128 MB
Average internal fragmentation per region	2 KB	1 MB

Huge pages slash TLB misses for large-memory workloads (databases, VMs) but waste memory for small or sparse regions; operating systems therefore support several page sizes (4 KB, 2 MB, 1 GB on x86-64 and RISC-V Sv39/48).

4.7 Protection

Not every process is allowed to access every page — e.g. supervisor-level privilege may be needed to access system pages. **Idea:** store access-control information on a page basis in the process's page table, and enforce access control at the same time as translation. The virtual memory system thus serves two functions today: **address translation** (for the illusion of a large physical memory) and **access control** (protection).

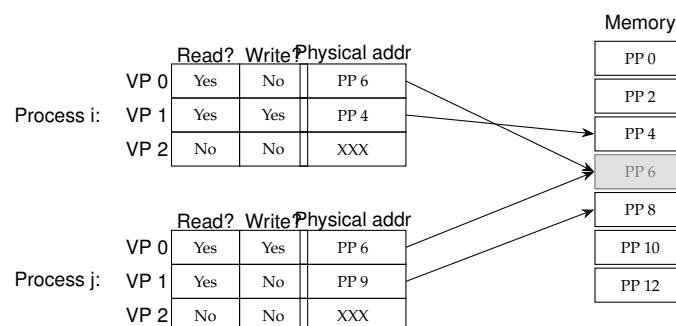


Figure 4.4: VM as a tool for memory-access protection: PTEs are extended with permission bits; both processes share physical page 6 (process *i* read-only, process *j* read-write).

The permission bits are checked on each access and during a page fault; if violated, an access-protection exception is generated. Privilege levels add a second dimension: x86 has four **rings** (0 = kernel ... 3 = applications) and combines page-directory and page-table permissions (user/supervisor, read-only/read-write); RISC-V has U, S and M modes, with the PTE's U bit controlling user access.

4.8 The three major issues

1. How large is the page table, and how do we store and access it?
2. How can we speed up translation and access-control checks?
3. When do we do the translation in relation to cache access?

(There are many other issues we will not cover in detail: what happens on a context switch? How can you handle multiple page sizes?)

4.8.1 Issue I: page-table size

Worked Example 4.5: The flat page table does not scale

64-bit VA and 40-bit PA, 4 KB pages: VPN = 52 bits, PPN = 28 bits; PA = PPN concatenated with the 12-bit offset. A flat table has 2^{52} entries $\times \sim 4$ bytes $\approx 16 \times 10^{15}$ bytes — and that is for just one process! And the process may not be using the entire VM space.

Where do we store it? In hardware? In physical memory (where is the PTBR?)? In virtual memory (where is the PTBR?)? How can we store it efficiently without requiring physical memory that can store all page tables? **Idea: multi-level page tables.** Only the first-level page table has to be in physical memory; the remaining levels are in virtual memory (but get cached in physical memory when accessed), and levels are allocated only for regions actually in use.

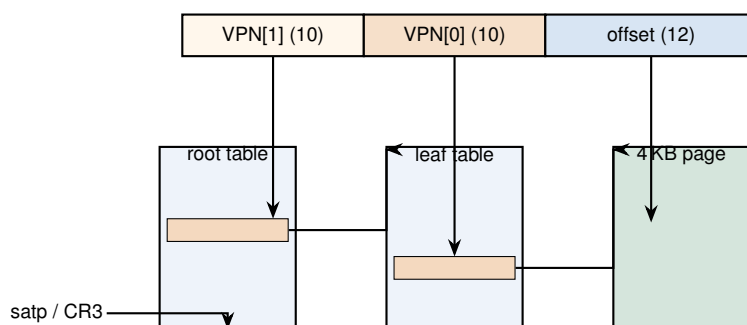


Figure 4.5: Two-level page table (x86 32-bit and RISC-V Sv32 both use 10 | 10 | 12): the root table is indexed by VPN[1], the selected leaf table by VPN[0].

Worked Example 4.6: Memory saved by a two-level table

A 32-bit process uses 4 MB of code and data at the bottom of its address space and 1 MB of stack at the top. Flat table: $2^{20} \times 4 \text{ B} = 4 \text{ MB}$. Two-level: the root table (4 KB) plus one leaf table per 4 MB region in use — one for the code/data, one for the stack: $3 \times 4 \text{ KB} = 12 \text{ KB}$, a $340\times$ saving.

Worked Example 4.7: An Sv39 page walk

RISC-V Sv39: 39-bit VA = VPN[2] (9) | VPN[1] (9) | VPN[0] (9) | offset (12); 8-byte PTEs, so each table is 512 entries = one 4 KB page. Translate VA `0x012345678`: VPN[2] = 0, VPN[1] = 145 (`0x91`), VPN[0] = 325 (`0x145`), offset `0x678`. With `satp.PPN = 0x800000`:

1. read PTE at $0x800000 \times 4096 + 0 \times 8 \rightarrow$ pointer to table at PPN `0x800001`;
2. read PTE at $0x800001 \times 4096 + 145 \times 8 \rightarrow$ pointer to PPN `0x800002`;
3. read PTE at $0x800002 \times 4096 + 325 \times 8 \rightarrow$ leaf, PPN `0x8ABCD`, R/W;
4. $PA = 0x8ABCD \times 4096 + 0x678 = 0x8ABCD678$.

Three memory accesses to translate one address. (If a level-1 PTE were a leaf, it would map a 2 MB “megapage”; a level-2 leaf maps a 1 GB “gigapage”.)

Page-table access. A page-table base register (CR3 in x86, `satp` in RISC-V) and a page-table limit register: if the VPN is out of bounds (exceeds the limit), the process did not allocate the virtual page $\Rightarrow$ access-control exception. The page-table base register is part of a process’s context,

just like the PC, status registers and general-purpose registers; it needs to be loaded when the process is context-switched in.

4.8.2 Issue II: speed of translation — the TLB

How fast is address translation? How can we make it fast? **Idea:** use a hardware structure that caches PTEs $\Rightarrow$ the **translation lookaside buffer**. What should be done on a TLB miss? What TLB entry to replace? Who handles the TLB miss — hardware or software? What should be done on a page fault? What virtual page to replace from physical memory? Who handles the page fault — hardware or software?

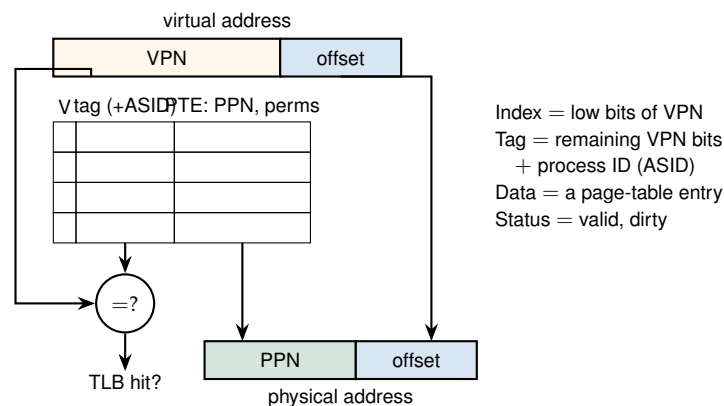


Figure 4.6: A TLB is essentially a cache of recent address translations; it avoids going to the page table on every reference. The usual cache design choices (placement, replacement policy, multi-level, etc.) apply here too.

Worked Example 4.8: Effective translation cost with a TLB

L1 TLB: 64 entries, 1-cycle lookup in parallel with the L1 cache; hit rate 99%. On a miss, a hardware walker performs 3 dependent memory accesses for Sv39; the PTEs usually hit in the L2 cache (12 cycles each), so a walk costs ≈ 36 cycles. Average translation overhead = $0.01 \times 36 = 0.36$ cycles per memory access. With 35% loads/stores, CPI increases by $0.35 \times 0.36 = 0.13$. A TLB hit rate of 95% would cost 0.63 CPI — which is why huge pages and second-level TLBs (e.g. 1536–2048 entries) are so important.

Handling TLB misses. The TLB is small; it cannot hold all PTEs. Some translations will inevitably miss in the TLB; we must access memory to find the appropriate PTE. This is called **walking** the page directory/table, and it carries a large performance penalty. Who handles TLB misses?

- **Hardware-managed** (e.g. x86, most RISC-V implementations, ARM): the hardware does the page walk, fetches the PTE and inserts it into the TLB; if the PTE is invalid, it raises a page fault. [+] No exception on a TLB miss, the instruction just stalls; independent instructions may continue; no extra instructions or data brought into the caches. [–] The page-directory/table organization is etched into the system: the OS has little flexibility in deciding these.
- **Software-managed** (e.g. MIPS, Alpha): the hardware raises an exception; the operating system does the page walk, fetches the PTE and inserts/replaces an entry in the TLB. [+] The OS can define the page-table organization; more sophisticated TLB replacement policies

are possible. [–] Need to generate an exception $\Rightarrow$ performance overhead due to pipeline flush, exception-handler execution and extra instructions brought into the caches.

4.8.3 Issue III: address translation and caching

When do we do the address translation? Before or after accessing the L1 cache? In other words, is the cache virtually addressed or physically addressed? Virtual versus physical cache. What are the issues with a virtually addressed cache?

Definition 4.2: Homonyms and synonyms

Homonym: the same virtual address can map to two different physical addresses (in two different processes). **Synonym:** different virtual addresses can map to the same physical address (shared pages, in the same or different processes).

- **Physically indexed, physically tagged (PIPT):** translate first, then access the cache — correct but the TLB lookup adds latency to every access.
- **Virtually indexed, virtually tagged (VIVT):** fastest, but homonyms require flushing the cache on a context switch (or tagging with ASIDs), and synonyms can create two copies of the same data that become inconsistent.
- **Virtually indexed, physically tagged (VIPT):** index the cache with page-offset bits (identical in VA and PA) while the TLB translates in parallel; compare the physical tag. If all index bits lie within the page offset, there are no synonym problems.

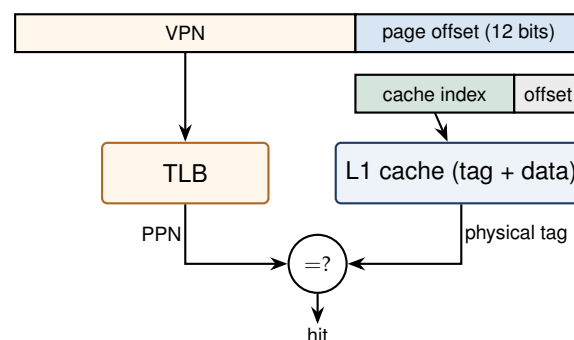


Figure 4.7: Virtually indexed, physically tagged L1: cache indexing and TLB lookup proceed in parallel because the index comes from the untranslated page offset.

Worked Example 4.9: How large can a VIPT L1 cache be?

Synonym-free VIPT requires index + block-offset bits $\leq$ page-offset bits, i.e. cache size/associativity $\leq$ page size. With 4 KB pages: a 32 KB cache must be at least 8-way; a 48 KB cache 12-way; a 64 KB cache 16-way. This is exactly why so many L1 data caches are 32 KB/8-way or 48 KB/12-way (with 4 KB pages), and why Apple's cores with 16 KB pages can use 128 KB/8-way L1 caches.

Solutions to the synonym problem.

- Limit the cache size to (page size $\times$ associativity) — get the index from the page offset.
- On a write to a block, search all possible indices that can contain the same physical block, and update/invalidate them — used in the Alpha 21264 and MIPS R10K.

- Restrict page placement in the OS: make sure $\text{index}(\text{VA}) = \text{index}(\text{PA})$ — called **page colouring**; used in many SPARC processors.

Advanced Insight: Virtual memory under virtualization

With a hypervisor, every guest-physical address must itself be translated to a host-physical address. Hardware *nested paging* (Intel EPT, AMD NPT, RISC-V H-extension G-stage) walks both tables: with 4-level guest and host tables, a TLB miss can require up to $(4 + 1) \times (4 + 1) - 1 = 24$ memory accesses. TLBs that cache the combined translation, page-walk caches for intermediate levels, and huge pages in both guest and host are what make virtualized workloads run close to native speed.

4.9 Virtual memory in action

The indirection of page tables is used for far more than extending memory:

- **Demand-zero pages:** a newly allocated heap or stack page initially maps to nothing (or to a shared read-only zero page); a physical frame is allocated only on first write.
- **Copy-on-write (fork):** parent and child share all pages read-only; the first write to a page faults and the OS copies just that page.
- **Shared libraries:** one physical copy of `libc`'s code is mapped into every process (read/execute only), possibly at different virtual addresses.
- **Memory-mapped files (mmap):** file pages are brought in on demand by page faults and written back when dirty, unifying file I/O with memory access.
- **Guard pages:** an unmapped page below each thread's stack turns a stack overflow into an immediate fault instead of silent corruption.

Worked Example 4.10: The cost of fork with copy-on-write

A 2 GB process (524 288 pages of 4 KB) forks and the child immediately calls `exec`. Copying all memory: 2 GB at 10 GB/s = 200 ms. With copy-on-write, `fork` copies only the page tables (about 4 MB of 8-byte leaf PTEs, well under a millisecond) and marks pages read-only; if the parent writes 1 000 pages before the child's `exec`, 1 000 faults of $\approx 2 \mu\text{s}$ each plus 4 MB of copying add only ≈ 2.4 ms.

4.9.1 Thrashing and the working set

If the pages a process actively uses — its **working set** — do not fit in its share of physical memory, it page-faults continuously: **thrashing**. Throughput collapses because the CPU waits for the disk most of the time. Operating systems estimate each process's working set (e.g. the pages referenced in the last Δ references, approximated with reference bits) and suspend processes rather than over-committing memory.

Worked Example 4.11: Onset of thrashing

A process makes 10^9 memory references per second. With 4 GB of frames its fault rate is 10^{-8} per reference (10 faults/s at 100 μs each on an SSD: 0.1% of time). Squeezed into 2 GB, its fault rate rises to 10^{-5} : 10^4 faults/s $\times$ 100 μs = 1 s of fault service per second of execution

— the process now runs at less than half speed, and on a disk (10 ms per fault) it would effectively stop. Small changes in available memory cause huge changes in performance once the working set no longer fits.

Chapter summary

- Virtual memory gives each process a large private address space via indirection: VA $\rightarrow$ page table $\rightarrow$ PA (or disk). Physical memory acts as a fully associative, write-back cache of pages.
- Page faults are handled by the OS; their huge cost requires very low fault rates.
- PTEs hold V, PPN, R/W/X/U permissions, accessed and dirty bits; CLOCK approximates LRU.
- Page size trades table size and TLB reach against fragmentation.
- Issue I (table size): multi-level tables (Sv32 10/10/12, Sv39 9/9/9/12). Issue II (speed): TLBs with hardware or software miss handling. Issue III (caching): VIPT with size/associativity $\leq$ page size avoids synonyms.

Exercises

E4.1 32-bit VA, 8 KB pages, 4-byte PTEs. How many bits of VPN and offset? Size of a flat page table?

Answer: VPN 19, offset 13; $2^{19} \times 4 = 2 \text{ MB}$.

E4.2 Translate VA `0x7FFFF123` with 4 KB pages if $\text{PTE}[0x7FFFF] = \{V = 1, \text{PPN} = 0x00042\}$.

Answer: `0x00042123`.

E4.3 Run FIFO and LRU with 3 frames on 7 0 1 2 0 3 0 4 2 3 0 3 2.

Answer: FIFO 10 faults, LRU 9 faults.

E4.4 An Sv48 system (four levels) has a TLB hit rate of 98% and walk accesses that cost 10 cycles each. Average translation cost per access?

Answer: $0.02 \times 40 = 0.8 \text{ cycles}$.

E4.5 What is the largest VIPT, synonym-free L1 for a 6-way cache with 4 KB pages? With 16 KB pages?

Answer: 24 KB; 96 KB.

E4.6 Explain why the page-table base register must be saved and restored on a context switch, and why the TLB must be flushed or tagged with an ASID.

5

The Memory System in Practice

The previous chapters studied caches, DRAM and virtual memory one at a time. Real systems combine them, overlap many accesses at once, and share them among several cores. This chapter follows a single load through the whole hierarchy, then examines the techniques that make modern memory systems fast — non-blocking caches and memory-level parallelism, victim caches, prefetching — and the problems that appear when several cores share memory: cache coherence, false sharing and memory consistency. It ends with the energy cost of data movement.

5.1 The journey of a load

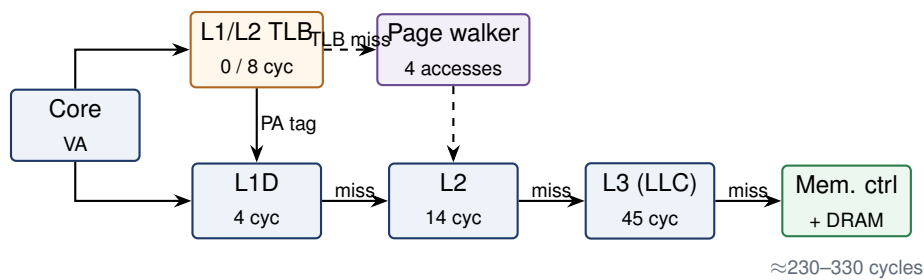


Figure 5.1: The path of a load through the memory system, with representative latencies.

Worked Example 5.1: Latency of one load in a modern system

A core at 3.5 GHz executes `ld a0, 0(a1)`. Representative latencies: L1 TLB 0 extra cycles (parallel with L1), L2 TLB 8 cycles, page walk (4 levels, PTEs in L2) 4×14 cycles; L1D 4 cycles, L2 14 cycles, L3 45 cycles, DRAM 80 ns ≈ 280 cycles beyond L3.

Scenario	Path	Load-to-use latency
Best case	L1 TLB hit, L1 hit	4 cycles
L1 miss	L1 TLB hit, L2 hit	$4 + 14 = 18$
L2 miss	L3 hit	$4 + 14 + 45 = 63$
LLC miss	DRAM (row hit)	$63 + \approx 230 = \approx 293$
LLC miss	DRAM (row conflict)	$63 + \approx 330 = \approx 393$
Worst	L2 TLB miss + walk + DRAM conflict	$8 + 56 + 393 = 457$

The spread is two orders of magnitude: from 4 to more than 450 cycles for the same instruction. The “practical example” of Chapter 1 already noted that the worst-case access latencies are 500+ cycles. Out-of-order cores hide part of this with independent work, but a dependent chain of misses (pointer chasing in a linked list) pays the full latency each time.

5.2 Non-blocking caches and memory-level parallelism

A **blocking** cache stops servicing requests while a miss is outstanding. A **non-blocking** (lockup-free) cache keeps servicing hits (*hit under miss*) and additional misses (*miss under miss*). Outstanding misses are tracked in **miss-status holding registers (MSHRs)**: each MSHR records the block address, which loads/stores are waiting for it (so that a second miss to the same block merges instead of issuing a new request), and where the returning data must go.

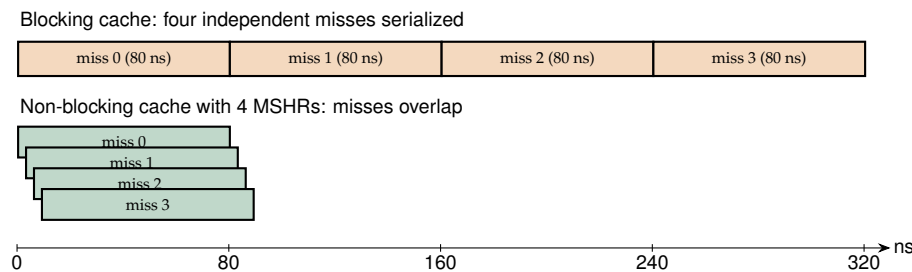


Figure 5.2: Memory-level parallelism: overlapping independent misses.

Definition 5.1: Memory-level parallelism (MLP)

The average number of long-latency memory accesses outstanding at the same time. With n independent misses overlapped, the effective miss penalty per miss approaches latency / n .

Worked Example 5.2: How many MSHRs does a core need? (Little's law)

Little's law: (requests in flight) = (throughput) $\times$ (latency). To sustain 20 GB/s of 64-byte block fills with an 80 ns memory latency:

$$\text{in flight} = \frac{20 \times 10^9 \text{ B/s}}{64 \text{ B}} \times 80 \times 10^{-9} \text{ s} = 312.5 \times 10^6 \times 80 \times 10^{-9} = 25.$$

A core with only 10 L1 MSHRs can sustain at most $10 \times 64 \text{ B}/80 \text{ ns} = 8 \text{ GB/s}$ from DRAM, regardless of channel bandwidth. This is why L2 caches have 32–64 MSHRs and hardware prefetchers issue requests from the L2: single-thread memory bandwidth is limited by outstanding-miss capacity, not by the DRAM pins.

5.3 Victim caches

A **victim cache** (Jouppi, 1990) is a small, fully associative buffer (4–16 blocks) that holds blocks recently evicted from a direct-mapped or low-associativity cache. On a miss in the main cache, the victim cache is checked; on a hit there, the blocks are swapped.

Worked Example 5.3: Rescuing a ping-pong conflict

A direct-mapped cache; addresses A and B map to the same set and are accessed alternately: A B A B ... Without a victim cache every access misses (Chapter 2). With a 4-entry victim cache: access A (miss, fill), B (miss; A moves to the victim cache), A (victim hit: swap A and B), B (victim hit: swap) ... After the first two compulsory misses, every access is a victim-cache hit costing 1–2 extra cycles instead of an L2 access. The victim cache adds associativity only where conflicts actually occur, at the cost of a few blocks of fully associative storage.

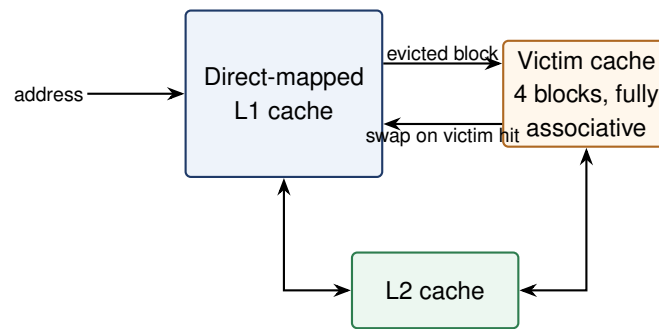


Figure 5.3: A victim cache sits beside the L1 and catches recently evicted blocks.

5.4 Prefetching

Prefetching fetches data before it is demanded, converting future misses into hits and attacking compulsory misses that no cache size can remove.

- **What** to prefetch: next-line (block $b + 1$ after a miss to b); stride (detect $a, a + s, a + 2s$ per load PC and fetch $a + ks$); stream buffers; spatial-pattern and correlation prefetchers for irregular accesses; software prefetch instructions.
- **When** to prefetch: early enough to hide the latency, late enough not to be evicted before use.
- **Where** to put prefetched data: in the cache (risk of pollution) or a separate buffer.

Definition 5.2: Prefetcher metrics

Coverage = (misses eliminated by prefetching)/(misses without prefetching). **Accuracy** = (useful prefetches)/(prefetches issued). **Timeliness**: the fraction of useful prefetches that arrive before the demand access.

Worked Example 5.4: Prefetch distance for a streaming loop

```
for (i = 0; i < N; i++) sum += a[i] * b[i];    // doubles, 64-byte blocks
```

Each iteration takes about 2 ns on the core; memory latency is 80 ns; one 64-byte block holds 8 doubles (8 iterations, 16 ns). To arrive on time, a prefetch must be issued $\lceil 80/16 \rceil = 5$ blocks ahead, i.e. `__builtin_prefetch(&a[i+40])` (and the same for `b`) once per block. With two streams and 5 blocks in flight each, 10 outstanding requests are needed — within the MSHR budget computed above. A hardware stride prefetcher learns the same distance automatically.

Inaccurate prefetching hurts

A prefetcher with 40% accuracy issues 2.5 prefetches per useful one: it wastes DRAM bandwidth and evicts useful blocks (cache pollution). In bandwidth-bound multi-core workloads, aggressive prefetching can slow the system down; real prefetchers throttle themselves using measured accuracy.

5.5 Caches in multi-core systems: coherence

When several cores cache the same memory block, a write by one core must eventually be seen by the others. Without additional mechanisms, private write-back caches would let cores read stale data.

Worked Example 5.5: The coherence problem

Cores P1 and P2 have private write-back caches; $x = 0$ in memory.

1. P1 reads x : P1's cache holds $x = 0$.
2. P2 reads x : P2's cache holds $x = 0$.
3. P1 writes $x = 1$: only P1's cache is updated (write-back).
4. P2 reads x : hits in its own cache and gets **0** — stale.

Definition 5.3: Cache coherence

A memory system is coherent if (1) a read by a core returns the value of the most recent write to that location (by any core), with writes eventually becoming visible, and (2) writes to the same location are *serialized*: all cores see them in the same order.

5.5.1 A snooping MSI protocol

In a bus-based system every cache controller observes (*snoops*) all bus transactions. Each cache block is in one of three states:

- **M (Modified)**: this cache has the only valid copy, and it is dirty;
- **S (Shared)**: one or more caches have clean, read-only copies;
- **I (Invalid)**: not present.

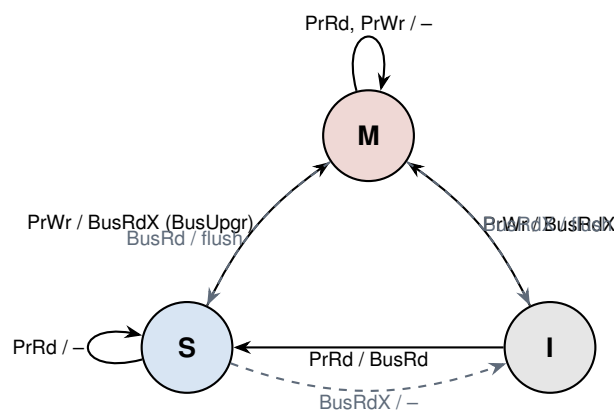


Figure 5.4: MSI protocol. Solid: transitions caused by the local processor (PrRd, PrWr) and the bus transaction they issue. Dashed: transitions caused by snooping another cache's bus transaction.

Worked Example 5.6: An MSI trace

Step	Action	P1 state	P2 state	Bus / data source
1	P1 reads x	S	I	BusRd, memory
2	P2 reads x	S	S	BusRd, memory
3	P1 writes x = 1	M	I	BusRdX (upgrade): P2 invalidates
4	P2 reads x	S	S	BusRd; P1 flushes 1 to memory and P2
5	P2 writes x = 2	I	M	BusRdX: P1 invalidates
6	P1 writes x = 3	M	I	BusRdX; P2 flushes 2, then P1 writes 3

At step 4, P2 now correctly reads 1. Every write by a core that does not hold the block in M costs a bus transaction that invalidates the other copies. Real protocols add an **E** (Exclusive, clean) state (MESI) so that a block read by only one core can be written without a bus transaction, and **O** (Owned) or **F** (Forward) states to supply data cache-to-cache. Large systems replace the snooping bus by a **directory** that tracks which caches hold each block.

5.5.2 False sharing

Coherence operates on whole blocks. If two cores repeatedly write *different* variables that happen to lie in the same 64-byte block, the block ping-pongs between their caches although no data is actually shared.

Worked Example 5.7: The cost of false sharing

```
struct { long c0; long c1; } counters;           // both in one 64-byte block
// thread 0: for (i=0;i<1e8;i++) counters.c0++;
// thread 1: for (i=0;i<1e8;i++) counters.c1++;
```

Every increment needs the block in M state; each ownership transfer between cores costs about 40–100 cycles. If the threads interleave, each of the 2×10^8 increments can incur a transfer: $\approx 2 \times 10^8 \times 60 = 1.2 \times 10^{10}$ cycles = 3.4 s at 3.5 GHz, versus about 0.06 s when each counter is in its own block (padding the struct to 64 bytes, e.g. `alignas(64)`). A $50\times$ slowdown from a layout decision.

5.5.3 Directory-based coherence

Snooping requires every cache to see every transaction — feasible for a handful of cores on a bus or ring, not for 64 cores on a mesh. A **directory** stores, for each memory block, its state and the set of caches holding it (e.g. a bit vector with one presence bit per core). Requests go to the block's *home* directory, which forwards invalidations only to the caches that actually hold copies.

Worked Example 5.8: Directory storage and traffic

64 cores, 64-byte blocks, full bit-vector directory: 64 presence bits + 2 state bits per block = 66 bits per 512-bit block, a 12.9% overhead if kept for all of memory. Real designs keep directory entries only for blocks that are cached (a “sparse” directory sized like the caches) or use coarse vectors. Traffic: a write to a block shared by 3 cores costs a request to the home, 3 invalidations and 3 acknowledgements — 7 messages — regardless of the total number of cores, whereas a broadcast would send 63 invalidations.

5.5.4 Coherence with I/O and accelerators

DMA engines and accelerators also read and write memory. If a device writes a buffer in DRAM while stale copies sit in a CPU cache, or the CPU's latest data are still dirty in its cache when the device reads DRAM, the two disagree. Systems either make I/O *coherent* (devices snoop the CPU caches, e.g. the Zynq's accelerator coherency port, or CXL.cache) or require software to *flush* dirty lines before a device reads and *invalidate* lines before the CPU reads device-written data. Forgetting this step is a classic embedded-systems bug: the code works in simulation (no caches) and fails on hardware.

5.6 Memory consistency in one paragraph

Coherence concerns a single location. **Consistency** concerns the order in which accesses to *different* locations become visible. Sequential consistency (SC) requires the result to be as if all operations of all cores were executed in some single order consistent with each program's order — intuitive, but it forbids common optimizations such as store buffers. RISC-V uses a relaxed model, **RVWMO**, in which loads and stores to different addresses may be reordered; programmers (and compilers, via C/C++ atomics) insert fence instructions or use acquire/release atomics (`lr/sc`, `amoswap.w.aq`) where order matters, e.g. between writing a data buffer and setting the flag that publishes it. x86 provides the stronger TSO model, which reorders only a store followed by a later load.

5.7 The energy of data movement

Table 5.1: Approximate energy per operation in a 7–16 nm-class technology (orders of magnitude; values vary with design and source).

Operation	Energy
64-bit integer add	≈ 0.1 pJ
64-bit floating-point FMA	≈ 1 – 2 pJ
64-bit read from a 32 KB L1 cache	≈ 5 – 10 pJ
64-bit read from a large (MB) on-chip SRAM	≈ 20 – 50 pJ
64-bit access to off-chip DRAM	≈ 1000 – 2000 pJ

Worked Example 5.9: Where does the energy go?

A kernel performs one FMA per 8 bytes loaded. If every operand comes from L1: $1.5 + 7 \approx 8.5$ pJ per FMA, 18% spent computing. If every operand comes from DRAM: $1.5 + 1500 \approx 1500$ pJ per FMA, 0.1% spent computing. At 1 TFLOP/s (5×10^{11} FMAs/s) the DRAM-bound version would draw 750 W. Caches, data reuse (tiling), compression and near-memory processing are therefore as much about energy as about speed.

Chapter summary

- A single load costs anywhere from 4 to 450+ cycles depending on TLB, cache and DRAM state.
- Non-blocking caches with MSHRs overlap misses; Little's law bounds per-core band-

width by $(\text{outstanding misses} \times \text{block size})/\text{latency}$.

- Victim caches absorb conflict misses; prefetchers attack compulsory misses and are judged by coverage, accuracy and timeliness.
- Multi-core caches need coherence (MSI/MESI snooping or directories); false sharing turns independent writes into coherence traffic; consistency models (RVWMO, TSO) define cross-location ordering and require fences.
- Moving data costs $10\text{--}1000\times$ more energy than computing on it.

Exercises

E5.1 With 80 ns latency and 64-byte blocks, how many outstanding misses are needed to use a 51.2 GB/s channel fully?

Answer: $51.2 \times 10^9 / 64 \times 80 \times 10^{-9} = 64$.

E5.2 Extend the MSI trace with step 7: P2 reads x. What happens?

Answer: P2: I→S via BusRd; P1 flushes 3 and goes M→S; both end in S.

E5.3 A stride prefetcher sees addresses 1000, 1064, 1128 from one load PC. What does it prefetch, and with a distance of 4?

Answer: Stride 64: 1192, then with distance 4 it requests $1128+4\times 64=1384$ ahead (and the blocks in between as it ramps up).

E5.4 Rewrite the false-sharing example so the two counters cannot share a block, and estimate the speedup.

Answer: Pad each counter to 64 bytes; $\approx 50\times$.

E5.5 In the journey-of-a-load table, compute the average load latency if 95% hit L1, 3% L2, 1.5% L3, and 0.5% go to DRAM (row hit), ignoring TLB misses.

Answer: $0.95 \cdot 4 + 0.03 \cdot 18 + 0.015 \cdot 63 + 0.005 \cdot 293 = 3.8 + 0.54 + 0.95 + 1.47 = 6.75$ cycles.

Solved Problems

Worked Example A.1: Designing an L1 data cache

Requirements: 48 KB, 64-byte blocks, 4 KB pages, VIPT without synonyms, 44-bit physical addresses, write-back with pseudo-LRU.

1. **Associativity:** size/associativity $\leq$ page size $\Rightarrow A \geq 48/4 = 12$: 12-way.
2. **Geometry:** blocks = 48 KB/64 = 768; sets = 768/12 = 64; offset 6 bits, index 6 bits (bits [11:6] — entirely inside the 12-bit page offset, as required).
3. **Tag:** 44 – 6 – 6 = 32 bits (physical tag from the TLB).
4. **Tag store:** per line 32 + V + D = 34 bits; tree pseudo-LRU for 12 ways needs 11 bits per set: $768 \times 34 + 64 \times 11 = 26\,816$ bits $\approx$ 3.3 KB (6.8% overhead).
5. **Access:** 12 tag comparators in parallel with the TLB lookup; a 12:1 way MUX.

Worked Example A.2: AMAT with TLB, two cache levels and DRAM

L1: 4 cycles, 4% miss rate. L2: 14 cycles, 25% local miss rate. Memory: 250 cycles. TLB: hits 99.5% (in parallel with L1); a miss costs a 30-cycle walk.

$$\begin{aligned} \text{AMAT} &= \underbrace{0.005 \times 30}_{\text{TLB}} + 4 + 0.04 \times (14 + 0.25 \times 250) \\ &= 0.15 + 4 + 0.04 \times 76.5 = 0.15 + 4 + 3.06 = 7.21 \text{ cycles.} \end{aligned}$$

Halving the L2 local miss rate (to 12.5%) gives $0.15 + 4 + 0.04 \times 45.25 = 5.96$ cycles (17% better); halving the L1 miss rate gives $0.15 + 4 + 0.02 \times 76.5 = 5.68$ (21% better). The L1 miss rate is the better lever here because every L1 miss pays the full L2 latency.

Worked Example A.3: Scheduling DRAM requests (FR-FCFS)

One bank, row buffer holding row 5. Queue (oldest first): R1 row 3, R2 row 5, R3 row 5, R4 row 3. Latencies: hit 15 ns, conflict 45 ns.

- **FCFS:** R1 (conflict 45), R2 (conflict 45), R3 (hit 15), R4 (conflict 45): 150 ns.
- **FR-FCFS** (row hits first, then oldest): R2 (hit 15), R3 (hit 15), R1 (conflict 45), R4 (hit 15): 90 ns — 40% less.

FR-FCFS maximizes row-buffer hits and is the baseline policy of real controllers, but it can starve an application with poor row locality — the motivation for fairness-aware schedulers.

Worked Example A.4: From virtual address to DRAM bank

Sv39, 4 KB pages; the TLB maps VA $0x012345678$ to PA $0x8ABCD678$ (the page walk of Chapter 4). The L1 is 32 KB/8-way/64 B: index = PA bits [11:6] = $(0x678 \gg 6) \& 63 = 25$ — available from the VA before translation completes. On a miss everywhere, the DRAM controller (cache-block interleaving, 8 banks: bank = bits [8:6]) selects bank $25 \& 7 = 1$; with row interleaving (bank = bits [15:13]) it would select bank $(0x8ABCD678 \gg 13) \& 7 = 6$ — a bit chosen by the OS's frame allocation, not by the program.

Worked Example A.5: Page-table memory for a sparse 64-bit process

Sv39, a process using three regions: 8 MB of code/data at the bottom, a 64 MB heap starting at 1 GB, and 1 MB of stack near the top of the 512 GB space. Each table is 4 KB (512 PTEs).

- Level 2 (root): 1 table.
- Level 1: one table per 1 GB region in use: 3 (bottom GB, second GB, top GB).
- Level 0: one table per 2 MB in use: code $8/2 = 4$, heap $64/2 = 32$, stack 1: 37 tables.

Total = $1 + 3 + 37 = 41$ tables = 164 KB, versus $2^{27} \times 8 \text{ B} = 1 \text{ GB}$ for a flat table. Using 2 MB megapages for the heap would remove 32 leaf tables.

Worked Example A.6: When does a cache hurt?

A streaming kernel reads 1 GB once with no reuse. With a write-allocate, write-back cache and 64-byte blocks, reads cost one block fill per 64 bytes — no worse than without a cache. But if the kernel also *writes* a 1 GB output array, write-allocate first *reads* every output block from DRAM before overwriting it: DRAM traffic = 1 GB (input) + 1 GB (output fills) + 1 GB (write-backs) = 3 GB instead of 2 GB. Non-temporal (streaming) stores, or “write-validate” policies with per-byte valid bits (sectoring), avoid the useless fill: a 33% traffic reduction.

Worked Example A.7: Memory bandwidth of a video pipeline

A camera SoC processes 4K video (3840×2160 pixels) at 60 frames/s. Each pixel is 2 bytes (YUV 4:2:2). The pipeline: the sensor writes each frame to DRAM; the ISP reads it and writes a processed frame; the encoder reads the processed frame and writes a reference frame, and reads two reference frames for motion estimation. Frame size = $3840 \times 2160 \times 2 = 16.6 \text{ MB}$. Per frame: writes 3 frames, reads 4 frames = 7 frame transfers = 116 MB; at 60 frames/s: 6.97 GB/s. A single LPDDR4X-4266 32-bit channel provides $4266 \times 10^6 \times 4 \text{ B} = 17.1 \text{ GB/s}$ peak; at a realistic 70% efficiency (row conflicts, refresh, read/write turnaround) that is 11.9 GB/s — the video pipeline alone uses 59% of it. This is why SoCs compress frame buffers in hardware and use on-chip SRAM line buffers to avoid round trips through DRAM.

Worked Example A.8: TLB reach and huge pages for an in-memory database

A database scans a 64 GB table randomly. L1 DTLB: 64 entries; L2 TLB: 2048 entries; 4 KB pages. Reach = $2048 \times 4 \text{ KB} = 8 \text{ MB} \ll 64 \text{ GB}$, so nearly every random access misses both TLBs. With a 4-level walk whose upper levels hit in cache but whose leaf PTE (one per 4 KB page, 16×2^{20} PTEs = 128 MB of PTEs) usually misses to DRAM, each access costs an extra DRAM access: the memory traffic roughly doubles. With 1 GB pages, 64 TLB entries cover the whole table: TLB misses essentially vanish. With 2 MB pages, 2048 L2-TLB entries reach

4 GB, and the leaf PTEs ($32\,768 \times 8\text{ B} = 256\text{ KB}$) fit in the L2/L3 cache, so walks are cheap.

Worked Example A.9: Counting coherence traffic

Two cores alternately increment a shared counter 1000 times each (perfect alternation) under MSI. Each increment is a read followed by a write. Core A: read miss (BusRd) $\rightarrow$ S; write (BusRdX/upgrade) $\rightarrow$ M. Core B then reads: BusRd, A flushes; B writes: BusRdX. So every increment costs 2 bus transactions and one cache-to-cache data transfer: 4000 transactions for 2000 increments. With MESI and an atomic `amoadd.w`, each increment is a single read-for-ownership (BusRdX) that obtains the block in M: 2000 transactions. Better still, keep per-core counters and add them at the end: 2 transactions in total.

B

A Trace-Driven Cache Simulator

The traces in Chapter 2 were produced with the short Python program below. It models a set-associative cache with LRU, FIFO or Belady-OPT replacement and reports the hit/miss pattern; students can extend it with write policies, multiple levels or a victim cache.

Listing B.1: cachesim.py — a minimal trace-driven cache simulator.

```
def simulate(addr, n_sets, ways, block, policy="LRU"):
    """Return a string of 'H'/'M' for each byte address in addr."""
    sets = [[] for _ in range(n_sets)]          # each set: list of block numbers
    blocks = [a // block for a in addr]
    out = []
    for k, b in enumerate(blocks):
        s = sets[b % n_sets]                    # index = block number mod #sets
        if b in s:                              # hit
            out.append("H")
            if policy == "LRU":                  # move to MRU position
                s.remove(b); s.append(b)
            continue
        out.append("M")                          # miss
        if len(s) == ways:                       # set full: choose a victim
            if policy in ("LRU", "FIFO"):
                s.pop(0)                         # front = LRU (or oldest)
            elif policy == "OPT":                # Belady: furthest next use
                future = blocks[k + 1:]
                nxt = lambda x: future.index(x) if x in future else float("inf")
                s.remove(max(s, key=nxt))
            s.append(b)
    return "".join(out)

if __name__ == "__main__":
    trace = [0x00, 0x40, 0x00, 0x48, 0x10, 0x20, 0x40, 0x80, 0x00, 0x10]
    for name, (sets, ways) in {"direct-mapped": (4, 1), "2-way": (2, 2),
                              "fully-assoc": (1, 4)}.items():
        print(f"{name:14s}", simulate(trace, sets, ways, 16))
    print("FA OPT", simulate(trace, 1, 4, 16, "OPT"))
```

Output:

direct-mapped	MMMMMMHMMH
2-way	MMHHMMHMMH
fully-assoc	MMHHMMHMMH
FA OPT	MMHHMMHMMH

Suggested extensions. (1) Count compulsory misses with a set of all blocks ever seen and classify the rest into capacity and conflict by also running a fully associative OPT cache of the same size. (2) Add a dirty bit and count write-backs for a write-back cache. (3) Chain two caches to measure local and global miss rates. (4) Feed it real address traces (e.g. from a RISC-V simulator's memory log) and plot miss rate versus cache size to find the working set.



Quick Review Questions

Short questions for revision; answers follow each question.

Q1. Why can a single memory level not be both large and fast?

Answer: Larger arrays have longer word/bit lines and deeper decoders (bigger is slower), and fast technologies (SRAM) cost more area per bit than dense ones (DRAM).

Q2. State the difference between temporal and spatial locality and name the cache parameter that exploits each.

Answer: Temporal: reuse of the same location — exploited by capacity/retention; spatial: use of nearby locations — exploited by block size.

Q3. What does $T_i = t_i + m_i T_{i+1}$ assume about m_i ?

Answer: It is the local miss rate of the accesses that reach level i .

Q4. Why is DRAM read destructive, and what restores the data?

Answer: The cell's charge is shared with the bit line; the sense amplifier latches the value and writes it back.

Q5. A cache has 256 sets, 4 ways and 32-byte blocks. What is its capacity?

Answer: 32 KB.

Q6. Why do most L1 caches use 8 or more ways?

Answer: To lower conflict misses and to allow a VIPT organization whose index lies within the page offset.

Q7. What is set thrashing, and which replacement policy suffers most?

Answer: A working set larger than the associativity cycling through a set; LRU (0% hits on cyclic patterns).

Q8. Why is Belady's OPT not optimal for execution time?

Answer: Miss costs differ (location, overlap), and OPT minimizes only the miss count.

Q9. What problem does the dirty bit solve?

Answer: It tells a write-back cache which evicted blocks must be written to the next level.

Q10. Give one advantage of write-through.

Answer: Lower levels are always up to date, simplifying coherence and error recovery.

Q11. Why are L1 caches split into I and D while L2 is unified?

Answer: IF and MEM access the L1 in the same cycle from different pipeline stages; L2 benefits from dynamic sharing.

Q12. Classify: a miss that would also occur in a fully associative OPT cache of the same size.

Answer: Capacity miss (if not compulsory).

Q13. List the three DRAM commands needed for a row conflict.

Answer: PRECHARGE, ACTIVATE, READ (or WRITE).

Q14. How many $\times 8$ chips make a non-ECC 64-bit rank?

Answer: 8.

Q15. What is the benefit of cache-block interleaving over row interleaving?

Answer: Consecutive blocks go to different banks and can be serviced in parallel.

Q16. What does FR-FCFS prioritize?

Answer: Row-buffer hits, then the oldest request.

Q17. What is a page fault, and who handles it?

Answer: An access to a page not present in physical memory (or a permission violation); the OS page-fault handler.

Q18. Why must exceptions be precise for demand paging?

Answer: The faulting instruction must be restartable after the page is loaded, with no later instruction having changed state.

Q19. Compute the TLB reach of 1536 entries with 4 KB pages.

Answer: 6 MB.

Q20. Why do multi-level page tables save memory?

Answer: Tables are allocated only for regions of the address space that are in use.

Q21. What is a synonym, and why is it a problem for virtually indexed caches?

Answer: Two VAs mapping to one PA; the data could be cached twice at different indices and become inconsistent.

Q22. What does an MSHR record?

Answer: An outstanding miss: block address, waiting requests and destination.

Q23. Name the three states of the MSI protocol and which one allows writing without a bus transaction.

Answer: Modified, Shared, Invalid; Modified.

Q24. What is false sharing?

Answer: Coherence traffic caused by different cores writing different data that share a cache block.

Q25. Why is moving data more expensive than computing on it?

Answer: Long wires and off-chip I/O dissipate far more energy ($pJ-nJ$) than an arithmetic operation ($\leq pJ$).



Formula Sheet

Quantity	Formula
Hierarchical latency	$T_i = t_i + m_i T_{i+1}$ (m_i local miss rate)
AMAT	$t_{\text{hit}} + \text{miss rate} \times \text{miss penalty}$
Global miss rate of level i	$\prod_{j \leq i} m_j$
Cache geometry	sets = $C / (B \cdot A)$; offset = $\log_2 B$; index = $\log_2(\text{sets})$; tag = $n - \text{index} - \text{offset}$
True-LRU state	$\lceil \log_2(A!) \rceil$ bits per set; tree PLRU $A - 1$ bits
3 Cs	conflict = total – compulsory – capacity (capacity from FA-OPT)
DRAM latency (to last data)	hit $t_{CL} + t_{BURST}$; empty $t_{RCD} + t_{CL} + t_{BURST}$; conflict $t_{RP} + t_{RCD} + t_{CL} + t_{BURST}$
Channel bandwidth	(transfers/s) $\times$ (bus width in bytes)
Little's law	outstanding requests = throughput $\times$ latency
Page-fault EAT	$(1 - p) t_{\text{mem}} + p t_{\text{fault}}$
Flat page table size	$2^{\text{VPN bits}} \times \text{PTE size}$
TLB reach	entries $\times$ page size
Synonym-free VIPT	cache size / associativity $\leq$ page size
SEC-DED check bits	$2^r \geq k + r + 1$, plus 1 parity bit