Week 2A: Mechanisms + First Observability Loop

Cache, locality, branch behavior, and perf stat

> "A number without explanation is noise."

Today's Plan

Session 1: Lecture (80 min):

- Part 1: OS Concepts — memory hierarchy, caches, working set, branch prediction
- First observability loop — `time` and `perf stat` (IPC, cache-misses, branch-misses)
  - **VM note:** hardware counters (cycles/cache/branch) are often unavailable in VMware/VirtualBox; keep the `perf` commands, but expect to use **Valgrind (cachegrind/callgrind)** as fallback
- Quick virtual memory intuition (minor vs major faults)
- Part 4: Lab Preview — what you'll measure in Lab 1

Session 2: Lab Workshop (80 min):

- Start Lab 1: multi-level performance analysis (quicksort)
- Get help with tools and experiment design

Learning Objectives

By the end of this session, you should be able to:

1. Design a controlled experiment (repeats, warm-up, clear baseline)
2. Use `perf stat` correctly (IPC, cache-misses, branch-misses)
    - **VM note:** on many VMware/VirtualBox guests, these *hardware* events may be `<not supported>` / 0; use **Valgrind cachegrind/callgrind** as a fallback
3. Distinguish CPU-bound vs memory-latency-bound using IPC and miss behavior
    - **VM note:** IPC requires `cycles` + `instructions` hardware events; if unavailable, focus on **timing trends** + cachegrind/callgrind evidence
4. Explain a causal mechanism (connect a number to an OS/hardware mechanism)

Quick Check: Last Week

Anyone have environment issues?

- VM not working?
- perf not running?
- Other problems?

Let's solve them now or at the break.

The Performance Anti-Pattern

```
Developer: "I optimized the code! It's 2x faster!"
You: "What mechanism made it faster?"
Developer: "..."
You: "If you can't explain the mechanism, you don't know if it will stay faster."
```

This course: Numbers are evidence. Mechanisms are explanations.

Why This Matters

Scenario 1: You optimize code, it's 2x faster. You ship it. Next week, it's slow again.

- Without understanding the mechanism, you can't prevent regression.

Scenario 2: Your service has p99 latency spikes. You add more servers. Spikes continue.

- More servers don't fix memory pressure or GC pauses.

The skill we train: Trace observed behavior → OS/hardware mechanism → targeted fix.

Part 1: OS Concepts

The Memory Hierarchy

Why Does Memory Hierarchy Exist?

The fundamental problem:

- CPU can execute **billions** of instructions per second
- But memory (DRAM) is **slow** — can't keep up with CPU

If every memory access went to DRAM:

- CPU would be idle 99% of the time, waiting for data

The solution: Build a hierarchy of progressively larger, slower storage.

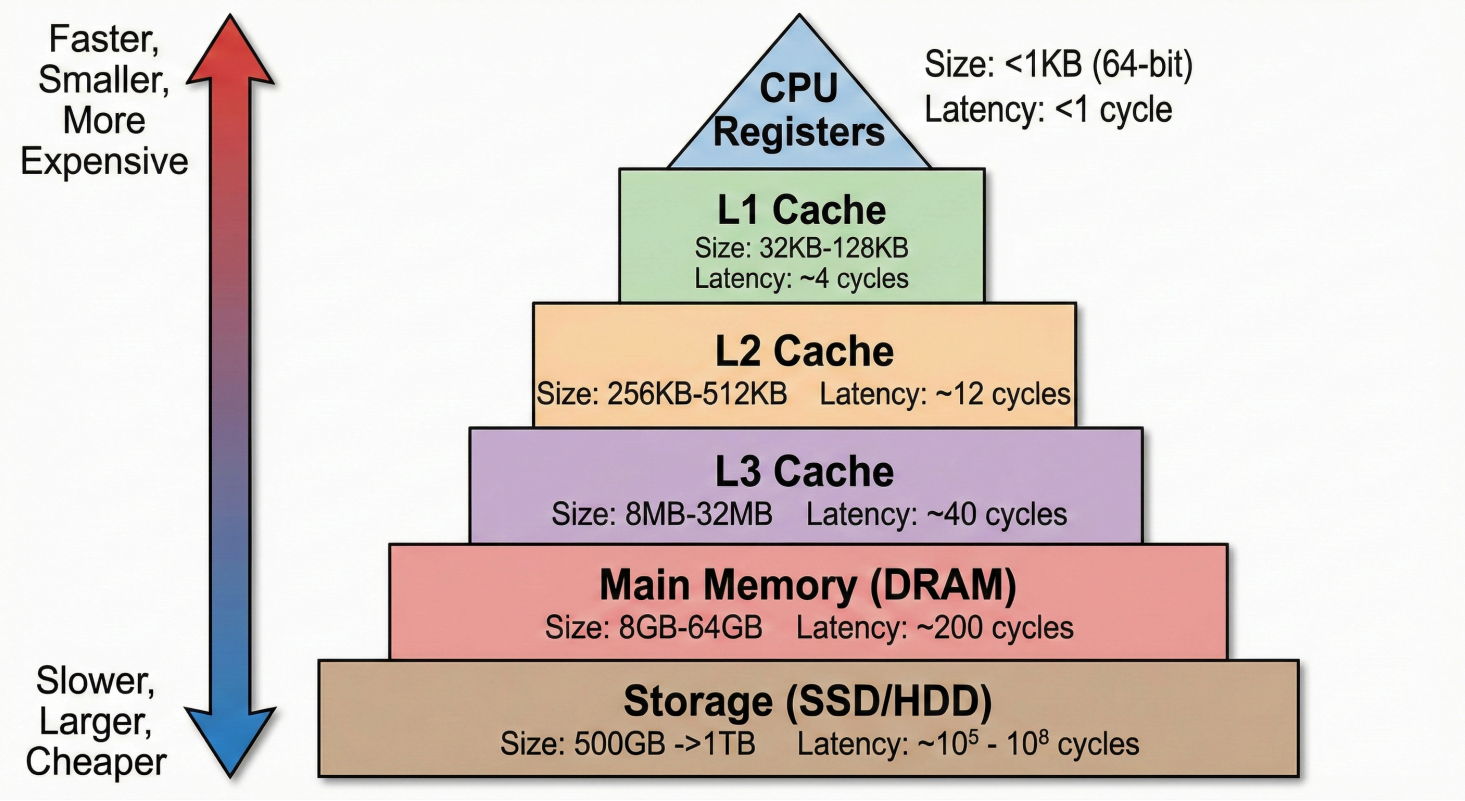
The Speed-Size Tradeoff

Physics constraint: Fast memory is expensive and hard to make large.

Memory Type	Speed	Typical Size	Cost/GB
Registers	0 cycles	~1 KB	$$$$$$
SRAM (cache)	4-40 cycles	32KB - 32MB	$$$$$
DRAM	200 cycles	8GB - 128GB	$$
SSD	50,000 cycles	256GB - 4TB	$
HDD	10,000,000 cycles	1TB - 20TB	¢

The hierarchy exploits this tradeoff.

The Memory Hierarchy Diagram



L1, L2, L3 — What's the Difference?

Level	Per-Core?	Size	Latency	What It Caches
L1	Yes	32-64 KB	4 cycles	Hot data + instructions
L2	Yes	256KB-1MB	12 cycles	Recent working set
L3	Shared	2-32 MB	40 cycles	Data used by any core

L1 is split:

- **L1i** (instruction cache) — stores code
- **L1d** (data cache) — stores data

Check Your Machine's Cache Sizes

```
# Method 1: getconf (most reliable)
$ getconf -a | grep CACHE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_DCACHE_SIZE                 32768
LEVEL2_CACHE_SIZE                  262144
LEVEL3_CACHE_SIZE                  8388608

# Method 2: Read from sysfs
$ cat /sys/devices/system/cpu/cpu0/cache/index*/size
32K     # L1d (index0)
32K     # L1i (index1)
256K    # L2  (index2)
8192K   # L3  (index3)

# Method 3: lscpu (may not show cache on all VMs)
$ lscpu | grep -i cache
```

Try these on your VM: What are your cache sizes?

You might not see anything if you are Apple Silicon. Nothing can be done there. But if you have an Intel CPU, try

`VBoxManage modifyvm "your-vm-name" --cpu-profile host`

Apple Silicon's L1/L2/L3 (M4 Example)

Level	Per-Core?	Size	Latency	What It Caches
L1	Yes	128 KB D + 192 KB I (P Core)	~3 cycles	Hot data + instructions
L2	Per-cluster shared	~16 MB (P-Core Cluster) / ~4 MB (E-Core Cluster)	~12 cycles	Recent working set
L3 (SLC)	All-cluster shared	8 MB	~18 cycles + 10-15 ns	Data used by any core/GPU

L1 is split:

- **L1i** (instruction cache) — stores code
- **L1d** (data cache) — stores data

The Cost of a Cache Miss

Level	Latency (cycles)	Latency (ns)	Analogy
Registers	0	0	Your hands
L1 hit	4	1	Grab book from desk
L2 hit	12	3	Walk to bookshelf
L3 hit	40	10	Walk to next room
DRAM	200	50-100	Drive to library
SSD	10K-100K	10-100 µs	Order online
HDD	10M+	5-10 ms	International shipping

Order-of-magnitude estimates — actual values vary by CPU, workload, and device.

The Key Insight

L1 miss → DRAM = 50x slower

If your inner loop misses L1 cache, you're leaving **50x performance** on the table.

```
// This loop runs billions of times
for (int i = 0; i < N; i++) {
    sum += data[i];  // If this misses L1 every time... disaster
}
```

The algorithm is O(N). The constant factor is 50.

Cache In Detail

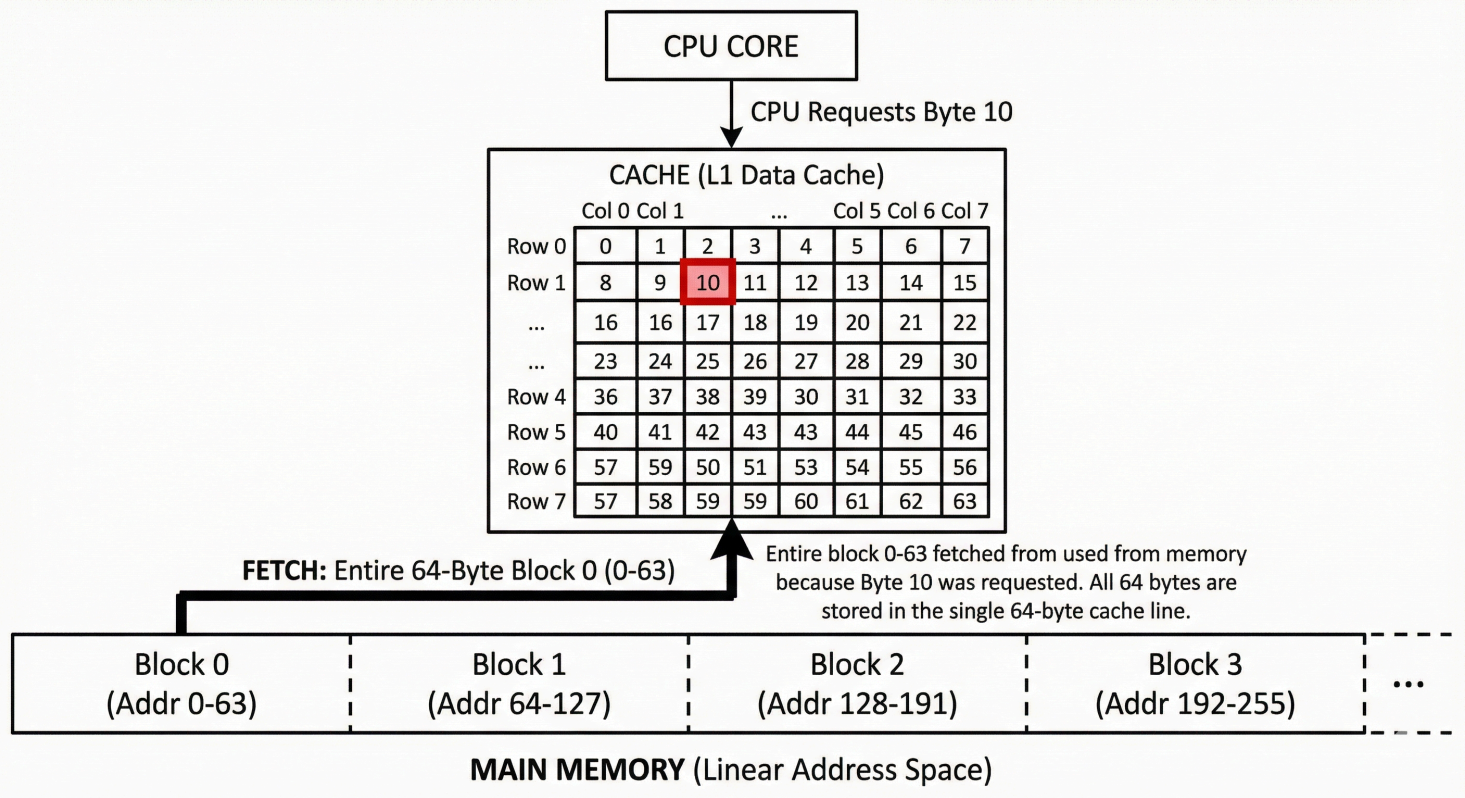
A cache is **small, fast memory** that stores **copies of recently used data**.

Key properties:

- **Capacity:** How much data can it hold? (e.g., 32KB for L1)
- **Line size:** How much data is fetched at once? (typically 64 bytes)
- **Associativity:** How flexible is placement? (e.g., 8-way)
- **Replacement policy:** Which data to evict? (usually LRU)

Cache Lines: The Unit of Transfer

CPU doesn't fetch **one byte** at a time. It fetches a **cache line** (typically 64 bytes).



If you access byte 10: CPU fetches bytes 0-63 (entire cache line).
Bytes 11-63 are now "free" to access.

Why 64 Bytes?

64 bytes is a balance between:

- **Spatial locality benefit:** Fetch nearby data together
- **Memory bandwidth:** Don't fetch too much unused data

64 bytes fits: 16 integers, 8 doubles, or 64 characters.

```
# Verify on your machine
$ getconf LEVEL1_DCACHE_LINESIZE
64
```

Locality: The Key to Cache Performance

Why do caches work? Programs exhibit **locality**:

Temporal Locality: Access X, likely access X again soon.

```
for (int i = 0; i < 1000; i++) {
    counter++;  // Same variable accessed 1000 times
}
```

Spatial Locality: Access X, likely access X+1 soon.

```
for (int i = 0; i < N; i++) {
    sum += arr[i];  // Sequential access
}
```

Good vs Bad Locality

Good: Sequential access

```
for (int i = 0; i < N; i++)
    sum += arr[i];  // Next element in same/next cache line
```

Bad: Strided access

```
for (int i = 0; i < N; i += 16)
    sum += arr[i];  // May skip cache lines
```

Terrible: Random access

```
for (int i = 0; i < N; i++)
    sum += arr[rand() % N];  // Every access likely misses
```

In-Class Exercise: Which Is Faster?

Option A: Row-major traversal

```
int arr[1000][1000];
for (int i = 0; i < 1000; i++)
    for (int j = 0; j < 1000; j++)
        sum += arr[i][j];
```

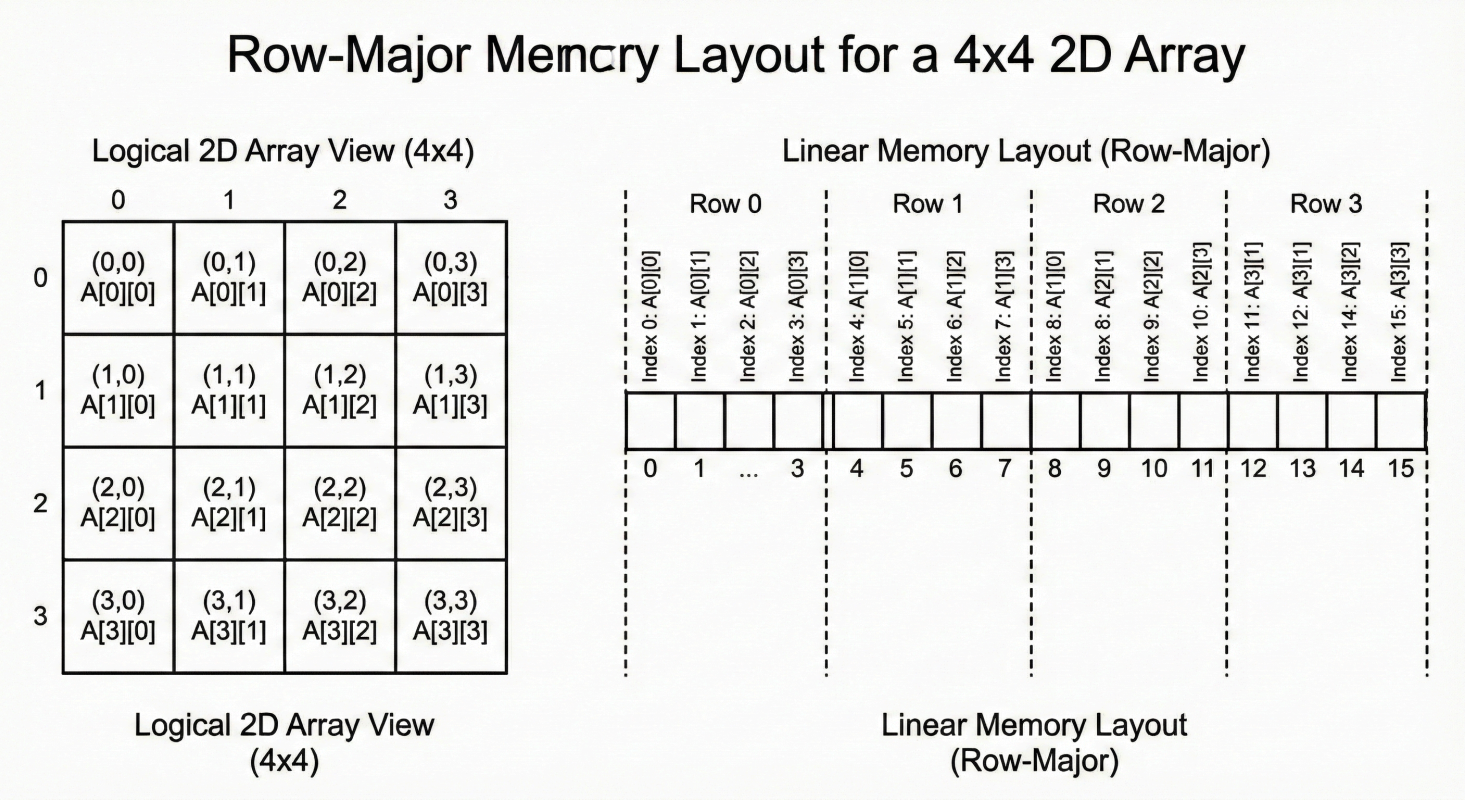
Option B: Column-major traversal

```
int arr[1000][1000];
for (int j = 0; j < 1000; j++)
    for (int i = 0; i < 1000; i++)
        sum += arr[i][j];
```

Think for 30 seconds...

How Are 2D Arrays Stored in Memory?

In C, 2D arrays are stored row-major:



Row elements are **contiguous** in memory.

Column elements are **separated** by row size.

Answer: Row-Major Is Much Faster

Row-major (Option A):

- Access: arr[0][0], arr[0][1], arr[0][2]...
- Sequential in memory → **excellent spatial locality**
- One cache miss per 16 elements

Column-major (Option B):

- Access: arr[0][0], arr[1][0], arr[2][0]...
- Jumps by 4000 bytes each time → **terrible locality**
- One cache miss per element!

Real-world impact: 10-50x performance difference!

Measuring Cache Behavior with perf

```
$ sudo perf stat -e cache-references,cache-misses ./row_major
    31,162,964      cache-references
        64,245      D cache-misses    # 0.8 % miss rate

$ sudo perf stat -e cache-references,cache-misses ./col_major
    31,162,964     cache-references
     1,001,764     D cache-misses    # 12.5% miss rate
```

VM note: these are hardware counter events. If your VM prints `<not supported>` / 0, skip straight to the cachegrind slide below.

Same algorithm, same data, 50x difference in cache misses!

VM note: `cache-references/cache-misses` are **hardware PMU** events. In many VMware Fusion / VirtualBox Ubuntu VMs, `perf stat` will show `<not supported>` or 0 for these events.

No good output from perf? (common in VMs)

```
sudo apt install valgrind
valgrind --tool=cachegrind --cache-sim=yes --branch-sim=no ./row-major
==4336== D refs:         8,045,656  (6,030,522 rd   + 2,015,134 wr)
==4336== D1  misses:        64,245  (   63,845 rd   +       400 wr)
==4336== LLd misses:        63,846  (   63,483 rd   +       363 wr)
==4336== D1  miss rate:        0.8% (      1.1%     +       0.0%  )

valgrind --tool=cachegrind --cache-sim=yes --branch-sim=no ./column-major
==4345== D refs:         8,045,648  (6,030,514 rd   + 2,015,134 wr)
==4345== D1  misses:     1,001,764  (1,001,344 rd   +       420 wr)
==4345== LLd misses:        64,863  (   64,483 rd   +       380 wr)
==4345== D1  miss rate:       12.5% (     16.6%     +       0.0%  )
```

Same algorithm, same data, 50x difference in cache misses!

Calculating Cache Miss Impact

Example: Inner loop with 10% cache miss rate

- Cache hit latency: 4 cycles
- Cache miss latency: 200 cycles (DRAM)
- Effective latency: 0.9 × 4 + 0.1 × 200 = **23.6 cycles**

That's **6x slower** than perfect cache hits!

Quick Quiz

A program has 5% L1 miss rate. All misses hit L2.

- L1 hit = 4 cycles
- L2 hit = 12 cycles

What is the effective access latency?

30 seconds to calculate...

Quiz Answer

Effective = 0.95 × 4 + 0.05 × 12 = 3.8 + 0.6 = **4.4 cycles**

Only 10% slower than perfect L1 hits.

But if L1 misses went to DRAM:
= 0.95 × 4 + 0.05 × 200 = 3.8 + 10 = **13.8 cycles** (3.5x slower!)

Lesson: L2 and L3 caches save you from disaster.

Working Set: The Key Concept

Working Set: The set of data your program actively uses.

Working Set Size	Cache Behavior
Fits in L1	Very fast
Fits in L3	Fast
Exceeds L3	Slow (DRAM)

Lab 1 asks: At what N does quicksort's working set exceed L3?

Virtual Memory and Page Faults

What is Virtual Memory?

Problem: Multiple programs need to share physical RAM.

Solution: Give each process its own "virtual" address space.

Benefits:

- **Isolation:** Process A can't access Process B's memory
- **Simplicity:** Every process thinks it starts at address 0
- **Flexibility:** Physical memory can be fragmented
- **Overcommit:** Allocate more virtual memory than physical RAM

Pages: The Unit of Virtual Memory

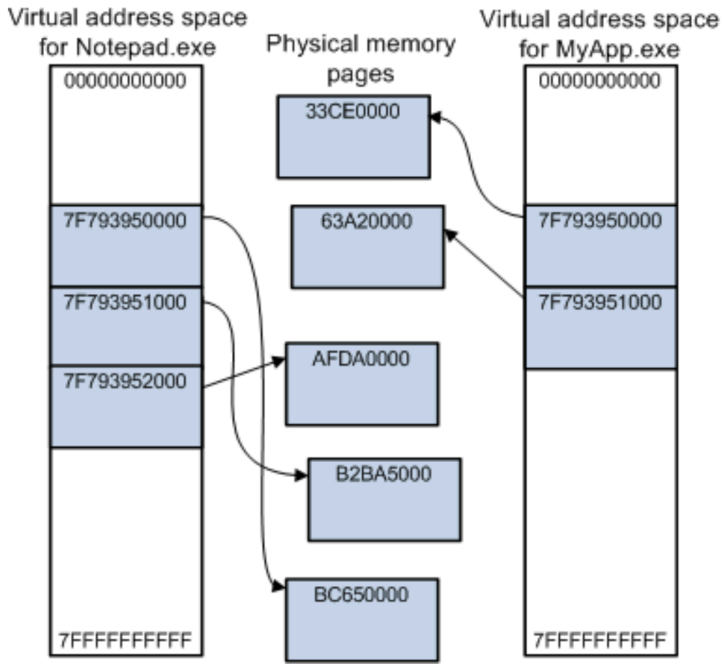
Page: A fixed-size chunk of memory (typically **4KB** on x86).

```
Virtual Address Space (per process):
[Page 0: 0x0000–0x0FFF][Page 1: 0x1000–0x1FFF][Page 2]...

Physical RAM (shared):
[Frame 0][Frame 1][Frame 2][Frame 3]...

Mapping (Page Table):
  Memory Page 0 → Frame 5
  Memory Page 1 → Frame 12
  Memory Page 2 → (on disk!)
```

Paging for Processes



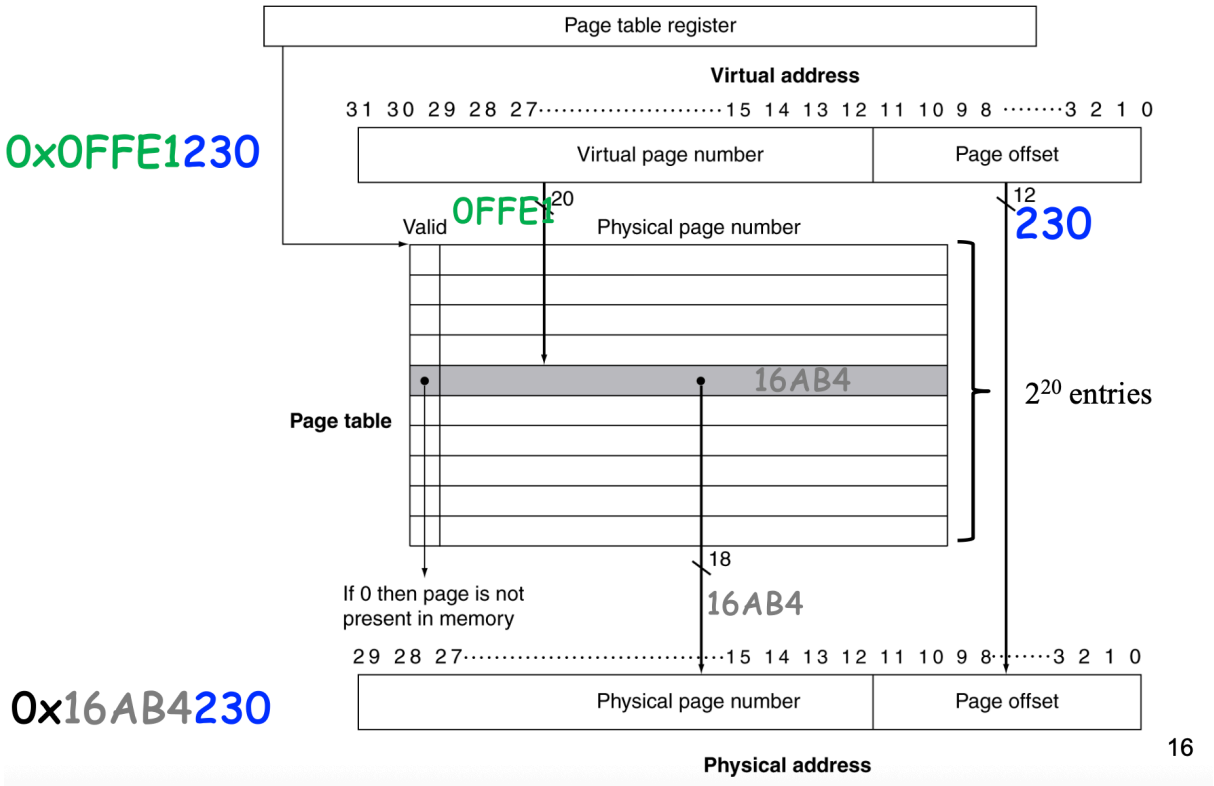
The Page Table

Each process has a page table mapping virtual pages to physical frames.

Virtual Page	Present?	Physical Frame	Permissions
0	Yes	5	Read/Write
1	Yes	12	Read/Execute
2	No	(on disk)	-
3	Yes	3	Read/Write

On every memory access, CPU translates virtual → physical.

Address Translation



Virtual address = Page Number + Offset

- Look up page number in page table → physical frame
- Physical address = frame × page_size + offset

The TLB: Translation Lookaside Buffer

Problem: Page table lookup adds latency to every access.

Solution: Cache recent translations in the TLB.

TLB Result	Latency
TLB hit	~1 cycle
TLB miss	10-100 cycles (page table walk)

Typical TLB: 64-1024 entries

- Covers 256KB to 4MB of virtual addresses

TLB Translation

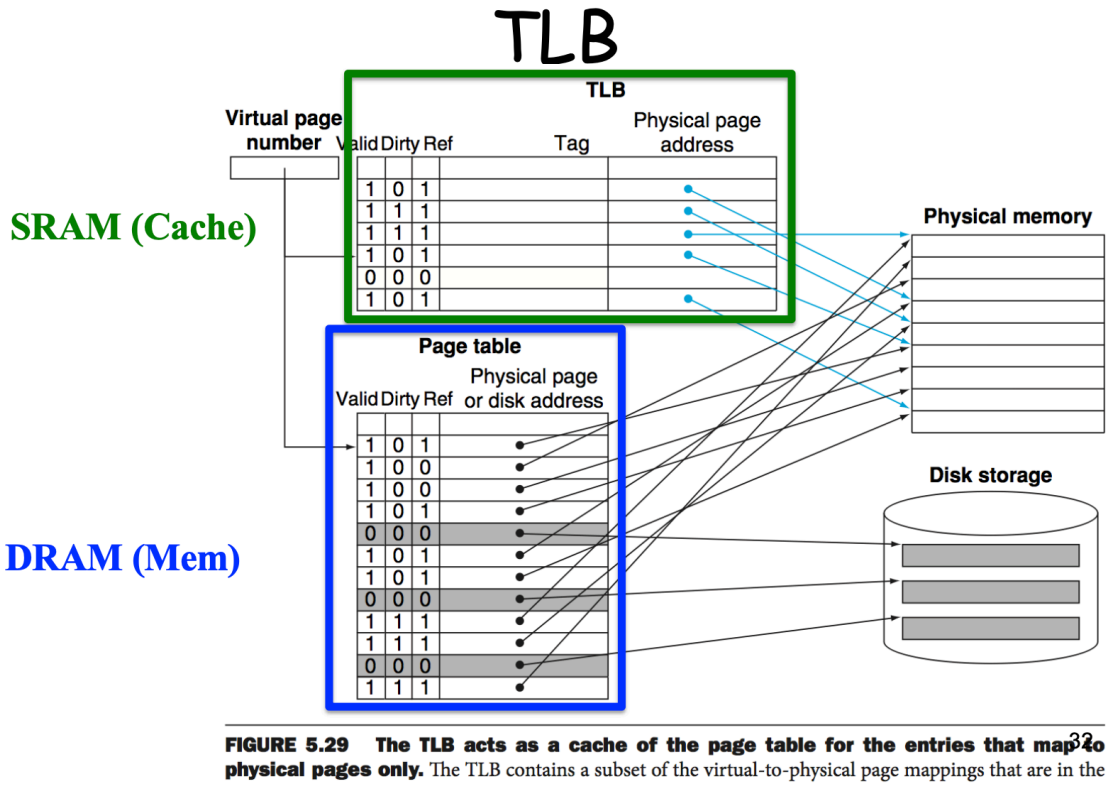


FIGURE 5.29 The TLB acts as a cache of the page table for the entries that map to physical pages only. The TLB contains a subset of the virtual-to-physical page mappings that are in the

TLB allows us to quickly get the DRAM physical frame of the virtual page.

Page Faults

What if the page is not in RAM? → **Page fault** exception.

Type	What Happens	Cost
TLB hit	Translation cached	~1 cycle
TLB miss	Walk page table	10-100+ cycles*
Minor fault	Page in RAM, not mapped	~1,000 cycles
Major fault	Page on disk, must load	**~10M cycles**

TLB miss cost varies: fast if page-walk hits cache, slow otherwise.

Major page fault ≈ 10,000x worse than cache miss!

When Do Page Faults Happen?

1. **First access to new memory** (malloc, stack growth)
    - Minor fault: allocate and zero a new page
2. **Accessing swapped-out memory** (memory pressure)
    - Major fault: load page from swap
3. **Copy-on-write after fork()**
    - Minor fault: copy the page on first write
4. **Memory-mapped file access**
    - Major fault: load file data

Measuring Page Faults

```
# Touch 512MB once, one byte per page
$ /usr/bin/time -v ./my_program --anon 512 --pattern seq --repeat 1 2>&1 | grep -i fault
Minor (reclaiming a frame) page faults: 262477
Major (requiring I/O) page faults: 0

# Or with perf
$ sudo perf stat -e page-faults,major-faults ./my_program --anon 512 --pattern seq --repeat 1
       1,234      page-faults
           5      major-faults
# Repeated
$ /usr/bin/time -v ./my_program --anon 512 --pattern seq --repeat 5 2>&1 | grep -i fault
# Repeated and force to drop anon pages so you get faults again
$ /usr/bin/time -v ./my_program --anon 512 --pattern seq --repeat 5 --madvise-dontneed 2>&1 | grep -i fault
```

Page Fault Example: Memory Pressure

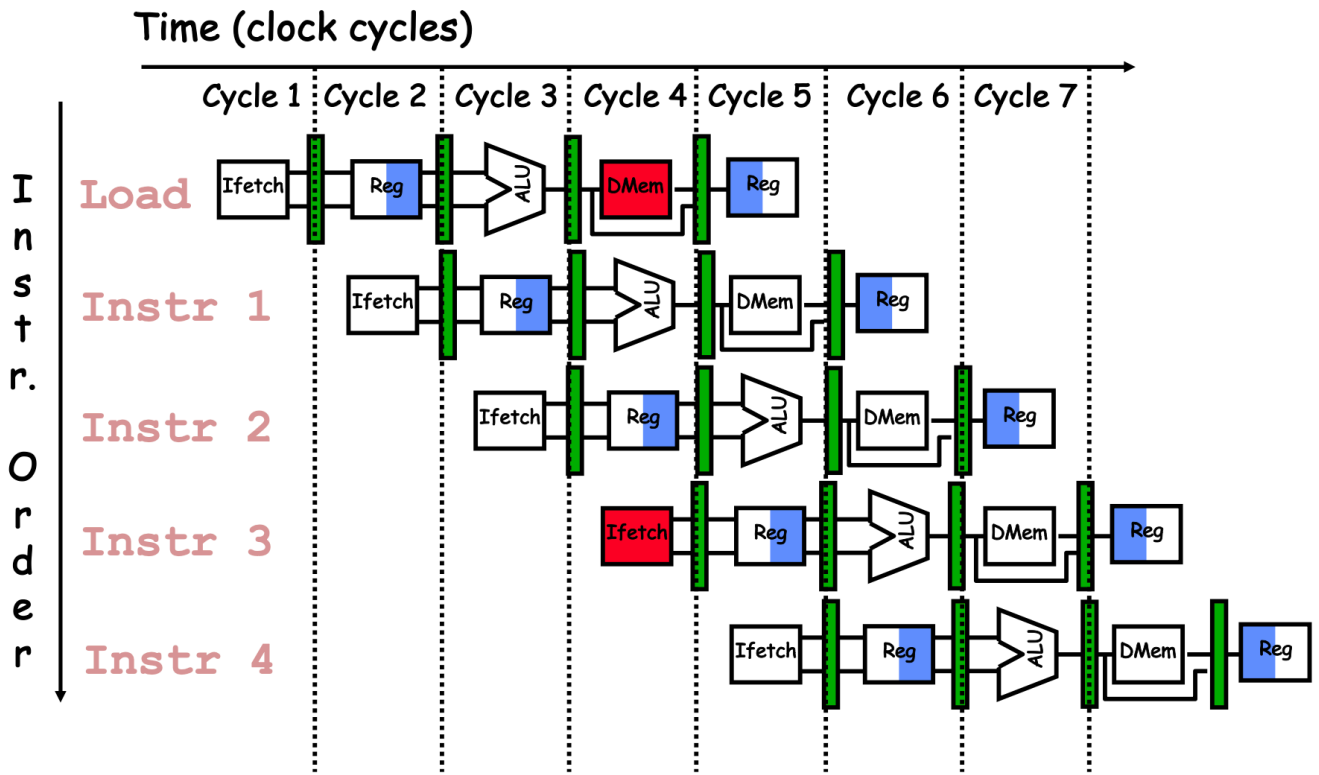
Scenario: Container has 512MB memory limit.

- Application uses 510MB
- Kernel reclaims pages to stay under limit
- Application accesses reclaimed pages
- **Major page faults → disk I/O → latency spikes**

The symptom is latency. The cause is memory. The mechanism is page faults.

Branch Prediction

The CPU Pipeline



Modern CPUs overlap instruction execution (pipelining).

Problem: At a branch, which instruction to fetch next?

The Branch Problem

```
if (x > 0) {
    y = a;  // path A
} else {
    y = b;  // path B
}
```

CPU doesn't know which path until `x > 0` is evaluated.

But pipeline has already fetched 15-20 instructions...

Without prediction: Stall for 15-20 cycles at every branch!

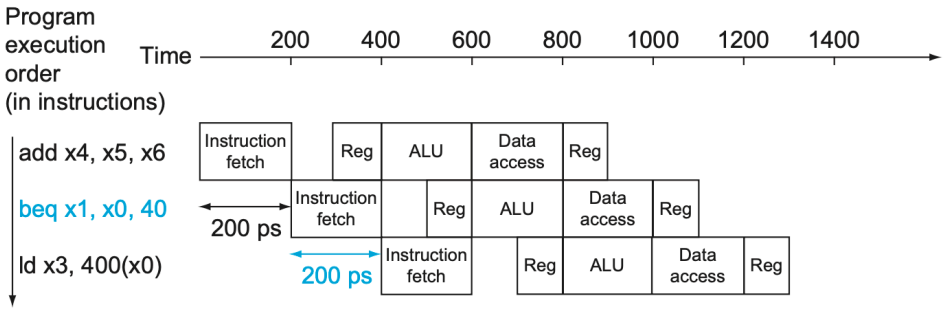
Speculative Execution

Solution: Guess which path to take!

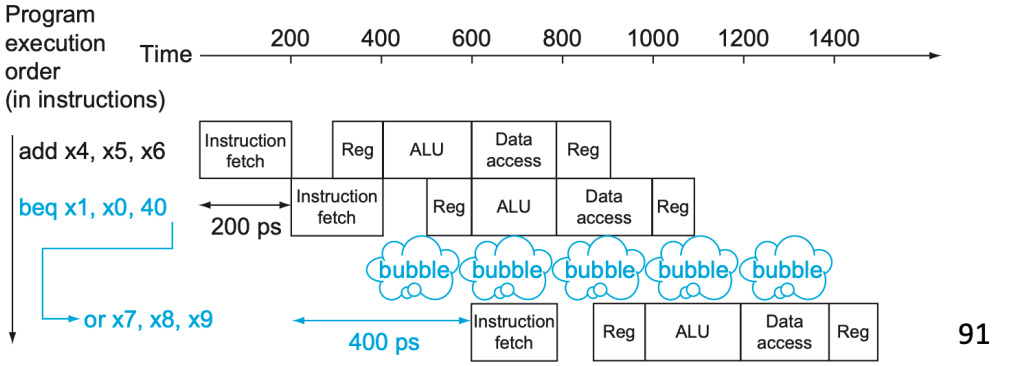
- **Branch predictor** guesses based on history
- CPU speculatively executes predicted path
- **Correct:** No penalty
- **Wrong:** Flush pipeline, **15-20 cycles wasted**

Branch and Prediction

Prediction correct



Prediction incorrect



91

Branch Prediction Accuracy

Easy to predict:

```
for (int i = 0; i < 1000000; i++) { }
// "Loop again" is taken 999,999 times → ~100% accuracy
```

Hard to predict:

```
if (rand() % 2 == 0) { } else { }
// Random → ~50% accuracy → many mispredictions
```

Measuring Branch Mispredictions

```
$ sudo perf stat -e branches,branch-misses ./my_program

   100,000,000      branches
     5,000,000      branch-misses    # 5% miss rate
```

VM note: `branches/branch-misses` are **hardware PMU** events and may be unavailable in VMware/VirtualBox.

If you see `<not supported>` or 0 counts, use cachegrind's branch simulation:

```
sudo apt install valgrind
valgrind --tool=cachegrind --cache-sim=yes --branch-sim=yes ./my_program
# See: "Branches:" and "Mispredicts:" in the output
```

Impact: 5% × 15 cycles = 0.75 cycles per branch (average)

If branches are frequent, this adds up!

The Sorted Array Example

```
// Unsorted: values are random (47, 182, 91, 203, 12...)
for (int i = 0; i < N; i++)
    if (arr[i] >= 128)  // Unpredictable!
        sum += arr[i];

// Sorted: values are ordered (1, 2, 3... 127, 128, 129...)
for (int i = 0; i < N; i++)
    if (arr[i] >= 128)  // First half: NO, second half: YES
        sum += arr[i];
```

Sorted is 3-6x faster! (branch becomes predictable)

OS Concepts Summary

Concept	Why It Matters	Cost of Getting It Wrong
Cache hierarchy	Data locality	50x slowdown
Cache lines (64B)	Memory layout	Wasted bandwidth
TLB	Page table caching	10-100x+ (varies)
Page faults	Memory pressure	10,000x for major
Branch prediction	Control flow	15-20 cycles/miss

Quick Check

Can you answer these?

1. Why is column-major traversal slow in C?
2. What happens on a TLB miss?
3. Why is a major page fault so expensive?
4. Why does sorting help branch prediction?

Ask now if any are unclear!

Part 1 References

Memory Hierarchy & Caching:

- Patterson & Hennessy, *Computer Organization and Design*, Chapter 5
- Ulrich Drepper, "What Every Programmer Should Know About Memory" (2007)

Virtual Memory:

- OSTEP, Chapters 18-23 (free online: ostep.org)

Branch Prediction:

- Agner Fog, "Microarchitecture of Intel/AMD CPUs"

Part 4: Lab Preview

What You'll Measure in Lab 1

Latency-Bound vs Bandwidth-Bound

When people say "memory-bound", they often mean two different bottlenecks:

Bottleneck	Intuition	Typical symptoms	Typical fix
Latency-bound (miss-driven)	Waiting for *a few* slow misses	IPC drops, stall cycles rise, LLC miss rate rises, bandwidth may stay moderate	Improve locality, reduce misses, better data layout
Bandwidth-bound (streaming)	DRAM can deliver bytes only so fast	Bandwidth near peak, IPC may be low but miss rate may be stable	Reduce bytes moved, improve cache reuse, NUMA placement

Takeaway: A higher miss rate does not imply bandwidth is the limiter, and high bandwidth does not imply lots of misses.

Optional Contrast Microbenchmark (Bandwidth-Bound)

To contrast quicksort (mixed branch/cache effects), you may also run a streaming kernel:

```
// Streaming sum / copy: predictable, sequential
for (size_t i = 0; i < N; i++)
    dst[i] = src[i] + 1;
```

What to look for:

- Throughput plateaus as N grows
- Memory bandwidth rises toward a stable ceiling

(Optional, for students who want a cleaner bandwidth story.)

Lab 1: Multi-Level Performance Analysis

Goal: Trace performance through multiple levels:

- Algorithm behavior
- Hardware counters (cache, branches)
- OS metrics (time, page faults)

Program: Quicksort (well-understood)

Question: At what N does quicksort become memory-bound?

Lab 1 Structure

Part	Difficulty	What You Do
Part A	Basic (required)	Profile quicksort on different inputs
Part B	Intermediate (required)	Find CPU→memory transition
Part C	Advanced (optional)	Apply to your own workload

Part A: What You'll Measure

Input types: Random, Sorted, Reverse, Nearly sorted

Metrics:

- Wall time, user/sys time
- Cache misses
- Branch mispredictions

Expected: Sorted is MUCH slower. Why?

Part B: The Key Question

At what N does quicksort transition from CPU-bound to memory-bound?

- Small N: working set fits in cache → CPU-bound
- Large N: working set exceeds cache → memory-bound (latency-limited by cache misses)

Your task: Find the transition point and explain the mechanism.

What Signals Should You Expect?

Hypothesis: When working set crosses L3 cache size...

Metric	Before Transition	After Transition
IPC (instructions/cycle)	High (2-3)	Drops (< 1)
LLC miss rate	Low (< 5%)	Increases significantly
Time scaling	~linear with N	Worse than linear

Template: "I expect IPC to drop from X to Y when N exceeds Z because..."

Note: Quicksort is also branch-sensitive — sorted input has different behavior!

In-Class Activity (5 min)

Write a short answer:

> "Given a program 10x slower than expected, list the first 5 things you would check, in order, and what tool for each."

5 minutes. We'll share answers.

Example Answer

1. **Time breakdown** — `time ./program` → user, sys, or waiting?
2. **CPU utilization** — `top` → using CPU or blocked?
3. **Cache behavior** — `perf stat -e cache-misses` → miss rate?
    - **VM fallback:** `valgrind --tool=cachegrind --cache-sim=yes ./program` (simulated cache misses)
4. **Hot spots** — `perf record && perf report` → where?
    - **VM fallback:** `valgrind --tool=callgrind ./program` + `callgrind_annotate` (or `kcachegrind`)
5. **Memory** — `perf stat -e page-faults` → page faults?

Measurement Toolkit (So Far)

`time`, `perf stat`, and Valgrind (cachegrind/callgrind/massif)

Goal: turn "it's slow" into evidence -> mechanism -> explanation.

One Rule: Always Ask "Rate vs Count?"

Many tools print **counts** (e.g., cache misses). Counts often grow with input size even if behavior is unchanged.

To understand a transition, you usually need a **rate**:

- cache miss rate ~= `cache-misses / cache-references`
- LLd miss rate (cachegrind) ~= `LLd misses / D refs`
- branch miss rate ~= `branch-misses / branches`
- IPC ~= `instructions / cycles`

Counts tell you "how much". Rates tell you "what changed".

Tool 1: `/usr/bin/time` (Real Timing + OS Counters)

Use `/usr/bin/time` (not the shell builtin) for consistent formatting.

```
/usr/bin/time -f "%e" ./qs datasets/random_200000.txt
# %e = elapsed wall time (seconds)
```

Verbose mode includes OS-level signals:

```
/usr/bin/time -v ./my_program 2>&1 | less
```

Key fields to read:

- **User time / System time**: CPU work in user vs kernel
- **Elapsed (wall clock) time**: includes waiting (I/O, scheduling)
- **Major / Minor page faults**: VM behavior
- **Maximum resident set size (RSS)**: peak physical memory footprint
- **Voluntary / involuntary context switches**: blocking vs preemption

How to Interpret `time -v` (Common Patterns)

If **elapsed >> user+sys**:

- your program is waiting (I/O, contention, sleeping, throttling)

If **system time** is high:

- lots of syscalls, page faults, or kernel work (I/O, mmap, paging)

If **major faults > 0**:

- you likely touched pages that required disk I/O (swap/file-backed paging)
- this can dominate runtime (orders of magnitude slower than cache misses)

If **max RSS** grows with N:

- memory footprint scales; higher risk of paging and cache pressure

Tool 2: `perf stat` (Hardware Counters)

`perf stat` measures **real CPU events** (when available).

```
sudo perf stat -e cycles,instructions,cache-references,cache-misses \
  ,branches,branch-misses,page-faults,major-faults \
  ./qs datasets/random_200000.txt
```

Flags you will use:

- `-e ...` choose events
- `-r 5` run 5 repeats and report variance
- `--` separates perf options from program args

VM note: in VMware/VirtualBox (and Apple Silicon guests), many PMU events can be `<not supported>` or 0.

Reading `perf stat`: The "Big 5" Metrics

Typical lines you care about:

- **cycles**: "how long" (in CPU cycles)
- **instructions**: "how much work" (retired instructions)
- **IPC = instructions/cycles**
  - high IPC (~2-4): CPU running efficiently
  - low IPC (<1): stalls (often memory latency)
- **cache-misses** and **cache-references**
  - look at **miss rate**, not just miss count
- **branch-misses** and **branches**
  - miss rate times mispredict penalty (~15-20 cycles) can be huge

Interpretation habit:

> If time increased, did cycles increase? If cycles increased, was it more instructions, or lower IPC (more stalls)?

`perf stat` in VMs: What to Do When It Fails

If you see `<not supported>` / zeros:

- do **not** try to interpret those numbers
- switch to Valgrind tools for simulated evidence

Fallback workflow (common in this course):

1. `/usr/bin/time` for real time trends
2. `valgrind --tool=cachegrind` for cache/branch trends
3. `valgrind --tool=callgrind` for hot spots
4. `valgrind --tool=massif` for peak memory and growth shape

Valgrind: What It Is (and Isn't)

Valgrind tools run your program under instrumentation:

- much **slower** than native
- but provides **detailed, stable, explainable** measurements

Important:

- cache/branch stats are **simulated** (model-based)
- use them for **comparisons and trends**, not exact hardware truth

Compile tip (helps attribution):

```
make clean && make CFLAGS="-O2 -g"
```

Tool 3: `cachegrind` (Cache + Branch Simulation)

Command:

```
valgrind --tool=cachegrind --cache-sim=yes --branch-sim=yes \
  --cachegrind-out-file=outputs/cg_%p.out \
  ./qs datasets/random_200000.txt
```

Why set the output file?

- avoids "Permission denied" issues
- keeps results organized (`%p` = PID)

Common fields:

- `I refs`, `I1 misses` (instruction fetch behavior)
- `D refs`, `D1 misses` (data cache behavior)
- `LLd misses` (last-level data misses; approx "LLC/L3-ish" depending on the model)

Reading `cachegrind`: What to Look For

For "working set exceeds cache" stories, focus on **data**:

- `D refs`: total data accesses (reads+writes)
- `D1 misses`: L1 data misses (small-cache locality)
- `LLd misses`: misses that reach the "last level" (proxy for expensive memory)

Two useful derived rates:

- **D1 miss rate** ~= `D1_misses / D_refs`
- **LLd miss rate** ~= `LLd_misses / D_refs`

Interpretation:

- rising **LLd miss rate** is a strong sign of growing memory latency pressure
- rising **D refs** can also mean the algorithm did more work (e.g., O(n^2) behavior)

Tool 4: `callgrind` (Where Did the Time Go?)

`callgrind` attributes cost to functions/lines (instruction-level model).

```
valgrind --tool=callgrind --callgrind-out-file=outputs/call_%p.out \
  ./qs datasets/random_200000.txt

callgrind_annotate --auto=yes outputs/call_*.out | head -60
# Optional GUI: kcachegrind (best for exploring call graphs)
```

How to read it:

- **Inclusive cost**: time in the function + callees (good for big picture)
- **Self cost**: time in the function body only (good for "what line is hot")

Use it to answer:

> Is the slowdown coming from partition/comparisons, recursion overhead, or I/O/parsing?

Tool 5: `massif` + `ms_print` (Heap Growth Over Time)

Massif is a **heap profiler** (peak memory, growth pattern, who allocated).

Run:

```
valgrind --tool=massif --massif-out-file=outputs/ms_%p.out \
  ./qs datasets/random_200000.txt

ms_print outputs/ms_*.out | less
```

Useful flags:

- `--stacks=yes` include stack(s) in profile (optional; adds overhead)
- `--time-unit=...` changes the x-axis notion of "time" (see manual)

How to Read `ms_print` Output (Mental Model)

`ms_print` typically shows:

1. Preamble (how it was run)
2. ASCII graph: memory over "time"
3. Snapshot list:
    - total snapshots collected
    - "detailed snapshots", including the **peak**
4. Detailed snapshot(s):
    - exact byte counts at that moment
    - breakdown of which allocation call stacks are responsible

The key step is: jump to the **peak snapshot** and read who allocated it.

Reading the Massif Graph (ASCII)

In the graph:

- **X-axis**: progression of execution (snapshots over time)
- **Y-axis**: memory usage (often KB/MB scale)
- each vertical column is a snapshot; taller means more memory at that moment

Below the graph, Massif tells you:

- `Number of snapshots: ...`
- `Detailed snapshots: [ ... (peak) ... ]`

Action:

- locate the snapshot marked **(peak)**
- scroll to that snapshot's detailed breakdown

If the graph is uninformative for short programs, try a different time unit:

```
valgrind --tool=massif --time-unit=B ...
# B = bytes allocated/deallocated (often makes the graph more informative)
```

Reading a Peak Snapshot: What the Numbers Mean

In a detailed snapshot, you will often see:

- `mem_heap_B` : useful heap bytes in use
- `mem_heap_extra_B` : allocator overhead / metadata
- `mem_stacks_B` : stack bytes (only if enabled / available)

Then you will see a "heap tree" (allocation call stacks).

How to use it:

- find the **largest** subtree/call stack
- that stack is your "who allocated the peak memory" answer
- connect it back to code/data structures (array copies, recursion depth, buffers)

Practical Hygiene: Always Write Outputs to `outputs/`

This avoids permission/file-clobber issues and keeps runs reproducible:

```
valgrind --tool=cachegrind --cachegrind-out-file=outputs/cg_%p.out ...
valgrind --tool=callgrind --callgrind-out-file=outputs/call_%p.out ...
valgrind --tool=massif    --massif-out-file=outputs/ms_%p.out ...
```

If you ever see "Permission denied" from Valgrind output:

- you likely have an old `cachegrind.out.*` owned by root (from earlier `sudo`)
- using explicit `--*-out-file=outputs/...` sidesteps the problem

Putting It Together: A Repeatable Investigation Loop

1. Start with timing
   - `/usr/bin/time -v` for OS signals (faults, RSS)
2. If hardware counters work
   - `perf stat` -> IPC + miss rates + branch miss rate
3. If hardware counters do not work (common in VMs)
   - `cachegrind` for cache/branch trends across N
4. Find the hot code
   - `callgrind` (annotate or kcachegrind)
5. Check memory growth
   - `massif` + read peak snapshot

The report should read like:

> "Runtime increased because X. Evidence: metric A changed at N~=..., which implies mechanism M."

References

- Valgrind Massif manual: https://valgrind.org/docs/manual/ms-manual.html
- `ms_print` man page: https://man7.org/linux/man-pages/man1/ms_print.1.html
- `perf` docs/wiki: https://perf.wiki.kernel.org/

Summary: What We Covered

Part	Key Takeaway
OS Concepts	Cache (50x), page faults (10,000x), branches (15 cycles)
Methodology	Use `perf stat`, control variance, explain the mechanism
Case Study	Connect symptoms to mechanisms
Lab Preview	Profile quicksort, find transition point

Part 4 References

Microbenchmarks and bandwidth:

- STREAM benchmark: https://www.cs.virginia.edu/stream/

Linux performance tools:

- Brendan Gregg, *Systems Performance*, 2nd ed. (2020)
- `perf` docs: https://perf.wiki.kernel.org/

Homework Reminder

Due before Week 2B:

1. Lab 1 report (Part A + B required)
2. Final project proposal (1 page)

Questions?

Let's Start Lab 1

Open `lab1_instructions.md` and let's get started!

Goals for today:

- Build the quicksort program
- Run first measurements
- Identify questions

Session 2: Lab Workshop begins