

Lab 5: Performance Optimization Guide

Goal: Make your code faster **without changing its output**.

Recall the Iron Law from lecture:

$$\text{Runtime} = \text{Instructions} \times \text{CPI} \times \text{Clock Period}$$

Since the ISA and clock are fixed in this lab, there are two main ways to improve performance:

- **Execute fewer instructions:** Remove unnecessary work.
- **Reduce CPI:** Avoid stalls, improve cache behavior, and expose more parallelism.

The most important rule: measure before optimizing. Don't spend an hour rewriting a loop that takes only 5% of your runtime.

1. How to Approach Optimization

Follow this process for each change:

1. **Profile:** Find the functions or loops taking the most time. Record a baseline.
2. **Investigate:** Use performance counters and assembly to understand *why* the code is slow.
3. **Form a hypothesis:** Predict what an optimization should improve.
4. **Change one thing:** Make one change at a time.
5. **Verify correctness:** Confirm that the output is still bit-exact.
6. **Benchmark again:** Measure multiple runs against your baseline.

If an optimization makes the program slower, revert it and try something else.

Amdahl's Law: Even completely removing a function that uses 10% of runtime gives only about a 1.11× overall speedup. Focus on the expensive parts first.

Lab restrictions

- **Bit-exact output:** Do not reorder floating-point operations or use approximate arithmetic.
- **Compiler:** GCC from scripts/setup.sh; use the lab Makefile defaults.
- **Optimization flags:** -O2 -g for all lab measurements.
- **Not allowed:** -O3, -ffast-math, LTO, or PGO.
- **Machine:** 4-core VM with a shared L3 cache. Some performance counters are missing or unreliable.
- **Permissions:** Use sudo for setup; benchmark as the regular user.

2. Diagnosing Performance Problems

First, use your profiler to find the *hot* functions. Then look at the relevant **perf** or APerf counters.

What you observe	Possible issue	What to investigate
High backend stalls and high L1 miss rate	Memory bottleneck	Cache locality and access patterns
High backend stalls but low L1 miss rate	Computation bottleneck	Dependency chains or expensive instructions
High IPC but still slow	Too much work	Instruction count and scalar loops
High branch mispredictions	Unpredictable branches	Data-dependent conditionals
High frontend stalls	Instruction fetch bottleneck	Code size and excessive unrolling
Poor scaling with more threads	Multicore bottleneck	False sharing, contention, or load imbalance
Occasional very slow runs	Setup or paging overhead	malloc , page faults

Counters are clues, not proof. Backend stalls, for example, can come from arithmetic as well as memory.

Confirm your hypothesis with a before-and-after timing experiment.

3. Common Optimization Techniques

A. Memory and Cache Optimization

Look here if: Cache misses and backend stalls are high, or larger inputs slow down disproportionately.

Memory accesses are expensive when data is not in cache. Try to reuse data while it is cached and avoid loading more than necessary.

- **Improve spatial locality:** Access consecutive memory locations. For a row-major array, traverse row by row.
- **Loop tiling (blocking):** Work on small chunks that fit in cache instead of repeatedly visiting a large array.
- **Loop fusion:** Combine passes over the same data to reduce repeated loads.
- **Improve data layout:** Keep commonly used fields together and avoid loading fields you do not need.
- **Reduce allocations:** Reuse buffers instead of repeatedly calling `malloc` and `free`.
- **Expose independent memory accesses:** Pointer chasing can create a long chain of dependent loads.

How to check: Benchmark several input sizes. A sharp slowdown near 64 KiB, 1 MiB, or 32 MiB (cache sizes on this system) is a clue that cache capacity matters.

The VM does not provide reliable DRAM bandwidth or memory latency counters, so input-size experiments are particularly useful.

B. Computation and Instruction-Level Parallelism (ILP)

Look here if: The loop executes many instructions, or backend stalls are high without many cache misses.

A CPU can work on independent instructions in parallel. If each instruction needs the previous result, the CPU may have to wait.

- **Remove redundant work:** Move loop-invariant calculations outside a loop.
- **Strength reduction:** Replace expensive operations with cheaper equivalent ones when correctness allows.
- **Loop unrolling:** Reduce loop-control overhead and expose independent work.
- **Independent accumulators:** Break a long dependency chain into multiple chains.
- **SIMD/vectorization:** Process multiple elements per instruction using NEON or SVE intrinsics.
- **Inlining:** Reduce the overhead of frequently called small functions.

Example: independent accumulators

A single accumulator creates a chain of dependent additions:

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

For an even `n`, two integer accumulators allow more independent work:

```
int sum0 = 0, sum1 = 0;

for (int i = 0; i < n; i += 2) {
    sum0 += a[i];
    sum1 += a[i + 1];
}
```

```
sum = sum0 + sum1;
```

This assumes the accumulator type has appropriate overflow behavior and that the surrounding code handles initialization. **Do not use this transformation for floating-point reductions in this lab:** changing the addition order can change the bits of the result.

How to check: Open the hot loop in `perf annotate`. Look for costly instructions and dependency chains. Check whether the compiler generated scalar or vector instructions.

Inspect generated assembly to check whether automatic vectorization occurred.

C. Branch Prediction

Look here if: `branch-mpki` (branch mispredictions per thousand instructions) is high.

The CPU predicts branches to keep working. When a prediction is wrong, it discards speculative work and recovers. Unpredictable branches can therefore be costly.

- **Branchless operations:** Replace conditionals with arithmetic or conditional selection when beneficial.
- **Lookup tables:** Replace repeated comparisons with indexed lookups.
- **Loop unrolling:** Reduce the number of loop-control branches.
- **Simplify conditionals:** Remove redundant checks or reorganize decision trees.

How to check: Use `perf annotate` to identify expensive branches. Compare different inputs: if mispredictions fall for a uniform image, branch behavior probably depends on the data.

The counter can include speculative, wrong-path branches. **Do not blindly remove every if statement.**

D. Instruction Fetch and Frontend

Look here if: Frontend stalls rise after inlining or unrolling.

The CPU must fetch and decode instructions before executing them. Large loops and functions can hurt instruction-cache locality.

- **Reduce code size:** Avoid excessive unrolling or inlining.
- **Keep hot code together:** Improve instruction locality.
- **Simplify control flow:** Arrange common paths to avoid unnecessary jumps.

How to check: Compare assembly and code size before and after the change. More unrolling is not always faster.

For these relatively small kernels, frontend problems are less likely than memory or computation bottlenecks.

E. Multicore Performance

Look here if: Adding threads gives little speedup, or performance gets worse with more threads.

More threads share hardware resources and may interfere with one another.

- **Load imbalance:** Some threads finish early. Divide work into smaller tasks.
- **Synchronization:** Threads spend time waiting for locks or shared state. Reduce contention.
- **False sharing:** Threads update different variables on the same cache line. Try per-thread storage or padding.
- **Memory bandwidth:** All cores compete for bandwidth. Improve locality before adding threads.
- **Serial work:** The nonparallel portion limits total speedup (Amdahl's Law).

How to check: Compare one-thread and four-thread runs. Read APerf's per-CPU measurements to see whether any cores are idle.

For coherence, you can investigate raw event `0x58` (L2 invalidations in Arm's standard numbering). Compare invalidations per instruction at one thread versus four. APerf reports system-wide events, whereas `perf` with `/u` limits the count to user-space events; compare like with like.

F. Other Sources of Slowdowns

Look here if: Your program has occasional slow runs or time is spent outside the main loop.

- **Allocation overhead:** Repeated `malloc` and `free` calls.
- **Page faults:** Memory pages allocated or mapped on demand.
- **Alignment:** Loads that cross cache-line boundaries.
- **Memory dependencies:** Loads or stores waiting on other memory operations.
- **VM variability:** Scheduling, frequency, or interference from other workloads.

To count page faults:

```
perf stat -e page-faults ./your_program
```

Use `perf report` to find expensive allocation or setup code.

Some architectural techniques, such as flush-to-zero floating-point behavior, are **not allowed** because they may

change the output.

4. Common Mistakes

Optimizing code that barely matters. Find the bottleneck first. Amdahl's Law limits gains from small functions.

Treating IPC as the goal. IPC helps diagnose performance, but the actual objective is lower *total runtime*.

Changing several things at once. When you change one thing at a time, you can tell what helped.

Trusting a single measurement. VM timings are noisy. Run multiple trials and compare the spread of results.

Treating unvalidated counters as fact. Some TLB and L2/L3 metrics are unavailable or unvalidated. Missing data is not a measured zero.

Skipping correctness checks. Verify bit-exact output after each change, especially around floating-point arithmetic.

5. Final Checklist

- ☐ Did I identify a real bottleneck?
- ☐ Do I understand why my change should help?
- ☐ Did I change only one thing?
- ☐ Is the output still bit-exact?
- ☐ Did I benchmark multiple times?
- ☐ Is the improvement larger than measurement noise?
- ☐ Did overall runtime decrease?

Bottom line: Profile → identify a likely cause → make one targeted change → verify correctness → measure again.