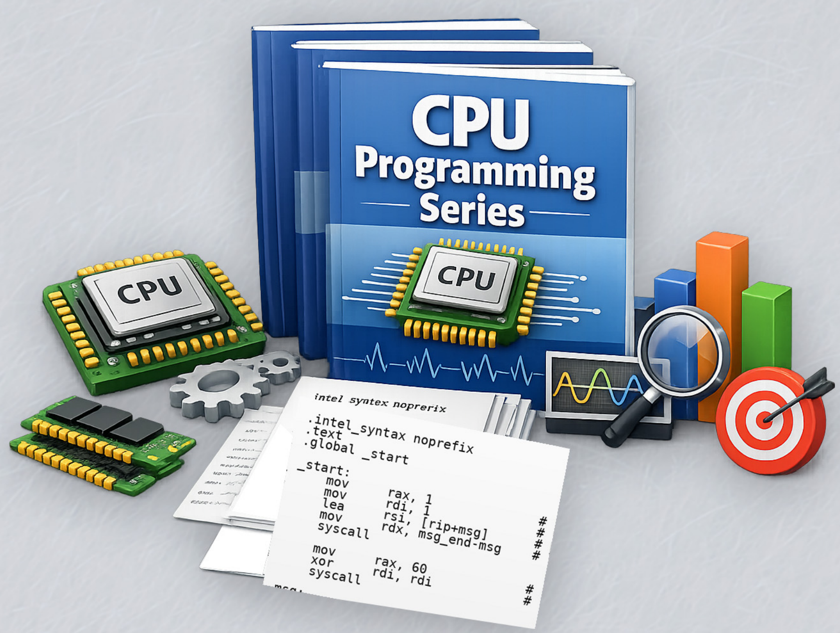
https://simplifycpp.org

CPU Programming Series

Memory, Caches, and the Cost of Access

Why Assembly Performance Is Usually About Memory



```
intel syntex noprerix

.intel_syntax noprefix
.text
.global _start
_start:
    mov     rax, 1
    mov     rdi, 1
    lea     rsi, [rip+msg]
    mov     rdx, msg_end-msg
    syscall

    mov     rax, 60
    xor     rdi, rdi
    syscall
```

4

Prepared by Ayman Alheraki

CPU Programming Series

Memory, Caches, and the Cost of Access

Why Assembly Performance Is Usually About Memory

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Preface — Why Memory Dominates Performance

Why Instruction Speed Is No Longer the Bottleneck

On modern CPUs, raw arithmetic and simple integer operations are typically far cheaper than fetching the data they operate on. Contemporary cores execute multiple instructions per cycle, overlap independent work, and keep deep pipelines busy through prediction and out-of-order execution. In contrast, data that is not already close to the core must travel through the memory hierarchy, and each step away from registers increases latency and reduces effective throughput. As a result, many real workloads are limited not by how fast the CPU can *compute*, but by how efficiently the program can *access* and *reuse* data.

The Illusion of "Fast Code" Without Memory Awareness

Many "optimizations" focus on instruction counts, clever arithmetic, or micro-tuning a loop body. These changes can look impressive in isolation but deliver little improvement when the program is **memory-bound**. In memory-bound code, the dominant cost is waiting for cache lines and translations, not executing the next instruction. A small reduction in instructions cannot compensate for:

- cache misses that stall dependent work,
- poor locality that defeats reuse,
- pointer-chasing patterns that serialize access,
- working sets that overflow cache capacity,
- or access patterns that waste bandwidth.

This booklet builds the mental model needed to detect when “faster instructions” are irrelevant and when data layout and access patterns are the real levers.

What This Booklet Explains — and What It Deliberately Excludes

This booklet explains the performance-critical concepts that apply across architectures:

- the registers $\rightarrow$ caches $\rightarrow$ RAM hierarchy as an execution reality,
- cache lines as the unit of transfer and the meaning of locality,
- a practical latency model for L1/L2/L3 and main memory (conceptual, not vendor-specific),
- TLB basics and why address translation can become a bottleneck,
- false sharing as a cache-coherence performance hazard (conceptual),
- access-pattern reasoning (sequential, strided, random, pointer chasing).

This booklet deliberately excludes:

- **atomics** and **memory ordering**,
- **memory barriers/fences**,
- **SIMD/vectorization**.

These topics require additional contracts and mechanisms. The goal here is to master the baseline: memory hierarchy behavior and the cost model of access.

How to Read Performance Discussions Correctly

To read performance claims with engineering rigor, apply these rules:

- **First classify the bottleneck:** is the workload compute-bound or memory-bound?
- **Think in units the hardware moves:** cache lines and pages, not individual variables.
- **Prefer evidence over intuition:** speedups must be tied to fewer misses, better locality, less bandwidth waste, or truly reduced dependency chains.
- **Beware single-number explanations:** "CPU is fast" or "RAM is slow" is not actionable; identify which level (L1/L2/L3/TLB/RAM) dominates.
- **Respect context:** results depend on working set size, access pattern, and data layout; a micro-benchmark may not represent the real program.

A Minimal Mental Checklist

Before believing an optimization, ask:

```
1) What data is being touched, and how often is it reused?
2) Does access have locality (spatial/temporal), or is it effectively
 ↪  random?
```

```
3) Does the working set fit in cache, or does it spill?
4) Are we limited by latency (stalling) or bandwidth (streaming)?
5) Could the same work be faster by changing layout/pattern, not
 ↪  instructions?
```

This booklet equips you to answer these questions without relying on platform-specific trivia, and to reason about performance using the same mental model across x86-64, ARM64, and RISC-V.

Chapter 1

From Registers to RAM: The Execution Reality

1.1 The Modern Memory Hierarchy

Modern CPUs execute instructions near the core, but the data they need is stored across a hierarchy of storage levels, each trading capacity for speed. Conceptually, the hierarchy is:

- **Registers:** the closest storage to execution units, extremely fast, very small.
- **Caches (L1/L2/L3):** fast on-chip storage that holds recently used data and instructions in cache-line units.
- **Main memory (RAM):** much larger but far higher latency and lower effective bandwidth per core than on-chip storage.

The CPU continuously moves data between these levels. Performance depends on how often the working set can be satisfied by the upper levels rather than by RAM.

1.2 Why Registers Exist

Registers exist because CPUs need immediate, low-latency operands to keep pipelines busy. They provide:

- **Single-cycle (or near) access** to values needed by arithmetic, address generation, and control logic.
- **High bandwidth** to feed multiple execution units in parallel.
- **A stable calling/ABI interface** for passing arguments, returning values, and preserving state efficiently.

If every operand access required memory, instruction throughput would collapse. Registers are the core enabler of fast execution.

1.3 Why RAM Is Slow (Relative, Not Absolute)

RAM is not "slow" in isolation; it is slow **relative to the CPU**. The gap comes from physics and system design:

- **Distance and signaling:** RAM is off-core (often off-chip), requiring longer paths and complex signaling compared to on-chip storage.
- **Protocol overhead:** memory access involves controllers, queues, scheduling, and row/bank management.
- **Latency dominance:** the time to fetch a single cache line from RAM is large compared to a CPU cycle, even if peak bandwidth is high.

Thus, a core can execute many instructions in the time it waits for one miss that must be serviced from main memory.

1.4 Cost Gaps Between Hierarchy Levels

The hierarchy exists because no single storage level can simultaneously be **very fast**, **very large**, and **low power/cost**.
Key cost-gap consequences:

- **A cache hit vs a cache miss** is often the dominant performance difference in tight loops.
- **Moving one cache line** from a lower level can stall dependent execution even if the CPU is otherwise capable of high throughput.
- **Latency is not uniform:** L1, L2, L3, and RAM represent progressively larger delays and different bandwidth constraints.

For performance reasoning, what matters is not the exact cycle numbers, but the *orders of magnitude* difference: upper levels are close enough to sustain instruction throughput; RAM is not.

1.5 Performance as a Data-Movement Problem

In many real programs, the limiting factor is not arithmetic, but **data movement**:

- If the working set fits and reuses well, caches supply most accesses quickly.
- If the working set is large or access is effectively random, misses force frequent fetches from lower levels, and the CPU spends time waiting.
- Instruction-level optimizations cannot overcome frequent long-latency misses; improving locality and layout often yields larger gains.

A practical way to think about performance is:

> Compute is cheap; fetching the next needed cache line is expensive.

1.6 Conceptual Summary

- The memory hierarchy exists to bridge the speed gap between cores and RAM.
- Registers enable high-throughput execution by providing immediate operands.
- RAM is slow relative to CPU cycles because of distance, protocol, and latency.
- The largest performance gaps often come from where data is served from (hit vs miss).
- Many workloads are best optimized by reducing data movement, not by reducing instruction count.

Chapter 2

The Mental Model: Registers → Cache → Memory

2.1 Conceptual View of the Hierarchy

The memory hierarchy should be understood as a **progressive distance from the execution core**. Each level exists to satisfy access requests when closer levels cannot:

- **Registers** provide immediate operands to execution units.
- **Caches** hold recently and nearby used data in cache-line granularity.
- **Main memory** supplies data when it is no longer resident on-chip.

At runtime, the program never "chooses" the level explicitly. The hardware resolves accesses automatically, but the *cost* depends entirely on where the data is found. Effective performance comes from structuring code so that most accesses are resolved at the upper levels.

2.2 Data Movement vs Computation

Modern CPUs can perform many arithmetic and logical operations per cycle, often far exceeding the rate at which new data can be delivered from memory. As a result:

- **Computation** is usually cheap once operands are available.
- **Data movement** dominates when operands are not already in registers or cache.

In practical terms, a loop with minimal arithmetic can be slower than a loop with heavier computation if the first triggers frequent cache or TLB misses. Performance analysis must therefore focus on *where data comes from*, not just what operations are performed.

2.3 Why CPUs Speculate and Prefetch

Because memory latency is high relative to CPU speed, modern processors attempt to **predict future accesses** and fetch data before it is explicitly requested. Two key mechanisms exist:

- **Speculative execution:** the CPU executes instructions along predicted paths to overlap useful work with potential waiting time.
- **Prefetching:** hardware detects access patterns and proactively loads cache lines expected to be needed soon.

These mechanisms do not make memory faster; they attempt to *hide latency* by ensuring data arrives before it becomes a blocking dependency. When predictions are correct, execution proceeds smoothly. When they fail, the cost of misprediction is paid.

2.4 What “Access” Actually Means in Hardware Terms

An “access” in source code (for example, reading a variable) expands into a sequence of hardware events:

- address calculation and translation (via the TLB),
- cache lookup at one or more levels,
- potential cache line fill from a lower level,
- update of cache state and coherence metadata.

Only after these steps does the value become available to the execution unit. The programmer-visible operation is simple, but the underlying cost varies widely depending on which levels are involved and whether the access can be overlapped with other work.

2.5 Why Latency Hiding Exists

Latency hiding exists because waiting for memory directly would waste most of the CPU’s execution capacity. Instead, CPUs attempt to overlap latency with useful work:

- out-of-order execution runs independent instructions while waiting,
- speculation keeps pipelines active,
- prefetching reduces the chance of a stall when data is needed.

Latency hiding has limits. When the program’s dependency structure is tight (for example, pointer chasing), or when working sets exceed cache capacity, latency becomes exposed and performance collapses to the speed of data delivery.

2.6 Conceptual Summary

- The hierarchy reflects increasing distance from the core, not just different memories.
- Performance is often constrained by data movement, not computation.
- CPUs speculate and prefetch to overlap work with memory latency.
- A single source-level access can trigger complex hardware activity.
- Latency hiding helps only when sufficient independent work and locality exist.

Chapter 3

Cache Lines: The Unit of Transfer

3.1 What a Cache Line Represents

A **cache line** is the fixed-size block of memory that the cache moves and stores as a unit. When the CPU needs a byte or a word that is not already in a cache, it does not fetch only that item; it fetches the *entire cache line* containing it. Conceptually, a cache line is:

- the minimum granularity of data movement between cache levels and memory,
- the unit used for caching decisions (hits, misses, replacement),
- and the unit of coherence tracking when multiple cores share memory.

The exact line size is platform-specific, but the key model is universal: **memory travels in lines, not in individual variables**.

3.2 Why CPUs Do Not Load Single Bytes

Fetching single bytes from main memory would be inefficient because:

- **Fixed access overhead:** memory access has protocol and latency costs that dominate the cost of transferring a few bytes.
- **Bandwidth efficiency:** transferring a larger contiguous block amortizes overhead and better uses the memory bus.
- **Locality exploitation:** programs often access nearby addresses soon after one another; bringing a full line increases the chance that subsequent accesses hit in cache.

Caches exist specifically to convert expensive, high-latency memory accesses into fewer, larger transfers that can be reused many times at low latency.

3.3 Spatial Locality Explained

Spatial locality means: if a program accesses address X, it is likely to access nearby addresses $X + \delta$ soon. Cache lines leverage this by bringing contiguous data together. Practical consequences:

- Iterating through arrays sequentially often performs well because adjacent elements share cache lines.
- Structures-of-arrays vs arrays-of-structures trade locality depending on which fields are accessed.
- Random access patterns defeat spatial locality and convert most accesses into cache misses or low reuse.

Spatial locality is not a theory; it is the fundamental reason cache lines are effective for common workloads.

3.4 Alignment and Cache Line Boundaries

Alignment affects how data maps onto cache lines:

- If an object fits entirely within one cache line and is aligned to avoid crossing a boundary, fewer line fills are needed.
- If an access spans a cache line boundary, it can require touching *two* lines, increasing traffic and potentially doubling the number of cache fills.

At the conceptual level, the important rule is:

> Crossing cache line boundaries increases the probability of extra transfers and extra misses.

This is especially relevant for frequently accessed objects, tight loops, and data structures where alignment can be controlled.

3.5 Wasted Bandwidth and Overfetching

Because caches move whole lines, the CPU may fetch data that is never used. This is **overfetching**. Overfetching wastes bandwidth and pollutes caches:

- **Bandwidth waste:** memory traffic increases without increasing useful work.
- **Cache pollution:** fetched but unused bytes occupy cache capacity, evicting data that *would* have been reused.
- **Miss amplification:** once useful data is evicted, future accesses miss more often, causing further traffic.

Overfetching is a common reason why code that “touches a little data” can still be slow: the hardware may be forced to move far more bytes than the program logically needs.

3.6 Conceptual Summary

- Cache lines are the unit of transfer and caching decisions.
- CPUs fetch lines, not single bytes, to amortize access costs and exploit locality.
- Spatial locality is the key property caches rely on to reduce average access cost.
- Alignment and boundary crossing influence how many lines must be touched.
- Overfetching wastes bandwidth and cache capacity, often dominating real performance.

Chapter 4

Temporal and Spatial Locality

4.1 Definition of Locality (Without Math)

Locality is the practical observation that program memory accesses are rarely uniform or random. Instead, programs tend to reuse the same data and to access nearby data in clusters. Caches exist because locality is common: if the hardware can keep recently used and nearby data close to the CPU, the average cost of access drops dramatically.
A good mental definition:

> Locality means that the next memory access is often related to the previous ones, either by reusing the same data or by accessing nearby addresses.

4.2 Temporal Locality: Reuse Over Time

Temporal locality means: if a program accesses an item now, it is likely to access the *same* item again soon. This is the foundation of caching.
Typical sources of temporal locality:

- loop variables and frequently used scalars,
- repeated updates to the same array region,
- hot metadata structures (sizes, counters, state flags),
- working sets that fit in cache across iterations.

When temporal locality is strong, cache hits increase and the effective memory latency seen by the core decreases.

4.3 Spatial Locality: Reuse Over Space

Spatial locality means: if a program accesses address X, it is likely to access addresses near X soon. Cache lines exploit this by fetching contiguous blocks.
Typical sources of spatial locality:

- sequential iteration over arrays and contiguous buffers,
- traversing packed structures where the needed fields are near each other,
- processing data in blocks (tiling) that stay close in memory.

Spatial locality is a major reason contiguous data structures often outperform pointer-based ones for the same logical work.

4.4 How Compilers Try to Exploit Locality

Compilers cannot change the fundamental algorithm, but they can apply transformations that often improve locality or reduce memory traffic when legality allows. Conceptually, common strategies include:

- **Inlining:** reduces call overhead and can expose larger regions for optimization, sometimes improving register reuse.
- **Loop optimizations:** unrolling, fusion, interchange, and blocking (when applicable) to increase reuse and reduce redundant loads.
- **Scalar replacement and promotion:** keeping frequently used values in registers rather than reloading from memory.
- **Strength reduction and common subexpression elimination:** reducing repeated address computations and redundant loads.

Important limitation: compilers cannot reliably fix poor data layout or fundamentally random access patterns. Data structure choice and access order are often the dominant factors.

4.5 When Locality Assumptions Fail

Locality fails when access patterns prevent reuse or prevent contiguous fetching from being useful. Common failure modes:

- **Working set exceeds cache capacity:** data is evicted before it can be reused, destroying temporal locality.
- **Pointer chasing:** linked structures and irregular graphs often force dependent loads that serialize progress and reduce spatial locality.
- **Large strides:** accessing every kth element can touch many cache lines while using only a small fraction of each line.
- **Unpredictable access:** input-dependent indexing patterns can defeat hardware prefetching and speculation.

- **Overfetch and pollution:** bringing in lines that are not reused can evict useful lines, reducing effective locality even when the code seems simple.

When locality fails, performance collapses toward the cost of lower memory levels, and instruction-level micro-optimizations become largely irrelevant.

4.6 Conceptual Summary

- Locality is the reason caches work: programs reuse data and access nearby data.
- Temporal locality is reuse of the same data over time; spatial locality is reuse of nearby data over space.
- Compilers can improve some locality through legal transformations and register reuse, but cannot repair fundamentally poor access patterns.
- Locality fails when working sets overflow caches, access is irregular, or cache lines are wasted and polluted.

Chapter 5

Cache Levels and Latency Model

5.1 Purpose of Multi-Level Caches

A single cache cannot be simultaneously **very fast**, **very large**, and **efficient in power/area**. Multi-level caches exist to balance these competing constraints:

- Keep the **most frequently used** data in the **closest and fastest** storage.
- Provide **larger fallback capacity** without forcing every access to pay the latency of main memory.
- Reduce average memory access time by ensuring most accesses are satisfied above RAM.

The hierarchy is a practical engineering compromise: small and extremely fast near the core, larger and slower as you move outward.

5.2 L1 vs L2 vs L3 (Conceptual Roles)

Although details vary by platform, the conceptual roles are stable:

- **L1 cache:** the first on-chip cache level, optimized for **minimal latency**. It is typically small and designed to feed the core at very high bandwidth.
- **L2 cache:** a larger, slower cache that reduces the miss rate seen by L1. It is a **capacity and filtering layer** that prevents many L1 misses from reaching deeper levels.
- **L3 cache (last-level cache):** the largest on-chip cache level, optimized for **capacity and sharing**. It reduces how often the system must go to RAM and often acts as a shared reservoir for multiple cores.

The general pattern:

> Higher levels prioritize latency; lower cache levels prioritize capacity.

5.3 Latency Differences as Execution Delays

From the perspective of a CPU core, latency is experienced as **delay in producing a needed operand**. If an instruction depends on data that is not ready, the core must wait or find independent work to execute.
Key points:

- A hit in a nearer cache level supplies data quickly, allowing dependent instructions to proceed.
- A miss forces the request to be serviced by a deeper level, increasing the time before the value becomes usable.

- Even with out-of-order execution, latency becomes visible when the dependency chain is tight or when many misses occur.

Thus, cache latency is not abstract; it directly translates into stalled cycles when the program cannot proceed without the missing data.

5.4 Why “Cache Miss” Is Not a Single Thing

A “cache miss” is often discussed as one event, but conceptually it can mean several different situations with very different costs:

- **L1 miss but L2 hit:** moderate delay; data is still on-chip.
- **L2 miss but L3 hit:** larger delay; still on-chip but farther and more contended.
- **Last-level miss:** data must come from RAM, producing the largest delay.

Additionally, misses can arise for different reasons (capacity pressure, conflicts, or first-touch). The important model is that “miss” is a **level-dependent event**, not a single universal cost.

5.5 Cost Amplification Across Levels

The hierarchy amplifies cost as requests fall through levels:

- Each deeper level increases latency and often reduces per-core effective bandwidth.
- A miss at a higher level can cause multiple downstream actions: filling a cache line, updating metadata, and possibly evicting an existing line.
- Evictions can trigger further traffic (for example, if modified data must be written back), increasing the effective cost beyond the original miss.

This amplification is why small changes in locality or working set size can create **large** performance swings: once the working set stops fitting in an upper cache, the program begins paying the cost of deeper levels repeatedly.

5.6 Conceptual Summary

- Multi-level caches balance latency, capacity, and cost.
- L1 is latency-first, L2 reduces L1 misses via more capacity, and L3 reduces RAM traffic and supports sharing.
- Latency differences appear as execution delays when operands are not ready.
- "Cache miss" is level-dependent; an L1 miss may still be an on-chip hit.
- Costs amplify across levels due to longer latency, reduced bandwidth, and eviction side effects.

Chapter 6

Cache Misses and Their Real Cost

6.1 Cold, Capacity, and Conflict Misses (Conceptual)

A **cache miss** occurs when the requested data is not present in the cache level being queried, forcing the request to be satisfied from a deeper level. Conceptually, misses are commonly classified as:

- **Cold (compulsory) miss:** the first time a cache line is accessed, it cannot already be in that cache. Cold misses are unavoidable for first-touch data, but their impact can be reduced by increasing useful work per fetched line.
- **Capacity miss:** the working set (the set of actively needed cache lines) exceeds the cache's effective capacity, so lines are evicted before they can be reused. Capacity misses indicate insufficient locality or an oversized working set for the cache level.
- **Conflict miss:** even if the working set could fit in capacity, cache placement constraints cause repeated evictions because multiple frequently used lines map to competing locations. Conflict misses are strongly influenced by access patterns and alignment.

These labels describe *why* a line is missing. The performance cost is determined by *how far down the hierarchy the request must go* to refill it.

6.2 Miss Penalties and Pipeline Impact

The **miss penalty** is the time until the requested cache line arrives from a deeper level and the load can be satisfied. The direct pipeline impact depends on dependencies:

- If later instructions depend on the loaded value, execution of those dependent instructions must wait.
- Out-of-order execution can continue with independent work, but only until it runs out of independent instructions or resources.
- When independence is exhausted, the core experiences a **stall** that exposes the full miss latency.

Thus, the penalty is not merely "slower memory"; it is a delay that can propagate through dependent instruction chains.

6.3 Why a Single Miss Can Stall Many Instructions

A single cache miss can stall many instructions because modern code is often structured around **critical dependencies**:

- Address dependencies: a pointer load is needed before the next address can be computed (pointer chasing).
- Data dependencies: the loaded value participates in arithmetic or in a branch decision.

- Control dependencies: a branch decision depends on a value that is not yet available, delaying correct-path execution.

Additionally, misses consume internal resources (load buffers, miss status handling structures, queues). If many misses accumulate, the core can become **memory-level parallelism limited**, meaning it cannot issue more outstanding misses efficiently, and stalling increases.

6.4 Why Misses Dominate Tight Loops

Tight loops often perform a small number of instructions per iteration. If each iteration triggers a miss (or insufficient reuse), the loop becomes dominated by waiting time:

- The instruction body is too small to hide latency.
- The same dependency chain repeats each iteration.
- The CPU may reach peak instruction throughput, yet overall progress is bounded by data arrival.

This is why "optimized assembly" can still be slow: if each iteration needs a new cache line from a lower level, the loop runs at the speed of cache line delivery, not at the speed of arithmetic.

6.5 Memory Stalls vs Instruction Throughput

It is essential to distinguish:

- **Instruction throughput:** how many instructions the core can execute per unit time when operands are ready and dependencies allow parallelism.

- **Memory stalls:** cycles where execution is limited by waiting for operands to arrive from the memory hierarchy.

A program can have excellent instruction throughput and still be slow if it spends significant time stalled on memory. Conversely, a program with more arithmetic per cache line can run faster because it increases work per fetched data and improves effective utilization of the core.

6.6 Conceptual Summary

- Cold, capacity, and conflict misses describe different reasons data is absent from cache.
- Miss penalties become visible when loads lie on the critical dependency path.
- One miss can stall many instructions due to dependency chains and limited memory-level parallelism.
- Tight loops are often dominated by misses because they cannot hide latency.
- Real performance is frequently limited by memory stalls rather than instruction throughput.

Chapter 7

Access Patterns and Performance

7.1 Sequential Access Patterns

Sequential access means touching memory locations in increasing (or decreasing) contiguous order. This pattern is typically the most cache-friendly because it aligns with how hardware moves data:

- Cache lines bring adjacent data together, so multiple consecutive accesses often hit after one line fill.
- Hardware prefetchers can often detect sequential streams and fetch future lines early.
- Translation costs (TLB) are amortized because many accesses fall within the same page range before moving on.

Sequential access tends to convert expensive memory operations into a streaming pattern where performance is limited primarily by cache or memory bandwidth rather than by exposed latency.

7.2 Strided Access Patterns

Strided access means accessing elements at a fixed step size (stride), such as every kth element. Stride changes the fraction of each cache line that is actually used:

- Small strides (within a cache line) can still exploit spatial locality.
- Large strides often touch one element per cache line, wasting most of the fetched bytes (overfetching).
- Certain strides can cause repeated conflicts in caches due to placement constraints, increasing conflict misses.

Strided access can also confuse or partially defeat hardware prefetching depending on stride regularity and magnitude. The conceptual rule is:

> As stride grows, useful bytes per cache line often shrink, and effective bandwidth is wasted.

7.3 Random Access Patterns

Random access means addresses appear effectively unpredictable and widely scattered relative to cache line and page structure. This pattern is typically the most expensive because it defeats the main mechanisms that make caches effective:

- Spatial locality is weak, so each access may require a new cache line.
- Temporal locality is often weak, so lines are evicted before reuse.
- Prefetchers cannot reliably predict future addresses, exposing full latency.
- TLB misses become more likely because accesses spread across many pages.

Random access often becomes **latency-bound**: each access waits for data rather than streaming efficiently.

7.4 Pointer Chasing and Linked Structures

Pointer chasing is a special case of access that is both irregular and dependency-serialized. The next address depends on the value loaded from the current address (for example, following a linked list):

- Each load must complete before the next address is known.
- This limits memory-level parallelism: the core cannot easily have many independent misses in flight.
- Out-of-order execution cannot hide latency because the dependency chain is inherently sequential.

This is why linked structures are often much slower than contiguous arrays for the same logical traversal. The cost is not only cache misses; it is the **inability to overlap misses** due to address dependencies.

7.5 Why Algorithms With the Same Complexity Differ Massively in Speed

Asymptotic complexity describes growth with input size, but it does not capture constant factors dominated by memory behavior. Two $O(n)$ algorithms can differ by large factors because of:

- **Working set behavior:** whether active data fits in cache or spills to lower levels.

- **Locality:** whether accesses reuse cache lines efficiently or waste most fetched bytes.
- **Bandwidth vs latency:** streaming contiguous data can be bandwidth-limited and fast; scattered data can be latency-limited and slow.
- **Dependency structure:** pointer chasing serializes access and prevents latency hiding.
- **Translation overhead:** scattered accesses increase TLB pressure and page working set size.

Therefore, performance engineering must consider:

> Not only how many operations you do, but how your data is laid out and how you touch it.

7.6 Conceptual Summary

- Sequential access is usually fastest because it matches cache lines and prefetching.
- Strided access can waste cache lines and trigger conflicts as stride grows.
- Random access defeats locality and prediction, exposing high latency and TLB costs.
- Pointer chasing is especially slow because address dependencies serialize memory access.
- Big performance gaps between same-complexity algorithms often come from locality, working set size, and latency hiding limits.

Chapter 8

The Translation Lookaside Buffer (TLB)

8.1 Virtual Memory Recap (Conceptual)

Modern systems use **virtual memory** to present each program with a contiguous address space while mapping it onto physical memory managed by the OS and hardware. Conceptually:

- Programs generate **virtual addresses**.
- Hardware translates these to **physical addresses** using page-based mappings.
- Memory protection, isolation, and flexible placement are enforced through this translation.

The key point for performance is that *every memory access requires address translation* before the cache or memory can be accessed.

8.2 What the TLB Does

The **Translation Lookaside Buffer (TLB)** is a small, fast cache that stores recent virtual-to-physical address translations.
Its role is to:

- avoid repeating expensive page table walks,
- provide near-register-speed translation for common accesses,
- allow the cache hierarchy to operate using physical addresses efficiently.

When a translation is found in the TLB (*TLB hit*), address translation completes quickly. When it is not found (*TLB miss*), the hardware must consult page tables, incurring additional latency.

8.3 Why Address Translation Has a Cost

Address translation is not free because:

- Page tables are multi-level structures stored in memory.
- Walking them may require several dependent memory accesses.
- These accesses themselves are subject to cache and memory latency.

Even when page table entries are cached, translation introduces extra steps before the data access can proceed. The TLB exists to hide this cost for the common case.

8.4 TLB Locality and Working Set Size

Just like caches, the TLB benefits from **locality**:

- **Temporal locality:** repeated accesses to the same pages reuse cached translations.
- **Spatial locality:** accessing many addresses within the same page amortizes translation cost.

The **TLB working set** is the set of pages actively accessed over a period of time. If this set fits in the TLB, translation overhead is minimal. If it exceeds TLB capacity, translations are evicted and misses increase.

8.5 When TLB Misses Become Dominant

TLB misses become a dominant cost when:

- Access patterns touch many pages with little reuse.
- Large data structures are accessed sparsely or randomly.
- Strided or pointer-based access crosses page boundaries frequently.
- The data working set fits in cache but spans too many pages for the TLB.

In such cases, performance can degrade even if cache hit rates are high. The program becomes **translation-bound** rather than cache- or compute-bound.

8.6 Conceptual Summary

- Virtual memory requires translating every address from virtual to physical.
- The TLB caches translations to make this fast in the common case.
- Address translation has real cost due to page table walks and dependencies.
- TLB effectiveness depends on page locality and working set size.
- TLB misses can dominate performance when access spans many pages with little reuse.

Chapter 9

False Sharing: When Cores Fight Over Data

9.1 What False Sharing Really Is

False sharing is a performance failure mode where multiple cores repeatedly interfere with each other even though they are not logically sharing the same variable. The interference happens because caches track and transfer data at the **cache line** granularity. If two independent variables reside on the same cache line and different cores modify them, the entire line becomes a shared point of contention.

Key idea:

> False sharing is not about shared variables; it is about shared *cache lines*.

9.2 Cache Coherence at a High Level (No Atomics)

In a multi-core system, each core may hold cached copies of memory. **Cache coherence** is the hardware mechanism that ensures cores observe a consistent view of memory for shared data. Conceptually:

- When a core **reads** a cache line, it may keep a local cached copy.
- When a core **writes** to a cache line, it must gain the right to modify it and ensure other cached copies are updated or invalidated.
- Coherence operates at cache-line granularity, not per-variable.

This coherence traffic is essential for correctness, but it can become a major performance cost when lines bounce between cores due to frequent writes.

9.3 Why Logically Independent Data Can Collide

Two threads may be logically independent (each updates its own counter or slot), yet still collide if their data sits within the same cache line. This happens because:

- The hardware must treat the whole line as a unit for ownership and modification.
- A write to any byte of the line forces coherence actions for the entire line.
- If two cores alternate writes, the line repeatedly transfers or invalidates across cores, even though the threads never intend to share.

Therefore, *layout* creates sharing even when the program logic does not.

9.4 How False Sharing Destroys Scalability

False sharing is most damaging under increasing core count and write frequency:

- It introduces high coherence traffic that consumes interconnect bandwidth.
- It forces repeated invalidation/ownership transitions, delaying useful work.
- It creates stalls that scale with contention, so adding cores can make performance worse.

The hallmark symptom is poor scaling: a workload that should speed up with more threads stagnates or slows down because cores spend time synchronizing cache lines instead of executing.

9.5 Recognizing False Sharing Patterns

False sharing tends to appear in predictable structural patterns:

- **Per-thread counters in a contiguous array:** adjacent counters may share a cache line.
- **Small structs in arrays:** different threads update different fields/elements but those fields/elements share a line.
- **Work queues and ring buffers:** head/tail indices or flags placed close together.
- **Hot flags or state bytes:** many threads update small control variables stored densely.

Conceptual recognition rules:

- If independent threads write frequently to data that is **close in memory**, suspect false sharing.
- If scaling degrades primarily with **write-heavy** activity, suspect coherence contention.

- If separating or padding data improves scaling, the root cause is often cache-line collision.

9.6 Conceptual Summary

- False sharing is contention on cache lines, not on variables.
- Coherence keeps caches consistent but can generate expensive traffic for write-heavy patterns.
- Independent variables collide when they share a cache line and are modified by different cores.
- False sharing destroys scalability by forcing line bouncing and coherence stalls.
- Common patterns include adjacent per-thread data, densely packed flags, and shared control indices.

Chapter 10

Stack vs Heap vs Global Data (Cache View)

10.1 How Stack Access Behaves in Caches

Stack allocation typically exhibits strong locality because stack usage follows structured, predictable patterns:

- **Contiguity:** stack frames are laid out in contiguous memory regions, so nearby locals often share cache lines.
- **Temporal locality:** locals are frequently reused within a function or within a short call chain.
- **Predictability:** stack growth and access patterns are often regular, which helps hardware prefetch and cache behavior.

However, stack locality is not guaranteed for all cases:

- very large stack frames can exceed cache capacity and evict useful data,
- deep recursion or many active frames can expand the working set,
- spilling under register pressure can increase stack traffic.

The common outcome is that stack-resident data is often cache-friendly *relative to* scattered heap allocations, not inherently "fast by definition."

10.2 Heap Allocation and Locality Loss

Heap allocation often loses locality because allocations are driven by runtime events and allocator policies rather than by a structured frame layout:

- **Fragmentation:** objects that are logically related may be physically far apart, increasing cache misses and TLB pressure.
- **Pointer indirection:** heap-heavy designs frequently rely on pointer chasing, which serializes access and defeats latency hiding.
- **Allocation churn:** frequent allocate/free activity can scatter objects across pages, expanding the page working set.

Heap data can still be cache-efficient when it is allocated in contiguous blocks (arenas, pools, vectors, slabs) and traversed sequentially. The performance problem is not "heap" itself, but **uncontrolled layout and indirection**.

10.3 Global Data and Sharing Behavior

Global (static-storage) data often interacts with caches through **sharing and contention** characteristics:

- **Read-mostly globals** can be efficient because shared reads typically do not cause coherence bouncing.
- **Write-shared globals** can become severe bottlenecks due to coherence traffic, especially if many cores update the same cache line.
- **False sharing risk:** multiple global variables placed near each other can collide on cache lines and degrade scalability.

Global data is also long-lived and widely visible, which increases the chance it becomes part of the “hot” shared working set in multi-threaded programs.

10.4 Why Allocation Strategy Affects Performance

Allocation strategy shapes **layout**, and layout determines **locality**. Key effects:

- **Spatial locality:** allocating related objects contiguously increases useful bytes per cache line.
- **Temporal locality:** keeping frequently reused objects resident in cache is easier when they cluster and do not compete with unrelated data.
- **TLB behavior:** packing data into fewer pages reduces translation overhead and improves locality at the page level.
- **Coherence behavior:** separating per-thread write-heavy data prevents cache line bouncing and false sharing.

This is why two programs with identical algorithms can differ massively in speed: one places data to match cache lines and pages, the other scatters it.

10.5 Lifetime vs Locality Trade-Offs

Storage duration choices often trade simplicity and safety against locality:

- **Stack (automatic storage):** short lifetime, structured layout, often good locality, but limited size and scope.
- **Heap (dynamic storage):** flexible lifetime and size, but locality depends on allocator strategy and data structure design.
- **Global (static storage):** stable lifetime and easy sharing, but high risk of contention and false sharing if write-heavy.

A disciplined view:

- Choose lifetime first for correctness.
- Then shape layout for locality using contiguous storage, pooling, and careful separation of hot write-shared data.

10.6 Conceptual Summary

- Stack access is often cache-friendly due to contiguity, reuse, and predictability, but can degrade with large frames and heavy spilling.
- Heap access can lose locality due to fragmentation and indirection; it becomes efficient when allocations are contiguous and traversal is regular.
- Global data is strongly affected by sharing; read-mostly is cheap, write-shared can be catastrophic due to coherence.

- Allocation strategy changes layout, and layout controls cache lines, pages, and coherence behavior.
- Lifetime decisions should be made for correctness; locality should be engineered within those constraints.

Chapter 11

Reading Performance Without Tools

11.1 Mental Simulation of Cache Behavior

You can often predict performance trends without profiling by simulating what the hardware must do at a high level. Use a cache-line mental model:

- Memory moves in **cache lines**, not variables.
- A line is either **present** (hit) or must be **fetched** (miss) from a deeper level.
- Performance depends on **reuse**: how many useful operations occur per fetched line.

A practical mental simulation loop:

1. Identify the data structure touched per iteration.
2. Determine whether accesses are contiguous, strided, random, or pointer-chasing.
3. Estimate how many distinct cache lines are touched for a unit of work.
4. Ask whether those lines are reused soon enough to remain in cache.

11.2 Predicting Misses From Code Shape

Code shape often reveals likely miss behavior:

- **Sequential loops over arrays:** typically few misses per many operations because each fetched line serves multiple elements.
- **Large working sets:** if the loop touches more data than cache can hold before reuse, expect capacity misses.
- **Strided indexing:** if stride approaches or exceeds cache-line size, expect low line utilization and more misses.
- **Indirect indexing/pointer chasing:** expect irregular misses and exposed latency due to address dependencies.
- **Nested loops:** reuse depends on loop order; the wrong order often destroys locality.

This is not about exact cycle counts; it is about recognizing whether the program forces frequent fall-through to deeper levels.

11.3 Recognizing Memory-Bound Code

A workload is typically **memory-bound** when the CPU spends most of its time waiting for data rather than executing instructions. Conceptual indicators include:

- Low arithmetic intensity: few computations per byte fetched.
- Performance strongly depends on data size: fast for small inputs, dramatically slower once data exceeds cache.

- Large gains from improving locality or layout, and small gains from reducing instruction count.
- Dominance of loads/stores and address generation over arithmetic in the critical path.

A simple mental test:

> If speeding up arithmetic would not change how often you fetch new cache lines, the code is likely memory-bound.

11.4 Why Instruction Counts Lie

Instruction counts can be misleading because instructions do not have uniform cost. The dominant cost often comes from **stall time**:

- A program can execute fewer instructions yet run slower if it triggers more cache/TLB misses.
- A program can execute more instructions yet run faster if it increases reuse (more work per fetched line) and hides latency.
- Modern cores overlap many instructions; the true limiter can be data availability, not instruction throughput.

Therefore, “this version uses fewer instructions” is not a sufficient performance argument unless memory behavior is also improved.

11.5 Common Performance Misinterpretations

Several recurring misconceptions cause poor optimization decisions:

- **“The CPU is slow.”** Often false; the CPU is waiting for memory. The bottleneck is data delivery, not execution capability.
- **“Cache miss” is one cost.** A miss can be an L1 miss but an L2 hit, or it can fall to RAM. The cost depends on how far the request travels.
- **“My algorithm is $O(n)$ so it must be fast.”** Complexity ignores locality, working set size, translation overhead, and dependency structure.
- **“Small code changes cannot cause big slowdowns.”** Small changes can shift working set over a cache boundary, change alignment, or alter access order, causing large miss-rate changes.
- **“If it fits in RAM, it should be fine.”** Fitting in RAM is irrelevant to speed. What matters is fitting in cache (and in the TLB) with reuse.

11.6 A Practical Non-Tool Checklist

```
1) What is the working set (bytes touched before reuse)?
2) Is access sequential, strided, random, or pointer-chasing?
3) How many cache lines are touched per unit of work?
4) Is there reuse before eviction (temporal locality)?
5) Is the pattern predictable enough for prefetching?
6) Does performance collapse when data exceeds cache (memory-bound
 ↪  signature)?
```

11.7 Conceptual Summary

- You can reason about performance by simulating cache-line movement and reuse.

- Code shape often predicts whether misses are likely (sequential vs irregular patterns).
- Memory-bound code is dominated by data delivery; arithmetic optimizations yield limited benefit.
- Instruction counts lie when stall time dominates.
- Many misinterpretations come from ignoring cache levels, working sets, and translation costs.

Chapter 12

What This Booklet Deliberately Excludes

12.1 Why No Atomics

Atomics introduce a separate correctness contract: **concurrent access coordination**. They are not merely “faster operations”; they define visibility and ordering rules between threads. Understanding atomics correctly requires:

- a formal memory model (what can be observed and when),
- the difference between atomicity and ordering,
- and the interaction between language rules and hardware coherence mechanisms.

This booklet focuses on the baseline cost model of **ordinary memory access** (cache lines, locality, TLB, misses). Mixing atomics into that discussion would blur the boundary between:

- **performance mechanics** (how fast data arrives),
- and **concurrency correctness contracts** (what values are allowed to be observed).

12.2 Why No Memory Barriers

Memory barriers (fences) exist to constrain reordering and visibility across threads and devices. They are meaningful only in a context that includes:

- the language concurrency model,
- the allowed reorderings of loads and stores,
- and the architecture-specific ordering guarantees.

A barrier changes **what the hardware may overlap and reorder**, which directly affects performance. But without first mastering the non-concurrent memory hierarchy model, barrier effects are easily misunderstood as "cache tricks" or "speed controls." This booklet keeps barriers out to preserve a clean, teachable foundation.

12.3 Why No SIMD

SIMD (vectorization) changes performance through two coupled effects:

- **compute throughput:** more arithmetic per instruction,
- **memory behavior:** wider loads/stores, alignment constraints, and different access granularity patterns.

SIMD is most beneficial when the program already has:

- strong locality,
- predictable access patterns,
- and sufficient data parallelism.

If the code is memory-bound due to misses, scattered access, or poor locality, SIMD often yields limited gains. Therefore, SIMD is a later topic that should be built on top of the memory-hierarchy understanding developed here.

12.4 Separation of Concerns in Performance Learning

This booklet enforces a disciplined learning order:

1. **First:** understand the cost of access for ordinary loads/stores (hierarchy, cache lines, locality, misses, TLB).
2. **Then:** add concurrency ordering (atomics, barriers) as a correctness and performance layer.
3. **Then:** add vector execution (SIMD) as a throughput and alignment layer.

This separation prevents common failure modes:

- blaming performance on “slow CPU” when the problem is cache misses,
- misusing fences to “fix” performance or correctness without understanding ordering,
- assuming SIMD automatically speeds up memory-bound code.

12.5 What Comes Next in the Series

After mastering memory hierarchy and the cost model of access, the natural next steps are:

- **Concurrency foundations:** atomic operations, visibility, and ordering as defined by language and ABI constraints.

- **Memory ordering and barriers:** how to reason about reordering, synchronization, and their costs.
- **Vector memory access (SIMD):** alignment discipline, contiguous layout requirements, and throughput vs bandwidth limits.
- **Architecture-specific cache behavior:** mapping the concepts here to concrete platform rules and practical measurement.

This booklet is intentionally the **baseline**: it provides the universal mental model required to understand why higher-level performance techniques work, when they fail, and what they actually cost.

12.6 Conceptual Summary

- Atomics and barriers belong to the concurrency model and require separate ordering semantics.
- SIMD changes both compute throughput and memory access constraints and is effective only after locality is understood.
- Separation of concerns produces transferable understanding instead of platform-specific memorization.
- The next series stages build on this foundation: concurrency ordering, then SIMD and architecture-specific analysis.

Appendices

Appendix A — Practical Memory Performance Rules

12.6.1 Rules of Thumb That Actually Hold

The following rules are stable across modern CPUs because they follow from cache-line transfer, locality, and latency fundamentals:

- **Think in cache lines and pages, not variables.** If you touch one byte, you often pay for a whole cache line; if you spread across pages, you pay in TLB capacity.
- **Prefer contiguous data for hot paths.** Contiguous traversal maximizes spatial locality and makes prefetching effective.
- **Maximize reuse before eviction.** Improve temporal locality by reusing data while it is still in upper caches.
- **Reduce working set size.** The fastest memory is the one you do not touch. Smaller working sets increase cache residency and reduce TLB pressure.
- **Avoid pointer chasing in performance-critical loops.** Dependency-serialized loads expose latency and reduce memory-level parallelism.

- **Use predictable access patterns.** Sequential/regular patterns are easier for the hardware to prefetch; irregular patterns expose misses.
- **Separate hot and cold fields.** If only a few fields are frequently accessed, keep them together and move rarely used fields away to reduce cache line waste.
- **Avoid false sharing in multi-core code.** Do not let independent thread-written data share the same cache line; coherence traffic can dominate runtime.

12.6.2 When Micro-Optimizations Matter

Micro-optimizations matter primarily when the bottleneck is **compute throughput** or **front-end overhead**, not memory stalls.
They tend to matter when:

- the working set fits in cache and hit rates are high,
- the loop body is compute-heavy relative to bytes moved,
- there is sufficient locality and few cache/TLB misses,
- the code path is extremely hot (executed very frequently).

They tend **not** to matter when:

- performance collapses as data size exceeds cache,
- the program is dominated by cache misses, TLB misses, or pointer-chasing,
- the critical path is waiting for memory rather than executing instructions.

Discipline rule:

> If access patterns force frequent misses, optimize data movement before optimizing instruction sequences.

12.6.3 When Data Layout Beats Algorithms

Asymptotic complexity is not the full story for real performance. Data layout can dominate when memory behavior is the limiting factor.
Layout often beats algorithmic micro-changes when:

- two alternatives have the same big-O but different locality (contiguous vs scattered),
- the working set is near a cache or TLB capacity boundary,
- the access pattern wastes most bytes in each fetched cache line,
- the design introduces indirection (pointers) where direct indexing would be possible.

Practical effect:

- A "worse" algorithm with strong locality can outperform a "better" one with poor locality, because the latter is dominated by miss latency and translation overhead.

12.6.4 When Caches Help — and When They Don't

Caches help when locality exists:

- **They help** when accesses reuse the same lines (temporal locality) or access nearby bytes (spatial locality).
- **They help** when the working set fits or is close enough that replacement does not destroy reuse.
- **They help** when patterns are predictable enough for prefetching and overlap.

Caches do not help much when locality is weak:

- **They don't help** with random access over a large address range: each access touches a new line with little reuse.
- **They don't help** with pointer chasing: the next address depends on the last load, exposing latency.
- **They don't help** when the working set is far larger than cache capacity and reuse happens only after eviction.

A core mental model:

> Caches accelerate reuse; they cannot accelerate absence of reuse.

12.6.5 Operational Checklist

```
1) Identify the working set (bytes touched before reuse).
2) Identify the pattern (sequential / strided / random / pointer
 ↪  chasing).
3) Estimate cache-line utilization (useful bytes per fetched line).
4) Check reuse distance (will the line still be resident when
 ↪  reused?).
5) Watch for page spread (TLB working set).
6) In multi-core code, check if independent writers share cache
 ↪  lines.
```

12.6.6 Summary

- Stable performance rules follow from cache lines, locality, and working set size.
- Micro-optimizations matter mostly when code is not dominated by memory stalls.
- Data layout can dominate performance even when algorithmic complexity is unchanged.

- Caches help when reuse exists; they cannot rescue fundamentally non-local access.

Appendix B — Common Misconceptions

12.6.7 “Caches are automatic, I don’t need to think about them”

Caches are automatic in operation, but **not automatic in outcome**. Hardware transparently moves data, yet performance depends on access patterns the programmer creates.
Clarifications:

- Caches exploit **locality**; if locality is weak, caches cannot help.
- Hardware cannot infer program intent beyond observed address streams.
- Poor layout, large working sets, or irregular access defeat caching regardless of cache size.

Correct mental model:

> Caches work best when the program gives them something to exploit.

12.6.8 “My algorithm is optimal, so it must be fast”

Asymptotic complexity describes growth with input size, not constant factors dominated by memory behavior.
Why this fails in practice:

- Two algorithms with the same big-O can have radically different locality.
- Cache and TLB misses introduce costs not reflected in complexity analysis.
- Dependency structure (e.g., pointer chasing) can serialize execution regardless of instruction count.

Practical consequence:

- An algorithm with worse asymptotic complexity but strong locality can outperform an "optimal" one with poor memory behavior.

12.6.9 "The CPU is slow"

Modern CPUs are rarely slow at executing instructions. When programs underperform, the core is often **idle waiting for data**.
Common indicators:

- Performance collapses as data size exceeds cache.
- Increasing clock speed or instruction-level tuning yields little improvement.
- Reordering data or improving locality yields large gains.

Correct reframing:

> The CPU is fast; the program is waiting.

12.6.10 "RAM speed is the only thing that matters"

Main memory speed matters, but it is not the dominant factor for most workloads.
Why this is misleading:

- Most accesses are served from **caches**, not directly from RAM.
- Cache hit rates, cache-level residency, and TLB behavior often dominate observed performance.
- Faster RAM does not compensate for poor locality that forces frequent misses.

What actually matters:

- How often data is reused before eviction.
- How many useful bytes are consumed per cache line.
- How much access can be overlapped (latency hiding).

12.6.11 Correct Mental Model

- Caches are automatic mechanisms, not automatic optimizers.
- Algorithmic optimality does not imply cache efficiency.
- CPUs are usually waiting on memory, not limited by execution units.
- RAM speed is only one component; cache behavior and locality usually dominate.

Understanding these misconceptions prevents wasted effort on instruction tuning when the real bottleneck is data movement and layout.

Appendix C — Preparation for Advanced Topics

12.6.12 Readiness for Atomics and Memory Ordering

You are ready to study atomics and memory ordering when you can reason about performance *before* introducing concurrency semantics:

- You understand ordinary loads/stores as cache-line transfers with locality and miss costs.
- You can distinguish **latency-bound** behavior from **bandwidth-bound** behavior.
- You recognize that correctness (visibility and ordering) is a separate contract layered on top of cache behavior.

This preparation is critical because atomics do not merely add cost; they restrict reordering and visibility. Without a solid baseline memory model, atomic costs are often misattributed to "slow caches" rather than to enforced ordering constraints.

12.6.13 Readiness for SIMD and Vector Memory Access

SIMD effectiveness depends as much on memory behavior as on arithmetic width. You are ready for SIMD when you can already predict:

- whether data is contiguous enough to feed wide loads/stores efficiently,
- whether alignment and cache-line boundaries will amplify or reduce traffic,
- whether the workload is compute-bound or already memory-bound.

SIMD increases **work per access**. If memory access dominates due to misses or poor locality, wider computation alone yields limited benefit. This booklet ensures SIMD is approached as a *throughput multiplier*, not a substitute for locality.

12.6.14 Readiness for Multithreaded Cache Behavior

Before studying multi-threaded performance, the reader should already understand single-thread cache behavior well enough to identify what changes under sharing:

- You can reason about cache-line residency and eviction.
- You understand false sharing as cache-line contention, not as a logical bug.
- You can predict how write-heavy patterns increase coherence traffic.

With this foundation, multi-threaded topics become additive:

- coherence protocols add constraints,

- ownership transfer adds latency,
- contention amplifies costs already present in single-thread access.

12.6.15 Mapping Concepts to Architecture-Specific Manuals

Architecture manuals are precise but dense. This booklet provides the abstraction needed to read them effectively:

- **Cache sections:** map line size, associativity, and levels to locality and working set concepts.
- **Memory ordering sections:** interpret rules as constraints layered on top of the baseline access model.
- **Performance sections:** separate throughput limits from latency exposure.

When reading architecture-specific documents, focus on answering:

- What is the cache-line size and alignment requirement?
- What are the observable penalties for misses at each level?
- What additional constraints does concurrency or vectorization impose on access?

12.6.16 Preparation Checklist

```
1) Can I identify the working set and access pattern?
2) Can I predict cache-line reuse and eviction?
3) Can I tell whether latency or bandwidth dominates?
4) Can I recognize false sharing risks from layout?
5) Can I explain performance changes without referencing instructions
 ↪  first?
```

12.6.17 Summary

- Atomics and ordering require a solid baseline memory cost model.
- SIMD is effective only when memory access patterns are already favorable.
- Multithreaded cache behavior amplifies single-thread locality issues.
- Architecture manuals become readable when mapped onto cache lines, locality, and access costs.

This appendix marks the transition from **foundational memory reasoning** to advanced performance and concurrency topics built on the same principles.

References

All Resources Used for This Booklet

This booklet is based on widely accepted, long-standing, and authoritative sources in computer architecture, operating systems, and performance engineering. The core concepts (memory hierarchy, cache lines, locality, cache miss costs, TLB behavior, and multi-core cache effects such as false sharing) are stable across modern platforms and are consistently defined across these references.

Computer Architecture (Caches, Locality, Latency, Memory Hierarchy)

- John L. Hennessy, David A. Patterson — *Computer Architecture: A Quantitative Approach*
- David A. Patterson, John L. Hennessy — *Computer Organization and Design: The Hardware/Software Interface*
- Andrew S. Tanenbaum, Todd Austin — *Structured Computer Organization*

Systems Programming (Memory Behavior, Practical Performance, Cache Effects)

- Randal E. Bryant, David R. O'Hallaron — *Computer Systems: A Programmer's Perspective*
- Ulrich Drepper — *What Every Programmer Should Know About Memory*
- Agner Fog — *Optimizing Software in C++* and *The Microarchitecture of Intel, AMD and VIA CPUs*

Operating Systems and Virtual Memory Foundations (Paging, Translation, TLB Context)

- Abraham Silberschatz, Peter B. Galvin, Greg Gagne — *Operating System Concepts*
- Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau — *Operating Systems: Three Easy Pieces*
- Robert Love — *Linux Kernel Development* (for practical VM and caching context)

Processor and ISA Manuals (Cache-Line Reality, Memory System Details, Platform Semantics)

- Intel — *Intel 64 and IA-32 Architectures Software Developer's Manual* (memory hierarchy, caching, and performance-relevant behavior)
- AMD — *AMD64 Architecture Programmer's Manual* (system and memory-related architectural behavior)

- Arm — *Arm Architecture Reference Manual (A-profile)* (memory system and architectural definitions)
- RISC-V International — *The RISC-V Instruction Set Manual* (unprivileged and privileged specifications for memory/VM context)

Concurrency-Caching Context (False Sharing and Coherence at the Concept Level)

- Maurice Herlihy, Nir Shavit — *The Art of Multiprocessor Programming* (conceptual coherence and scalability pathologies)
- Standard computer architecture references listed above (coherence and cache-line ownership concepts)

How These References Map to This Booklet

- **Hierarchy and latency model:** architecture textbooks + vendor manuals (conceptual + architectural constraints).
- **Cache lines, locality, misses:** architecture textbooks + systems programming texts (mechanism + programmer-facing impact).
- **TLB and translation cost:** OS texts + architecture texts (virtual memory model + hardware translation reality).
- **False sharing and scalability:** architecture + multiprocessor references (cache-line coherence effects).

Scope Note

This booklet intentionally relies on stable fundamentals and broadly consistent definitions across authoritative sources. Architecture-specific numerical latencies and vendor-dependent microarchitectural details are treated as implementation variation and are reserved for architecture-specific study later in the series.