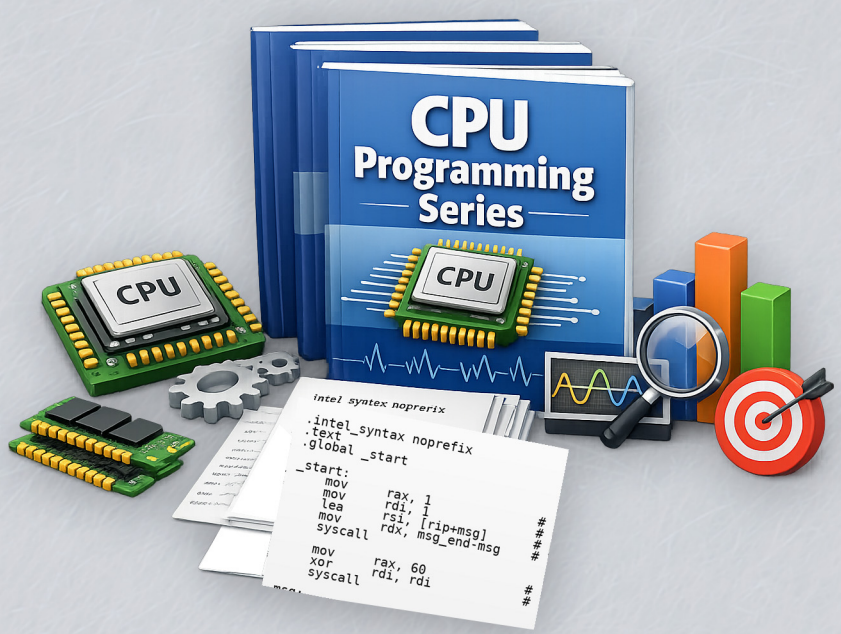
https://simplifycpp.org

CPU Programming Series

Branch Prediction & Speculation

Writing Control Flow That the CPU Likes



18

Prepared by Ayman Alheraki

CPU Programming Series

Branch Prediction & Speculation

Writing Control Flow That the CPU Likes

Prepared by Ayman Alheraki

simplifycpp.org

February 2026

Contents

Preface

Motivation and Objectives

Modern CPUs achieve high performance not by executing instructions faster in isolation, but by executing many instructions speculatively and in parallel. Among all microarchitectural mechanisms, **branch prediction and speculative execution** are the most critical — and the most misunderstood — sources of performance variation in real programs.

The primary objective of this booklet is to explain, with precision and clarity, how branch prediction works in modern processors, why mispredictions are costly, and how speculative execution shapes performance behavior far beyond what source code alone suggests. This booklet is not focused on compiler theory or abstract models; it is written from the perspective of how real CPUs behave and how programmers can align their control flow with that behavior.

The goal is to enable readers to reason about performance at the control-flow level, understand why seemingly small code changes can have large performance consequences, and write code that cooperates with the CPU rather than fighting it.

Target Audience

This booklet is intended for programmers and engineers who work close to performance-sensitive systems, including:

- Systems programmers working in C, C++, Rust, or similar languages
- Performance-oriented application developers
- Compiler and toolchain enthusiasts
- Embedded and low-level developers
- Readers studying CPU microarchitecture from a programmer's perspective

The content is written for readers who want to understand *why* performance behaves the way it does, not merely how to apply surface-level optimizations.

Assumed Background Knowledge

The reader is expected to have basic familiarity with:

- General CPU architecture concepts
- Instruction pipelines and execution stages
- Conditional branches and control flow
- Basic performance metrics such as cycles, latency, and throughput

Prior exposure to assembly language is helpful but not required. When assembly examples are used, they are presented to clarify behavior rather than to teach instruction syntax.

Scope and Limitations

This booklet focuses strictly on **branch prediction and speculative execution** as they relate to program behavior and performance. It deliberately avoids mixing concepts from unrelated domains such as vectorization, atomics, memory ordering models, or compiler internals unless they directly intersect with speculation.

The discussion is architectural and microarchitectural in nature. Exact implementation details may vary across CPU vendors and generations, but the principles described here apply broadly to modern out-of-order, superscalar processors.

Security topics related to speculation are discussed only to the extent necessary to understand architectural consequences and performance trade-offs, not as a security handbook.

Position Within the CPU Programming Series

This booklet is part of **Phase Five: Performance, Atomics, and SIMD** of the CPU Programming Series. Within that phase, it focuses exclusively on control-flow performance and speculative execution.

It builds upon concepts introduced in earlier booklets covering pipelines, out-of-order execution, and memory behavior, and prepares the reader for later booklets that address SIMD execution, atomic operations, and fine-grained performance tuning.

Each booklet in the series is designed to be read independently, but together they form a coherent progression from architectural fundamentals to advanced performance reasoning.

Chapter 1

Introduction to Branch Prediction

1.1 Control Flow as a Performance Bottleneck

On modern out-of-order superscalar CPUs, peak throughput depends on keeping the **frontend** (fetch, decode, branch prediction) continuously supplying the correct instruction stream to the **backend** (rename, schedule, execute, retire). Control flow disrupts this stream because the next instruction address depends on a runtime condition.
A conditional branch is effectively a question the CPU must answer immediately:

> *Which instruction address should be fetched next?*

If the CPU guesses incorrectly, all speculatively fetched and executed instructions on the wrong path are discarded, and the pipeline must restart from the correct target. When this happens frequently, performance becomes dominated by frontend recovery rather than useful computation.

1.1.1 Example: Same code, different performance due to predictability

```
#include <cstdint>
```

```cpp
#include <vector>

std::uint64_t count_positives(const std::vector<int>& a) {
    std::uint64_t c = 0;
    for (int x : a) {
        if (x > 0) {      // predictability depends on data distribution
            ++c;
        }
    }
    return c;
}
```

If the values in `a` are clustered (many positives followed by many negatives), the branch outcome becomes predictable and mispredictions are rare. If values are random or alternate frequently, mispredictions can dominate runtime even though the loop body performs minimal work.

1.2 Branch Instructions in Modern ISAs

Modern instruction set architectures provide multiple forms of control-flow instructions. While mnemonics and encodings differ, the prediction problem is largely identical across architectures:

- **Conditional branches** based on comparison results
- **Unconditional direct jumps** with encoded targets
- **Indirect branches** where the target is computed at runtime
- **Calls and returns** with structured control transfer

These categories stress different prediction mechanisms:

- Conditional branches require **direction prediction**
- Indirect branches require **target prediction**
- Returns rely on specialized return-target prediction

1.2.1 Example: Conditional and indirect control flow (x86-64)

```
.intel_syntax noprefix

/* conditional branch: direction prediction */
cmp  edi, 0
jg   positive

/* direct unconditional jump */
jmp  done

positive:
    /* ... */
done:

/* indirect branch: target stored in register */
mov  rax, rdi
jmp  rax
```

1.2.2 Example: Switch lowering to a jump table

Compilers often lower dense `switch` statements into jump tables, transforming direction prediction into target prediction:

```
int f(int x) {
```

```
    switch (x) {
        case 0: return 10;
        case 1: return 20;
        case 2: return 30;
        case 3: return 40;
        default: return -1;
    }
}
```

In such cases, performance depends heavily on how stable the runtime distribution of `x` is and how accurately indirect targets can be predicted.

1.3 Why CPUs Cannot Wait for Branch Resolution

Modern CPUs are both **deep** (many pipeline stages) and **wide** (multiple instructions issued per cycle). Waiting for each branch to resolve before fetching subsequent instructions would severely underutilize execution resources:

- The fetch and decode stages would frequently idle
- The backend would run out of ready instructions
- Overall IPC would degrade toward in-order execution levels

To avoid this, CPUs predict branch outcomes and continue fetching along the predicted path. Correctness is preserved because instructions only update architectural state when they retire in order.

1.3.1 Example: Branch resolution depends on late-arriving data

```
#include <cstddef>
#include <cstdint>
```

```
std::uint64_t count_threshold(const int* p, std::size_t n, int t) {
    std::uint64_t c = 0;
    for (std::size_t i = 0; i < n; ++i) {
        int x = p[i];           // may be delayed by cache or memory latency
        if (x >= t) ++c;        // branch cannot resolve until x is
        ↪  available
    }
    return c;
}
```

Even if the branch outcome is predictable, prediction is required to keep instruction fetch moving while the data dependency is resolved.

1.4 Prediction Accuracy vs Execution Speed

Branch prediction exists to maximize throughput, not to ensure correctness. Two competing constraints define predictor design:

- **Accuracy**: fewer mispredictions mean fewer pipeline flushes
- **Latency**: predictions must be available at fetch time

A highly accurate predictor that is too slow cannot be used in the critical fetch path. Conversely, a fast but inaccurate predictor causes frequent pipeline recovery. Modern CPUs therefore employ layered and hybrid predictors, balancing speed and accuracy.

1.4.1 Example: Predictable versus unpredictable branches

```
int clamp_nonnegative(int x) {
    // Often highly biased: negatives are rare in many workloads
```

```
    if (x < 0) return 0;
    return x;
}

std::uint64_t count_alternating(std::uint64_t n) {
    // Alternating pattern challenges simple predictors
    std::uint64_t c = 0;
    for (std::uint64_t i = 0; i < n; ++i) {
        if ((i & 1u) == 0) ++c;
    }
    return c;
}
```

The optimization goal is not to eliminate branches indiscriminately, but to minimize unpredictable branches in performance-critical paths.

1.4.2 Example: Branch versus conditional move

```
.intel_syntax noprefix

/* branch-based form */
cmp  edi, 0
jl   neg
mov  eax, edi
jmp  done
neg:
xor  eax, eax
done:

/* conditional-move-based form */
cmp  edi, 0
```

```
mov  eax, edi
mov  edx, 0
cmovl eax, edx
```

Conditional moves can avoid mispredictions at the cost of additional data dependencies. The trade-off depends on predictability and surrounding instruction context.

1.5 Overview of Modern Prediction Strategies

Modern CPUs must predict both branch direction and control-flow targets. Contemporary prediction systems combine several techniques:

- Static heuristics for cold or early execution
- Dynamic history-based predictors
- Local history capturing per-branch behavior
- Global history capturing cross-branch correlation
- Hybrid or tournament selection mechanisms
- Specialized predictors for indirect branches and returns

1.5.1 Example: Highly predictable loop backedge

```
#include <cstddef>
#include <cstdint>

std::uint64_t sum(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {  // taken many times, not-taken
    ↪  once
```

```
        s += static_cast<std::uint64_t>(p[i]);
    }
    return s;
}
```

Loop backedges typically exhibit stable behavior and are among the easiest branches to predict accurately.

1.5.2 Example: Indirect call target variability

```
using Fn = int(*)(int);

int apply(Fn f, int x) {
    // Frequent changes in f increase target misprediction risk
    return f(x);
}
```

When call targets vary across iterations or inputs, performance depends heavily on the quality of target prediction and the stability of runtime behavior.

Transition to Subsequent Chapters

The following chapters will examine concrete predictor models, quantify misprediction costs, analyze speculative execution behavior, and derive practical coding guidelines for writing control flow that modern CPUs can execute efficiently.

Chapter 2

CPU Pipelines and Control Hazards

2.1 Instruction Pipelines Recap

A **pipeline** divides instruction processing into stages so that multiple instructions are in flight simultaneously. Even if a single instruction requires many internal steps, pipelining allows the CPU to complete (retire) roughly one or more instructions per cycle when the pipeline stays full.
A simplified conceptual pipeline:

- **Fetch**: read instructions from the instruction cache using the current program counter (PC)
- **Decode**: translate instruction bytes into internal operations (micro-ops or equivalent)
- **Rename**: map architectural registers to physical registers to remove false dependencies
- **Dispatch/Schedule**: place operations into queues and select ready ones for execution
- **Execute**: perform ALU operations, address generation, loads/stores, branches

- **Retire/Commit**: update architectural state in-order (ensuring precise exceptions)

The key performance principle is simple: **steady fetch and decode** must continuously feed the backend. If the frontend stalls, the backend soon runs out of work, regardless of how powerful it is.

2.1.1 Example: Pipeline fill vs steady state

```
#include <cstdint>

std::uint64_t hot_loop(std::uint64_t n) {
    std::uint64_t s = 0;
    for (std::uint64_t i = 0; i < n; ++i) {
        s += i;  // tiny work per iteration
    }
    return s;
}
```

In a simple hot loop, once the pipeline is filled, performance is determined by how consistently the CPU can keep the pipeline full. Any disruption (especially in control flow) reduces throughput disproportionately because the loop body itself is cheap.

2.2 Control Hazards Explained

A **control hazard** occurs when the CPU cannot immediately know the next fetch address. Conditional branches, indirect branches, calls, and returns all change the PC in ways that depend on runtime behavior.
A conditional branch introduces two possible next PCs:

- **Fall-through**: next sequential instruction

- **Taken target**: branch destination

If the CPU waits until the branch condition is computed, fetch must stop. To avoid frontend starvation, the CPU predicts which path will be taken and continues fetching along that predicted path.

2.2.1 Example: Control hazard in the smallest form

```
.intel_syntax noprefix

cmp  edi, 0
je   zero_path     /* control hazard: next PC depends on flags */
add  eax, 1
jmp  done
zero_path:
sub  eax, 1
done:
```

The branch direction is not known at fetch time. It becomes known only after the compare is executed and flags are ready.

2.3 Branch Resolution Stages

Branch resolution is the point at which the CPU determines:

- whether a conditional branch is taken, and
- what the correct next PC should be (fall-through or target).

Resolution generally requires:

- the operands used in the comparison to be available,

- the comparison to execute (or flags to be produced),
- the target address to be available (for direct branches it is encoded; for indirect branches it is computed).

On real CPUs, the precise cycle when a branch resolves varies with:

- dependency chains (data readiness),
- execution port availability (backend pressure),
- micro-op decomposition,
- indirect-target computation latency.

2.3.1 Example: Resolution delayed by a dependency chain

```cpp
#include <cstddef>
#include <cstdint>

std::uint64_t count_nonzero(const int* p, std::size_t n) {
    std::uint64_t c = 0;
    for (std::size_t i = 0; i < n; ++i) {
        int x = p[i];       // may arrive late due to cache/memory behavior
        if (x != 0) ++c;    // branch resolves only after x is available
    }
    return c;
}
```

Even if the predictor is perfect, the branch cannot be *resolved* until `x` is loaded. Prediction exists because fetch cannot wait for that.

2.3.2 Example: Indirect branch resolution includes target computation

```
.intel_syntax noprefix

/* rax is computed from earlier work; target is unknown until rax is
 ↪  ready */
mov  rax, rdi
and  rax, 0xff
lea  rdx, [rip + table]
mov  rax, [rdx + rax*8]    /* load target address */
jmp  rax                   /* indirect branch: resolve target at
 ↪  runtime */
```

Indirect branches create a control hazard where both direction (always transfers) and target are runtime-dependent, increasing frontend pressure.

2.4 Pipeline Flushes and Bubbles

When a branch is mispredicted, the CPU must:

- discard (flush) instructions fetched and possibly executed on the wrong path,
- restore the correct fetch PC,
- refill the frontend with the correct instruction stream.

Flush refers to removing wrong-path work from the pipeline. **Bubbles** are empty slots where no useful instruction can be issued/retired because the pipeline is refilling or waiting on correct-path instructions.
Two distinct costs arise:

- **Recovery latency**: time to redirect fetch and restart decode

- **Opportunity cost**: lost cycles where the backend cannot do useful work

2.4.1 Example: Unpredictable branch causes repeated flush/refill

```cpp
#include <cstdint>

std::uint64_t branchy_mix(std::uint64_t n) {
    std::uint64_t s = 0;
    for (std::uint64_t i = 0; i < n; ++i) {
        if ((i * 1103515245u + 12345u) & 0x80000000u) {  // chaotic pattern
            s += i;
        } else {
            s -= i;
        }
    }
    return s;
}
```

The arithmetic pattern can make the branch hard to predict. Each misprediction flushes wrong-path work and injects bubbles while the correct path is refetched.

2.4.2 Example: Turning a hard branch into dataflow can reduce bubbles

```cpp
#include <cstdint>

std::uint64_t branchless_mix(std::uint64_t n) {
    std::uint64_t s = 0;
    for (std::uint64_t i = 0; i < n; ++i) {
        std::uint64_t cond = (std::uint64_t)(((i * 1103515245u + 12345u) >>
        ↪  31) & 1u);
        std::uint64_t add  =  (std::uint64_t)(-(std::int64_t)cond);   // 0
        ↪  or all-ones
        std::uint64_t sub  = ~add;
```

```
        s += (i & add);
        s -= (i & sub);
    }
    return s;
}
```

This form trades control dependence for data dependence. It can reduce misprediction flushes but may increase ALU work and dependency chains. The correct choice depends on the workload and microarchitecture.

2.5 Frontend vs Backend Effects

It is useful to separate performance effects into:

- **Frontend-bound behavior**: the backend is underutilized because fetch/decode cannot deliver enough correct instructions.
- **Backend-bound behavior**: the frontend supplies instructions, but execution resources or memory limit progress.

Branch mispredictions are primarily a **frontend** problem:

- wrong-path fetch wastes bandwidth,
- pipeline redirects take time,
- decode/rename queues may drain,
- backend starves after a short delay.

However, the backend also influences control hazards:

- a branch depending on a long dependency chain resolves later,

- contention for execution ports can delay compare/branch execution,
- long-latency loads delay branch operands, extending the unresolved window.

2.5.1 Example: Same branch, frontend-bound vs backend-influenced

```cpp
#include <cstddef>
#include <cstdint>

std::uint64_t sum_if_nonzero(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        int x = p[i];           // backend + memory can delay x
        if (x) s += (std::uint64_t)x;  // frontend cost if unpredictable;
        ↪  backend cost if x is late
    }
    return s;
}
```

If `x` is in cache and `x != 0` is highly predictable, the loop can be backend-friendly. If `x` arrives late (cache misses) the branch resolves late, increasing the unresolved window. If `x` is unpredictable, mispredictions make it frontend-bound, even with fast memory.

2.5.2 Operational takeaway

Control hazards sit at the boundary between frontend and backend. Branch prediction exists because the frontend must make progress without waiting for backend resolution. When prediction fails, the frontend disrupts the entire machine, and the backend quickly becomes idle despite being capable of high throughput.

Chapter 3

Cost of Branch Misprediction

3.1 What Happens on a Misprediction

A **branch misprediction** occurs when the CPU fetches and speculatively executes the wrong control-flow path. Correctness is preserved because speculative instructions do not permanently update architectural state until they **retire/commit** in order. When the branch finally resolves and the CPU discovers the prediction was wrong, several things happen conceptually:

- **Redirect**: the fetch unit is pointed to the correct next PC (fall-through or taken target).
- **Flush**: wrong-path instructions in the pipeline are invalidated and will not retire.
- **Restart**: the frontend refills from the correct path, decode and rename restart, and the backend resumes useful work.

The wasted work is not only the branch itself. A wide machine may have fetched, decoded, renamed, and even executed many wrong-path instructions during the speculative window.

3.1.1 Example: Wrong-path work exists even for tiny branches

```cpp
#include <cstddef>
#include <cstdint>

std::uint64_t count_lt(const int* p, std::size_t n, int t) {
    std::uint64_t c = 0;
    for (std::size_t i = 0; i < n; ++i) {
        if (p[i] < t) {    // a single unpredictable branch can dominate
            ++c;
        }
    }
    return c;
}
```

If `p[i] < t` is close to 50/50 and data-dependent, the CPU may frequently fetch the wrong path, flush, and restart. The loop body is small, so misprediction recovery becomes a major part of total time.

3.2 Pipeline Recovery Mechanisms

Recovery is implemented with microarchitectural mechanisms that restore a precise architectural view while discarding speculative state. Although details vary across CPUs, the conceptual recovery steps are stable:

- **Frontend redirect**: fetch begins at the correct PC.
- **Squash younger work**: all instructions younger than the mispredicted branch are removed from rename/scheduler/ROB state.
- **Restore rename mappings**: physical register allocation and rename tables are rolled back to the state at the branch.

- **Refill queues**: the decode/rename/dispatch queues fill with correct-path instructions.

The key property is **precise recovery**: the architectural state appears as if no wrong-path instruction ever executed.

3.2.1 Example: Why recovery costs scale with width

A wide frontend can fetch and decode many instructions per cycle. That is excellent when prediction is correct, but it also means wrong-path work accumulates quickly when prediction is wrong. Recovery must discard more in-flight work and then refill that wide machine again.

3.3 Typical Cycle Penalties

A misprediction penalty is the time between:

> *the point where the CPU could have continued on the correct path*

and

> *the point where the CPU actually resumes useful work on the correct path after recovery.*

In practice, the penalty is influenced by:

- **pipeline depth** (how far ahead the wrong-path fetch went),
- **frontend latency** (redirect and refill time),
- **branch resolution point** (how late the CPU discovers the correct outcome),
- **uop/decoder effects** (decode bandwidth, instruction length variability),
- **target type** (direct vs indirect vs return).

Two practical rules are stable across modern designs:

- **Mispredictions are expensive**: the cost is often comparable to dozens of simple ALU operations.
- **Unresolved time matters**: branches that resolve late (due to dependencies) waste more speculative bandwidth when mispredicted.

3.3.1 Example: A late-resolving branch multiplies the pain

```cpp
#include <cstddef>
#include <cstdint>

std::uint64_t count_hot(const int* p, std::size_t n) {
    std::uint64_t c = 0;
    for (std::size_t i = 0; i < n; ++i) {
        int x = p[i];           // late if cache misses or dependent chain
        if ((x & 7) == 3) {     // unpredictable + resolves after x arrives
            c += (std::uint64_t)x;
        }
    }
    return c;
}
```

If `x` arrives late, the branch stays unresolved longer, so the CPU may speculate further down the wrong path. When mispredicted, more work is discarded and the frontend refill cost becomes more visible.

3.4 Impact on IPC and CPI

Instruction throughput is often described via:

- **IPC** (instructions per cycle): higher is better.
- **CPI** (cycles per instruction): lower is better.

Mispredictions reduce IPC because cycles are spent doing non-retiring work:

- During recovery, the backend may be partially or fully starved.
- Fewer useful instructions retire per cycle.
- CPI increases because the same amount of architectural work now consumes more cycles.

A practical performance model for hot code is:

$$\textit{Total cycles} \approx \textit{base execution cycles} + (\textit{mispredictions} \times \textit{mispredict penalty})$$

This model is not exact, but it is extremely useful: even a modest misprediction rate can dominate if the penalty is large and the loop body is small.

3.4.1 Example: How a small mispredict rate can dominate a tiny loop

```
#include <cstddef>
#include <cstdint>

std::uint64_t sum_if(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        int x = p[i];
        if (x > 0) s += (std::uint64_t)x;
    }
    return s;
}
```

If the loop body without mispredictions would take only a few cycles per iteration, then each misprediction injects a large additional cycle cost. The net effect is a steep IPC drop and a visibly higher CPI.

3.5 Performance Measurement Considerations

To reason correctly about misprediction cost, measurements must distinguish between:

- **branch frequency**: how many branches execute in the hot path,
- **misprediction rate**: how many of those are mispredicted,
- **penalty magnitude**: how expensive recovery is on the specific CPU,
- **confounders**: cache misses, front-end fetch stalls, decode limits, and frequency scaling.

Key measurement principles:

- **Warm up**: cold-start effects (instruction cache, predictor warmup) can mislead.
- **Stabilize inputs**: branch predictability depends on data distributions; randomizing inputs changes the branch problem.
- **Avoid I/O and system noise**: pin threads, reduce OS interference, and measure steady-state loops.
- **Use hardware counters when available**: mispredicted branches and frontend stalls are observable at the microarchitectural level.

3.5.1 Example: Create predictable vs unpredictable datasets intentionally

```cpp
#include <algorithm>
#include <cstddef>
#include <cstdint>
#include <random>
#include <vector>

std::vector<int> make_clustered(std::size_t n) {
    std::vector<int> v(n);
    // First half positive, second half negative: highly predictable branch
    ↪  regions.
    for (std::size_t i = 0; i < n; ++i) v[i] = (i < n/2) ? 1 : -1;
    return v;
}

std::vector<int> make_random_sign(std::size_t n, std::uint32_t seed = 1) {
    std::vector<int> v(n);
    std::mt19937 rng(seed);
    std::uniform_int_distribution<int> d(0, 1);
    for (std::size_t i = 0; i < n; ++i) v[i] = d(rng) ? 1 : -1;
    return v;
}

void consume(std::uint64_t x) {
    // Prevent trivial dead-code elimination in simple benchmarks.
    volatile std::uint64_t sink = x;
    (void)sink;
}
```

Using controlled data lets you demonstrate the effect of predictability on mispredictions, IPC, and CPI. Without controlling inputs, a benchmark can accidentally measure data distribution rather than microarchitectural behavior.

3.5.2 Example: Microbenchmark structure (conceptual)

```cpp
#include <chrono>
#include <cstddef>
#include <cstdint>
#include <vector>

std::uint64_t count_positives(const std::vector<int>& a);

std::uint64_t bench(const std::vector<int>& a, int iters) {
    std::uint64_t total = 0;
    for (int k = 0; k < iters; ++k) {
        total += count_positives(a);
    }
    return total;
}

double seconds_for(const std::vector<int>& a, int iters) {
    auto t0 = std::chrono::steady_clock::now();
    auto x  = bench(a, iters);
    auto t1 = std::chrono::steady_clock::now();
    volatile std::uint64_t sink = x;
    (void)sink;
    std::chrono::duration<double> dt = t1 - t0;
    return dt.count();
}
```

This structure encourages steady-state behavior and amortizes timer overhead. In real investigations, cycle counters and hardware performance counters provide more direct visibility into mispredictions and frontend stalls.

Transition to Subsequent Chapters

With a clear understanding of what mispredictions cost and why, the next chapters examine how predictors learn branch behavior (static and dynamic schemes), why some patterns are inherently difficult, and how code structure can improve predictability in hot paths.

Chapter 4

Static Branch Prediction

4.1 Compile-Time Prediction Concepts

Static branch prediction means choosing a branch direction without using runtime history. The decision may be made by:

- fixed hardware rules (a built-in heuristic used when no dynamic history is available), and/or
- compiler-driven layout decisions (how code is arranged so the common path is the fall-through).

Static prediction matters most in three situations:

- **Cold code**: first-time execution before dynamic predictors learn.
- **Disruptions**: context switches, migration, power-state changes, or events that effectively reduce predictor usefulness.

- **I-cache and layout**: even with good prediction, poor layout can waste fetch bandwidth or cause I-cache misses.

A core practical idea: **even when hardware uses advanced dynamic prediction, the compiler still shapes performance** by selecting fall-through paths, placing likely targets close, and reducing frontend penalties.

4.1.1 Example: Shaping the common path as fall-through

```c
int handle(int x) {
    // Prefer the common path as straight-line fall-through.
    if (x == 0) {
        return 0;           // rare case exits early
    }
    // hot path continues without a taken branch
    return x + 1;
}
```

In many ISAs and frontends, the fall-through path is naturally fetch-friendly (no taken-branch redirection), especially when the rare path is handled as an early exit.

4.2 Always-Taken and Always-Not-Taken

The simplest static models assume one direction for all conditional branches:

- **Always-not-taken**: keep fetching sequentially; if taken, redirect.
- **Always-taken**: assume the branch will jump; if not, correct to fall-through.

These rules are not sophisticated, but they demonstrate two key truths:

- Predicting *not-taken* favors sequential fetch and can reduce frontend work when most branches fall through.

- Predicting *taken* can help for loop backedges (often taken) and backward branches.

4.2.1 Example: Loop backedge is usually taken

```cpp
#include <cstddef>
#include <cstdint>

std::uint64_t sum(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {  // branch taken many times
        s += (std::uint64_t)p[i];
    }
    return s;
}
```

The branch that jumps back to the loop head is taken for all iterations except the last. Any strategy that assumes “taken” for backward branches tends to do well here.

4.2.2 Example: Error paths are often not taken

```cpp
int parse_digit(char c) {
    // For valid inputs, the branch is mostly not taken.
    if (c < '0'  c > '9') return -1;
    return c - '0';
}
```

Many real systems have hot fast paths with rare error handling. Structuring code so the error path exits early keeps the hot path linear.

4.3 Direction-Based Heuristics

A more realistic static approach uses **direction-based heuristics**:

- **Backward branches** (targets at lower addresses) are assumed taken (common for loops).
- **Forward branches** (targets at higher addresses) are assumed not taken (common for conditionals and error exits).

This heuristic is effective because it matches typical code structure generated by compilers:

- Loops are formed using backward jumps.
- Conditionals often jump forward to skip code or reach an error path.

4.3.1 Example: Forward conditional to skip a rare block

```
.intel_syntax noprefix

/* if (x == 0) goto rare; else fall-through hot path */
test edi, edi
je   rare              /* forward branch: typically predicted
 ↪  not-taken */
    /* hot path */
    add eax, 1
    jmp done
rare:
    /* rare path */
    xor eax, eax
done:
```

The forward jump targets a rare block; structuring it this way makes the hot path fall-through.

4.3.2 Example: Backward branch in a tight loop

```
.intel_syntax noprefix

mov  rcx, rsi          /* n */
xor  rax, rax          /* sum = 0 */
loop_head:
    add  rax, [rdi]    /* sum += *p */
    add  rdi, 8        /* p++ */
    dec  rcx
    jne  loop_head     /* backward branch: typically predicted taken
     ↪  */
```

The loop branch is taken repeatedly, matching the classic backward-taken heuristic.

4.4 Compiler Branch Hints

Compilers may expose ways to communicate branch likelihood. Conceptually, hints can affect:

- **code layout**: making the likely path fall-through,
- **block placement**: placing hot blocks contiguously to reduce I-cache pressure,
- **instruction selection**: enabling conditional moves or other forms,
- **profile-guided optimization (PGO)**: using measured frequencies to guide layout and decisions.

Two practical classes exist:

- **source-level hints**: programmer expresses expectations (must be used carefully).

- **profile-driven guidance**: compiler uses real execution profiles (generally more reliable).

4.4.1 Example: Likely/unlikely macros (portable style)

```
#if defined(__GNUC__)  defined(__clang__)
    #define LIKELY(x)   __builtin_expect(!!(x), 1)
    #define UNLIKELY(x) __builtin_expect(!!(x), 0)
#else
    #define LIKELY(x)   (x)
    #define UNLIKELY(x) (x)
#endif

int parse_digit(char c) {
    if (UNLIKELY(c < '0'  c > '9')) return -1;  // expected rare
    return c - '0';
}
```

This kind of hint typically influences code layout and may help the compiler keep the fast path linear. It does not guarantee hardware prediction behavior, but it can reduce frontend costs by improving fall-through and placement.

4.4.2 Example: Modern standard hint style (conceptual)

```
int handle(int x) {
    if (x == 0) [[unlikely]] {
        return 0;
    }
    return x + 1;
}
```

Such annotations are intended to guide optimization and layout. They must reflect reality; incorrect hints can reduce performance.

4.4.3 Example: Profile-guided optimization as the strongest hint

```
int classify(int x) {
    // When compiled with profiling + PGO, the compiler can learn:
    // - which branch is hot
    // - how to arrange blocks
    // - how to shape fall-through
    if (x < 0) return -1;
    if (x == 0) return 0;
    return 1;
}
```

With profiling, the compiler can reorder blocks so the most frequent outcomes produce the most fetch-friendly instruction stream.

4.5 Limitations of Static Prediction

Static prediction and static hints have fundamental limits:

- **Data-dependent behavior**: branch direction depends on runtime data distributions that may change across inputs.
- **Phase changes**: what is "likely" can change over time (different request types, different user behavior, different program phase).
- **Microarchitecture dependence**: the same layout can behave differently across CPU families and generations.
- **Indirect control flow**: targets vary in ways static analysis cannot reliably predict (virtual dispatch, function pointers, JIT, interpreters).
- **Speculation and caching interactions**: even correct prediction can be slowed by instruction-cache misses or frontend bandwidth limits.

Therefore:

- Use static hints primarily to shape **code layout** for obvious fast/slow paths.
- Prefer **PGO** when available; it reflects real behavior.
- Measure; do not assume a hint helps.

4.5.1 Example: A hint that becomes wrong when input distribution changes

```
int handle_request(int type) {
    // If the workload shifts (e.g., type==0 becomes common),
    // an incorrect hint can hurt by mis-shaping layout.
    if (UNLIKELY(type == 0)) {
        return 0;
    }
    return type + 10;
}
```

Static assumptions are brittle when the environment or input distribution is not stable.

Transition to Subsequent Chapters

Static prediction provides useful fallbacks and layout guidance, but high performance on modern CPUs depends on **dynamic** predictors that learn patterns at runtime. The next chapter introduces the feedback-driven mechanism by which CPUs learn branch behavior and refine predictions over time.

Chapter 5

Dynamic Branch Prediction Basics

5.1 Runtime Feedback Loop

Dynamic branch prediction uses runtime behavior to predict future branch outcomes. The CPU treats each branch as a repeating event that can be learned statistically. The core idea is a feedback loop:

- Predict an outcome when the branch is encountered (to keep fetch running).
- Execute speculatively along the predicted path.
- Eventually resolve the real outcome.
- Feed the result back into the predictor so the next prediction improves.

This is not a single mechanism but a coordinated system in the frontend. The predictor must operate at fetch time with extremely low latency and high bandwidth.

5.1.1 Example: A loop produces a stable feedback signal

```
#include <cstddef>
#include <cstdint>

std::uint64_t sum(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {   // taken many times, not-taken
    ↪  once
        s += (std::uint64_t)p[i];
    }
    return s;
}
```

The loop backedge produces a predictable pattern: taken repeatedly, then not-taken once. A dynamic predictor quickly learns this behavior and maintains high accuracy.

5.1.2 Example: Data-driven branches produce unstable feedback

```
#include <cstddef>
#include <cstdint>

std::uint64_t count_gt(const int* p, std::size_t n, int t) {
    std::uint64_t c = 0;
    for (std::size_t i = 0; i < n; ++i) {
        if (p[i] > t) ++c;      // outcome depends on data distribution
    }
    return c;
}
```

If the distribution of values shifts over time or is near-random, the predictor sees noisy feedback and cannot converge to high accuracy.

5.2 Predictor State and Learning

Dynamic predictors maintain **state** that summarizes recent behavior. The simplest state is a confidence-like memory: “this branch tends to be taken” or “tends to be not-taken.” More advanced predictors store richer history patterns.
Learning is incremental:

- Correct predictions reinforce current state.
- Mispredictions push state toward the opposite direction.
- Stronger confidence requires more evidence to flip (stability).

Two practical properties matter:

- **Stability**: avoid overreacting to one anomaly.
- **Adaptability**: respond when behavior truly changes.

5.2.1 Example: A branch with bias learns quickly

```
int parse_digit(char c) {
    if (c < '0'  c > '9') return -1;  // typically rare
    return c - '0';
}
```

If invalid input is rare, the predictor rapidly learns “not-taken” for the error branch, and the frontend runs smoothly.

5.2.2 Example: A branch with periodic behavior requires history

```
#include <cstdint>
```

```
std::uint64_t periodic(std::uint64_t n) {
    std::uint64_t c = 0;
    for (std::uint64_t i = 0; i < n; ++i) {
        if ((i & 7u) == 0u) {   // true once every 8 iterations
            ++c;
        }
    }
    return c;
}
```

A predictor that remembers only bias may still do well here, but richer history helps distinguish periodic patterns from noise when multiple branches interact.

5.3 Branch History Storage

To learn patterns, the CPU must record recent outcomes. Conceptually, predictors use one or both:

- **Per-branch (local) history**: a small history associated with a particular branch (identified by its address).
- **Global history**: a record of the outcomes of recent branches in the dynamic instruction stream.

These histories are stored in small, fast hardware structures. Because hardware tables are finite, different branches can map to the same entry, causing **aliasing** (interference). Aliasing is a fundamental trade-off: larger structures reduce interference but cost area and power; smaller structures are faster but collide more.

5.3.1 Example: Two different branches can interfere conceptually

```
int f(int a, int b) {
    int s = 0;

    if (a > 0) s += 1;     // branch A
    if (b > 0) s += 2;     // branch B

    return s;
}
```

If predictor indexing causes A and B to share state, one branch's updates can distort the other's learned behavior. This is one reason prediction accuracy depends on workload composition and instruction address patterns.

5.3.2 Example: Global correlation can make prediction easier

```
int g(int mode, int x) {
    if (mode == 0) {           // branch 1
        if (x < 10) return 1; // branch 2 tends to correlate with mode
        return 2;
    } else {
        if (x < 10) return 3; // branch 3
        return 4;
    }
}
```

The outcome of later branches depends on earlier control decisions. Global history enables a predictor to capture such correlations.

5.4 Prediction vs Update Phases

Dynamic prediction has two distinct phases:

- **Prediction phase**: when the branch is fetched, the predictor must output a direction (and often a target) immediately.
- **Update phase**: when the branch outcome is resolved, the predictor state is updated with the true outcome.

These phases are separated in time by many cycles, especially on deep pipelines. During the gap:

- the CPU may have fetched far beyond the branch,
- multiple branches may be in flight,
- updates may arrive out of order (resolved in different pipeline stages),
- the predictor must remain consistent despite speculative execution.

A useful mental model is that the predictor is a *fast online learner* operating under latency and bandwidth constraints.

5.4.1 Example: Many in-flight branches

```
#include <cstddef>
#include <cstdint>

std::uint64_t complex_loop(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        int x = p[i];
```

```
        if (x > 0) s += (std::uint64_t)x;      // branch A
        if ((x & 3) == 0) s += 1;              // branch B
        if (x == 7) break;                     // branch C (rare exit)
    }
    return s;
}
```

The frontend must predict each branch as it is fetched, while the updates may occur later and at different times. The predictor's design must handle overlapping speculation windows and avoid unstable behavior.

5.5 Cold Start Behavior

At program start, or when encountering a branch that has not been seen recently, the predictor has limited or no history. This is called **cold start**. Cold behavior also appears after disruptive events that reduce predictor usefulness (for example, code paths that were not executed for a long time).
Cold start consequences:

- Early iterations of loops may be mispredicted until the predictor converges.
- Rare branches may never accumulate enough history to become strongly predicted.
- Benchmark results can be skewed if measured only during warmup or only during early cold execution.

Modern CPUs mitigate cold start with:

- default static-like biases for unknown branches,
- multi-level predictors where a fast component provides an initial guess,
- fast-learning state machines that quickly stabilize for biased branches.

5.5.1 Example: Short loops may never fully warm up

```cpp
#include <cstddef>
#include <cstdint>

std::uint64_t short_loop(const int* p, std::size_t n) {
    // If n is very small, the predictor may not reach a stable state.
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        if (p[i] & 1) ++s;   // branch sees few samples
    }
    return s;
}
```

If `n` is small and called many times from different contexts, the predictor may repeatedly operate in a quasi-cold regime for this branch, making static layout and branchless alternatives more relevant.

5.5.2 Example: Warming up deliberately in microbenchmarks

```cpp
#include <cstddef>
#include <cstdint>

std::uint64_t count_odds(const int* p, std::size_t n);

std::uint64_t warm_then_measure(const int* p, std::size_t n, int warm_iters,
↪  int meas_iters) {
    std::uint64_t x = 0;

    // warmup: allow predictor and caches to stabilize
    for (int k = 0; k < warm_iters; ++k) {
        x += count_odds(p, n);
    }
```

```
    // measurement phase
    for (int k = 0; k < meas_iters; ++k) {
        x += count_odds(p, n);
    }

    return x;
}
```

Without warmup, measurements may capture cold-start mispredictions rather than steady-state performance.

Transition to Subsequent Chapters

Dynamic prediction is built from concrete predictor models. The next chapter introduces the simplest widely used learning element: saturating counters and the state-machine view of branch confidence, which forms the foundation for more advanced history-based predictors.

Chapter 6

Two-Bit and Saturating Counter Predictors

6.1 Motivation for Multi-Bit Predictors

Dynamic predictors must balance two opposing goals:

- **Adapt quickly** when a branch behavior truly changes.
- **Do not overreact** to rare anomalies (noise).

A single bit of history (“last outcome”) adapts extremely fast, but it is unstable for common patterns such as loop exits. Multi-bit predictors introduce **hysteresis**: the predictor requires multiple consecutive opposite outcomes before it flips its prediction. This improves stability without needing complex history structures.
The simplest and most influential form is the **two-bit saturating counter**, which remains a foundation even in modern hybrid predictors.

6.1.1 Example: The loop-exit problem motivates hysteresis

```cpp
#include <cstddef>
#include <cstdint>

std::uint64_t sum(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {  // taken many times, not-taken
    ↪  once
        s += (std::uint64_t)p[i];
    }
    return s;
}
```

The backedge branch is taken repeatedly and then not taken once at exit. A stable predictor should not permanently "flip" direction because of a single exit.

6.2 One-Bit Predictor Weaknesses

A **one-bit predictor** predicts the next outcome to be the same as the last observed outcome. This looks reasonable, but it fails badly on two high-frequency patterns:

- **Loop branches**: one misprediction at loop exit *and* another at loop re-entry on the next invocation (if the predictor flipped).
- **Alternating branches**: a strict alternation causes a misprediction every time.

6.2.1 Example: One-bit predictor loop penalty (two mispredicts per run)

Assume the predictor stores "taken/not-taken" as the last outcome for the loop backedge. For each call:

- During the loop, the last outcome is mostly **taken**.
- At the final iteration, the branch becomes **not-taken** (exit) → mispredict (predict taken).
- Predictor updates to **not-taken**.
- Next time the function is called, the first backedge is **taken** → mispredict (predict not-taken).

This yields roughly **two mispredictions per invocation** for a simple loop branch, which is unacceptable in hot code.

6.2.2 Example: Alternation defeats one-bit prediction

```
#include <cstdint>

std::uint64_t alternating(std::uint64_t n) {
    std::uint64_t c = 0;
    for (std::uint64_t i = 0; i < n; ++i) {
        if ((i & 1u) == 0u) ++c;  // true, false, true, false, ...
    }
    return c;
}
```

When outcomes alternate, predicting “same as last time” is guaranteed to be wrong.

6.3 Two-Bit Saturating State Machine

A **two-bit saturating counter** represents four states, typically interpreted as two levels of confidence for each direction:

- **Strongly Not Taken** (SNT)

- **Weakly Not Taken** (WNT)
- **Weakly Taken** (WT)
- **Strongly Taken** (ST)

The prediction is:

- predict **Not Taken** in SNT or WNT
- predict **Taken** in WT or ST

Updates move the state toward taken on taken outcomes, and toward not-taken on not-taken outcomes, but **saturate** at the extremes.

6.3.1 State machine (direction and update rules)

```
States: 00=SNT, 01=WNT, 10=WT, 11=ST

Predict:
  if state in {00,01} -> predict Not-Taken
  if state in {10,11} -> predict Taken

Update on Taken:
  00->01, 01->10, 10->11, 11->11

Update on Not-Taken:
  11->10, 10->01, 01->00, 00->00
```

This design introduces hysteresis: one opposite outcome does not fully reverse the predicted direction; it only weakens confidence.

6.3.2 Example: Minimal implementation of a 2-bit counter

```
#include <cstdint>

struct Sat2 {
    // 0=SNT, 1=WNT, 2=WT, 3=ST
    std::uint8_t s = 3;

    bool predict_taken() const {
        return s >= 2;
    }

    void update(bool taken) {
        if (taken) {
            if (s < 3) ++s;
        } else {
            if (s > 0) --s;
        }
    }
};
```

This models the core behavior used by many practical predictors at some level of the hierarchy.

6.4 State Transitions and Stability

The primary benefit of the two-bit scheme is stability in the presence of rare anomalies. Consider a branch that is usually taken but occasionally not taken:

- A one-bit predictor flips immediately after a single anomaly.
- A two-bit predictor usually stays predicting taken, merely weakening confidence.

6.4.1 Example: “mostly taken” branch

```cpp
#include <cstddef>
#include <cstdint>

std::uint64_t sum_until_sentinel(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        int x = p[i];
        if (x == 0) break;      // usually not taken, but taken once at end
        s += (std::uint64_t)x;
    }
    return s;
}
```

The branch is “not taken” repeatedly and then “taken” once to exit. A stable predictor should avoid flipping permanently due to that one exit event.

6.4.2 Example: Why two-bit reduces loop mispredicts

For a loop backedge that is taken many times and not taken once at exit:

- The predictor tends toward **ST** during the loop.
- At exit (not taken), it moves from **ST** to **WT** and mispredicts once.
- On the next invocation, the first taken outcome moves back toward **ST** without forcing an extra mispredict.

Compared to one-bit prediction, this typically reduces the “exit + re-entry” pair of mispredicts to **about one mispredict per loop run** (the exit), assuming the loop is long enough to keep the predictor in a taken-biased state.

6.5 Predictor Accuracy Characteristics

Two-bit counters work best for:

- **Biased branches**: strongly favor taken or not-taken.
- **Loop backedges**: repeated outcomes with a single opposite event at exit.
- **Rare error paths**: predictable fall-through with occasional jumps.

They work poorly for:

- **True alternation**: taken/not-taken repeating patterns can still be difficult.
- **Chaotic data-dependent branches**: near-random outcomes cannot be learned reliably.
- **Phase changes**: if behavior flips frequently between regimes, any small-state predictor will lag.

6.5.1 Example: Alternation remains hard even with two-bit

```cpp
#include <cstdint>

std::uint64_t alternating(std::uint64_t n) {
    std::uint64_t c = 0;
    for (std::uint64_t i = 0; i < n; ++i) {
        if ((i & 1u) == 0u) ++c;
    }
    return c;
}
```

A two-bit counter still tends to be wrong frequently on strict alternation because the state machine lags behind a pattern that flips every iteration. This motivates predictors that use explicit history (local/global) rather than only a confidence counter.

6.5.2 Example: Periodic pattern can be learned partially

```
#include <cstdint>

std::uint64_t periodic_8(std::uint64_t n) {
    std::uint64_t c = 0;
    for (std::uint64_t i = 0; i < n; ++i) {
        if ((i & 7u) == 0u) ++c;    // taken once every 8
    }
    return c;
}
```

A two-bit predictor may converge to “not taken” (the majority outcome) and mispredict the periodic taken events. History-based predictors can do better by recognizing the repeating pattern.

Transition to Subsequent Chapters

Two-bit saturating counters provide hysteresis and strong performance for biased branches and loop structures, but they cannot exploit correlations between branches. The next chapter extends the model by introducing local and global history predictors that use sequences of past outcomes to improve accuracy on structured patterns.

Chapter 7

Local and Global History Predictors

7.1 Local History Predictors

A two-bit counter captures *bias* (mostly taken / mostly not-taken) but it does not capture *patterns*. Many branches are not purely biased; they follow repeating sequences. A **local history predictor** addresses this by keeping a short history of recent outcomes *for each branch*.

Conceptually, a local predictor has two main structures:

- **Local History Table (LHT)**: indexed by (part of) the branch address; stores an h-bit shift register of recent outcomes for that branch.
- **Pattern History Table (PHT)**: indexed by the local history pattern; stores a small saturating counter (often 2-bit) that predicts the next outcome for that pattern.

The local history is updated on each resolved branch, and the corresponding PHT counter learns whether that particular pattern tends to be followed by taken or not-taken.

7.1.1 Example: A branch with a repeating pattern (not pure bias)

```
#include <cstdint>

std::uint64_t periodic_4(std::uint64_t n) {
    std::uint64_t c = 0;
    for (std::uint64_t i = 0; i < n; ++i) {
        if ((i & 3u) == 0u) {  // true once every 4 iterations
            ++c;
        }
    }
    return c;
}
```

A bias-only predictor may converge to “not taken” and still miss the periodic taken events. A local-history predictor can learn that the branch outcome depends on the last few outcomes and can predict the repeating pattern more accurately.

7.1.2 Example: Local predictor conceptual model

```
#include <cstdint>

struct LocalHistoryPredictor {
    // Conceptual only: tiny tables to illustrate mechanics.
    static constexpr std::uint32_t LHT_SIZE = 64;
    static constexpr std::uint32_t PHT_SIZE = 1u << 4; // 4-bit local
    ↪ history

    std::uint8_t lht[LHT_SIZE] = {};   // each entry holds 4-bit history
    std::uint8_t pht[PHT_SIZE] = {};   // 2-bit counters: 0..3

    static std::uint32_t idx_lht(std::uint32_t pc) { return pc & (LHT_SIZE -
    ↪ 1); }
```

```
    bool predict(std::uint32_t pc) const {
        std::uint8_t h = lht[idx_lht(pc)] & 0xF;
        return pht[h] >= 2;
    }

    void update(std::uint32_t pc, bool taken) {
        std::uint32_t i = idx_lht(pc);
        std::uint8_t h  = lht[i] & 0xF;

        std::uint8_t& c = pht[h];
        if (taken) { if (c < 3) ++c; } else { if (c > 0) --c; }

        lht[i] = (std::uint8_t)(((h << 1) | (taken ? 1 : 0)) & 0xF);
    }
};
```

This illustrates the essential idea: *history selects a counter*, and counters learn the next outcome for that history.

7.2 Per-Branch Correlation

Per-branch correlation means that the next outcome of a branch is correlated with its own recent outcomes. Common sources of per-branch correlation:

- **Loop structure**: repeated taken outcomes and a final not-taken exit, possibly with nested loops.
- **Periodic checks**: "do this every k iterations" patterns.
- **State machines**: branches that follow a fixed cycle of internal state transitions.

Local predictors are strong when a branch is largely independent from other branches and has a consistent local pattern.

7.2.1 Example: Per-branch correlation from a small state machine

```
#include <cstdint>

std::uint64_t state_machine(std::uint64_t n) {
    std::uint64_t c = 0;
    int st = 0;
    for (std::uint64_t i = 0; i < n; ++i) {
        // This branch depends on the local state evolution.
        if (st == 0) {
            ++c;
            st = 1;
        } else if (st == 1) {
            st = 2;
        } else {
            st = 0;
        }
    }
    return c;
}
```

Outcomes are not random; they are governed by a local cycle. Local-history-based prediction can learn such patterns with short histories.

7.3 Global History Predictors

Some branches are not well predicted by their own past alone. Their outcomes depend on *earlier control-flow decisions elsewhere*. A **global history predictor** records outcomes of the most recent branches in the dynamic stream using a **Global History Register (GHR)**.

Conceptually:

- The GHR is an h-bit shift register: each resolved branch shifts in 1 (taken) or 0 (not taken).
- A PHT (or similar table) is indexed using the GHR (often combined with branch address bits) to select a saturating counter.

This allows the predictor to capture **cross-branch correlation**: the outcome of branch B may depend on the earlier outcome of branch A.

7.3.1 Example: Global correlation from mode selection

```
int classify(int mode, int x) {
    if (mode == 0) {              // branch A
        if (x < 10) return 1;     // branch B1 correlated with A
        return 2;
    } else {
        if (x < 10) return 3;     // branch B2 correlated with A
        return 4;
    }
}
```

The "`x < 10`" decision occurs in both paths, but it is reached under different prior control decisions. A global history predictor can use the outcome of branch A to help predict the later branch behavior.

7.3.2 Example: Global predictor conceptual model

```
#include <cstdint>

struct GlobalHistoryPredictor {
```

```
    // Conceptual only.
    static constexpr std::uint32_t H = 8;
    static constexpr std::uint32_t PHT_SIZE = 1u << H;

    std::uint8_t ghr = 0;                 // 8-bit history
    std::uint8_t pht[PHT_SIZE] = {};      // 2-bit counters: 0..3

    bool predict() const {
        return pht[ghr] >= 2;
    }

    void update(bool taken) {
        std::uint8_t& c = pht[ghr];
        if (taken) { if (c < 3) ++c; } else { if (c > 0) --c; }

        ghr = (std::uint8_t)((ghr << 1) | (taken ? 1 : 0));
    }
};
```

Real designs incorporate branch address bits to reduce aliasing and to make predictions branch-specific; the example isolates the global-history concept.

7.4 Correlating Independent Branches

Global history is powerful because program behavior is often structured:

- Earlier branches select a **mode** or **phase**.
- Later branches behave differently depending on that mode.
- Nested conditionals create correlated outcomes across multiple branches.

A global predictor can learn that "after the pattern 1011 in recent branches, this branch is likely taken." This correlation can exist even when the branch's *own* local history does not explain it.

7.4.1 Example: Correlation across separate tests

```
int h(int a, int b, int x) {
    // Branch A and Branch B select a phase.
    bool phase = (a > 0);           // branch A
    if (b == 7) phase = !phase;     // branch B (rare toggle)

    // Branch C outcome correlates with phase, not necessarily with C's own
    ↪ history.
    if (phase) {                    // branch C (depends on prior outcomes)
        return (x & 1) ? 10 : 11;
    } else {
        return (x & 1) ? 20 : 21;
    }
}
```

Even if `x` is random, the selection of the `phase` influences which region of code is executed next, creating correlation patterns that global history can exploit.

7.5 Strengths and Trade-Offs

Local and global predictors each solve a different class of problems:

7.5.1 Strengths

- **Local history** excels when the branch has a stable per-branch pattern: loops with structured exits, periodic checks, and local state machines.

- **Global history** excels when branch outcomes depend on earlier control-flow decisions: mode switches, nested conditionals, and phase-dependent behavior.

7.5.2 Trade-offs

- **Aliasing / interference**: finite tables cause unrelated branches or histories to share entries. Global history can increase interference because many branches share the same GHR-dependent index.
- **Warmup cost**: history-based predictors may require more observations to converge.
- **Complexity**: combining history with indexing and multiple tables increases hardware complexity.
- **Sensitivity to phase changes**: global history can become stale when the program switches behavior abruptly.

In practice, modern CPUs typically combine multiple predictors (local + global + others) and select between them dynamically. This is the motivation for hybrid and tournament predictors introduced next.

Transition to Subsequent Chapters

Local history captures patterns intrinsic to a branch; global history captures correlations across different branches. Since real workloads contain both, modern CPUs commonly combine predictors and use a selection mechanism to choose the best source of prediction for each branch. The next chapter introduces hybrid and tournament predictors and explains why they are a practical necessity.

Chapter 8

Hybrid and Tournament Predictors

8.1 Why Single Predictors Are Not Enough

No single predictor family performs best on all branch behaviors. Real programs contain a mixture of:

- strongly biased branches (error exits, bounds checks),
- loop backedges (taken many times, not-taken once),
- periodic patterns (every k iterations),
- phase behavior (mode switches),
- cross-branch correlation (earlier decisions influence later outcomes),
- indirect control flow (virtual calls, jump tables).

A predictor optimized for bias (simple counters) fails on structured patterns. A predictor optimized for local patterns can fail on cross-branch correlation. A predictor optimized

for global correlation can suffer from interference and aliasing. Therefore, practical CPUs combine multiple predictors and choose among them.
The **hybrid** idea is simple:

> *Use more than one predictor, and select the one that has been more accurate for this branch.*

8.1.1 Example: Local wins on periodicity, global wins on correlation

```cpp
#include <cstdint>

std::uint64_t periodic(std::uint64_t n) {
    // Per-branch periodic pattern: local history is often effective.
    std::uint64_t c = 0;
    for (std::uint64_t i = 0; i < n; ++i) {
        if ((i & 7u) == 0u) ++c;
    }
    return c;
}

int correlated(int mode, int x) {
    // Cross-branch correlation: global history can be effective.
    if (mode == 0) {
        if (x < 10) return 1;
        return 2;
    } else {
        if (x < 10) return 3;
        return 4;
    }
}
```

A hybrid design can let the periodic branch lean on local history while the correlated branch leans on global history.

8.2 Combining Local and Global Predictors

A common hybrid pattern combines:

- a **local-history predictor** (per-branch history → pattern table),
- a **global-history predictor** (global history → pattern table),
- a **chooser** (meta-predictor) that decides which predictor to trust.

At fetch time, both local and global components produce candidate predictions. The chooser selects the final direction (and may participate in target prediction decisions as well). Conceptually:

```
local_pred  = LocalPredictor.predict(PC)
global_pred = GlobalPredictor.predict(PC, GHR)

choice      = Chooser.predict(PC, GHR)  // select local vs global

final_pred  = (choice == LOCAL) ? local_pred : global_pred
```

In practice, indexing functions typically mix branch address bits with history to reduce aliasing and make predictions branch-specific.

8.2.1 Example: Conceptual hybrid skeleton in code

```cpp
#include <cstdint>

struct Sat2 {
    std::uint8_t s = 1; // 0..3
    bool predict_taken() const { return s >= 2; }
    void update(bool taken) {
        if (taken) { if (s < 3) ++s; } else { if (s > 0) --s; }
```

```
    }
};

struct Hybrid {
    // Conceptual: very small tables.
    static constexpr std::uint32_t H = 8;
    static constexpr std::uint32_t PHT = 1u << H;

    std::uint8_t ghr = 0;

    Sat2 local[PHT]  = {};
    Sat2 global[PHT] = {};
    Sat2 chooser[PHT]= {}; // predicts whether to trust global (>=2) or
     ↪  local (<2)

    bool predict(std::uint32_t pc, std::uint8_t local_hist) const {
        bool lp = local[local_hist].predict_taken();
        bool gp = global[(std::uint8_t)(ghr ^ (pc &
         ↪  0xFF))].predict_taken();
        bool use_global = chooser[(std::uint8_t)(ghr)].predict_taken();
        return use_global ? gp : lp;
    }

    void update(std::uint32_t pc, std::uint8_t local_hist, bool actual,
                bool lp, bool gp) {
        local[local_hist].update(actual);
        global[(std::uint8_t)(ghr ^ (pc & 0xFF))].update(actual);

        // Train chooser only when predictors disagree; reinforce the
         ↪  correct one.
        if (lp != gp) {
            bool global_was_right = (gp == actual);
            chooser[(std::uint8_t)(ghr)].update(global_was_right);
```

```
        }

        ghr = (std::uint8_t)((ghr << 1) | (actual ? 1 : 0));
    }
};
```

This example is intentionally small and conceptual. Real predictors use larger tables, additional hashing, multiple components, and separate handling for different branch types.

8.3 Meta-Predictor Logic

The **meta-predictor** (chooser) is itself a predictor. Its job is not to predict taken/not-taken directly, but to predict **which component** is likely to be correct for this branch under current behavior.
A standard approach is a saturating counter:

- states biased toward **local** selection,
- states biased toward **global** selection.

Training policy (common and effective):

- Update the chooser only when local and global disagree.
- Move the chooser toward whichever component was correct.
- If both were correct or both were wrong, do not update (no signal).

This keeps chooser training stable and reduces noise.

8.3.1 Example: Chooser update truth table

```
local_pred  global_pred  actual  chooser_update
-----------------------------------------------
T           T            T       none
T           T            N       none
N           N            T       none
N           N            N       none

T           N            T       move chooser toward LOCAL
T           N            N       move chooser toward GLOBAL

N           T            T       move chooser toward GLOBAL
N           T            N       move chooser toward LOCAL
```

8.4 Selection and Adaptation

A hybrid predictor must adapt across:

- **different branches**: some are best served by local history, others by global history.
- **different phases**: a single branch can change behavior across time (e.g., initialization vs steady state).

Adaptation is complicated by aliasing:

- Multiple branches share table entries, which can corrupt learning.
- Global history can increase collisions because many branches consult the same GHR-derived indices.

Therefore, selection logic must be:

- responsive enough to track real phase changes,

- stable enough not to oscillate due to noise or interference.

8.4.1 Example: A branch that changes behavior by phase

```cpp
#include <cstddef>
#include <cstdint>

std::uint64_t phase_branch(const int* p, std::size_t n) {
    std::uint64_t s = 0;

    // Phase 1: warmup (predictable)
    for (std::size_t i = 0; i < n/8; ++i) {
        if ((p[i] & 1) == 0) s += (std::uint64_t)p[i];  // may be biased
    }

    // Phase 2: steady state (less predictable, different distribution)
    for (std::size_t i = n/8; i < n; ++i) {
        if ((p[i] & 1) == 0) s += (std::uint64_t)p[i];  // same branch,
        ↪  different behavior
    }

    return s;
}
```

The same branch can behave differently in different phases. A hybrid predictor aims to keep accuracy high by shifting reliance between components when needed.

8.5 Scalability in Modern CPUs

Modern CPUs must predict at very high bandwidth:

- multiple branches can be fetched per cycle,

- the predictor must respond with low latency,
- the predictor must scale across large code footprints and many active branches.

Scalability constraints drive design choices:

- **Multi-level structure**: a fast predictor provides immediate guesses; slower predictors refine accuracy.
- **Hashed indexing**: mixing PC bits with history reduces destructive aliasing without excessive storage.
- **Partitioned resources**: different branch types (conditional vs indirect vs return) may use specialized mechanisms.
- **Energy and area limits**: predictor storage is limited; designs trade table size against power/latency.

The important programmer-visible consequence is:

> *Predictability is a property of the runtime stream. Hybrid predictors improve average accuracy, but they cannot eliminate the cost of truly chaotic control flow.*

8.5.1 Example: Two styles of "hard" control flow

```
#include <cstddef>
#include <cstdint>

std::uint64_t random_branch(const int* p, std::size_t n) {
    // If p[i] behaves randomly, no predictor can achieve high accuracy.
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        if (p[i] & 1) s += (std::uint64_t)p[i];
```

```
    }
    return s;
}

using Fn = int(*)(int);

int indirect_targets(const Fn* table, std::size_t n, int x) {
    // Many varying call targets create target-prediction pressure.
    int r = 0;
    for (std::size_t i = 0; i < n; ++i) {
        r += table[i](x);
    }
    return r;
}
```

Hybrid direction prediction helps for structured branches, but it cannot “solve” randomness. For indirect branches, direction is trivial (always transfers) while the key challenge becomes target prediction, handled by separate mechanisms discussed later in the booklet.

Transition to Subsequent Chapters

Hybrid and tournament predictors address the reality that no single predictor model dominates across all workloads. The next chapter transitions from prediction to execution, explaining how speculative execution consumes predicted paths, how the CPU maintains precise architectural state, and why mispredictions trigger flush-and-restart behavior.

Chapter 9

Speculative Execution Fundamentals

9.1 Executing Before Branch Resolution

Speculative execution is the CPU's strategy for maintaining high throughput when control flow is uncertain. When a branch is fetched, its outcome is not yet known. Waiting would stall the frontend and starve the backend. Instead, the CPU:

- predicts the branch direction (and target when needed),
- continues fetching and decoding along the predicted path,
- executes instructions before the branch is resolved.

If the prediction is correct, the speculative work becomes useful work and retires normally. If the prediction is wrong, the CPU discards the wrong-path work and resumes from the correct path.

9.1.1 Example: The CPU must choose a path immediately

```
.intel_syntax noprefix

cmp  edi, 0
jl   negative        /* outcome not known at fetch time */
add  eax, 1          /* wrong-path if branch is actually taken */
jmp  done
negative:
sub  eax, 1
done:
```

The CPU cannot stop fetching after `jl`. It must fetch either the fall-through sequence or the target block. Prediction is the mechanism that chooses.

9.1.2 Example: Speculation keeps a tiny loop fast

```
#include <cstddef>
#include <cstdint>

std::uint64_t sum(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {  // branch predicted taken
    ↪  repeatedly
        s += (std::uint64_t)p[i];
    }
    return s;
}
```

The loop backedge is highly predictable. Speculation allows the CPU to keep fetching the next iterations without pausing at the branch.

9.2 Speculative vs Architectural State

A crucial distinction:

- **Architectural state**: what the ISA defines as the visible machine state (registers, flags, memory) at a precise point in program order.
- **Speculative (microarchitectural) state**: temporary internal state created while executing ahead of unresolved branches.

Speculative execution is safe for correctness because architectural state is updated only when instructions **retire/commit** in order. However, speculative work can still affect microarchitectural structures (e.g., caches), which matters for both performance and security. Correctness is about architectural state; performance is influenced by both.

9.2.1 Example: Speculation cannot change architectural state until commit

```
.intel_syntax noprefix

/* Architecturally, eax should update only if this path commits. */
cmp  edi, 0
jl   take
add  eax, 5          /* may execute speculatively */
jmp  done
take:
sub  eax, 5          /* may execute speculatively */
done:
```

Even if `add` executes on a wrong path, it must not become architecturally visible. The CPU enforces this by separating speculative results from committed results until retirement.

9.3 Commit and Rollback Mechanisms

To execute out of order and speculatively while preserving precise architectural behavior, modern CPUs use mechanisms that provide:

- **in-order retirement**: instructions become architecturally visible in program order,
- **precise recovery**: on misprediction or exception, the CPU can roll back to a well-defined state.

A practical mental model includes:

- **rename mappings**: architectural registers map to physical registers; speculative instructions write new physical registers.
- **a reorder structure**: tracks in-flight instructions and their completion status.
- **commit point**: when an instruction is known to be correct and can update architectural state.

On a misprediction, the CPU:

- discards younger speculative instructions,
- restores rename mappings to the state at the branch,
- redirects fetch to the correct path,
- continues execution.

9.3.1 Example: Why rollback must be precise (exceptions and correctness)

```
#include <cstdint>

int f(int* p, int x) {
    // If p is invalid, the store must not "partially happen"
    ↪ architecturally.
    if (x > 0) {
        *p = 123;       // may fault; must be precise
        return 1;
    }
    return 0;
}
```

If the CPU speculatively executes the store on a path that later turns out to be wrong, it still must not commit that store architecturally. Similarly, if a fault occurs, the program must observe precise exception behavior. Commit/rollback machinery ensures these properties.

9.3.2 Example: Branch misprediction triggers rollback

```
#include <cstddef>
#include <cstdint>

std::uint64_t branchy(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        int x = p[i];
        if (x & 1) {           // if mispredicted, wrong-path work is
        ↪ discarded
            s += (std::uint64_t)x;
        } else {
            s -= (std::uint64_t)x;
        }
```

```
    }
    return s;
}
```

When the branch resolves differently than predicted, the CPU squashes wrong-path instructions, restores the correct speculative context, and refills from the correct path.

9.4 Speculative Windows

The **speculative window** is the amount of work the CPU can have in flight beyond unresolved control-flow points. It is not a single number, but a dynamic range bounded by:

- frontend capacity (fetch/decode bandwidth and queue sizes),
- rename and scheduling resources (physical registers, issue queue capacity),
- reorder/retirement capacity (how many in-flight instructions can be tracked),
- branch resolution latency (how long branches remain unresolved).

A larger speculative window increases potential throughput when predictions are correct, but it also increases the amount of wasted work when predictions are wrong.

9.4.1 Example: Late-resolving branch increases wasted window

```
#include <cstddef>
#include <cstdint>

std::uint64_t sum_if(const int* p, std::size_t n, int t) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        int x = p[i];           // may arrive late
```

```
        if (x > t) {              // branch resolution waits for x
            s += (std::uint64_t)x;
        }
    }
    return s;
}
```

If `x` arrives late due to cache misses or dependent computation, the branch remains unresolved longer. The CPU may speculate further, so a misprediction discards more work and costs more cycles.

9.5 Interaction with Out-of-Order Execution

Out-of-order (OoO) execution and speculation are tightly coupled:

- OoO reorders execution of independent instructions to keep units busy.
- Speculation supplies those independent instructions by fetching beyond branches.
- Retirement remains in order to preserve architectural correctness.

This creates a powerful combination:

- The frontend predicts and provides a stream of instructions.
- The backend executes ready operations as soon as operands are available, not strictly in program order.
- Commit/rollback preserves the illusion of in-order execution.

But it also creates failure modes:

- A wrong prediction wastes OoO work and drains backend queues.

- A branch depending on a long dependency chain delays resolution, increasing speculative depth.
- Control hazards can turn a backend-capable workload into a frontend-bound workload.

9.5.1 Example: OoO finds work only if speculation supplies it

```cpp
#include <cstddef>
#include <cstdint>

std::uint64_t mixed(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        int x = p[i];
        s += (std::uint64_t)(x * 3 + 7);  // independent ALU work
        if (x < 0) {                      // control hazard
            s ^= 0x9e3779b97f4a7c15ull;
        }
    }
    return s;
}
```

When the branch is predictable, speculation keeps the stream flowing and OoO execution overlaps ALU operations efficiently. When the branch is unpredictable, frequent flushes reduce available independent work, and the OoO engine cannot maintain high throughput.

Transition to Subsequent Chapters

Speculation explains how the CPU can execute beyond unresolved branches without losing correctness. The next chapter focuses on speculative **memory** behavior: speculative loads, store buffers, forwarding, cache side effects, and the performance consequences of executing memory operations ahead of control-flow resolution.

Chapter 10

Speculation and Memory Operations

10.1 Speculative Loads

Speculation is not limited to arithmetic and control flow. Modern CPUs also perform **speculative memory operations**, especially loads, because memory latency is high and hiding it is essential for throughput.
A **speculative load** is a load that executes before the CPU can fully prove it is safe with respect to:

- the final control-flow path (branch outcome not yet resolved),
- ordering constraints with older stores whose addresses or data may not be known yet.

Speculative loads are valuable because they can:

- start cache accesses early,
- overlap memory latency with other work,
- enable the backend to stay busy.

Correctness is preserved because:

- load results are held in speculative state until commit,
- if a violation or misprediction is detected, the CPU squashes and re-executes as needed.

10.1.1 Example: Loads inside unpredictable branches still execute speculatively

```
#include <cstddef>
#include <cstdint>

std::uint64_t sum_conditional(const int* a, const int* b, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        if (a[i] & 1) {                // unpredictable branch
            s += (std::uint64_t)b[i]; // load may execute speculatively
        }
    }
    return s;
}
```

Even if the branch is later found not taken, the CPU may already have speculatively issued the load from `b[i]`. Architecturally, the load does not "happen" unless the path commits, but microarchitectural effects (such as cache state) may still change.

10.2 Load–Store Ordering Assumptions

A major difficulty in speculative memory execution is dealing with older stores. Consider a younger load:

- If it is known to be independent of all older stores, it can execute early.

- If it might alias an older store (same address), executing it early could read stale data.

To keep performance high, CPUs typically make **speculative assumptions** such as:

- most loads do not alias most older stores,
- many store addresses can be computed early enough,
- if a rare alias is detected later, recovery is cheaper than always waiting.

This is the load/store analog of branch prediction: *speculate for the common case, recover for the rare case*.

10.2.1 Example: Potential alias between earlier store and later load

```
#include <cstdint>

int store_then_load(int* p, int* q) {
    *p = 1;          // older store, address may be computed
    return *q;       // younger load: can it execute before store is known
     ↪  safe?
}
```

If `p == q`, the load must observe the value written by the store. If `p != q`, executing the load early can hide memory latency. CPUs often speculate that `p != q` unless proven otherwise.

10.2.2 Example: Address uncertainty increases the speculative risk

```
#include <cstdint>

int tricky(int* base, int i, int j) {
    base[i] = 7;          // store address depends on i
```

```
    return base[j];        // load address depends on j
}
```

Until `i` and `j` are known and effective addresses are computed, the CPU cannot know whether the store and load alias. Many designs allow the load to execute and later verify whether it was safe.

10.3 Store Buffers and Forwarding

Stores are typically not written directly to cache/memory immediately. Instead, they are placed in a **store buffer** (or store queue) until it is safe and efficient to make them globally visible.
Key purposes of a store buffer:

- decouple store retirement from cache/memory latency,
- combine and schedule writes efficiently,
- allow younger loads to proceed while stores are pending.

Store-to-load forwarding is the mechanism that ensures a younger load reads the value of an older store to the same address, even if the store has not yet been written to cache. Forwarding preserves the program-order illusion.

10.3.1 Example: Forwarding required for correctness

```
#include <cstdint>

int f(int* p) {
    *p = 123;      // store enters store buffer
    return *p;     // load must see 123, usually via forwarding
}
```

If the store is still in the store buffer when the load executes, the CPU must forward the stored value. If forwarding fails due to uncertainty or partial overlap, the CPU may delay the load or re-execute after resolving the store.

10.3.2 Example: Partial overlap can complicate forwarding

```
#include <cstdint>

struct Pair { std::uint32_t a, b; };

std::uint32_t g(Pair* p) {
    p->a = 0x11111111u;    // store 4 bytes
    p->b = 0x22222222u;    // store 4 bytes
    return p->a;           // load overlaps only with the first store
}
```

Forwarding logic must match address and size. When multiple stores are pending to nearby addresses, the CPU must select the youngest matching store or combine data correctly.

10.4 Cache Effects of Speculation

Even when wrong-path instructions do not commit architecturally, speculative memory operations can affect microarchitectural state:

- instruction cache and decoder streams change due to wrong-path fetch,
- data cache lines may be brought in by speculative loads,
- cache replacement state and prefetch behavior can shift.

From a performance perspective, speculative loads can be beneficial:

- if the prediction is correct, the cache line is already warm when needed.
- even if the prediction is wrong, the line might be useful soon due to locality.

But they can also be harmful:

- useless wrong-path loads can evict useful cache lines,
- wasted memory bandwidth can reduce performance on bandwidth-bound workloads.

10.4.1 Example: Wrong-path prefetching can help or hurt

```
#include <cstddef>
#include <cstdint>

std::uint64_t maybe_use(const int* hot, const int* cold, std::size_t n,
↪  const int* key) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        // key[i] may be unpredictable; CPU may speculatively touch cold[i]
        if (key[i] & 1) {
            s += (std::uint64_t)cold[i];
        } else {
            s += (std::uint64_t)hot[i];
        }
    }
    return s;
}
```

If `cold` is rarely used, speculative loads into `cold` may waste cache and bandwidth. If usage alternates frequently, speculative loads might partially warm the needed lines. The effect depends on predictability, locality, and cache capacity.

10.5 Memory Consistency Implications

Speculative memory execution must respect the architectural memory model. Two distinct notions are important:

- **Architectural ordering**: what the ISA guarantees other cores/threads can observe.
- **Microarchitectural speculation**: what the core tries internally to improve performance.

CPUs may execute loads and stores out of order internally, but they must ensure that:

- each thread observes its own program order according to the language/ISA rules,
- other threads observe memory effects in a manner consistent with the architectural model,
- fences and synchronization operations enforce required ordering.

For programmers, the practical consequence is:

> *Speculation changes when work is performed internally, but it must not violate the architectural memory model.*

However, speculation can still matter for performance and for the visibility of microarchitectural side effects. Synchronization constructs (atomics, fences) can restrict reordering and speculation, often reducing performance but providing correctness in concurrent code.

10.5.1 Example: Why ordering constraints limit speculation (conceptual)

```
#include <atomic>
```

```
int data = 0;
std::atomic<int> flag{0};

void producer() {
    data = 42;
    flag.store(1, std::memory_order_release);
}

int consumer() {
    if (flag.load(std::memory_order_acquire) == 1) {
        return data;  // must see 42 after acquire if producer signaled
    }
    return 0;
}
```

Acquire/release ordering prevents certain reorders that the CPU might otherwise attempt, ensuring correctness across threads. Internally, CPUs still speculate, but they must not allow outcomes that violate the required ordering.

Transition to Subsequent Chapters

Speculative execution of memory operations is essential for hiding latency, but it introduces subtle interactions: alias checks, forwarding, cache side effects, and ordering constraints. The next chapter focuses on the security and observability implications of speculative microarchitectural effects, and why speculation has consequences beyond pure performance.

Chapter 11

Security Implications of Speculation

11.1 Microarchitectural Side Effects

Speculative execution is architecturally invisible: wrong-path instructions do not retire and must not change architectural state. However, speculation can still change **microarchitectural state**, such as:

- cache contents and cache replacement state,
- TLB entries and page-walk caches,
- branch predictor state,
- prefetcher state,
- internal queues and contention patterns.

These effects are not defined by the ISA as part of the architectural contract, but they influence timing and resource behavior. That timing influence is the basis of many side-channel attacks.

11.1.1 Example: A wrong-path load can still warm a cache line

```
#include <cstddef>
#include <cstdint>

std::uint64_t speculative_touch(const int* a, const int* b, const int* key,
↪  std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        if (key[i] & 1) {
            // If mispredicted, this access may still bring b[i] into
            ↪  cache.
            s += (std::uint64_t)b[i];
        } else {
            s += (std::uint64_t)a[i];
        }
    }
    return s;
}
```

Even if the branch prediction is wrong and the path is squashed, the data cache may still contain the speculatively touched line. This can change later timings.

11.2 Architectural vs Observable State

Two notions of “state” matter:

- **Architectural state**: registers/memory as defined by the ISA at the retirement boundary.
- **Observable state**: anything an attacker can infer indirectly (often via timing), including effects of caches, predictors, and other shared resources.

A critical security lesson of speculation is:

Architectural rollback does not automatically undo microarchitectural footprints.

Thus, the system can remain correct by ISA rules while still leaking information through timing differences.

11.2.1 Example: Timing reveals cache residency (conceptual)

```
#include <cstdint>
#include <cstddef>

static inline std::uint64_t rdtsc_like(); // conceptual placeholder

std::uint64_t time_read(const std::uint8_t* p) {
    std::uint64_t t0 = rdtsc_like();
    volatile std::uint8_t x = *p;
    (void)x;
    std::uint64_t t1 = rdtsc_like();
    return t1 - t0;
}
```

If `*p` hits in cache, it is typically faster than a miss. Speculation can influence whether `p` is in cache, so timing can act as an observation channel.

11.3 Misprediction as a Leakage Channel

A misprediction creates a controlled period where the CPU executes along a wrong path. If that wrong path performs data-dependent memory accesses, it can imprint secret-dependent patterns into caches or other structures. Even though the CPU later squashes the wrong-path instructions, the footprint can remain observable.
For leakage, three ingredients are commonly present conceptually:

- **Transient execution**: instructions execute speculatively before the true control-flow is known.
- **Secret-dependent access**: speculative path reads a value that should not influence architecturally visible behavior.
- **Amplification**: the secret influences which cache line or resource is touched, making timing differences measurable.

11.3.1 Example: Secret-dependent index (conceptual pattern)

```
#include <cstddef>
#include <cstdint>

extern std::uint8_t secret;
extern std::uint8_t probe[256 * 4096];

void transient_pattern(std::size_t idx) {
    // Conceptual: if idx check is mispredicted as "in-bounds",
    // secret-dependent access can occur transiently.
    if (idx < 16) {
        std::uint8_t v = secret;                 // sensitive data
        volatile std::uint8_t x = probe[v * 4096]; // touches a
        ↪  secret-dependent line
        (void)x;
    }
}
```

Architecturally, an out-of-bounds `idx` should prevent the access. But if the CPU transiently executes the body due to misprediction, the cache footprint of `probe[v * 4096]` may reveal information about `v`.

11.4 Spectre-Class Vulnerabilities (Conceptual)

"Spectre-class" vulnerabilities (conceptually) exploit the fact that speculative execution can:

- follow a mispredicted path,
- perform operations that access secrets,
- leave observable microarchitectural traces,
- then be squashed architecturally.

From a programmer's performance perspective, the key conceptual takeaway is that predictors and caches are not merely optimization structures; they are also **shared microarchitectural resources** that can encode information.
This booklet does not attempt to be a security manual. The goal here is to provide a correct conceptual model that explains why mitigations exist and why they can affect performance.

11.4.1 Example: Training, transient use, and observation (conceptual steps)

```
1) Train predictor to favor a particular branch direction.
2) Trigger misprediction so CPU transiently executes an unintended path.
3) During transient execution, touch a cache line based on secret-derived
↪  value.
4) After squash, measure which cache line is fast to infer information.
```

The important point is not the exact exploit technique, but the mechanism: *transient execution can create observable footprints even when architectural correctness is preserved.*

11.5 Performance vs Security Trade-Offs

Mitigations that reduce speculative leakage typically do so by restricting speculation or by preventing the speculative path from creating a usable footprint. These restrictions can reduce performance because speculation is a major source of throughput.
Common mitigation categories conceptually include:

- **Speculation barriers**: prevent certain speculative behaviors across sensitive boundaries.
- **Retpoline-style techniques**: reduce exposure of certain indirect branch prediction behaviors.
- **Fence usage and serialization**: constrain execution ordering in sensitive sequences.
- **Isolation strategies**: reduce sharing of microarchitectural state across security domains.

Programmers should understand two practical consequences:

- Security hardening can increase branch and indirect-call overhead.
- Some mitigations reduce effective predictor quality or limit speculative depth, lowering IPC.

11.5.1 Example: A speculation barrier changes the performance profile (conceptual)

```
.intel_syntax noprefix

/* Conceptual sequence: after a bounds check, prevent further
↪  speculation
   from using transiently computed values in certain contexts. */
```

```
/* if (idx < n) */
cmp  rdi, rsi
jae  out

/* barrier-like instruction may be placed here by compiler or
 ↪  manually
   in hardened code paths (exact instruction depends on platform). */

mov  al, byte ptr [rdx + rdi]   /* safe access if in-bounds */
jmp  done

out:
xor  eax, eax
done:
```

Adding barriers can reduce performance by limiting speculation and by introducing serialization points. However, such measures may be required in high-assurance code that crosses trust boundaries.

11.5.2 Example: Branchless transformations can be security-relevant but not universal

```
#include <cstddef>
#include <cstdint>

// Conceptual: attempt to avoid a branch that could be mistrained.
// Note: branchless code is not automatically "secure"; it is one tool.
std::uint8_t masked_load(const std::uint8_t* a, std::size_t n, std::size_t
 ↪  idx) {
```

```
    std::size_t m = (idx < n) ? ~(std::size_t)0 : 0;
    std::size_t safe = idx & m;          // forces idx=0 when out of range
    return a[safe];
}
```

This shows the general idea of controlling indices without a control-flow decision. Whether such approaches are appropriate depends on the threat model, the compiler, and the platform.

Transition to Subsequent Chapters

Speculation is a performance feature with observable microarchitectural consequences. Understanding those consequences explains why certain mitigations exist and why they can change performance characteristics. The final chapter focuses purely on programming practice: how to write control flow that modern CPUs predict well, how to avoid branch-hostile patterns in hot paths, and how to reason about when branchless techniques help or hurt.

Chapter 12

Writing Branch-Friendly Code

12.1 Predictable vs Unpredictable Branches

A branch is **predictable** when its outcome is strongly biased or follows a stable pattern that the predictor can learn. It is **unpredictable** when outcomes are close to random, frequently alternate, or change rapidly across phases.
Practical signals of predictability:

- **Strong bias**: taken rate near 0% or near 100% over the hot interval.
- **Stable pattern**: loop backedges, periodic checks, state-machine cycles.
- **Stable correlation**: branch outcomes depend on earlier branches in a consistent way.

Common sources of unpredictability:

- **Data-dependent noise**: outcomes depend on input with high entropy.
- **Alternation**: true/false toggling or short-period patterns not captured by the predictor used.

- **Phase changes**: the “likely” path flips across program phases.

12.1.1 Example: Predictable error check vs unpredictable data test

```
int parse_digit(char c) {
    // Often predictable: invalid input is usually rare in well-formed
    ↪  streams.
    if (c < '0'  c > '9') return -1;
    return c - '0';
}

#include <cstddef>
#include <cstdint>

std::uint64_t count_parity(const int* p, std::size_t n) {
    // Often unpredictable if parity is random-like.
    std::uint64_t c = 0;
    for (std::size_t i = 0; i < n; ++i) {
        if (p[i] & 1) ++c;
    }
    return c;
}
```

The first branch tends to be strongly biased; the second may be close to 50/50 depending on the data source.

12.1.2 Example: Alternation is branch-hostile

```
#include <cstdint>

std::uint64_t alternating(std::uint64_t n) {
    std::uint64_t c = 0;
    for (std::uint64_t i = 0; i < n; ++i) {
```

```
        if ((i & 1u) == 0u) ++c;    // true, false, true, false, ...
    }
    return c;
}
```

Alternation can defeat simple predictors and reduce effective frontend throughput, even though the computation is trivial.

12.2 Data-Oriented Control Flow

When branches are unpredictable, the core optimization approach is to reduce **control dependence** by shifting decisions into **dataflow**:

- compute masks, indices, or predicates,
- use arithmetic/bitwise selection,
- process data in groups where control is more uniform,
- separate hot predictable fast paths from cold complex paths.

This is often called a data-oriented or branch-minimizing style, and it is especially effective in hot loops where branch mispredictions dominate.

12.2.1 Example: Split hot/cold paths by filtering

```
#include <cstddef>
#include <cstdint>
#include <vector>

std::uint64_t sum_positives_filtered(const int* p, std::size_t n) {
```

```
    // Phase 1: filter indices of positives (can be vectorized or optimized
    ↪  separately).
    std::vector<std::size_t> idx;
    idx.reserve(n);

    for (std::size_t i = 0; i < n; ++i) {
        if (p[i] > 0) idx.push_back(i);
    }

    // Phase 2: sum only positives, now branch-free.
    std::uint64_t s = 0;
    for (std::size_t i : idx) {
        s += (std::uint64_t)p[i];
    }
    return s;
}
```

This trades memory and two passes for a branch-free summation phase. It can win when mispredictions are expensive and the dataset is large enough.

12.2.2 Example: Reorder data to increase locality and predictability

```
#include <algorithm>
#include <cstddef>
#include <cstdint>
#include <vector>

std::uint64_t sum_clustered(std::vector<int> v) {
    // Group values to create a highly biased branch region.
    std::stable_partition(v.begin(), v.end(), [](int x){ return x > 0; });

    std::uint64_t s = 0;
    for (int x : v) {
```

```
        if (x > 0) s += (std::uint64_t)x; // now: long run of taken, then
        ↪  long run of not-taken
    }
    return s;
}
```

Grouping by predicate can drastically improve predictability and instruction-cache behavior, at the cost of a preprocessing step.

12.3 Branchless Programming Techniques

Branchless programming removes a control-flow decision and replaces it with arithmetic selection. This can reduce misprediction penalties but may increase:

- instruction count,
- data dependency chains,
- register pressure,
- power usage.

Therefore, branchless forms are best used when the original branch is genuinely unpredictable and on the critical path.

12.3.1 Example: Conditional selection with the ternary operator

```
#include <cstdint>

int clamp_nonnegative(int x) {
    // This may compile to cmov or to a branch depending on optimization
    ↪  and target.
```

```
    return (x < 0) ? 0 : x;
}
```

The source expresses branchless selection. Whether it becomes branchless machine code depends on the compiler and the target.

12.3.2 Example: Mask-based selection (explicit dataflow)

```
#include <cstdint>

std::uint32_t select_u32(std::uint32_t cond, std::uint32_t a, std::uint32_t
↪  b) {
    // cond must be 0 or 1
    std::uint32_t mask = (std::uint32_t)-(std::int32_t)cond; // 0x00000000
    ↪  or 0xFFFFFFFF
    return (a & mask) | (b & ~mask);
}
```

This pattern makes the selection explicit. It avoids a branch but introduces extra ALU operations.

12.3.3 Example: Branch vs cmov-style lowering (conceptual x86-64)

```
.intel_syntax noprefix

/* branch form */
cmp  edi, 0
jl   neg
mov  eax, edi
jmp  done
neg:
```

```
xor  eax, eax
done:

/* conditional-move style (conceptual) */
cmp  edi, 0
mov  eax, edi
mov  edx, 0
cmovl eax, edx
```

The conditional-move form avoids a control hazard but may create additional data dependencies. It is often a good trade when the branch is unpredictable and the selected values are already available.

12.4 Loop Structure and Branch Behavior

Loops are the dominant source of conditional branches in hot code. Many loop branches are highly predictable, but loop bodies often contain data-dependent branches that are not.
Key loop guidelines:

- Keep loop backedges simple; they are usually predicted well.
- Minimize unpredictable branches inside the loop body.
- Prefer early exits only when the exit condition is rare and predictable.
- Consider unrolling when it reduces branch frequency and increases ILP, but measure because unrolling can increase I-cache pressure.

12.4.1 Example: Highly predictable backedge

```
#include <cstddef>
```

```cpp
#include <cstdint>

std::uint64_t sum(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        s += (std::uint64_t)p[i];
    }
    return s;
}
```

The loop branch is taken many times and not taken once, which is structurally friendly to dynamic predictors.

12.4.2 Example: Unpredictable branch inside the loop body

```cpp
#include <cstddef>
#include <cstdint>

std::uint64_t sum_if_random(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        int x = p[i];
        if (x & 1) s += (std::uint64_t)x; // may be close to 50/50
    }
    return s;
}
```

Here the backedge is fine; the data-dependent branch is the likely performance limiter.

12.4.3 Example: Reduce branch frequency by unrolling (conceptual)

```cpp
#include <cstddef>
#include <cstdint>
```

```
std::uint64_t sum_unrolled2(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    std::size_t i = 0;

    for (; i + 1 < n; i += 2) {
        s += (std::uint64_t)p[i];
        s += (std::uint64_t)p[i + 1];
    }
    for (; i < n; ++i) {
        s += (std::uint64_t)p[i];
    }
    return s;
}
```

Unrolling reduces loop-branch frequency and can increase instruction-level parallelism, but it can also increase code size.

12.5 Practical Optimization Guidelines

Branch optimization should be driven by measurement and guided by predictable structure:

- **First: identify hot branches.** Optimize only branches in hot loops or hot paths.
- **Measure predictability.** If a branch is already highly predictable, branchless rewrites often lose.
- **Make the hot path straight-line.** Structure code so the common case is fall-through and the rare case exits early.
- **Avoid unpredictable conditionals inside tight loops.** Use filtering, partitioning, or branchless selection if needed.

- **Be cautious with hints.** Hints help only if they reflect reality; incorrect hints can hurt.
- **Consider cache and code size.** A branchless transformation that expands code can degrade I-cache behavior.
- **Prefer stable data layouts.** Grouping similar cases together improves predictability and locality.

12.5.1 Example: Early-exit rare path keeps hot path linear

```cpp
#include <cstddef>
#include <cstdint>

std::uint64_t process(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        int x = p[i];

        // Rare case exits early: avoids mixing slow path in the hot
        ↪  stream.
        if (x == 0) return s;

        // Hot path: straight-line work.
        s += (std::uint64_t)(x * 3 + 7);
    }
    return s;
}
```

When `x == 0` is genuinely rare, the branch is highly predictable (mostly not taken), and the hot path stays compact and fetch-friendly.

12.5.2 Example: Branchless accumulation using boolean-to-integer

```cpp
#include <cstddef>
#include <cstdint>

std::uint64_t sum_positives_branchless(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        int x = p[i];
        // (x > 0) is 0 or 1; multiply selects contribution.
        s += (std::uint64_t)(x > 0) * (std::uint64_t)(x);
    }
    return s;
}
```

This avoids a branch but adds operations and dependencies. It is often beneficial when the original branch is unpredictable and the loop is hot.

Final Note

Writing branch-friendly code means aligning control flow with what modern CPUs can predict and fetch efficiently. Do not fight the predictor when it already succeeds; instead, remove or restructure only those branches whose unpredictability is proven to dominate the hot path. Measure in steady state, with realistic inputs, and treat code size and memory behavior as first-class constraints.

Appendices

Appendix A — Terminology and Predictor Glossary

Branch Direction

Branch direction describes the binary outcome of a conditional branch instruction:

- **Taken**: control flow transfers to the branch target address.
- **Not-taken**: control flow continues at the fall-through (next sequential) address.

Direction prediction is distinct from **target prediction**. For direct conditional branches, the target address is encoded and known; only the direction is uncertain. For indirect branches, the direction is always a transfer, while the target is unknown.

Example: Direction determines the next PC

```
.intel_syntax noprefix

cmp  edi, 0
jl   L_taken        /* taken vs not-taken selects next PC */
add  eax, 1         /* executed only if branch is not-taken */
jmp  L_done
```

```
L_taken:
sub  eax, 1
L_done:
```

Practical note

Direction mispredictions are costly because they invalidate the frontend's speculative work and force a pipeline redirect.

Taken and Not-Taken

Taken and **not-taken** describe whether a branch transfers control, not whether the branch instruction itself executes. The instruction always executes; the outcome only determines the next fetch address.
Typical structural patterns include:

- **Loop backedges**: taken for most iterations, not-taken once at loop exit.
- **Fast-path checks**: usually not-taken, with rare taken paths for errors or slow cases.

Example: Loop backedge behavior

```
#include <cstddef>
#include <cstdint>

std::uint64_t sum(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {  // taken many times, not-taken
    ↪  once
        s += (std::uint64_t)p[i];
    }
    return s;
}
```

Example: Rare taken branch with fast fall-through

```
int parse_digit(char c) {
    if (c < '0'  c > '9') return -1;  // rarely taken
    return c - '0';                     // common fall-through
}
```

Prediction Accuracy

Prediction accuracy is the fraction of branch predictions that match the resolved outcomes over a given execution window.
Two related concepts matter in practice:

- **Misprediction rate**: the fraction of wrong predictions.
- **Misprediction cost**: the cycle penalty paid for each wrong prediction.

Even a low misprediction rate can dominate performance when:

- the branch executes very frequently,
- the loop body is small,
- the pipeline is deep and recovery latency is high.

Example: Accuracy depends on data distribution

```
#include <cstddef>
#include <cstdint>

std::uint64_t count_positive(const int* p, std::size_t n) {
    std::uint64_t c = 0;
    for (std::size_t i = 0; i < n; ++i) {
        if (p[i] > 0) ++c;
```

```
    }
    return c;
}
```

If positives are clustered, accuracy is high. If signs are random-like, accuracy degrades.

Speculative Execution Window

The **speculative execution window** is the amount of work in flight beyond unresolved branches and other unresolved control-flow decisions.
It is bounded by:

- frontend fetch and decode bandwidth,
- rename, scheduling, and reorder capacities,
- branch resolution latency,
- memory latency that delays branch operands.

A larger window:

- increases throughput when predictions are correct,
- increases wasted work and recovery cost when predictions are wrong.

Example: Late data extends speculation depth

```
#include <cstddef>
#include <cstdint>

std::uint64_t sum_if(const int* p, std::size_t n, int t) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
```

```
        int x = p[i];         // late load delays branch resolution
        if (x > t) s += (std::uint64_t)x;
    }
    return s;
}
```

When `x` is delayed by cache misses, speculation proceeds further before the branch resolves.

Commit Point

The **commit point** (also called **retirement**) is where an instruction's effects become architecturally visible.
Before commit:

- results live in speculative physical registers,
- stores reside in internal buffers,
- wrong-path work can be discarded safely.

Commit typically occurs in program order to ensure:

- precise exceptions,
- clean rollback on misprediction,
- architectural state consistent with the ISA.

Example: Speculative updates vs architectural state

```
.intel_syntax noprefix

cmp  edi, 0
```

```
jl   L_take
add  eax, 5          /* speculative until commit */
jmp  L_done
L_take:
sub  eax, 5          /* speculative until commit */
L_done:
```

Only instructions that reach the commit point on the correct path update architectural state; wrong-path effects are squashed.

Example: Commit guarantees precise behavior

```
#include <cstdint>

int f(int* p, int x) {
    if (x > 0) {
        *p = 123;    // becomes architectural only if the path commits
        return 1;
    }
    return 0;
}
```

The commit point ensures that stores, register updates, and exceptions appear as if the program executed strictly in order, even though execution was speculative and out of order internally.

Appendix B — Common Branch Prediction Pitfalls

Data-Dependent Branch Patterns

A frequent mistake is assuming that “the same code” has the same predictability. In reality, predictability is often a property of the **runtime data distribution**. Branches driven by high-entropy inputs can behave close to random and defeat predictors.

Common data-dependent patterns that cause trouble:

- **Near 50/50 outcomes**: minimal bias produces high uncertainty.
- **Frequent alternation**: short-period patterns can cause repeated mispredictions.
- **Phase shifts**: predictability changes across program phases (warmup vs steady state).
- **Indirect target variability**: virtual calls / function pointers that change targets often.

Example: Same branch, different predictability across datasets

```
#include <cstddef>
#include <cstdint>

std::uint64_t count_positive(const int* p, std::size_t n) {
    std::uint64_t c = 0;
    for (std::size_t i = 0; i < n; ++i) {
        if (p[i] > 0) ++c;
    }
    return c;
}
```

If positives are clustered (many positives then many negatives), the branch becomes highly predictable. If signs are random-like, it becomes much harder to predict.

Example: Alternation is branch-hostile

```
#include <cstdint>

std::uint64_t alternating(std::uint64_t n) {
    std::uint64_t c = 0;
    for (std::uint64_t i = 0; i < n; ++i) {
        if ((i & 1u) == 0u) ++c;  // true, false, true, false, ...
```

```
    }
    return c;
}
```

Even though the loop body is tiny, mispredictions can dominate.

Over-Optimization Without Measurement

A common pitfall is rewriting code to "avoid branches" without proving that branches are the bottleneck. Branchless rewrites can lose due to:

- increased instruction count,
- longer dependency chains (less ILP),
- higher register pressure (spills),
- increased code size (worse I-cache behavior),
- missed vectorization opportunities due to added complexity.

The correct rule:

> **Do not remove a branch unless it is on the hot path and measurably mispredicting.**

Example: Branchless is not automatically faster

```
#include <cstddef>
#include <cstdint>

std::uint64_t sum_if_branch(const int* p, std::size_t n) {
    std::uint64_t s = 0;
```

```
    for (std::size_t i = 0; i < n; ++i) {
        int x = p[i];
        if (x > 0) s += (std::uint64_t)x;  // can be very fast if
        ↪  predictable
    }
    return s;
}

std::uint64_t sum_if_branchless(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        int x = p[i];
        s += (std::uint64_t)(x > 0) * (std::uint64_t)x; // avoids branch,
        ↪  adds ops + deps
    }
    return s;
}
```

If the branch is highly predictable, the branch version can outperform the branchless version. If the branch is truly unpredictable, the branchless version may win.

Example: Turning a branch into a cmov may increase dependency pressure

```
.intel_syntax noprefix

/* branch form */
cmp  edi, 0
jl   L_neg
mov  eax, edi
jmp  L_done
L_neg:
xor  eax, eax
```

```
L_done:

/* cmov-style form (conceptual) */
cmp  edi, 0
mov  eax, edi
mov  edx, 0
cmovl eax, edx
```

The cmov-style form avoids a control hazard but can create data dependencies that reduce parallelism. Which is better depends on predictability and surrounding context.

Misinterpreting Benchmark Results

Branch prediction behavior is often mis-measured due to benchmark design errors. Common mistakes:

- **Measuring cold start**: predictor and caches are not warmed, inflating mispredictions.
- **Unrealistic inputs**: synthetic random data may not match real distributions.
- **Dead-code elimination**: compiler removes work, turning the benchmark into nonsense.
- **Timer noise**: OS scheduling, frequency scaling, and interrupts dominate short runs.
- **Mixed effects**: cache misses and memory bandwidth issues masquerade as branch effects.

Measurement discipline:

- warm up before timing,
- keep input distributions realistic and controlled,

- prevent trivial optimization removal,
- run long enough to amortize noise,
- interpret results with hardware counters when available.

Example: Warmup then measure

```cpp
#include <chrono>
#include <cstddef>
#include <cstdint>
#include <vector>

std::uint64_t count_positive(const std::vector<int>& a);

std::uint64_t run_iters(const std::vector<int>& a, int iters) {
    std::uint64_t s = 0;
    for (int k = 0; k < iters; ++k) s += count_positive(a);
    return s;
}

double time_seconds(const std::vector<int>& a, int warm, int iters) {
    volatile std::uint64_t sink = run_iters(a, warm); (void)sink;

    auto t0 = std::chrono::steady_clock::now();
    sink = run_iters(a, iters);
    auto t1 = std::chrono::steady_clock::now();
    (void)sink;

    std::chrono::duration<double> dt = t1 - t0;
    return dt.count();
}
```

Without warmup, you often measure predictor cold-start rather than steady state.

Example: Preventing dead-code elimination

```cpp
#include <cstdint>

static inline void consume(std::uint64_t x) {
    volatile std::uint64_t sink = x;
    (void)sink;
}
```

Benchmarks that do not consume results risk being optimized away.

Hardware-Specific Behavior

Branch prediction is highly microarchitecture-dependent. Pitfalls include assuming that:

- a transformation that helps on one CPU helps on all CPUs,
- the same compiler produces the same branch structure across targets,
- unrolling or inlining always improves predictability,
- a branch hint has uniform impact across compilers and processors.

What varies across CPUs:

- predictor structure and capacity (aliasing behavior),
- frontend width and fetch policy,
- misprediction recovery latency,
- indirect branch target prediction behavior,
- interaction with I-cache, TLB, and prefetchers.

Therefore:

- measure on the target CPU that matters,
- treat branch hints as optional, not guaranteed,
- keep optimizations readable and justified by evidence.

Example: Branch layout changes with compilation choices

```
int clamp_nonnegative(int x) {
    return (x < 0) ? 0 : x;  // may become a branch or cmov depending on
    ↪  target + flags
}
```

Different compilers, optimization levels, and targets can produce different control-flow forms, changing prediction and performance behavior even when the source code is identical.

Appendix C — Practical Rules of Thumb

When Branches Are Cheap

Branches are often “cheap” (low net cost) when they are **highly predictable** and do not disrupt the frontend:

- **Strongly biased outcomes**: almost always taken or almost always not-taken.
- **Structured loops**: backedges taken many times, not-taken once at exit.
- **Rare slow paths**: error handling or uncommon cases that are truly rare.
- **Branch resolves early**: branch condition depends on already-available registers, not on late memory.

In these cases, the predictor learns quickly, the pipeline stays full, and the CPU pays little or no recovery cost.

Example: Cheap branch due to strong bias (rare error path)

```
int parse_digit(char c) {
    if (c < '0'  c > '9') return -1;  // typically rare: predictable
    ↪  not-taken
    return c - '0';
}
```

Example: Cheap branch due to loop structure

```
#include <cstddef>
#include <cstdint>

std::uint64_t sum(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {  // backedge is highly
    ↪  predictable
        s += (std::uint64_t)p[i];
    }
    return s;
}
```

Example: Early-resolving condition avoids long speculative windows

```
.intel_syntax noprefix

/* Condition uses registers already available: resolves quickly */
test edi, edi
jz   L_zero
add  eax, 1
```

```
jmp  L_done
L_zero:
xor  eax, eax
L_done:
```

When Branchless Code Wins

Branchless transformations tend to win when the branch is **both hot and unpredictable**, and the branchless replacement does not create worse bottlenecks.
Strong signals that branchless code can win:

- **Outcome near 50/50** on the hot path.
- **High-entropy data dependence** (effectively random).
- **Short loop body** where misprediction recovery dominates total cycles.
- **Work per iteration is small** and easily expressed as arithmetic selection.

But branchless can lose when it increases:

- instruction count and decode pressure,
- dependency chains (reducing ILP),
- register pressure (spills),
- code size (hurting I-cache),
- memory traffic.

Example: Branchless accumulation using predicate-to-integer

```
#include <cstddef>
#include <cstdint>

std::uint64_t sum_positives_branchless(const int* p, std::size_t n) {
    std::uint64_t s = 0;
    for (std::size_t i = 0; i < n; ++i) {
        int x = p[i];
        s += (std::uint64_t)(x > 0) * (std::uint64_t)x;
    }
    return s;
}
```

Example: Explicit mask selection

```
#include <cstdint>

std::uint32_t select_u32(std::uint32_t cond01, std::uint32_t a,
↪   std::uint32_t b) {
    std::uint32_t mask = (std::uint32_t)-(std::int32_t)cond01; // 0 or
    ↪   0xFFFFFFFF
    return (a & mask) | (b & ~mask);
}
```

Example: Branch vs conditional-move trade-off (conceptual x86-64)

```
.intel_syntax noprefix

/* branch form */
cmp  edi, 0
jl   L_neg
mov  eax, edi
```

```
jmp  L_done
L_neg:
xor  eax, eax
L_done:

/* cmov-style form (conceptual) */
cmp  edi, 0
mov  eax, edi
mov  edx, 0
cmovl eax, edx
```

The cmov form avoids control hazards but can increase data dependencies. It tends to help when the branch is unpredictable and surrounding code can tolerate the dependency chain.

Measuring Mispredictions Correctly

Reliable branch conclusions require measuring the right thing under stable conditions.
Checklist:

- **Warm up**: allow caches and predictors to stabilize before timing.
- **Use realistic inputs**: branch predictability depends on real data distributions.
- **Avoid dead-code elimination**: ensure results are consumed.
- **Separate effects**: cache misses and bandwidth limits can look like branch problems.
- **Use steady-state timing**: run long enough to amortize OS noise and timers.
- **Use hardware counters when available**: look at mispredicted-branch events and frontend-stall events.

Example: Warmup + measurement structure

```cpp
#include <chrono>
#include <cstddef>
#include <cstdint>
#include <vector>

std::uint64_t kernel(const std::vector<int>& a);

static inline void consume(std::uint64_t x) {
    volatile std::uint64_t sink = x;
    (void)sink;
}

double bench_seconds(const std::vector<int>& a, int warm_iters, int
↪  meas_iters) {
    consume([&]{
        std::uint64_t s = 0;
        for (int k = 0; k < warm_iters; ++k) s += kernel(a);
        return s;
    }());

    auto t0 = std::chrono::steady_clock::now();
    std::uint64_t s = 0;
    for (int k = 0; k < meas_iters; ++k) s += kernel(a);
    auto t1 = std::chrono::steady_clock::now();
    consume(s);

    std::chrono::duration<double> dt = t1 - t0;
    return dt.count();
}
```

Example: Control data distributions explicitly

```
#include <cstddef>
#include <vector>

std::vector<int> make_clustered(std::size_t n) {
    std::vector<int> v(n);
    for (std::size_t i = 0; i < n; ++i) v[i] = (i < n/2) ? 1 : -1; // long
    ↪ runs
    return v;
}

std::vector<int> make_alternating(std::size_t n) {
    std::vector<int> v(n);
    for (std::size_t i = 0; i < n; ++i) v[i] = (i & 1) ? 1 : -1;   //
    ↪ alternation
    return v;
}
```

Clustered data often yields high predictability; alternating data stresses many predictors.

Portability Across Microarchitectures

Branch prediction performance is **microarchitecture-specific**. Code that helps on one CPU may be neutral or harmful on another due to differences in:

- predictor design and capacity (aliasing behavior),
- frontend width and fetch policy,
- misprediction recovery latency,
- indirect branch handling,

- I-cache and TLB behavior.

Portable guidance:

- Prefer **clean structure**: common case as fall-through, rare case as early exit.
- Apply branchless rewrites only to **hot, proven-unpredictable** branches.
- Minimize code size blowups; instruction-cache pressure is cross-platform.
- Benchmark on the target CPU(s) that matter; do not generalize from one machine.
- Keep optimizations maintainable and reversible.

Example: Same source, different lowering across targets

```
int clamp_nonnegative(int x) {
    return (x < 0) ? 0 : x;  // may compile to a branch or to cmov
    ↪  depending on target/flags
}
```

Even when the source is unchanged, the control-flow shape and predictor interaction can vary with compiler, optimization level, and microarchitecture.

Closing note

Use these rules as fast heuristics, not as proof. For hot code, the only reliable method is: measure steady-state behavior with realistic inputs and confirm whether the bottleneck is branch misprediction, frontend bandwidth, cache/memory, or dependency limits.

References

CPU Microarchitecture Manuals

x86 / x86-64

- Intel® 64 and IA-32 Architectures Optimization Reference Manual.
- Intel® 64 and IA-32 Architectures Software Developer's Manual (SDM), especially the performance monitoring (PMU) and memory-ordering topics.
- Intel® Architecture Instruction Set Extensions and Future Features Programming Reference (for ISA-level details that interact with control flow and speculation).
- AMD64 Architecture Programmer's Manual, Vol. 1–5 (AMD), including memory model, performance monitoring, and microarchitectural guidance.
- AMD Software Optimization Guide for AMD Family of processors (family-specific microarchitecture tuning notes).

ARM / AArch64

- Arm® Architecture Reference Manual for A-profile Architecture (Armv8-A / Armv9-A), including memory model and exception behavior.

- Arm® Cortex-A Series Programmer's Guide / Software Optimization Guides (core-specific guidance on control flow and performance).
- Arm® System Developer's Guide (performance analysis and system-level tuning patterns).

RISC-V (Conceptual and Systems Context)

- The RISC-V Instruction Set Manual (Unprivileged ISA), for control-flow instruction semantics.
- The RISC-V Privileged Architecture Specification, for trap/exception semantics relevant to precise state and recovery.
- RISC-V psABI documentation (calling conventions and code generation constraints that influence branch shape and layout).

Academic Branch Prediction Research

Foundational predictor concepts

- Classic two-bit saturating-counter predictors (state-machine hysteresis model).
- Two-level adaptive predictors (history-based prediction: local history and global history).
- Correlation-based prediction (global-history-driven predictors and branch correlation).
- Hybrid and tournament predictors (meta-predictor selection among components).

Representative research threads (topic categories)

- History length and aliasing trade-offs (how table size and indexing affect interference).
- Geometric and tagged predictors (reducing aliasing while capturing long histories).
- Indirect branch prediction and return prediction structures (BTB variants, RAS behavior).
- Speculative execution side effects and transient-execution attack models (conceptual foundations of Spectre-class behavior).

Suggested academic reading venues

- ISCA (International Symposium on Computer Architecture) proceedings.
- MICRO (IEEE/ACM International Symposium on Microarchitecture) proceedings.
- HPCA (IEEE International Symposium on High-Performance Computer Architecture) proceedings.
- ACM Transactions on Architecture and Code Optimization (TACO).

Compiler Optimization Documentation

Core compiler toolchains

- LLVM/Clang documentation on optimization passes, inlining, loop transforms, and profile-guided optimization (PGO).
- GCC documentation on optimization options, inlining heuristics, if-conversion, and PGO/LTO workflows.

- MSVC documentation on optimization switches, PGO, LTCG, and code generation settings affecting control flow.

Key topics to study (compiler-side)

- Block layout and fall-through shaping (placing the likely path as linear code).
- If-conversion and predication/conditional-move formation (reducing unpredictable branches).
- Loop unrolling, peeling, and vectorization interactions with branch structure.
- Profile-guided optimization (training-based branch probability and layout decisions).
- Link-time optimization (whole-program layout and inlining affecting branch locality).

Example: Compiler-visible patterns that impact prediction

```
int fast_path(int x) {
    if (x == 0) return 0;       // rare case as early exit can keep hot path
    ↪  linear
    return x + 1;
}

int clamp_nonnegative(int x) {
    return (x < 0) ? 0 : x;     // may become a branch or a conditional move
}
```

Performance Analysis and Profiling Literature

Hardware performance counters and methodology

- PMU-based measurement methodology: interpreting branch-mispredict events, frontend stalls, and cycles-per-instruction metrics.
- Benchmarking discipline: warmup vs steady state, controlling input distributions, preventing dead-code elimination.
- Separating branch effects from cache/memory effects: identifying whether the bottleneck is control, bandwidth, or latency.

Tools and workflows (conceptual study topics)

- Sampling profilers vs instrumentation: overhead trade-offs and attribution accuracy.
- Top-down performance analysis approaches (frontend bound vs backend bound breakdown).
- Microbenchmark pitfalls and validation techniques (repeatability, noise control, and statistical confidence).

Example: Warmup then measure (baseline pattern)

```cpp
#include <chrono>
#include <cstddef>
#include <cstdint>
#include <vector>

std::uint64_t kernel(const std::vector<int>& a);

static inline void consume(std::uint64_t x) {
    volatile std::uint64_t sink = x;
```

```
    (void)sink;
}

double bench_seconds(const std::vector<int>& a, int warm_iters, int
↪  meas_iters) {
    std::uint64_t warm = 0;
    for (int k = 0; k < warm_iters; ++k) warm += kernel(a);
    consume(warm);

    auto t0 = std::chrono::steady_clock::now();
    std::uint64_t s = 0;
    for (int k = 0; k < meas_iters; ++k) s += kernel(a);
    auto t1 = std::chrono::steady_clock::now();
    consume(s);

    std::chrono::duration<double> dt = t1 - t0;
    return dt.count();
}
```

Example: Data distribution control for branch studies

```
#include <cstddef>
#include <vector>

std::vector<int> make_clustered(std::size_t n) {
    std::vector<int> v(n);
    for (std::size_t i = 0; i < n; ++i) v[i] = (i < n/2) ? 1 : -1;
    return v;
}

std::vector<int> make_alternating(std::size_t n) {
    std::vector<int> v(n);
    for (std::size_t i = 0; i < n; ++i) v[i] = (i & 1) ? 1 : -1;
```

```
    return v;
}
```

Closing guidance

For branch prediction and speculation topics, the most trustworthy workflow is:

- learn ISA-visible rules from architecture manuals,
- learn predictor models from peer-reviewed architecture research,
- learn code-shaping effects from compiler documentation,
- validate on target hardware using disciplined profiling methodology.