Pruning the Unchangeable: Static Safety Proofs, Cardinality-Specialized Dispatch, and Self-Captured Data Relayout

— Bit-Exact Switch-Level Simulation of a Whole Console (a technical monograph)

Ying-Wei Li

Independent Researcher, Taiwan

baxermux@gmail.com

Abstract

Switch-level simulation executes a chip as its transistors — the only software model that reproduces pass-transistor buses, dynamic charge storage and precharge behaviour bit-for-bit, which is why it remains the reference tool for die-level preservation of 1980s NMOS parts. It is also notoriously slow: every event re-resolves a channel-connected component (CCC) through pointer-chasing graph walks, and a widespread engineering impression holds that managed languages are unsuited to such kernels. We present AprVisual, a C# whole-console simulator of the NES (2A03 + 2C02, ~14.7K nodes / ~26.8K transistors) whose resolution semantics are functionally MOSSIM II. On one Zen 2 core it sustains **~136K master-clock half-cycles/s** — an equivalent ~68 kHz silicon master clock, ~2.5× its direct C++ ancestor (MetalNES) and ~5.6× a literal chipsim.js port — while remaining **bit-exact** (full-state checksum verification). The speed comes from three families: (1) *provably-null event suppression* — load-time structural proofs (including a capacitance-dominance argument for charge-share tie-break immunity) that delete ~21% of all re-evaluations before they are enqueued; (2) *cardinality-specialized dispatch* — runtime proofs that the active CCC has size ≤ 2, resolved inline; (3) *self-captured data relayout* — an in-process inspector-executor pass that re-derives the node id space from the production cascade's first-touch order at every load. A nine-stage same-day ablation with hardware counters quantifies each step and shows the engine is latency-chain-bound, not miss-count-bound: the relayout cuts L1d misses by 67% while wall-clock moves single digits, and L1i misses stay at 0.2–0.3 MPKI — the architectural reason interpretation beats compilation here.

Keywords: switch-level simulation; event-driven simulation; channel-connected components; event suppression; data layout; bit-exactness; hardware preservation; NES

Reading note. This monograph is the long-form edition of the project's measurement record. Every performance number is a measurement made on the machine described in Chapter 8 (raw data reproduced in the appendices); negative results are reported with the same care as positive ones, and techniques that exist in prior art are labelled as such. Generative AI tools were used extensively in the engineering, the measurements, and the writing; in line with arXiv and IEEE

guidance on AI-assisted manuscripts, the full disclosure appears in the Acknowledgments. The human author is the sole author of this work and takes full responsibility for its content, scientific accuracy, and originality.

Contents

Chapter 1. Introduction

This monograph is about making the slowest credible model of a computer run fast enough to be useful, without ever letting it become a different model. The system under study is AprVisual: a bit-exact, event-driven, switch-level simulator of the complete Nintendo Entertainment System — the 2A03 CPU, the 2C02 PPU, and the board's TTL glue, roughly 14.7K electrical nodes and 26.8K NMOS transistors after lowering — written in C# [21]. On one core of a commodity desktop CPU it sustains approximately 136,000 master-clock half-cycles per second, an equivalent silicon master clock of about 68 kHz, which is roughly 2.5× the throughput of its direct C++ ancestor and 5.6× that of a literal port of the original JavaScript simulator, measured on the same machine with the same workload (all figures measured; Section 1.5). Every optimization that produced that number is gated by full-state checksums, and every optimization that failed that gate — or that succeeded at it and still lost time — is reported with the same care as the wins. The result is intended to be read on two levels: as an engineering account of a specific simulator, and as a measured map of where the single-core performance boundary of event-driven switch-level simulation actually lies.

1.1 Why Switch-Level, and Why Bit-Exactness Is Non-Negotiable

The chips in question have no register-transfer-level description. The 2A03 and 2C02 are commercial NMOS designs from the early 1980s whose only authoritative record is the silicon itself; the netlists we simulate were recovered from die photographs by the Visual 6502 project and by Quietust's Visual 2A03 / Visual 2C02 efforts — polygon-level segment definitions, transistor definitions, and a partial name map [15, 16]. What that recovery yields is not a logic design but a physical connectivity graph: which diffusion regions touch, which polysilicon gates cross which channels, which nodes carry depletion pull-ups. Any model imposed above that level is an interpretation; the switch-level model is, for practical purposes, the highest abstraction that is still a transcription.

Bryant's switch-level model [1] treats the chip as a set of charge-storing nodes connected by bidirectional switches, with signal strength and node capacitance resolving conflicts. The crucial property of this model — the reason it exists at all — is that *storage is emergent rather than declared*. A node that becomes disconnected from every driver does not assume an unknown value; it holds its charge. A group of mutually connected floating nodes resolves by charge sharing, which our engine implements as the documented tie-break: the purely-floating group takes the previous state of its largest-capacitance member (strict comparison, first-seen winning ties). No data structure in the simulator is labelled "register" or "RAM cell." Registers, latches, and memories arise because the netlist happens to contain configurations in which charge parked on a floating node survives until it is next read.

The 2C02's sprite memory (OAM) is the canonical example. It is not a RAM macro: it is on the order of 2,300 floating capacitors, written through pass transistors and retained purely as charge. Its capacity, its timing, and its well-known fragility are consequences of physics that the switch-

level model reproduces and that an RTL model would have to hand-write as behavioural annotations — at which point the model documents the modeller's beliefs, not the chip. The same applies to the design idioms that pervade both dies: bidirectional pass-transistor buses, precharge-and-conditional-discharge logic, bus contention resolved by drive strength, and address and data signals multiplexed onto the same physical nodes in the PPU. RTL's core assumptions — unidirectional signal flow, declared state elements, strength-free values — are false of this silicon in load-bearing ways. For the preservation use case, where the deliverable is *evidence about what the artifact does* rather than a convenient reimplementation, switch-level simulation is therefore not one option among several; it is the only software level at which the interesting behaviours emerge instead of being asserted [1, 15].

This is also why bit-exactness is non-negotiable rather than merely desirable. In a conventional performance project, an optimization that changes results slightly can be traded against its speedup. Here the floating tie-break *is* the chip's storage mechanism, and the within-wave Gauss-Seidel evaluation order is observable semantics: changing either does not degrade the simulation, it simulates a different machine. We verified this the hard way. An experiment that capped the settle loop's depth — abandoning only the deepest fraction of a percent of settle activity — diverged within fewer than 1,000 half-cycles, and aggressive caps derailed the CPU's program counter inside the first frame (measured; the negative-results catalogue in this monograph returns to it). There is no graceful degradation regime: the simulated machine either runs or it crashes in ways indistinguishable from a netlist bug. Consequently every optimization in this work is admitted under a single, mechanical definition of correctness:

> **Property 1.1 (Bit-exactness gate).** An engine change is *bit-exact* iff the full-state FNV-1a checksums over all node states are unchanged at 300k, 400k, and 1M half-cycles on the reference workload (goldens 0x794A43ABDF169ADA, 0x9174E19D961CB6E5, 0x6D4CCBCE2E9CD599), and a 10M-half-cycle sprite-heavy *Super Mario Bros.* gate passes.

Throughout this monograph, "fast" always means "fast subject to Property 1.1." Where we discuss techniques that would relax it — behavioural memories, cone abstraction, settle restructuring — we discuss them explicitly as departures from the model, and in this engine we reject them.

1.2 The Cost of Fidelity

The price of emergent storage is paid per event. The engine is event-driven in the lineage of Ulrich's selective-trace principle [4]: only nodes whose inputs changed are re-evaluated. Each re-evaluation of a node must, in general, discover the set of nodes currently connected to it through ON pass transistors — the channel-connected component (CCC) of the moment, which we call the group walk — then resolve the group: OR the members' drive flags, classify through a 256-entry priority lookup table (grounded beats powered beats externally driven beats pulled-up beats held), or, if the group is purely floating, apply the capacitance tie-break. State changes are written back and the gate and channel neighbours of changed nodes are enqueued into the next settle wave; waves drain until the netlist is quiescent, and only then does the clock toggle again.

The structure of this workload is dictated by the netlist, and the netlist is unhelpful. Measured on the current engine, a 100k-half-cycle run performs about 42.85 million node re-evaluations — roughly 418 per half-cycle — of which 39.5% require a group walk at all. The walks themselves are tiny: 77.1% of groups have exactly two members, the mean BFS depth is 1.13, the 99th percentile is 3, and the maximum ever observed is 14. The mean conducting group is 1.13–1.4 nodes. Settle waves average 12.06 per half-cycle with a maximum of 45. In other words, the simulator's "graph algorithm" is in reality tens of millions of dependent-pointer-chase sequences of two or three loads each, interleaved with queue maintenance, and 80.1% of all re-evaluations end with no state change at all — a structural residue dominated by pull-up nodes (42.0%) and supply-adjacent nodes (38.1%) that are woken by neighbours and resolve to what they already were.

This shape sets a floor that conventional acceleration cannot reach under. Each step of a group walk loads an address that depends on the previous load; the engine's critical path is a chain of dependent loads against a working set small enough to live in the L1 data cache, which means the bound is load-to-use *latency*, not bandwidth, not arithmetic throughput, and — as the hardware counters in our evaluation show directly — not even miss counts. There is no large regular computation to batch, vectorize, or offload, and we did not take that on faith: in this project's own measured history, an IR interpreter was 2.5% slower than the event-driven engine, ahead-of-time code generation was 3–6× slower, a single-instance GPU port was about 10.7× slower, a bit-parallel BFS formulation was about 156× slower, and splitting the two chips across two threads was about 15× slower. Every one of these failures has the same root cause: the event-driven engine already operates at the netlist's natural granularity, and that granularity is a handful of nanoseconds of serial pointer-chasing.

One further presumption deserves to be named: a widespread impression in performance-engineering practice — one for which we could find no formal source — that managed languages are unsuited to such pointer-chasing kernels. This book tests that impression empirically. The engine's hot state is unmanaged structure-of-arrays storage — one-byte node states, 16-byte node records with a 12-byte inline payload, 16-bit transistor lists, double-buffered wave lists with a deduplication hash — and its inner loops are bounds-check-free. On this substrate, the C# engine outperforms the native C++ implementations of its own family on the same machine and workload (Section 1.5); throughout this monograph we show, by measurement, that on this latency-bound kernel the language runtime itself is not the deciding factor.

1.3 Problem Statement and Approach

The problem this monograph addresses can be stated narrowly: *given a fixed switch-level netlist and fixed observable semantics (resolution order, priority lattice, and charge tie-break), maximize sustained half-cycles per second on a single commodity core under Property 1.1, and characterize — with measurements, not arguments — where the remaining boundary lies.* The narrow statement matters. We do not change the model, we do not approximate, we do not parallelize (we measured why not), and we do not accept speed claims that are not checksum-gated. Within that frame, the engine's measured profile in Section 1.2 admits exactly three lines of attack, and the work reported here is organized around them.

Family 1: prove null, and suppress. If 80.1% of re-evaluations conclude "no change," the cheapest re-evaluation is the one never enqueued. The difficulty is that "this event cannot change anything" is a semantic claim about a charge-sharing model, and naive versions of it are wrong in ways that corrupt storage: our first same-state prune produced a black screen, traced to floating dynamic nodes whose hold-previous tie-break the prune had silently bypassed. The family that survived (P-1 through P-4) therefore couples each suppression rule to a load-time *structural safety proof*: a taint classification identifies the node populations for which the rule's soundness argument holds (pull-up-backed nodes; nodes outside ForceCompute channel components), and a capacitance-dominance argument un-taints nodes that provably can never win a floating tie-break because their capacitance is strictly below that of every channel neighbour. Together these prunes delete about 21% of all node re-evaluations before they enter the queue, bit-exactly.

Family 2: prove small, and specialize. The group-size distribution says that almost every walk discovers one or two nodes. The dispatch layer therefore spends a few cycles attempting a cheap cardinality proof before committing to the general walk: a node all of whose pass gates are currently OFF is its own group and resolves in O(1) (the R-1 dynamic singleton); a node with exactly one ON gate whose neighbour's ON channels all lead back to the seed forms a two-node group resolved inline (the B1 pair path), replicating the general path's order-sensitive details — member-hash clearing, both supply scans, the tie-break form, the write order — byte for byte, with every bail-out taken before any mutation. A static class of 3,929 nodes (26.7%) has no pass channels at all and never walks. These are specializations of known ideas, as Section 1.4 is careful to say; their value here is the measured payoff of treating cardinality as something to *prove per event* at dispatch time.

Family 3: self-derive the layout. What remains after suppression and specialization is the dependent-load chain itself, and the only lever left is where the data lives and what the hot loop must load to decide anything. The engine rebuilds its own node id space at every load, in three passes: it classifies nodes into prune classes and renumbers class-major, so that the per-node safety lookups of Family 1 collapse into register comparisons against three id boundaries; it then warms up, captures 32,768 half-cycles of the production cascade's true first-pop order through a cold instrumented copy of the settle loop, and rebuilds again with that order as the locality key. The classification is re-derived and verified against ground truth at every reset, with a safe-degenerate fallback on mismatch — the deoptimization-guard pattern [13]. The lineage is the inspector–executor tradition of irregular-computation reordering [7, 8, 9] and trace-driven data layout [10, 11, 12, 20]; the configuration — an event-driven simulator capturing its own cascade in-process, with no profile files and no workload assumptions — and the empirical finding about *why* it works (order, not cache-line density) are where this work adds something.

Beneath all three families sits a fourth element we treat as a first-class artifact rather than scaffolding: the correctness infrastructure. Golden checksums at three horizons plus a long sprite-heavy gate; per-node state-dump differencing for localizing divergences (it is how the black-screen prune was diagnosed); and a measurement discipline — same-day round-robin interleaving, medians, paired ordering for sub-1% effects — without which roughly half of this

monograph's percentages would be thermal noise. In a model where approximation fails catastrophically rather than gracefully, the verification machinery is not overhead on the contribution; it is the precondition for any of the contributions being claims at all.

1.4 Contributions

We state the contributions with explicit honesty labels, because the literature adjacent to this work is fifty years deep and several of our mechanisms are best described as variants of known ideas. The resolution semantics themselves are not ours: functionally, the engine implements Bryant's MOSSIM II model [1] without the X state, the group walk is CCC evaluation, and the floating tie-break is his node-size lattice. The dispatch and suppression machinery sits on top of that model; the nearest neighbours are credited per item.

(1) A prune family with static safety proofs (P-1 through P-4). We suppress an enqueue when the resulting re-evaluation is provably a no-op, using only static structure plus the two endpoints' live states. P-1, the same-state turn-on prune, is honestly a variant: endpoint-equality bypasses are in the spirit of IRSIM's equipotential handling [3] and of selective trace generally [4]; what we add is the structural safety taint taxonomy — the classification of exactly when an equal-state merge is still unsafe (no-pull-up nodes; ForceCompute channel components), which is where the naive version corrupts storage. P-2, the turn-off isolation prune for degree-1 driverless leaves, promotes what is a runtime outcome in IRSIM-class simulators into a load-time proof applied *before* enqueue, saving the entire enqueue–pop–resolve chain; we assess it as plausibly original, though minor. P-3 and P-4 are the strongest claim of the family: a capacitance-dominance argument — a node whose capacitance is strictly below that of all its channel neighbours can never be the maximum-capacitance member of any group it joins, hence can never determine a floating tie-break, hence equal-state merges through it are provably null. The nearest prior practice is IRSIM's manual node-size coarsening for resolution efficiency [3]; ours is an automatically derived inequality spent on event suppression, exact rather than approximate. Measured together, the family deletes ~21% of re-evaluations and contributes the largest single step of the ablation ladder.

(2) Cardinality-specialized dispatch (R-1 and B1). Honest label: engineering variants, not new theory. Dynamic sizing of the active subnetwork is IRSIM's territory [3]; static CCC extraction and compilation is COSMOS's [2]. Our versions are an early-out execution filter over that idea: prove at dispatch time that the group is exactly {*n*} (all pass gates OFF) or exactly {seed, neighbour} (one ON gate, all return channels closed), and resolve inline while replicating the general walk's observable semantics exactly. The measured value is large: R-1 was worth +18.7%, and B1 contributes +7.0% on the ablation ladder (S6→S7, same-day interleaved; about +8.9% after separating the attribution from the adjacent housekeeping commit).

(3) Self-captured first-touch relayout. The three-pass load of Section 1.3: class-major renumbering that turns prune metadata into id-range register compares, plus an in-process capture of the production cascade's first-pop order as the locality key, verified per reset with a deoptimization-style fallback [13]. Each component has lineage — inspector–executor reordering

[7, 8, 9], trace-driven and cache-conscious layout [10, 11, 12], profile-guided layout in compiler practice [20] — and we credit it; we assess the synthesis (an event-driven simulator capturing its own event cascade in-process, every load, zero configuration, immune to profile staleness by construction) as having no precedent we or our prior-art consultations could find. The accompanying empirical finding is, we believe, new and is corroborated by hardware counters: the key's value lies in the *pruned cascade's order*, not in cache-line density. A capture taken with the prunes disabled achieves equal line density and gains nothing (±0.0%); the same capture with prunes on gains +6.17%; and the stage that cuts L1d misses by half buys only single-digit wall-clock, while the stage that buys +5% barely moves the miss counters.

(4) A zero-configuration safety methodology (presented as implementation, not as a theoretical claim). Every safety classification above — which nodes are storage, which are bus fabric, which merges are tie-break-immune — is derived from physics, never from names: pull-up flags, union-find over pass channels, capacitance comparisons, driven-pin exclusion. Storage excludes itself from unsafe prunes because large capacitance is what storage physically *is*. Automatic transistor-to-gate-level logic extraction and subcircuit recognition via subgraph isomorphism are established EDA practice [5, 6]; in contrast to that library- and structural-pattern-driven family of automated identification, our physics-property-based identification — used as the soundness substrate for event suppression, applied to a die-derived netlist with no human annotation — is positioned as methodology supporting contributions (1)–(3).

(5) A falsifiable performance boundary map for single-core switch-level simulation. The negative results are contributions, stated with the same precision as the wins: every abstraction route (IR, AOT/codegen, cone compression) and every parallelization route (GPU, bit-parallel, per-chip threading) measured slower, with mechanisms identified; a ~7% maintenance floor on maintained-runtime-fact optimizations, mapped to centimetre precision by the P-5 experiments, including the semantic distinction — live state versus resolution-time snapshot — that explains why the profitable variant cannot be made sound cheaply; the catastrophic (not graceful) failure of under-settling; and the structural facts beneath all of it: 94% of the dependency graph is a single bidirectional strongly-connected component, and the no-change residue has no static subset (confirmed three independent ways). A nine-stage same-day ablation with hardware performance counters quantifies every adopted step. The map is falsifiable in the plain sense: each boundary is a measurement that future hardware or a better idea can overturn — as indeed several of our own earlier published "ceilings" were overturned by later measurements within this very project.

1.5 Headline Results

All numbers below were measured on one machine (AMD Ryzen 7 3700X, Zen 2, Windows 11; .NET 10/11 per stage), workload `full_palette.nes` (a community open-source NES test ROM displaying the full NES palette as a static screen, from the public nes-test-roms collection), one benchmark process at a time, stages interleaved round-robin within the same session so that thermal drift hits every stage equally; medians are reported and every run passed the checksum gate of Property 1.1. The central result is the ablation ladder: eight historical engine stages rebuilt

from their actual commits and benchmarked side by side on the same day. From the baseline fork at 67,955 half-cycles per second, the adopted techniques compound to 132,243 median (135,828 best) — a cumulative **+94.6%** — with the largest steps being the R-1 dynamic singleton (+18.7%), the P-1 same-state prune (+26.4%), the P-2/P-3/P-4 prune completion (+7.2%), the class-major range-prune relayout (+4.2%), the self-captured first-touch key (+5.0%), and the B1 pair path (+7.0%).

> **Headline.** ~136K half-cycles/s on one Zen 2 core — an equivalent ~68 kHz silicon master clock — bit-exact under Property 1.1, with the entire +94.6% same-day ablation checksum-gated at every stage. The remaining gap to real time (42,954,552 hc/s) is approximately 316×.

Table 1.1. The switch-level NES/6502 family on one machine, one workload, one unit (measured 2026-06-08; AprVisual updated to its current figure). The original projects ship no built-in hc/s measurement; we experimentally instrumented their source to obtain same-unit figures (details in Section 2.4). perfect6502 simulates the 6502 alone and is listed for context, not ranked.

Simulator	Scope	Language	Throughput (hc/s)	Relative
VisualNes [19]	whole NES	C++ (literal chipsim.js port)	~24K	0.18×
MetalNES [17]	whole NES	C++ (optimized; our ancestor)	~54K	0.40×
AprVisual [21]	whole NES	C#	**~136K**	**1.0×**
perfect6502 [18]	6502 only	C	~29K (different unit)	—

The hardware-counter story can be told in one paragraph, and it inverts the framing a cache-conscious-optimization paper would be expected to adopt. Across the ladder, retired instructions per million half-cycles fall by a third (155.6 billion to 104.4 billion) — the prunes delete work, not just queue entries — while IPC holds between 2.1 and 2.4. The relayout stage cuts L1d misses from 27.3 to 13.2 MPKI, yet wall-clock moves only +4.2%; the self-capture stage gains +5.0% while barely moving the miss counters at all; and L1i misses sit at 0.17–0.30 MPKI at every stage, confirming that the fully-inlined interpreter loop fits in the instruction cache — the counter-level explanation for why every compiled variant lost (compilation explodes code footprint exactly where the interpreter pays nothing). The engine is dependent-load-*latency*-bound on an L1-resident working set, not miss-*count*-bound, and the decisive layout win is the order of the pruned cascade, not line packing.

Finally, the honest distance: real time on this hardware would require 42.95 million half-cycles per second, and the current engine stands approximately 316× away. Nothing in this monograph claims that gap is closable on one core; Chapter 9's boundary map argues, with measurements, that it is not — while noting that every previous ceiling this project published fell to a measurement rather than to an argument, which is the appropriately uncomfortable note on which to make any such claim.

1.6 Organization of This Monograph

Chapter 2 establishes the background: the switch-level model and its resolution semantics, the die-derived netlists and their data formats, and the lineage from chipsim.js through MetalNES to this engine [15, 16, 17]. Chapter 3 surveys related work in four strands — the switch-level school [1, 2, 3], event suppression and selective trace [4], storage extraction [5, 6], and runtime data reorganization [7–12, 20] — with an explicit verdict on what our mechanisms are variants of. Chapter 4 describes the engine itself: the unmanaged structure-of-arrays layout, the settle loop, the dispatch classes, and the correctness infrastructure. Chapters 5 through 7 develop the three families in depth — the prune family and its safety proofs (Chapter 5), cardinality-specialized dispatch (Chapter 6), and the self-derived layout (Chapter 7) — each with mechanism, soundness argument, and measured effect. Chapter 8 presents the evaluation in full: the ablation ladder, the hardware-counter profiles, the order-versus-density study, and the measurement-hygiene methodology. Chapter 9 assembles the boundary map from the negative results. Chapter 10 concludes with what we believe transfers beyond this artifact, and what is honestly specific to it. The engine, the benchmark package, and the public leaderboard are available at the project repository [21].

Chapter 2. Background: The Switch-Level Model, the NES, and the Simulator Family

This chapter assembles the four pieces of context that the rest of the monograph stands on. First, the *semantic model*: every simulator discussed in this book — including ours — executes a discrete switch-level model whose essential content was formalized by Bryant in 1984 [1], and we state plainly at the outset which parts of that model we inherit unchanged. Second, the *subject*: the two NMOS chips of the Nintendo Entertainment System, whose circuit idioms (ratioed logic, pass-transistor networks, precharged buses, dynamic charge storage) are exactly the behaviors that force simulation down to the switch level. Third, the *algorithm*: the event-driven settle loop that every generation of the simulator family shares, which we walk through on a small worked example so that later chapters can refer to its steps by name. Fourth, the *family* itself: the lineage from the Visual 6502 project's JavaScript simulator through MetalNES to AprVisual, including what each generation added and where the present work sits. A final section fixes terminology and units — in particular the master-clock half-cycle, the unit in which every throughput number in this book is expressed.

2.1 The Bryant Switch-Level Model

2.1.1 Networks of bidirectional switches

Gate-level abstraction treats a digital circuit as a directed graph of Boolean operators: each gate has inputs, an output, and a function from the former to the latter. For the NMOS chips of the late 1970s this abstraction is not merely lossy but wrong in kind. A MOS transistor is not a logic gate; it is a *switch* whose channel, when conducting, connects two nodes symmetrically. Designers of the era exploited that symmetry systematically: a single pass transistor implements a multiplexer arm, a register-file port, or a bus connection, and current — therefore information — flows through it in whichever direction the surrounding circuit demands at that instant. No static assignment of inputs and outputs exists for such structures, so no gate-level netlist can be extracted from them without design knowledge that a die photograph does not carry.

The switch-level model, introduced by Bryant [1], is the weakest abstraction that remains faithful to this reality while staying discrete. A circuit is a set of *nodes* (the electrical nets) connected by *transistors* modeled as voltage-controlled switches: the gate node's logical state determines whether the channel between the two channel terminals conducts. Node states are drawn from the set {0, 1, X}, where X represents an unknown or invalid level. Computation proceeds not gate by gate but *component by component*: at any instant, the conducting channels partition the circuit's non-rail nodes into **channel-connected components** (CCCs) — maximal sets of nodes mutually reachable through transistors that are currently ON — and the steady-state value of every node in a component is determined jointly by everything connected to that component. Because the partition depends on gate states, which themselves change as the simulation runs, the

components must in general be discovered dynamically. This dynamic discovery — in our engine, the per-event breadth-first *group walk* — is the algorithmic heart of every simulator in this book, and it is what the literature calls CCC evaluation.

Bryant's model also fixes *how* a component's joint value is resolved, and it is here that the model earns its keep. Two physical phenomena must be captured discretely. The first is *ratioed drive*: when a component is connected both to a strong path (say, to ground through an ON enhancement transistor) and to a weak one (a depletion pull-up load), the strong path wins — this is precisely how an NMOS inverter produces a logic 0 while its pull-up is still conducting. The second is *charge storage*: a component connected to no driving path at all does not assume an arbitrary value; it retains charge, and its nodes hold their previous state, with larger capacitances dominating smaller ones when previously-disagreeing nodes are merged.

2.1.2 Strengths, sizes, and charge sharing

To make both phenomena decidable without analog computation, Bryant's formalization assigns each transistor a discrete *conductance strength* and each node a discrete *capacitance size*, each drawn from a small ordered set [1]. The steady-state response of a channel-connected component is then defined by an ordering on signals: a path to a supply rail through stronger transistors overrides a path through weaker ones; any driven path overrides stored charge; and among stored charges, the largest node size dominates. Conflicts between equally strong opposing signals, and charge whose history is unknown, yield X. The model's central theorem is that this discrete resolution agrees with the qualitative behavior of the underlying RC network — which is exactly what licenses replacing analog simulation with table lookups and graph traversals.

Charge sharing deserves emphasis, because it is the part of the model that casual ports tend to break and because it carries real information in our subject chips. When a conducting path forms between a floating node and a larger driven or charged structure, the smaller node's charge is overwhelmed: discretely, the merged component takes the value associated with the largest-capacitance participant. Conversely, when a pass transistor opens and strands a node, that node keeps its last value indefinitely — the model has no decay. Dynamic MOS design depends on both halves of this behavior: a dynamic latch *is* a deliberately stranded node, and a precharged bus *is* a deliberate charge-sharing fight that the precharge is sized to win or lose by design. A simulator that resolves floating components incorrectly does not produce slightly wrong waveforms; it produces a machine whose registers forget their contents.

Bryant's model spawned a family of academic simulators whose design axes anticipate much of this monograph. MOSSIM II [1] is the reference event-driven implementation of the model itself. COSMOS [2] showed that the per-component resolution can be *compiled*: channel-connected components are extracted statically, each is translated once into Boolean evaluation code, and simulation becomes execution of that code — trading model generality for throughput. IRSIM [3] took the incremental path instead: an event-driven simulator with linear-RC timing in which only the *active* subnetwork around each event is rebuilt and re-resolved, with explicit rules for charge preservation. The event-driven discipline these systems share descends from Ulrich's principle of exclusive simulation of activity [4]: in a large network where almost nothing changes per time

step, work must be proportional to what changed, not to what exists. All three systems reappear in Chapter 3 as the prior art against which our techniques are positioned; here they serve to fix the vocabulary.

2.1.3 Our resolution semantics: MOSSIM II minus the X state

We now state the resolution semantics that AprVisual — and, up to implementation detail, every simulator in the visual6502 lineage — actually executes, and we are deliberately blunt about its provenance. **The model is functionally Bryant's, with the X state removed.** Node states are binary. Each node carries a small set of flags: `Gnd` and `Pwr` mark the two supply rails; `SetHigh` / `SetLow` mark external pin drive; `PullUp` marks a depletion pull-up load; `ForceCompute` marks certain board-level bus nodes with a special contention rule. Resolution of a channel-connected component is a two-stage function: the flags of all members are OR-ed together, and the resulting byte indexes a 256-entry priority table.

> **Property 2.1 (resolution function).** Let F be the bitwise OR of the flags of all members of a channel-connected component (with supply rails contributing their flag but never joining the walk). The component's new value is determined by the first matching rule: (i) if F contains `ForceCompute` together with both `Gnd` and `Pwr`, both rail flags are cancelled before proceeding; (ii) `Gnd` → 0; (iii) `Pwr` → 1; (iv) `SetHigh` → 1; (v) `SetLow` → 0; (vi) `PullUp` → 1; (vii) otherwise the component is purely floating, and every member takes the previous state of the member with the strictly largest capacitance proxy, the first member encountered winning ties.

The correspondence to Bryant's lattices is direct but coarsened. The strength order “rail > external drive > pull-up > stored charge” is a fixed four-level instance of his conductance-strength hierarchy — sufficient because the NMOS processes of these chips used exactly one flavor of strong device and one flavor of weak load. The floating tie-break is his node-size comparison, with a node's static connection count standing in as the capacitance proxy: the netlists carry no extracted capacitance values, and the count of attached channel terminals is the topological surrogate the family inherited from its JavaScript ancestor [15]. In our engine the entire priority ladder is precomputed into a 256-entry lookup table indexed by the OR-ed flag byte (a MetalNES contribution, Section 2.4.3), so stage one of resolution is a graph walk and stage two is a single load.

```
// The priority ladder, as actually executed (precomputed into FlagsToState[256]).
if (F has ForceCompute and Gnd and Pwr)  F := F - {Gnd, Pwr};   // bus contention cancels
if (F has Gnd)      return 0;        // strong path to ground wins
if (F has Pwr)      return 1;
if (F has SetHigh)  return 1;        // external pin drive
if (F has SetLow)   return 0;
if (F has PullUp)   return 1;        // depletion load: weakest driven signal
// purely floating: charge sharing — largest-capacitance member holds
return prevState(member with strictly greatest connection count; first seen wins ties);
```

What is lost by deleting X? Three things, each compensated structurally rather than semantically. Unknown initial charge cannot be represented; the family substitutes a deterministic power-on procedure that enqueues every node once and settles the whole network to a reproducible fixed point, after which all charge has a defined history. Drive conflicts cannot yield “invalid”; rule (ii) resolves them deterministically in favor of ground, which matches NMOS electrical reality for rail fights and is patched for the one place it is wrong — board-level bus nodes where opposing always-on paths meet — by the `ForceCompute` cancellation of rule (i). Oscillation cannot be flagged by X-propagation; it is bounded operationally by the settle-loop structure of Section 2.3.1, and we note that on these netlists settling in fact converges within at most 45 waves (measured; Section 2.3.1).

We insist on the formulation “we *are* functionally MOSSIM II minus the X state” rather than some softer phrase, for two reasons that shape the entire monograph. The first is honesty of attribution: nothing in our resolution semantics is new, and a reader who encounters our priority table or our floating tie-break should be told immediately that they are reading Bryant's model as filtered through chipsim.js [15, 1]. Every contribution claimed in later chapters is a contribution of *scheduling, suppression, proof, and memory layout over an unchanged forty-year-old semantic model* — never of the model itself. The second reason is methodological: our central correctness contract, bit-exactness (Section 2.5), is only meaningful because the semantics is fixed, known, and shared across implementations. A simulator that “improves” the model has no golden reference to be bit-exact against; a simulator that freezes the model can treat any byte-level divergence, anywhere in 14.7K nodes over ten million half-cycles, as a bug by definition. The freeze is the foundation of the method.

2.2 The NES at the Transistor Level

2.2.1 The two custom chips

The Nintendo Entertainment System (the 1983 Famicom in its 1985 export form) is built around two custom NMOS chips. The **2A03** is the CPU device: a licensed MOS 6502 core with the decimal-correction circuitry disabled, sharing its die with the audio processing unit (two pulse channels, triangle, noise, and a sample channel), a sprite-DMA engine, and the controller I/O ports. The **2C02** is the picture processing unit (PPU): it fetches background and sprite pattern data, evaluates up to 64 sprites against the current scanline from its on-die object attribute memory (OAM), composes a 256×240 picture, and generates the video signal timing directly. The two chips run from a single master clock and communicate over a multiplexed bus fabric; the CPU divides the master clock by 12 and the PPU by 4, so the machine's natural global time base is finer than either chip's instruction or pixel granularity (Section 2.5).

Both dies have been imaged, and their full transistor-level netlists extracted from the photographs, by Quietust [16], following the methodology established for the original 6502 by the Visual 6502 project [15]: successive die layers are photographed, polygons are traced for diffusion, polysilicon, and metal, and the polygon stack is compiled into nodes and transistors.

The result is not a schematic drawn from documentation but the silicon itself in graph form, including every design idiosyncrasy, every undocumented behavior, and — critically for this book — every structure whose function exists only at the charge level.

It is worth pausing on what such a netlist is *not*. It carries no labels of intent: no “this is a register”, no “this bus is precharged”, no clock-domain annotations (the node-name files do name some internal signals, but names are documentation, not structure, and our methodology refuses to depend on them). It carries no extracted electrical parameters: no capacitances, no transistor geometries usable as strengths. It is a bipartite connectivity graph plus a pull-up marking — and everything a simulator does must be derivable from that alone. This austerity is what makes the later chapters' “identify structure by its physics, not its name” discipline both necessary and possible.

2.2.2 NMOS circuit idioms

Four idioms of late-1970s NMOS design dominate these dies, and each maps onto a specific feature of the model of Section 2.1.

Ratioed enhancement/depletion logic. The basic NMOS gate is an always-conducting depletion-mode pull-up load over an enhancement pull-down network. A logic 0 is produced not by disconnecting the high side but by *overpowering* it: the pull-down path is sized to win the fight. In the model this is rule ordering — `Gnd` outranks `PullUp` in Property 2.1 — and in the netlist data it appears as a per-node pull-up flag rather than as explicit load transistors: we measured that the seventh (“weak”) column of the transistor records is false on every row of both chips' netlists, the extraction having folded all depletion loads into per-segment pull-up markings instead.

Pass-transistor networks. Multiplexers, register read/write ports, and the internal bus switches are built from bare pass transistors. These are the bidirectional structures of Section 2.1.1, and they are the reason channel-connected components must be discovered per event: which nodes form a component depends on which gates are high *now*. The measured shape of this dynamism on the running machine is striking and recurs throughout the book: the average conducting component contains only 1.13–1.4 nodes, and of the component walks that reach the general resolver in the optimized engine, 77.1% have exactly two members. The chips are a sea of switches, but at any instant almost all of them are open.

Precharged dynamic buses. Long internal buses are precharged high during one clock phase and conditionally discharged during the other, replacing slow ratioed pulls with a timed charge-sharing discipline. The simulator does not model this specially; it *emerges* from rules (vi)–(vii) and the clock. Its cost, however, is structural: precharge traffic recurs every phase whether or not the bus value changes, and we measured that roughly a quarter of all node re-evaluations in the running engine are memory- and bus-fabric activity of this clock-driven kind. Several optimization chapters return to this number.

Dynamic charge storage. Registers, pipeline latches, and above all the PPU's sprite memory store bits as charge on deliberately stranded nodes, refreshed by the two-phase clock. The 2C02's OAM is the extreme case: it is not a RAM macro but roughly 2,300 floating storage capacitors. In the model, every one of these bits lives in rule (vii) — the purely-floating largest-capacitance tie-break *is* the chip's storage mechanism. This single fact drives much of the book: it is why the tie-break order is observable semantics (Section 2.3.5), why approximations that perturb floating resolution fail catastrophically rather than gracefully, and why the capacitance-dominance safety proofs of the prune chapters have to exist at all. We note that recognizing storage structures in flat transistor netlists is itself a classical EDA problem [5, 6]; what is unusual here is that the simulator must get their *behavior* right implicitly, on every event, forever.

2.2.3 The netlist interchange format

The netlists are distributed in the Visual 6502 project's three-file JavaScript format [15, 16], summarized in Table 2.1. The `segdefs` file lists silicon polygons: each record names the node the polygon belongs to, a pull marking (`'+'` for pull-up, `'-'` for pull-down), a layer index, and the polygon coordinates. The `transdefs` file lists transistors as (gate, channel-terminal-1, channel-terminal-2) triples with bounding-box and geometry fields; the 2A03/2C02 variants append the seventh weak/depletion boolean noted above. The `nodenames` file maps human-readable names to node numbers — supply rails, clocks, external pins, and (for these two chips) many internal signals, including the CPU's architectural registers bit by bit.

Table 2.1. The Visual 6502 / Quietust netlist interchange format [15, 16].

File	Record	Contents	Notes
`segdefs`	polygon	node id, pull (`'+'` / `'-'`), layer, vertex list	pull markings carry the depletion loads; we retain both `'+'` and `'-'`
`transdefs`	transistor	name, gate, c1, c2, bbox, geometry [, weak]	c1/c2 interchangeable (bidirectional channel); weak column measured false on every row of both chips
`nodenames`	name → id	e.g. `vcc` , `clk0` , `res` , `a0..a7` , `pcl0..pcl7`	names may carry `/` `#` `~` `_` prefixes; the 2C02 multiplexes its `ab` / `db` buses onto shared nodes

Two properties of this format matter downstream. First, channel terminals are unordered: c1 and c2 are interchangeable because the device is symmetric, and any engine data structure that imposes an orientation must do so as a normalization step, not as a semantic claim. Second, the format's connectivity is per chip; building a console requires a composition layer, which the family did not acquire until MetalNES.

2.2.4 Board-level composition and behavioral memory

A console is more than its two custom chips. The NES-001 board adds 2 KB of CPU work RAM, 2 KB of video RAM, the cartridge (in the mapper-less NROM configuration we simulate: program ROM and character ROM), and TTL glue logic such as the address latch between CPU and

cartridge. AprVisual inherits MetalNES's composition machinery [17]: a system is described by module definition files, each declaring its pins, sub-module instances, inter-module connections, per-node pull-ups, the `ForceCompute` node set, and any behavioral memory attached to the module. During loading, each module instance receives a private node-id range, and a cross-module connection is realized as an always-on transistor joining the two nodes.

Those always-on connection transistors are semantic no-ops — two nodes joined by a permanently conducting channel are one electrical net — and our loader's *lowering* pass merges them away, fusing 441 nodes and 530 transistors. The composed and lowered whole-console graph that every number in this book refers to contains approximately 14.7K nodes and 26.8K transistors. Its structural census, computed at load time, is: 3,929 nodes (26.7%) have no pass-transistor channels at all and can never join a multi-node component; 10,784 nodes (73.2%) have channels and are therefore dynamic-component candidates; and a residue of roughly 16 nodes carries behavioral callbacks. These three populations are the dispatch classes cls1, cls2, and cls0 of the optimization chapters (Section 2.5).

Memory is the one place the family deliberately departs from transistor fidelity. Simulating 4 KB of board SRAM plus the ROMs as transistors would multiply the netlist size for no archival benefit — the board memories, unlike the on-die OAM, are ordinary commodity parts. MetalNES introduced, and we retain, *behavioral memory handlers*: callbacks registered on the relevant bus nodes (mechanically, a “fake transistor” that fires when its node resolves) implement RAM and ROM array semantics directly [17]. The boundary is principled: everything on the two custom dies is simulated at the switch level, including every dynamic storage node; everything that was a socketed commodity chip on the board is behavioral. The `ForceCompute` contention rule of Property 2.1(i) exists for the same boundary — certain bus nodes where this composition makes both rails reachable need the cancellation to resolve as the real open-collector fabric does.

2.3 The Algorithm Every Generation Shares

2.3.1 Events, waves, and the settle loop

All simulators in this family are event-driven in Ulrich's sense [4]: work is proportional to state change. The unit of external stimulus is a toggle of the board clock node — one master-clock half-cycle, *hc* for short — after which the engine must propagate consequences until the network is quiescent. The propagation structure is a double-buffered worklist, and its drain is the inner loop of everything this book measures.

```
// One half-cycle: toggle the clock, then settle to a fixed point.
toggle(clk); enqueue(clk);
while (nextList is not empty) {            // each iteration = one settle WAVE
    swap(currentList, nextList);
    for nn in currentList {                // pop order = enqueue order (FIFO)
        if (dedupHash[nn] cleared) continue;   // absorbed into an earlier group this wave
        group  = groupWalk(nn);            // BFS over ON pass transistors (Sec. 2.3.2)
        v      = resolve(group);           // Property 2.1
        for m in group: setState(m, v);    // in place; may append to nextList
    }
}
// setState(m, v): if states[m] == v return;  states[m] = v;
//   for each transistor t gated by m:
//     if v == 1  enqueue(t.c1);              // turn-on: walk reaches c2 through the channel
//     else       enqueue(t.c1); enqueue(t.c2);  // turn-off: the sides may now separate
```

Several details of this loop are load-bearing. The two lists are *buffered by wave*: nodes enqueued while draining the current list land on the next list, so propagation advances in breadth-first generations — we call each drain a **settle wave**. A per-node hash deduplicates enqueues within a wave, and the group walk clears the hash entry of every node it absorbs, cancelling that node's pending pop in the current wave: a node that has just been resolved as part of someone else's component must not be redundantly re-evaluated an instant later. Gate transitions enqueue asymmetrically: a gate going high makes its channel conduct, so enqueueing one endpoint suffices (the walk will cross the channel and find the other); a gate going low may split a component in two, so both endpoints must re-resolve independently.

On the real workload the loop's measured shape is as follows (all figures per 100k hc on the composed console running a standard test ROM): approximately 418 node evaluations (“pops”) per half-cycle; a mean of 12.06 settle waves per half-cycle with an observed maximum of 45; and component walks whose breadth-first depth averages 1.13 levels with a 99th percentile of 3 and a maximum of 14. The picture these numbers paint — thousands of tiny, shallow, latency-bound graph probes per half-cycle, with no large regular structure anywhere — is the terrain every optimization in this book had to fight on, and the quantitative refutation of every batching, compiling, and parallelizing strategy we later report as a negative result. We also note, ahead of the methodology chapters, that the settle depth is not slack: experiments that capped the wave count diverged from the reference within fewer than a thousand half-cycles. The deep tail of the settle distribution is the netlist's real critical path, not an artifact.

2.3.2 The group walk and resolution

The **group walk** is the dynamic CCC discovery of Section 2.1.1, specialized to the event at hand. Starting from the popped node as seed, the walk maintains a growing member list (which doubles as the BFS queue) and an OR-accumulator of member flags. For each member, the engine scans that node's channel-terminal records: for every transistor whose channel touches the member, if the transistor's gate node is currently high, the far terminal joins the component — unless it is a supply rail, in which case the walk does not cross but ORs the rail's flag into the accumulator.

Membership is deduplicated with a per-node marker, so the walk runs in time linear in the component's incident transistor records. When the queue drains, the accumulated flag byte goes through the 256-entry table of Property 2.1; if it is entirely empty, the floating branch scans the member list for the largest connection count and adopts that member's previous state.

The resolved value is then written back to *every* member through the state-change filter of the settle loop: members already at the value are untouched (and crucially generate no further events); members that change trigger the gate-side enqueues described above, seeding the next wave. The walk-then-broadcast structure means a single pop can settle several nodes at once, and conversely that a node's value can be rewritten by a walk seeded elsewhere — both facts matter when later chapters reason about which enqueues are provably redundant.

It bears repeating how small these components are in practice. The mean conducting component on this netlist holds 1.13–1.4 nodes; among walks reaching the general resolver, 77.1% hold exactly two members, 16% three, and only 5.7% four or more. The model permits arbitrarily large components — and the power-on settle does briefly produce large ones — but the steady-state machine is overwhelmingly a machine of singletons and pairs. The optimization chapters' dispatch-class architecture (cls1/cls2 fast paths, the pair path) is nothing but this distribution taken seriously.

2.3.3 The floating tie-break is the storage mechanism

Rule (vii) of Property 2.1 deserves its own subsection because it is the single most consequential line in the engine. When a component has no flag at all — no rail, no external drive, no pull-up — the model declares it charge-isolated and assigns it the previous state of its largest-capacitance member. Every dynamic latch, every pipeline register bit, every OAM cell on these dies stores its bit *only* through this rule. There is no latch primitive anywhere in the system: storage is an emergent property of the resolution function applied to deliberately stranded nodes.

Three consequences follow. First, correctness of the tie-break is not negotiable in the small: a change that perturbs which member wins a floating resolution, even on a handful of obscure nodes, corrupts architectural state — our project history contains a black-screen regression traced to exactly such a perturbation on nodes lacking pull-ups. Second, the rule's inputs are part of the observable semantics: the strict-greater comparison, the first-seen-wins tie order, and therefore the order in which the walk encounters members, must be preserved byte-for-byte by any rewrite of the walk (the pair-path optimization of a later chapter carries an explicit obligation list for precisely this reason). Third, and constructively: because storage nodes are *physically* the large-capacitance participants — having dominant capacitance is what makes a node useful as storage — structural inequalities over the capacitance proxy can prove certain re-evaluations unable to change any value. That observation grows into the capacitance-dominance prunes, the strongest originality claim of this work; here we only note that it is rule (vii) that makes it possible.

2.3.4 A worked example: one event through a dynamic latch

We now trace the algorithm through one event on a deliberately minimal circuit. *The circuit is an illustrative invention for this section, not an excerpt of either netlist*, but it is the idiom — an NMOS inverter feeding a pass-transistor dynamic latch — that the real dies instantiate thousands of times.

```
          VCC                          (illustrative example)
           |
          [pull-up]                nodes:  IN   externally driven input
           |                               OUT  inverter output (PullUp flag)
   IN ──gate─| T1                          CLK  externally driven clock
           |        T2                     STO  storage node (no pull-up, no drive)
   OUT ──────┴────────/ ──────── STO    transistors:
           |      gate                     T1: gate=IN,  channel OUT–GND
          GND      |                     T2: gate=CLK, channel OUT–STO
                  CLK
```

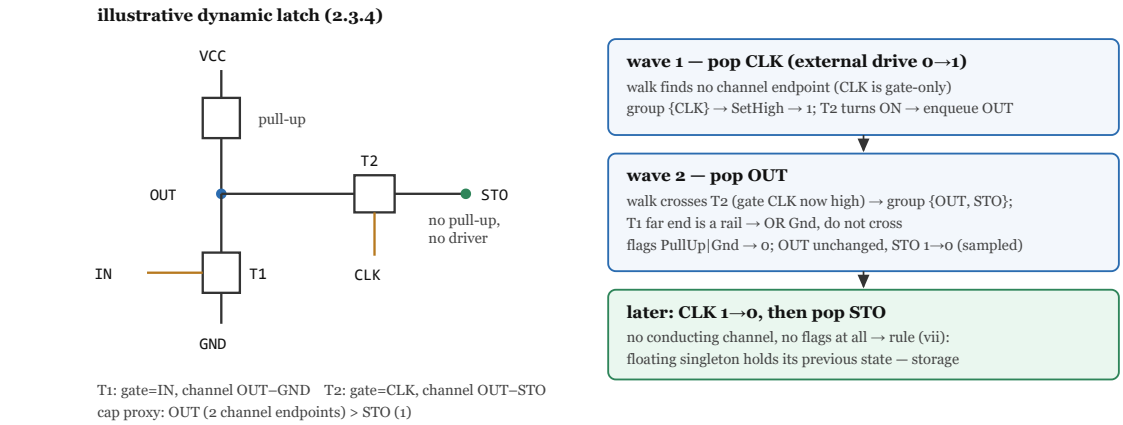

Figure 2.1. The worked example of Section 2.3.4: an NMOS inverter feeding a pass-transistor dynamic latch. Left: the circuit and the capacitance proxy; right: the CLK rising-edge event unrolled wave by wave, and how the post-falling-edge pop of STO lands in rule (vii)"s floating hold branch — storage emerges here.

Let the initial state be: IN driven high (`SetHigh`), so T1 conducts; CLK driven low, so T2 is open; OUT = 0 (its component resolves through rule (ii): the OR of `PullUp` and `Gnd` is a ground win — the ratioed inverter at work); and STO = 1, a previously sampled bit held by charge. The connection-count proxy gives OUT (two channel terminals) a larger capacitance than STO (one). The event: the external driver flips CLK from low to high.

The drive change rewrites CLK's flags from `SetLow` to `SetHigh` and enqueues CLK; the settle loop starts. **Wave 1** pops CLK. The group walk seeded at CLK finds no channel terminals at all — CLK is gate-only in this circuit — so the component is {CLK} and the OR-ed flags are `SetHigh` : rule (iv) resolves 1. The write-back changes CLK 0→1, so CLK's gate list is scanned: T2 turns on, and the turn-on case enqueues one channel endpoint, OUT. **Wave 2** pops OUT. The walk from OUT examines OUT's channels: T2's gate (CLK) is now high, so STO joins the component; T1's

gate (IN) is high, but its far terminal is the ground rail, so the walk does not cross — it ORs `Gnd` into the accumulator. The component is {OUT, STO} with flags `PullUp | Gnd`; rule (ii) resolves 0. Write-back: OUT stays 0 (no change, no events); STO changes 1→0. STO gates nothing here, so nothing is enqueued, the next list is empty, and the network is quiescent after two waves, two pops, and one two-member walk — precisely the modal event shape of the measured distribution. The latch has transparently sampled NOT(IN).

Now let the clock fall: CLK 1→0. The turn-off case enqueues *both* endpoints of T2, since the channel may now separate them. The next wave pops OUT: its component is {OUT} alone (T2 is open), flags `PullUp | Gnd`, value 0, no change. It pops STO: the walk finds no conducting channel and no flags whatsoever — STO has no pull-up, no drive, and is no rail — so the floating branch of rule (vii) runs over the singleton component and returns STO's own previous state, 0. Nothing changes; the machine is quiescent again.

The second event is the instructive one. The bit survives: from now until T2 next conducts, STO holds 0 with no driver anywhere — if IN now changes and OUT swings high, STO is unaffected, because OUT's activity stops at the open channel. That is dynamic storage emerging from rule (vii), exactly as promised in Section 2.3.3. And the pop of STO was, provably, a waste: a node whose only channel just opened, with no driver of its own, *must* float and hold — re-evaluating it cannot ever change anything. Multiplied across the die and across every clock edge, such guaranteed no-ops are a measurable fraction of all work, and suppressing classes of them before they are enqueued — with load-time structural proofs of safety — is the subject of the prune chapters.

2.3.5 Within-wave order is observable semantics

One property of the settle loop must be stated with care, because it constrains every optimization in this book: **the order of evaluation within a wave is part of the semantics**. The state array is updated in place, so a pop later in a wave reads values written by earlier pops of the same wave — the iteration is Gauss–Seidel, not Jacobi. The enqueue order of one wave fixes the pop order of the next; the pop order determines which node seeds a given component's walk; the seed and the adjacency order determine the walk's member encounter order; and the encounter order feeds the first-seen-wins clause of the floating tie-break, which is storage. There is no slack anywhere in that chain.

This is not a theoretical nicety. The family's reference implementations all execute the same FIFO discipline, which is why they can agree bit-for-bit at all; and our own experiments confirmed the contrapositive — within-wave reordering attempts changed simulation results and were abandoned as semantically invalid, not merely unprofitable. The consequence for engineering is a hard rule we will invoke repeatedly: *an optimization may delete work that provably cannot change state, and it may resolve a component by any method that reproduces the walk's writes byte-for-byte, but it may never reorder the work that remains.* Parallelization across the wave is excluded by the same argument, independently of its (also measured, also fatal) synchronization costs.

> The Gauss–Seidel order, the FIFO wave discipline, and the first-seen tie-break are inherited contingencies of chipsim.js [15], not laws of physics; a different reference order would define a different — equally defensible — fixed point. But a preservation project must pick one semantics and freeze it, and the family froze this one. Bit-exactness in this book always means "bit-exact with respect to that frozen order".

2.4 The Simulator Family

The simulators that execute this model on these netlists form a single family tree, shown in Figure 2.2. One piece of JavaScript is the common ancestor; the branches differ in scope, in faithfulness to that ancestor, and — as Chapter 1 previewed and the evaluation chapters quantify — by a factor of roughly five and a half in throughput between the most literal descendant and ours.

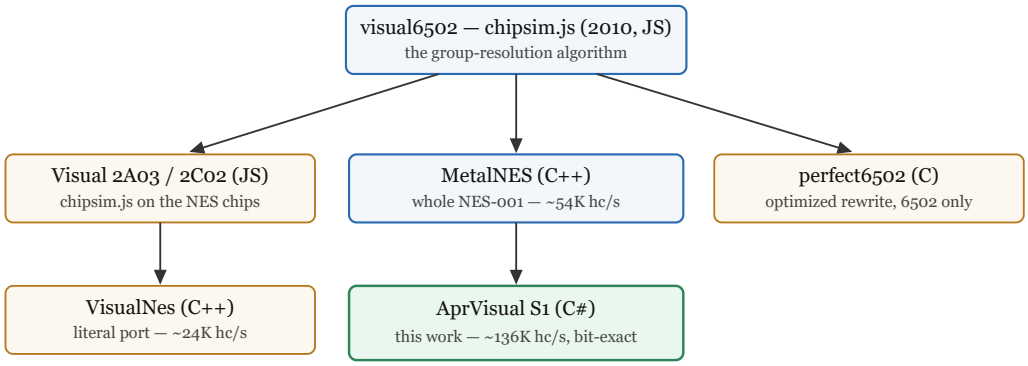

Figure 2.2. The simulator family: the Visual 6502 project's chipsim.js [15] and the Quietust 2A03/2C02 netlists [16] descend into VisualNes [19] (a literal port), perfect6502 [18] (an optimized 6502-only rewrite), and MetalNES [17] (a re-engineered whole-console composition), from which AprVisual derives [21].

2.4.1 visual6502 and the netlists

The Visual 6502 project [15] established the entire genre: extract a netlist from die photographs, then animate it in the browser with a few hundred lines of JavaScript — `chipsim.js` — implementing exactly the event-driven settle, group walk, and priority resolution of Section 2.3. The simulator's purpose was explanatory and archival, not fast; its legacy is that its *semantics*, including every contingent ordering decision discussed above, became the de facto reference model for the family. Quietust's Visual 2A03 and Visual 2C02 [16] applied the same extraction methodology to the two NES chips and published netlists in the same format, with two notable enrichments: the seventh transdefs column (Section 2.2.3) and node-name files that label the CPU's internal registers bit by bit — a convenience for debugging that our methodology deliberately declines to depend on.

2.4.2 The three descendants

Three independent projects then took the JavaScript off the browser, and their differing ambitions make them, fortuitously, a controlled experiment in implementation quality. **VisualNes** [19] is a line-by-line C++ port of the two chip simulators wired into a console; it preserves the JavaScript's data structures and inefficiencies (including a quadratic membership test in the group walk) so faithfully that it serves this book as the *unoptimized baseline* — the algorithm's cost when translated but not engineered. **perfect6502** [18] is the opposite pole: a genuinely optimized C rewrite with bitmap state storage and precomputed dependency lists, but scoped to the bare 6502 and — decisive for our purposes — without a charge-retention model for floating components. For a standalone 6502 that simplification is survivable; for a console whose video chip stores its sprite memory as stranded charge (Section 2.2.2) it is disqualifying, which is why no whole-NES branch grew from it.

MetalNES [17] is the re-engineered descendant and our direct ancestor: a C++ transistor-level simulation of the complete NES-001 board. Its measured throughput on our reference machine (~54K hc/s, versus ~24K for VisualNes; Section 2.4.4) quantifies what disciplined native engineering buys on an unchanged algorithm — roughly a factor of two over the literal port.

2.4.3 What MetalNES added

MetalNES's contributions define the platform AprVisual started from, and we credit them explicitly. First, *whole-system composition*: the module-definition machinery of Section 2.2.4 that assembles two chip netlists plus board TTL into one simulated console — no earlier family member simulated a system. Second, the *256-entry resolution table*: precomputing the priority ladder of Property 2.1 into a lookup keyed by the OR-ed flag byte, replacing a branch cascade in the innermost loop, and distinguishing “driven high” from “held high” in the flag taxonomy. Third, *behavioral memory handlers*: the callback (“fake transistor”) mechanism that exempts commodity board memories from transistor simulation. Fourth, *ForceCompute*: the rail-cancellation rule for composed bus nodes. MetalNES also extends in a direction we deliberately do not follow — analog output ladders that model the video and audio DACs at the voltage level — and it omits two things AprVisual had to add: it keeps only the `'+'` pull markings of segdefs, and it has no bit-exactness methodology and no checksum discipline.

2.4.4 AprVisual's position in the family

AprVisual began as a function-for-function C# port of MetalNES's wire core — the porting was conservative by design, so that the starting point was a known quantity — and then diverged along two axes that the rest of this monograph documents. The first axis is *verification*: full-state checksums at fixed half-cycle counts, golden values frozen across the project's history, and a sprite-heavy ten-million-half-cycle gate (Section 2.5). The second axis is *performance over an unchanged model*: the dispatch classes, prune families, renumbering, and layout techniques of the later chapters, every one gated on the verification axis. Table 2.2 gives the family's measured standings — same machine, same ROM, same unit, all compiled and benchmarked by us rather than quoted from their authors.

Table 2.2. The family measured on one machine, one ROM, one unit (master-clock half-cycles per second; measured 2026-06-08, AprVisual's current figure 2026-06-12). The perfect6502 row uses a 6502-clock half-cycle — an order of magnitude coarser — and is therefore not rankable against the whole-console rows.

Simulator	Scope	Language	Relationship to chipsim.js	Throughput
VisualNes [19]	whole NES	C++	literal line-by-line port	~24K hc/s
MetalNES [17]	whole NES	C++	re-engineered descendant	~54K hc/s
perfect6502 [18]	6502 only	C	optimized rewrite, no charge model	~29K (6502 hc; different unit)
AprVisual [21]	whole NES	C#	MetalNES port + this book's program	~136K (current record)

The VisualNes, MetalNES and perfect6502 numbers in the table are not native outputs of those projects — none of the originals ships a throughput measurement; for a same-unit comparison we experimentally instrumented each project's source with half-cycle counting and timing identical in definition to ours, and measured on the same machine and workload (Chapter 8).

Read as a single progress bar — 24K for the unoptimized algorithm, ~54K for disciplined native engineering, ~136K for the present work — the table separates three contributions that are often conflated: the algorithm, its implementation quality, and the optimization program built on top. The gap between the last two rows is the subject of this book; the gap that remains above all of them is also part of the story, and we state it here once for calibration: real silicon runs at 42,954,552 half-cycles per second, so the current engine stands approximately 316× from real time. The monograph's evaluation chapters argue, with measured negative results rather than intuition, that this residual gap is structural for a single core executing this model faithfully.

2.5 Terminology and Units

Throughout this book, performance is expressed in **master-clock half-cycles per second (hc/s)**. One *hc* is one toggle of the board's clock node — the engine's fundamental stimulus step (Section 2.3.1) and the finest time base the composed system has. All of the console's clocks are fixed integer divisions of it, so a throughput in hc/s converts directly into “how fast the simulated silicon is running”; Table 2.3 collects the conversions used everywhere in the book.

Table 2.3. Time-base conversions for the NTSC NES. R denotes a measured throughput in hc/s.

Quantity	Relation	Real-hardware value
master-clock half-cycle (hc)	one toggle of the board `clk` node	42,954,552 hc/s
master clock	hc / 2	21.477 MHz
PPU pixel clock	hc / 8	≈5.37 MHz
CPU clock	hc / 24	≈1.79 MHz
one video frame	714,732 hc	60.0988 frames/s
equivalent frame rate	R / 714,732	—
distance from real time	42,954,552 / R	—

The unit was chosen for honesty across the family: it counts the work a whole-console simulator actually performs, it is independent of which chip is busy, and it exposes unit mismatches that per-“cycle” figures hide (the perfect6502 footnote of Table 2.2 being the standing example). At the current ~136K hc/s, the engine renders roughly a fifth of a frame per second; the real-time requirement is the 42.95M hc/s of Table 2.3.

Bit-exact, used as a technical term from here on, means the following concrete protocol. The engine computes an FNV-1a hash over the complete node-state array; a configuration is bit-exact if it reproduces the frozen golden checksums at 300k, 400k, and 1M half-cycles (0x794A43ABDF169ADA, 0x9174E19D961CB6E5, and 0x6D4CCBCE2E9CD599 respectively) and additionally survives a ten-million-half-cycle gate on a sprite-heavy commercial workload (Super Mario Bros.). Because the hash covers every node, agreement implies per-node, per-snapshot equality of the entire simulated die state, not merely of architecturally visible registers; and because the semantics is frozen (Section 2.1.3), any divergence anywhere is a defect by definition. Every optimization reported in this book passed this protocol before its performance was even considered.

Finally, the engine-internal vocabulary used throughout: a **pop** is one node evaluation drawn from the settle worklist; a **settle wave** is one drain of the double-buffered list; the **group walk** is the per-event BFS of Section 2.3.2 (the literature's CCC evaluation [1]); and the **dispatch classes** partition nodes by the structural census of Section 2.2.4 — **cls1** for static singletons (no pass channels: the component is provably {*n*} forever), **cls2** for dynamic-singleton candidates (channels exist, but whenever all their gates are momentarily off the component is {*n*} right now), and **clso** for the callback/ForceCompute residue that must always take the general path. These names recur in every optimization chapter; their populations — 26.7%, 73.2%, and a fraction of a percent of nodes respectively — are the static skeleton on which the dynamic measurements of Section 2.3.1 hang.

This book keeps the project"s internally grown names; Table 2.4 maps them to the customary terms of the EDA and systems literature. Each mechanism"s chapter restates the correspondence on first use.

Table 2.4. Correspondence between this book"s terms and customary literature terms.

This book	Customary literature term	Refs
group walk	channel-connected component (CCC) evaluation	[1, 2]
settle wave	one drain of the event queue in event-driven simulation; kin of selective trace	[4]
flags-OR + priority LUT resolution	switch-level strength-lattice resolution	[1]
floating tie-break (largest capacitance holds)	charge-sharing resolution; capacitive strength ordering	[1]
dynamic singleton (R-1), pair path (B1)	cardinality-specialized early-outs over dynamic active-subnetwork sizing	[3]
same-state turn-on prune (P-1), turn-off isolation prune (P-2)	event suppression in load-time structural-proof form	[3, 4]
capacitance-dominance un-taint (P-3/P-4)	nearest neighbour: IRSIM"s maxnode size-coarsening practice (differences in §3.2)	[3]
class-major renumbering + range compares (range-prune)	data reorganization under inspector–executor separation	[7–9]
self-captured first-touch key	trace-driven data layout; first-touch reordering	[8, 10–12]
safe fallback (range verification fails → prunes off)	deoptimization guard	[13]

Chapter 3. Related Work

Every optimization reported in this monograph was subjected to an adversarial prior-art mapping before it was allowed to be called a contribution: an external model was instructed to *refute* novelty rather than confirm it, each technique was mapped to its closest named precedent, and the surviving "plausibly original" items were then checked by title-level searches constructed so that a collision, if one existed, would surface. The working principle was that we would rather learn we had reinvented something forty years old than claim false novelty. This chapter is the prose record of that exercise. We use a fixed three-level verdict vocabulary throughout: **KNOWN** (standard literature; we cite it and claim nothing), **VARIANT** (the concept is known; our form or engineering differs in ways worth stating), and **PLAUSIBLY ORIGINAL** (no precedent found by adversarial mapping plus title-level search — a claim about searches performed, not a proof of absence).

> The premise that makes every comparison below meaningful: our resolution model **is functionally Bryant's MOSSIM II** [1] minus the X (unknown) state. The priority resolution and the largest-capacitance floating tie-break are his discrete-strength / node-size lattice formalization, inherited through the Visual 6502 project's `chipsim.js` [15]. What the engine calls a "group walk" is what the literature calls **channel-connected component (CCC) evaluation**. We admit this first; the differences claimed afterwards are differences *on top of* that model, not differences of model.

3.1 Switch-Level Simulation

Two distinct lineages converge in this work: the academic switch-level school of the 1980s, and the die-shot reverse-engineering lineage of the 2010s, which reimplemented the school's core algorithm largely outside the literature. We summarize both, then place our dispatch-level techniques against them.

3.1.1 The Classical School: MOSSIM II, COSMOS, IRSIM

Bryant's switch-level model [1] is the semantic foundation of everything in this monograph. It abstracts an MOS network into nodes carrying discrete states and transistors acting as switches, and — critically for our purposes — it formalizes *charge sharing*: when a set of nodes is connected through ON transistors with no driver present, the resolved state is determined by a lattice of discrete node sizes, with the largest-capacitance member dominating. Our 256-entry priority LUT (ground beats power beats external drive beats pull-up beats hold) and our largest-capacitance floating tie-break are direct descendants of this formalization. The one semantic subtraction is the X state: the die-shot lineage discarded ternary simulation in favor of deterministic two-state relaxation, which is also why within-wave Gauss-Seidel order becomes observable semantics in our engine (Chapter 4).

Before Bryant, Ulrich [4] articulated the founding insight of all event-driven simulation: simulate only activity. A network in which most of the state is quiescent most of the time should be charged only for the parts that change. Our event queue, our settle waves, and ultimately our entire prune family (Section 3.2) descend from this principle; the prunes can be read as progressively sharper answers to Ulrich's question "what counts as activity?" under switch-level semantics.

COSMOS [2] took the compiled route: a network preprocessor (Anamos) statically partitions the transistor network into channel-connected components, derives each component's Boolean steady-state behavior, and compiles the result into executable code, eliminating the interpretive walk entirely. COSMOS matters to us twice. First, its static CCC extraction is the named precedent for our cls1 dispatch class (Section 3.1.3). Second, its static subgraph template matching — recognizing transmission gates and latches and emitting specialized code for them — is the precedent against which our B1 pair path must be measured. We note, without elaborating here, that our own measured experience with compiled and oblivious evaluation on this netlist was uniformly negative (3–6× slower than the event-driven engine; Chapter 9); the COSMOS trade-off is real but does not pay on a netlist whose conducting groups average 1.13–1.4 nodes.

IRSIM [3] is the closest ancestor in spirit: an incremental, event-driven switch-level simulator with linear RC timing, in which evaluation operates over the *dynamic* subnetwork of currently-ON transistors rather than a static partition. Three IRSIM ideas recur throughout Section 3.2 as nearest neighbors of our techniques: its dynamic active subnetworks, its charge-preservation behavior on isolated nodes, and its node-size coarsening practice for charge-share resolution.

3.1.2 The Die-Shot Lineage

The second lineage begins with the Visual 6502 project [15], which derived a transistor-level netlist of the MOS 6502 from die photographs and implemented a two-state Bryant-style relaxation in JavaScript (`chipsim.js`). Quietust extended the methodology to the NES chipset, producing the Visual 2A03 and Visual 2C02 netlists [16] that this project consumes. The algorithm was subsequently reimplemented several times: perfect6502 [18], a C port of the 6502 simulation; VisualNes [19], a C++ port of the 2A03/2C02 simulators; and MetalNES [17], a C++ composition of both chips with the NES-001 board's TTL logic into a whole-console simulation. MetalNES is our direct ancestor: our engine began as a C# port of its wire-level core.

This lineage is engineering-driven and largely disconnected from the academic school whose algorithm it carries; one purpose of this monograph is to reconnect them. For calibration, we measured the family on one machine, one ROM, and one unit (master-clock half-cycles per second, whole-console where applicable) — none of these original implementations ships an hc/s measurement of its own, so we experimentally instrumented their source code to obtain same-unit figures (details in Section 2.4): VisualNes sustains roughly 24K hc/s, MetalNES roughly 54K, and the engine described here roughly 136K; perfect6502 measures roughly 29K in a 6502-only unit that is not directly rankable against whole-console figures [21]. The ~2.5× over our direct ancestor is the cumulative subject of Chapters 5–8.

3.1.3 Where Our Dispatch Techniques Sit

The engine resolves each event through one of several dispatch classes (Chapter 4), and each class has a literature address.

cls1, the static singleton, is KNOWN. A node with no pass-transistor channels can never join a multi-node conducting group; its resolution is O(1) by static structure alone. This is the degenerate case of static CCC extraction as practiced by MOSSIM II and COSMOS [1, 2] — COSMOS goes further and compiles such unidirectional logic directly into Boolean gates. In our netlist, 26.7% of nodes (3,929) fall into this class. We cite the precedent and claim nothing.

R-1, the dynamic singleton, is a VARIANT. At the moment an event is processed, if every pass gate incident on the node is OFF *right now*, the conducting group is exactly the singleton {*nn*} and can be resolved in O(1), bit-identically to the full walk. The underlying concept — that the relevant network is the dynamic one over currently-ON switches — is IRSIM's dynamic active subnetwork [3]. Our novelty is implementation-level rather than theoretical: the runtime *cardinality proof* is used as an *early-out dispatch*, an O(1) inline path that bypasses the generic relaxation machinery entirely. The honest framing, which we adopt, is "dynamic sizing applied as an early-out execution filter." Measured in the ablation of Chapter 8, this step is worth +18.7% on its own — the largest single step in the project — which we read as evidence about modern memory hierarchies, not about new simulation theory.

B1, the pair path, is a VARIANT with a dynamic twist. When exactly one incident gate is ON and the neighbor's ON channels all lead back to the seed, the group is provably {seed, neighbor}; the pair is then resolved inline, replicating the generic walk's order-sensitive semantics byte for byte (member-hash clearing, both supply scans, the tie-break form, the write order), with every bail-out taken before any mutation. The named precedent is COSMOS/Anamos static template matching [2], which recognizes transmission-gate and latch subgraphs at compile time. Ours differs in binding time: it is *runtime topological pattern matching on the active channel graph*. Theoretically this is nothing more than loop-unrolling the BFS for group size two; we present it as effective engineering, not theory. In the ablation ladder of Chapter 8, this step measures +7.0% (S6→S7, same-day interleaved); about +8.9% after separating the attribution from the adjacent housekeeping commit.

3.2 Event Suppression Across Abstraction Levels

The prune family (P-1 through P-4; Chapter 6) suppresses events whose processing is provably a no-op. Suppressing provably-null work is among the oldest ideas in simulation; what distinguishes the members of our family is *what is proved, when, and what the proof is spent on*. This section walks the abstraction stack from IRSIM upward and places each prune precisely.

3.2.1 Selective Trace, Equipotential Bypass, and P-1

Ulrich's selective trace [4] established that no-change need not propagate. IRSIM applies a switch-level instance of the same instinct: when a transistor turns on between nodes at equal potential, the connection changes nothing and work can be bypassed [3]. Our P-1 — suppress the turn-on enqueue when the two channel endpoints hold equal states — is a VARIANT of this standard idea, and we label it as such.

What we believe is worth stating is the *taxonomy of when an equal-state merge is still unsafe* under discrete charge sharing. In a Bryant-style two-state model, the floating tie-break is not an edge case: it *is* the chip's storage mechanism, and an equal-state merge can still alter which member's capacitance dominates a later floating resolution. Our first, naive implementation of P-1 shipped a black screen — the diverging nodes were precisely the no-pull-up dynamic storage nodes (OAM and palette RAM cells) living on the floating tie-break. The production form encodes the unsafety as a load-time structural taint: no-pull-up nodes and ForceCompute channel components are excluded from the prune. The taxonomy, not the bypass, is the contribution; the bug that motivated it is the literature's own quirk made empirical.

3.2.2 The Nearest Neighbor: IRSIM's Node-Size Practice versus Capacitance Dominance (P-3/P-4)

P-3 and P-4 carry the strongest claim in this monograph, so their nearest neighbor deserves the most careful treatment. The semantic substrate is again Bryant's node-size lattice [1]: in a purely-floating group, the largest-capacitance member's previous state wins. P-3/P-4 derive, at load time and fully automatically, a strict dominance inequality: a driverless node whose capacitance is strictly below that of *all* of its channel neighbors can never be the largest-capacitance member of any group it joins, hence can never influence the floating tie-break, hence an equal-state merge through it is provably a no-op even though the node sits on the storage mechanism. The proof un-taints a subset of the nodes that P-1's safety taint had excluded, and the un-taint is spent on event suppression — measured at +5.96% (P-3) and +1.71% (P-4), bit-exact.

The title-level scoop scan found no direct collision, but it did surface a must-cite nearest neighbor: IRSIM's node-size (`maxnode`) practice [3], in which users manually assign coarse discrete sizes — high-capacitance nodes such as precharged buses a larger size, ordinary nodes a default, storage nodes optionally the smallest — to simplify and accelerate charge-share resolution. The resemblance is real (both exploit capacitance ordering among nodes) and the differences must therefore be drawn explicitly. They differ on four axes. *Authorship:* IRSIM's sizes are hand-assigned by the user; our inequality is derived automatically from extracted capacitances. *Epistemic status:* size assignment is a coarsening — an approximation the user vouches for; ours is a proof, and the result is bit-exact against the unpruned engine. *Mechanism:* IRSIM's sizes feed the resolution function itself; our inequality never touches resolution — it gates enqueues. *Purpose:* resolution simplification versus event suppression. A satisfying corollary of the relational form (dominance over *all* channel neighbors) is that heavy storage cells exclude themselves: having the locally dominant capacitance is precisely what makes a node storage, so the proof never fires where it would be dangerous.

3.2.3 P-2: From Runtime Outcome to Load-Time Proof

IRSIM's charge preservation rules ensure that a node isolated by a turn-off retains its charge — but in IRSIM this is a *runtime evaluation outcome*: the event is processed, the isolation is discovered, the state is held [3]. P-2 makes the inverse move: for a node that is statically degree-1 and driverless, the outcome of its only channel turning off is decidable at load time — the node isolates and holds, always. The event is therefore suppressed *before the queue*, saving the entire enqueue→pop→resolve chain rather than just the downstream propagation. The title-level scan found no collision (only thematically adjacent titles), and we advance P-2 as a PLAUSIBLY ORIGINAL, secondary claim: modest in mechanism, but representative of the family's signature move — promoting a runtime discovery to a load-time structural proof and spending it earlier in the pipeline than the literature does.

3.2.4 Suppression at Other Abstraction Levels

Analogous suppression machinery exists above and below the switch level, and the distinctions are worth drawing because they bound what our methodology can claim. Fast-SPICE simulators bypass numerical integration for latent subcircuits — a dynamic, runtime detection over a continuous model, with accuracy traded explicitly; our masks are static, load-time, and bit-exact. RTL synthesis and simulation exploit observability don't-cares and clock gating — but RTL has explicit register primitives, so "this storage element cannot change" is syntactically available; we must infer implicit state from a flat sea of transistors. Closest in technique is the EDA tradition of automated extraction and pattern recognition over transistor netlists: LOGEX [5] automatically extracts gate-level logic from transistor-level netlists, and SubGemini [6] identifies subcircuits in a netlist via fast subgraph isomorphism — both belong to the library- and structural-pattern-driven automated extraction family, an established practice of nearly four decades. Our "identify memory by its physics, not its name" pass (pull-up flags, channel-component union-find, capacitance comparison, driven-pin exclusion; zero hand-listed nodes) identifies by physical properties rather than by structural pattern matching — a different approach within the same tradition — and claims no algorithmic novelty. The application differs: extraction historically answers *what to evaluate or abstract*; ours answers *what to drop* — it parameterizes the safety classes of an event-culling system. Following the adversarial consult's advice, we present this as supporting methodology (Chapter 6), not as a contribution.

3.3 Data Layout and Locality

The third family — class-major renumbering, range-encoded prune predicates, and the self-captured first-touch key (Chapter 7) — sits in a different literature entirely: runtime data reorganization and cache-conscious layout. Every component has a clear lineage; the synthesis, we will argue, does not.

3.3.1 The Inspector-Executor Lineage

The paradigm of inspecting an irregular computation's access pattern at runtime and reordering data (and/or iterations) before executing an optimized schedule is the inspector-executor model, established by the CHAOS runtime library [7], extended to cache-oriented data and computation reorganization by Ding and Kennedy [8], and given a compile-time composition framework by Strout, Carter, and Ferrante [9]. Our three-pass load is recognizably an inspector-executor: the classification pass and the instrumented capture pass constitute the inspector; the final rebuild is the data reordering; the production run is the executor. Ding and Kennedy's first-touch packing [8] — reorder array elements by the order of first access observed by an inspector — is the closest algorithmic ancestor of our first-touch key, and we name it as such.

Three differences separate our instance from the lineage. First, *what is inspected*: not the index arrays of an irregular parallel loop nest, but the event cascade of a discrete-event simulator — an object that exists only dynamically and is observable only by running the engine itself, which is why the inspector is a cold instrumented copy of the settle loop rather than an analysis of inputs. Second, *process topology*: the entire cycle is in-process and self-contained, re-derived from the netlist at every load in about 1.3 seconds, with no profile files; staleness — the classic failure mode of offline profile-guided methods — is excluded by construction, and the online key measurably beat a loaded offline profile of the same form by +4.94% in a same-executable head-to-head (16/16 paired wins). Third, *what is reordered*: the simulator's entire node-id space, with semantic invariance under permutation established constructively and then verified (Section 3.3.3).

3.3.2 Trace-Driven and Cache-Conscious Layout

A parallel lineage approaches layout from the allocator, garbage collector, and compiler: cache-conscious structure layout and splitting [10], hot-data-stream abstractions for reference locality [11], compiler-directed data placement driven by profiles [12], and the patented compiler-level form of profile-guided data layout [20]. These establish that temporal access behavior, not static structure, should govern placement — our premise as well. The differences are again of layer and of mechanism: those systems place heap objects or structure fields using offline or sampled profiles consumed by an allocator or compiler; we rebuild a simulator's id space from its own production cascade at load time.

One empirical finding of ours sharpens this literature rather than merely instantiating it: the value of the captured key lies in the *order* of the pruned production cascade, not in cache-line density. Measured with an in-engine instrument counting hot-structure cache lines touched per half-cycle: the blind structural key touches 118.4 lines/hc; an offline profile key, 109.5 (+1.55% wall-clock); a true capture taken with the prunes disabled reaches equal density (110) yet yields ±0.0%; the same capture with the prunes enabled — the order the production engine will actually execute — reaches 105.9 and +6.17% (20/20). Equal density, different order, opposite outcomes. The cache-conscious layout literature optimizes density proxies; our data say density was necessary but not sufficient here.

3.3.3 The Deoptimization-Guard Pattern

The correctness regime around the relayout is a direct application of a known pattern, and we cite rather than claim it. Hölzle, Chambers, and Ungar's dynamic deoptimization [13] established the discipline of running aggressively optimized code under guards, with a semantically transparent fallback to a baseline when an assumption is violated. Our prune masks are recomputed from ground truth at every reset; the range-derived bits are verified against them; on any mismatch the engine degrades to a safe boundary — prunes off, supply invariants kept, correctness preserved. This is the deoptimization-guard pattern transplanted from JIT compilers to a simulator's load path.

3.3.4 Prefetchers and Memory-Level Parallelism

Why should order matter at equal density? The hardware account comes from the prefetching and memory-level-parallelism literature: spatial memory streaming [14] documents that prefetch effectiveness depends on recurring access *sequences* correlated with code behavior, not merely on which lines are resident. A layout matching the production schedule presents the settle loop with near-monotone access sequences that train stream-oriented prefetchers and keep load queues full; an equally dense but non-production order touches the same hot lines in a scrambled sequence within each half-cycle. We offer this as the plausible mechanism for the order-versus-density result, with the caveat that we did not isolate prefetcher counters directly.

3.3.5 Practical Analogues Outside the Academic Literature

Three engineering practices are honest spiritual twins of the range-encoding half of this work. Entity-component-system (ECS) archetype storage in modern game engines sorts entities into contiguous memory blocks by component ownership, so hot loops drop per-entity membership checks — a one-to-one analogue of our class-major renumbering, where a static predicate becomes a contiguous id range and a hot-loop table lookup becomes a register compare. Database clustered indexes and range partitioning realize the same trick declaratively: cluster on (class, captured-order) and a class predicate becomes a contiguous scan. Copying garbage collectors are the analogue of the rebuild itself: the copy order (the collector's traversal) determines the spatial locality of co-accessed objects, and our rebuild passes amount to a relocating trace whose order is measured rather than reachability-derived.

None of these — nor the academic lineages above — capture an event-driven simulator's *own cascade, in-process, as the layout key*. The game engine sorts by static component sets; the database by declared keys; the collector by reachability; the inspector-executor systems by loop index streams; the profile-guided compilers by offline traces of other runs. Our key is the measured first-pop order of the production pruned cascade — an artifact of the system's dynamics, not its structure. Structure-based orders demonstrably fail on this workload: a rules-based BFS from a stable snapshot measured *worse* than the blind structural key (174 versus 118.4 lines/hc), because a bidirectional pass-transistor network's topology does not map to its execution flow. That gap — dynamics-derived versus structure-derived order, executed entirely inside the process being optimized — is the specific synthesis we claim in Section 3.5.

3.4 The Verdict Matrix

Table 3.1 condenses the mapping. Method: each row's verdict was produced by the adversarial consult (an external model instructed to find the strongest refuting precedent), then checked by title-level scoop scans over IEEE Xplore, the ACM Digital Library, and Google Scholar using query sets designed to surface a collision if one exists (for example, combinations of "switch-level simulation" with "charge sharing" and "event suppression", or "inspector-executor" with "event-driven simulation"). Verdicts are bounded by the effort and recall of those searches; they are claims of due diligence, not proofs of absence. Full records — claim wordings, search keys, per-item citations — are preserved in the project repository [21].

Table 3.1. Prior-art verdict matrix. KNOWN = standard literature, cited and not claimed; VARIANT = known concept, our form differs as stated; PLAUSIBLY ORIGINAL = no precedent found by adversarial mapping plus title-level search.

Our technique	Nearest named prior art	Verdict	One-line difference
cls1 static singleton dispatch	Static CCC extraction; compiled Boolean components [1, 2]	KNOWN	None claimed; standard practice, cited.
R-1 dynamic singleton	IRSIM dynamic active subnetworks [3]	VARIANT	Runtime cardinality proof used as an O(1) early-out dispatch, not a new sizing theory.
B1 pair path	COSMOS/Anamos static template matching [2]	VARIANT (dynamic twist)	Runtime topological matching on the active channel graph; semantically an unrolled size-2 walk, replicated byte for byte.
P-1 same-state turn-on prune + safety taint	Selective trace [4]; IRSIM equipotential bypass [3]	VARIANT	Adds the taxonomy of when an equal-state merge is still unsafe under discrete charge sharing, encoded as a load-time structural taint.
P-2 turn-off isolation prune	IRSIM charge preservation (a runtime evaluation outcome) [3]	PLAUSIBLY ORIGINAL (secondary)	Elevates the hold outcome to a load-time static proof; suppresses before the queue, saving the whole enqueue→pop→resolve chain.
P-3/P-4 capacitance-dominance un-taint	Bryant node-size lattice [1]; IRSIM node-size (`maxnode`) practice [3]; storage extraction [5]	PLAUSIBLY ORIGINAL (strongest)	Automatically derived dominance inequality proving tie-break immunity, spent on bit-exact event suppression — versus manual size coarsening for resolution simplification.
Physics-based memory identification	State-element extraction [5, 6]; Fast-SPICE latency bypass; RTL ODC/clock gating	KNOWN technique, novel application	Extraction spent on safety classes for event culling ("what to drop"), on implicit state in a flat netlist; presented as methodology, not a claim.
Range-encoded prune predicates (id-range compares)	Index-space partitioning practice; ECS archetypes; cache-conscious layout [10, 12]	VARIANT (practice)	No single named origin in the literature; a component of the relayout system, not claimed independently.
Self-captured first-touch relayout	Inspector-executor [7, 8, 9]; trace-driven layout [10, 11, 12]; profile-guided layout patent [20]	PLAUSIBLY ORIGINAL (synthesis)	In-process, self-contained capture of an event-driven simulator's own pruned cascade as the layout key; value shown to be order, not density.

Verification + safe-degenerate fallback	Dynamic deoptimization guards [13]	KNOWN	Direct application of the JIT guard pattern to a simulator's load path; cited.

3.5 Scope of the Novelty Claims

The mapping above leaves exactly two top-level claims and one secondary claim standing, and we state them here in the form we are prepared to defend.

Claim 3.1 (capacitance-dominance tie-break immunity; P-3/P-4). In a Bryant-style discrete charge-sharing model, the statically derived strict inequality "a driverless node's capacitance is less than that of every channel neighbor" is a sufficient condition for tie-break immunity: the node can never determine a purely-floating group's resolution. This proof safely enables equal-state event suppression on a subset of dynamic storage nodes, bit-exactly, with the heaviest storage cells excluding themselves by the same physics that makes them storage.

Claim 3.2 (self-captured trace relayout). An in-process, self-contained inspector-executor pass that re-derives a switch-level simulator's node-id space from the production pruned cascade's first-touch order at every load — range-encoding static safety predicates as contiguous id blocks, guarded by ground-truth verification with a safe-degenerate fallback — is, to the best of our search, without precedent as a synthesis; and its measured value lies in the cascade's order rather than cache-line density.

Claim 3.3 (secondary; P-2). Statically identifying degree-1 driverless leaves and suppressing their disconnection events before the queue promotes IRSIM-style runtime charge preservation to a load-time proof, deleting the full event-processing chain rather than the propagation alone.

Equally important is what we explicitly do *not* claim. We do not claim the resolution model: it is Bryant's [1], inherited via the Visual 6502 project [15], and we say "functionally MOSSIM II" in every framing. We do not claim CCC evaluation, static CCC extraction, or event-driven selective trace [1, 2, 4]. We do not claim theoretical novelty for R-1 or B1: they are engineering variants of IRSIM's dynamic subnetworks and COSMOS's template matching respectively, and the consult's suggested third claim — a "cardinality-dispatch architecture" — was deliberately demoted to variant framing rather than advanced as a contribution. We do not claim the physics-based identification of memory as an algorithmic contribution; it is forty-year-old extraction redeployed, and it appears in this monograph as methodology. We do not claim the deoptimization-guard pattern [13], the inspector-executor paradigm [7], or first-touch packing [8]; our relayout claim is strictly about the synthesis and the order-not-density finding. We do not

claim that any verdict in Table 3.1 proves absence of prior work: title-level scans bound the search effort, and the one near-collision they did surface (IRSIM's `maxnode` practice) is distinguished, not dismissed, in Section 3.2.2. Finally, we do not claim generality beyond what was measured: all primary numbers come from one netlist family and one host microarchitecture (Zen 2), supplemented by community leaderboard datapoints [21].

Chapter 4. Engine Architecture and the Correctness Infrastructure

This chapter describes the engine as it actually runs: the memory layout of the hot state, the event loop that drains it, the build pipeline that produces it from the raw netlists, and — given equal weight, deliberately — the correctness infrastructure that made the optimization campaign of Chapters 5–7 possible at all. The algorithmic content is conservative by design: the resolution semantics are those of the Visual 6502 family [15][16][17] and are functionally a restriction of Bryant's switch-level model [1] (Chapter 3 develops that correspondence). What is specific to this engine is the data organization, the discipline that every transformation be provably or verifiably bit-exact against that semantics, and the apparatus — golden checksums, per-node dump-diffing, and a permutation-aware power-on procedure — that turns "bit-exact" from an aspiration into a mechanical gate. Throughout, numbers are measured on the reference machine and workload described in Chapter 8; we quote them here only to motivate structural choices.

4.1 Data Layout

4.1.1 Structure-of-arrays on an unmanaged heap

The engine is a single static class whose hot state lives entirely in unmanaged memory, allocated once per `Reset()` through aligned-allocation wrappers and freed in one sweep. The choice of a structure-of-arrays (SoA) decomposition over an object graph is the foundational layout decision. The natural object-oriented encoding — a `Node` object holding its state, flags, and adjacency lists — survives in the engine only as the *build-time* representation (Section 4.3); before simulation begins, everything the inner loop touches is flattened into parallel arrays indexed by node id. The rationale is the standard one from the cache-conscious-layout literature [10][12]: the engine's behavior is dominated by dependent loads over a working set of a few hundred kilobytes, so co-locating fields by *access pattern* rather than by *entity* determines how many cache lines each event touches.

The decomposition is access-pattern-driven down to individual fields. The per-node logic value is a single byte in `NodeStates` — the most frequently read array in the engine, since every gate test during a group walk and every same-state comparison during enqueueing reads it. The per-node descriptor `NodeInfo` (Section 4.1.2) holds the flags and channel adjacency read during evaluation. Two fields that a naive design would place inside `NodeInfo` are split into separate "cold" arrays precisely because their access patterns differ: `NodeConnections`, the capacitance proxy, is read only by the purely-floating tie-break (under 1% of group walks reach it), and `NodeTlistGates`, the index of the node's gate-fanout list, is read only by `SetNodeState` at writeback time — once per group member per state change, not once per BFS visit. Keeping them out of the 16-byte hot record doubles the number of hot records per cache line. This is structure splitting in the sense of Chilimbi et al. [10], applied with the split boundary chosen from measured access frequency rather than field size.

4.1.2 The 16-byte NodeInfo and the inline/overflow union

Each node's evaluation-time descriptor is an explicit-layout, 16-byte structure — four per 64-byte cache line. The first four bytes are a flags byte (pull-up, external drive, supply, ForceCompute, callback — the bits that participate in group resolution), an `Inline` discriminator, a count of inline channel pairs, and a packed pair of nibble counts for ground/power channels. The remaining twelve bytes are a union with two interpretations selected by the discriminator:

- **Inline form** (~96% of nodes): six 16-bit slots holding the node's complete channel adjacency directly — first the (gate, other-end) pairs of its pass channels to normal nodes, then the gate ids of its channels to ground, then the gate ids of its channels to power. A node qualifies whenever 2·pairs + gnd + pwr ≤ 6.
- **Overflow form** (~4% of nodes — buses, clock fanout trees): three 32-bit indices into the shared flattened adjacency array (Section 4.1.3), one per channel bucket.

The two forms are mutually exclusive and every hot access is gated on the discriminator, so the union is safe. The point of the inline form is the elimination of a dependent pointer chase: for the overwhelming majority of nodes, the dispatch test "are all of this node's pass gates OFF?" (the R-1 dynamic-singleton check, Section 4.2.2), the O(1) singleton resolution, and the BFS expansion step all complete out of the *one* cache line that the descriptor load already brought in. On a memory-latency-bound walk whose average conducting group is 1.13–1.4 nodes, the difference between one dependent load and two is a first-order effect; the inline-payload change (named S2-A in the engineering log) was among the larger single layout wins. As a secondary benefit, inline nodes no longer emit their channel sublists into the shared adjacency array at all, shrinking the array that the remaining overflow walks traverse.

The bucketing of channels into *normal* / *to-ground* / *to-power* classes is itself semantic, not just organizational. A channel whose far end is a supply node can never grow the group — its only effect is to contribute the Gnd or Pwr flag — so the BFS expansion loop iterates pairs only, and supply contact is detected by a flat scan with an early break (or a branchless OR on the fast path). This is the layout-level encoding of the observation that supply nodes must never enter a group or a queue.

4.1.3 The flattened adjacency and the gate-fanout lists

All adjacency that does not fit inline lives in `TransistorList`, a single array of 16-bit node ids organized as zero-terminated sublists; index 0 is a sentinel zero so that an "empty sublist" reference of 0 immediately reads a terminator. Two representational decisions matter. First, ids are 16-bit: the lowered system has ~14.7K nodes, comfortably below 65,536, and halving the element width halved what was then the engine's hottest array from 697 KB to 350 KB — a pure working-set reduction with no algorithmic content, worth more than most instruction-level tricks on this workload. Second, the array carries at least four trailing zero pads, so that loops which read *two* (c1, c2) pairs per iteration as a single 64-bit load (Section 4.2.4) can never fault by over-reading past the final terminator; the padding reads as terminators and is harmless.

Every node, inline or not, additionally references a *gate-fanout* sublist: the (c1, c2) endpoint pairs of every transistor whose *gate* is this node. This list is the engine's event-propagation edge set — it is read exactly when a node's state actually changes, to decide which channel endpoints must be re-evaluated — and is kept in the shared array for all nodes because its access pattern (sequential burst at writeback) differs from the descriptor's (random single-line probe).

4.1.4 Wave queues, the dedup hash, and the absence of bounds checks

The pending-event structure is a double-buffered pair of queues with a companion membership byte-array per buffer: `RecalcList/RecalcListNext` (32-bit ids, append-only within a wave) and `RecalcHash/RecalcHashNext` (one byte per node, 0/1). "Hash" is historical naming from the ancestor implementations [15][17]; in this engine it is a direct-indexed flag array, and its narrowing from `int` to `byte` (58 KB → 14 KB per buffer) was again a working-set decision. The flag serves three duties at once: O(1) deduplication on enqueue, the "still pending this wave" test at pop time, and — crucially for semantics — the mechanism by which a group walk *absorbs* the pending evaluations of every member it visits (Section 4.2.3). The group-walk scratch state follows the same pattern: `_groupBuf` (16-bit ids, 29 KB) doubles as both the result list and the BFS work queue, and `_inGroup` (one byte per node, 14 KB) is the walk's dedup flag, cleared sparsely — only the previous group's entries — between walks.

Bounds checks are absent from the hot loop not because they were suppressed but because the representation makes them inexpressible: every hot array is a raw pointer into unmanaged memory, so the runtime has no array object against which to check. The safety argument is structural rather than per-access. Indices originate either from the build pipeline (which validates them once, at composition time, against the node count) or from the lists themselves (which are zero-terminated, with node 0 reserved); ids are 16-bit by construction and the arrays are sized to the node count; the only deliberate out-of-sublist reads are the padded 64-bit dual-pair loads. We note plainly that this trades memory safety for speed within a closed, build-time validated index space — a defensible trade in a simulator kernel, and one whose residual risk the correctness infrastructure of Section 4.4 is designed to catch as divergence rather than corruption going unnoticed.

Table 4.1. The hot arrays of the engine at the lowered system size (~14.7K node ids including handler fake nodes). Sizes are as documented in the engineering notes; the per-half-cycle *active* set is far smaller (~15 KB, L1-resident — Chapter 8).

Array	Element	Approx. size	Role
`NodeStates`	byte	15 KB	logic value 0/1; the most-read array
`NodeInfos`	16-byte struct	235 KB	flags + inline channel payload or overflow indices
`TransistorList`	ushort	≤350 KB	flattened zero-terminated sublists (overflow channels; gate fanout for all nodes)
`NodeConnections`	int	58 KB	cold: capacitance proxy for the floating tie-break
`NodeTlistGates`	int	58 KB	cold: index of each node's gate-fanout sublist
`RecalcList` (×2)	int	58 KB each	double-buffered wave queues (current / next)
`RecalcHash` (×2)	byte	14 KB each	queue membership / pending flag per buffer
`_groupBuf`	ushort	29 KB	group member list ≡ BFS work queue
`_inGroup`	byte	14 KB	group-walk dedup flag (sparsely cleared)
`IsPureLogic`	byte	15 KB	dispatch class per node (cls0/cls1/cls2)
`FlagsToState`	byte	256 B	flags-OR → resolved value priority LUT

4.2 The Hot Loop

4.2.1 StepCycle and the settle loop

The unit of simulated time is the master-clock half-cycle (hc): one toggle of the board `clk` node. `StepCycle` flips the clock node's external-drive flag branchlessly (the clock always toggles, so no direction branch is needed), enqueues it, and calls `ProcessQueue` to propagate to quiescence:

```
procedure ProcessQueue():
    while |nextList| > 0:
        swap(curList, nextList); swap(curHash, nextHash)     // wave boundary
        for nn in curList, in append order:                  // drain one settle wave
            if curHash[nn] != 0:                             // not absorbed by an earlier group
                RecalcNode(nn)
                curHash[nn] := 0
    InvokeCallbacks()                                        // behavioral periphery, post-settle
```

The double buffer defines the *settle wave*: evaluations triggered by state changes in wave k are appended to the next buffer and run in wave k+1. Within a wave, however, the semantics are Gauss–Seidel, not Jacobi: `RecalcNode` writes resolved states immediately, so a node evaluated

later in the same wave observes the updated values of nodes evaluated earlier, and a group walk that absorbs a member cancels that member's still-pending evaluation in the *current* wave by clearing its flag. We emphasize — because it constrains every optimization in this monograph — that within-wave order is therefore **observable semantics**, not an implementation detail: the floating-group tie-break (Section 4.2.3) reads previous states, so two drain orders can legitimately produce different (both physically defensible) results. Bit-exactness is defined against this order. Any transformation that reorders pops, splits waves, or defers writebacks changes the model, not merely the schedule; the measured failures of wave restructuring are discussed with the negative results in Chapter 9.

On the reference workload a half-cycle settles in 12.06 waves on average (maximum observed 45), with ~418 evaluations per half-cycle (measured). Release builds run the settle loop with *no* iteration cap: the netlist's deepest observed settle is ~45 waves across two real workloads, an in-loop guard measured +2.77% (the cold diagnostic block inhibited code generation of the fully inlined loop), and a separate study (Chapter 9) established that under-settling is catastrophic rather than graceful — capping even the deepest 0.58% of settles diverges within a thousand half-cycles. Debug builds keep a 128-pass tripwire purely as a development aid.

4.2.2 Dispatch in RecalcNode

Every evaluation begins with a one-byte class load from `IsPureLogic`, populated at Reset (Section 4.3.3):

```
procedure RecalcNode(nn):
    cls := IsPureLogic[nn]
    if cls = 1: RecalcNodeFast(nn); return        // static singleton: no pass channels at all
    if cls = 2:                                   // dynamic-singleton candidate (R-1)
        scan nn's (gate, other) channel pairs:
            on the FIRST gate that is ON:
                attempt the inline pair proof (B1); on success resolve {nn, other} inline
                otherwise goto BFS
        RecalcNodeFast(nn); return                // all pass gates OFF => group is exactly {nn}
BFS:
    v := ComputeNodeGroup(nn)                     // full walk + flags-OR + LUT
    for each member m of the group: SetNodeState(m, v)
    if the OR-ed flags contain HasCallback:
        enqueue the callback of every member that has one
```

Class 1 (*static singleton*, 3,929 nodes, 26.7%) covers the classical NMOS gate output: a node with channels only to supply and no pass channel to any normal node. Its conducting group is provably {*nn*} for all time, so evaluation reduces to OR-ing the states of its ground-channel and power-channel gates into the flags word and one LUT lookup — `RecalcNodeFast`, a loop-free, branch-light resolution that reproduces the general path's result byte-for-byte, including the floating case (empty flags ⇒ hold previous value ⇒ no write at all). Class 2 (*dynamic-singleton candidate*, 10,784 nodes, 73.2%) covers nodes that *do* have pass channels but carry none of the excluded flags; for these the singleton property is tested at runtime by scanning the (usually inline) gate

list, and the scan doubles as the entry point of the B1 two-node inline resolution. Class 0 (~16 nodes, 0.4% of evaluations) is everything that must take the general walk unconditionally — callback targets and ForceCompute-flagged nodes. The dispatch and its measured economics are the subject of Chapter 6; here we record only its architectural shape: a data-driven three-way branch off a byte array, with the rare class resolved last.

4.2.3 The group walk and resolution

The general path is the literature's channel-connected-component evaluation [1][2][3], restricted to the *currently conducting* subgraph: a breadth-first walk from the seed across ON pass transistors, accumulating a flags-OR over members, followed by a 256-entry priority-table resolution.

```
procedure ComputeNodeGroup(seed):
    clear the PREVIOUS group's inGroup flags (sparse)
    groupFlags := 0;  group := [seed];  inGroup[seed] := 1;  curHash[seed] := 0
    groupFlags |= flags[seed]
    i := 0
    while i < |group|:                          // group doubles as the BFS queue
        nn := group[i];  i := i + 1
        for (gate, other) in channels(nn):      // inline payload, or 64-bit dual-pair
            if state[gate] = 1 and inGroup[other] = 0:      //   loads on overflow nodes
                inGroup[other] := 1;  append other to group
                curHash[other] := 0             // absorb other's pending evaluation
                groupFlags |= flags[other]
        if any ground-channel gate of nn is ON: groupFlags |= Gnd   (early break)
        if any power-channel gate of nn is ON:  groupFlags |= Pwr   (early break)
    if groupFlags != 0: return FlagsToState[groupFlags]
    // purely floating: strict largest-capacitance member's previous state, first seen wins
    return state of argmax over group of NodeConnections (strict >, insertion order)
```

Three mechanisms deserve comment. First, `_groupBuf` serving as both result list and BFS queue makes the queue free: there is no separate work-list allocation, push, or pop, only a read index chasing a write index. This is why an experiment replacing BFS with DFS measured −2.28% — the orders are semantically interchangeable here (the group is a set, and resolution is order-free over the OR), but DFS needs a stack the BFS gets for nothing. Second, the member-flag clear (`curHash[other] := 0`) is load-bearing semantics: a node absorbed into a group has been evaluated *as part of that group*, and letting its stale pending entry pop later in the same wave would re-evaluate it against post-write states. Third, the walk body is compiled as one unit: the engine hoists the static array pointers and the group count/flags into locals for the duration of the walk (measured +3.2%), and the whole chain — dispatch, walk, resolution, writeback — inlines into the wave-drain loop of `ProcessQueue`, which is deliberately *not* force-inlined into its many callers (measured −1.4% when it was).

Resolution itself is the flags-OR → LUT scheme inherited from the family [15][17]: the 256-entry table is built once from a pure priority function — ForceCompute groups containing both Gnd and Pwr cancel both; then Gnd > Pwr > SetHigh > SetLow > PullUp > hold — so the per-group cost is

one byte load. The purely-floating branch (empty OR) is the chip's storage mechanism, not an error case: a dynamic node isolated by an opening pass gate retains its value as trapped charge, modeled as "the strictly-largest-capacitance member's previous state, first seen winning ties," with capacitance proxied by total connection count. On this workload the floating branch is reached by under 1% of walks, which is why `NodeConnections` can be a cold array; but its semantics taint every optimization that touches evaluation order, since the value it returns depends on members' *previous* states. Measured distributions (Chapter 6): 77.1% of walks build a group of size 2, 16% size 3, 5.7% size 4 or more; BFS depth averages 1.13 with p99 = 3 and maximum 14. The walk machinery, in other words, exists for a conducting component that is nearly always tiny — the observation that motivates both the dispatch classes of Chapter 6 and the failure of every batching scheme in Chapter 9.

4.2.4 SetNodeState and the enqueue walks

State writeback is where events propagate, and where the prune family of Chapter 5 acts. The structure is a same-value early-out, the store, and then a walk over the node's gate-fanout list — the transistors this node gates — specialized by the new value, since a gate going high and a gate going low have different propagation obligations:

```
procedure SetNodeState(nn, v):
    if state[nn] = v: return                      // no event
    state[nn] := v
    for (c1, c2) in gateFanout(nn):               // read as 64-bit dual-pair loads
        if v = 0:                                 // channel turns OFF: ends may DISCONNECT;
            if c1 >= S and nextHash[c1] = 0: append c1     //   both endpoints need re-eval
            if c2 >= S and nextHash[c2] = 0: append c2     //   (S: turn-off skip boundary)
        else:                                     // channel turns ON: ends MERGE; one side
            if (c1 < A or c1 >= B                 //   suffices (the walk reaches the other)
                or state[c1] != state[c2])        //   P-1 same-state prune, unsafe-class
               and nextHash[c1] = 0: append c1    //   exempted by the A/B range compares
```

The asymmetry between the two legs is physical. A turning-on gate *merges* the channel's two ends into one conducting group, so enqueueing either endpoint suffices — the group walk will traverse the now-ON channel to the other — and the build pipeline normalizes any supply endpoint onto c2 so that c1 is always a legal enqueue target. A turning-off gate *disconnects*, and each end must independently re-resolve from whatever drivers remain on its side. The inequalities against the boundaries A, S, B are the range-compiled forms of the prune safety classes (turn-off skip ⇔ id < S; turn-on prune-unsafe ⇔ id < A or id ≥ B); their derivation, proof obligations, and verification are Chapter 7's subject — at the architectural level the relevant fact is that the hot loop's safety metadata has been compiled into the id space itself, so the walk reads *no* per-node mask array at all. The fanout list is consumed two (c1, c2) pairs at a time via a single 64-bit load (measured +~1.2%): unlike the random-access gathers of the group walk, whose instruction overhead hides under memory-latency stalls, this sequential walk is throughput-sensitive, and halving its loop branches and load count pays. The same dual-pair trick is applied inside the group walk only on

the long (overflow-node) adjacency scans, where it measured +1.4% — and was rejected on short walks, where it lost; the lesson that identical micro-optimizations sign-flip with context recurs throughout Part III.

4.3 The Composition Pipeline

4.3.1 Parsing and module composition

The input artifacts are the Visual 6502-family netlists [15][16]: `segdefs` (silicon polygons, of which the engine consumes the per-node pull-up annotation), `transdefs` (NMOS transistors as gate/c1/c2 node triples, with a weak-device column that is uniformly false in the 2A03/2C02 datasets — depletion loads are encoded as pull-up segments instead), and `nodenames` (name → node id, including the 2A03's internal registers). Around the two dies, the system is assembled MetalNES-style [17] from module definition files: a module declares pins, sub-module instances, pull-up lists, ForceCompute lists, behavioral memory regions, and *connections* — and the composition rule for a connection is the engine's most elegant simplification: **a connection is an always-on transistor**, a device whose gate is the power node. Instantiation allocates a fresh node-id block per instance, resolves pin and wildcard name references across module boundaries, and emits one flat transistor list for the entire board: the 2A03, the 2C02, the board glue, and the cartridge wiring as one uniform switch-level graph, with no special-cased inter-chip communication.

4.3.2 Lowering

The flat netlist then passes through a behavior-preserving lowering, a deliberately minimal analogue of the netlist preprocessing tradition [2]. Three transformations run, all justified by construction. (1) *Static-group merge*: two normal nodes joined only by an always-on, non-weak device are the same electrical node forever; a union-find collapses each such class onto one representative. The merged node's capacitance proxy is set to the *maximum* over the class — exactly the value the floating tie-break would have selected among the class members had they remained distinct — so even that corner case is bit-identical to the un-lowered model. (2) *Dead-device removal*: a transistor whose gate is ground can never conduct and is dropped. (3) *Compaction*: survivors are renumbered densely (ids 0–2 reserved for the sentinel and the two supplies), self-loops left behind by consumed connections are dropped, and duplicate (gate, c1, c2) triples are de-duplicated. On the composed NES the merge collapses 441 nodes and removes 530 transistors, yielding the ~14.7K-node / ~26.8K-transistor system simulated everywhere in this monograph. Lowering also *defines* the golden node numbering: every checksum in Section 4.4 is stated in this id space, and comparisons against the un-lowered model (a diagnostic A/B retained behind a flag, measured −3.7%) are made through a mapped checksum computed in the old id space.

4.3.3 Classification passes, renumbering, and Reset

After handlers are attached (Section 4.5) — which must precede `Reset()`, because callbacks add fake nodes and transistors and Reset sizes every hot array to the final node count — `Reset()` performs the build-to-runtime handoff: it allocates the SoA arrays, builds the 256-entry LUT from the pure priority function, walks the managed node graph once to emit the inline payloads and the flattened sublists (bucketing each node's channels into normal/ground/power as it goes), sets power-on state (pull-up nodes start high, everything else low/floating), stamps the supply and ForceCompute flags, and finally runs the classification passes: the dispatch classifier (cls0/1/2), the prune-safety taint pass (a union-find over channel components for ForceCompute taint, plus the no-pull-up taint), and the turn-off-skip pass, which also performs the range verification described in Section 4.4.4.

The full production load is a *three-pass* version of this sequence: pass 0 composes and classifies under identity ids to capture each node's prune class; pass 1 rebuilds with a class-major permutation and a temporary structural ordering key, warms the chip past the reset transient (1,024 hc), and captures the true first-evaluation order of the production cascade over 32,768 hc through a cold instrumented copy of the settle loop; pass 2 rebuilds finally with the class blocks and the captured locality key. The whole load costs ~1.3 s and is re-derived from the actual chip on every load — there is no profile file to go stale. The design and measured economics of this self-captured layout are treated in their own chapter; what matters here is the correctness obligation it creates, addressed next: the engine must be *provably indifferent* to its own node numbering. Power-on itself ends with a sweep that enqueues every node and settles the whole chip once, followed by a faithful reset sequence — assert the board reset line, run 192 half-cycles, deassert — rather than any shortcut initialization.

4.4 Correctness as Infrastructure

The engine's optimization campaign accepted only transformations that left the simulation bit-identical. That policy is only meaningful if "bit-identical" is cheap to test, hard to fool, and diagnostic when it fails. This section describes the apparatus; we regard it as a contribution of equal standing with the speedups it gated, because every dead end in Chapter 9 was identified — and several latent soundness holes in plausible-looking optimizations were caught — by this machinery rather than by inspection.

4.4.1 Golden checksums and the SMB1 gate

The primary gate is an FNV-1a 64-bit hash over the complete `NodeStates` array — a full-state fingerprint, not a sampled trace, so two runs that match at the same time index are bit-identical in every one of the ~14.7K node values. Three golden values anchor the reference workload at increasing depths (300k, 400k, and 1M half-cycles: `0x794A43ABDF169ADA`, `0x9174E19D961CB6E5`, `0x6D4CCBCE2E9CD599`), and every benchmark run in this monograph — all 72 runs of the ablation ladder included — is checksum-gated as a matter of course.

A single workload, however measured deeply, is a survivorship risk: a transformation can be unsound precisely on circuitry the workload leaves quiet. This is not hypothetical. The adversarial review of one abandoned optimization (the maintained-fact bypass, Chapter 9) found latent holes that the reference workload could never expose because the sprite OAM bus is essentially idle in it; its goldens were survivorship. The response is a fourth gate: a 10-million-half-cycle run of a sprite-heavy commercial title (Super Mario Bros.), which exercises the OAM and sprite-evaluation datapaths — precisely the dynamic-storage circuitry on the floating tie-break — for minutes of simulated time. The combination is deliberately heterogeneous: a synthetic full-screen test ROM measured deeply plus a real program measured long.

4.4.2 Layout independence by construction

The self-captured renumbering of Section 4.3.3 permutes the engine's entire id space on every load. For the golden checksums to remain meaningful — and for a renumbered run to be comparable with the identity-numbered history at all — the simulation must be invariant under the permutation. The engine achieves this by construction rather than by hope:

> **Property 4.1.** The simulated event sequence and all golden checksums are invariant under any node-id permutation that fixes the reserved ids, because (i) the only id-order-dependent computation in the engine — the power-on sweep that enqueues every node before the first settle, whose order feeds the floating hold-previous tie-break — iterates in *original* id order through the permutation; and (ii) the checksum likewise hashes `NodeStates` in original id order through the permutation. All other orders in the engine (wave append order, group-walk order, fanout-list order) are induced by the structural lists, which are rebuilt with their relative order preserved.

This was audited, not assumed: the power-on sweep is documented in the source as the single id-order-dependent site, and the property is exercised continuously, since every production run is a renumbered run checked against checksums recorded under the identity numbering. The same discipline yields the *mapped checksum* rule for transformations that legitimately change the node set (such as lowering): the gate becomes the checksum computed in the old id space through the transformation's mapping, compared against an untransformed reference — bit-exactness of the surviving state, with the re-baselining made explicit instead of silently accepted.

4.4.3 Per-node dump-diff

A checksum is a one-bit oracle; diagnosis needs coordinates. The complementary tool dumps the full per-node state vector (in original id order, with names attached from `nodenames`) at a chosen time and diffs two runs node by node. Its decisive use was the post-mortem of the first same-state prune attempt: the naive prune passed short runs and then diverged into a black screen, and the dump-diff localized the first divergent nodes to OAM and palette RAM cells — no-pull-up dynamic storage whose values flow through the floating tie-break — overturning an initially plausible but wrong theory (ForceCompute interaction) and directly inducing the structural safety

taint that made the prune sound. The general pattern deserves stating: in this methodology, a failed checksum plus a per-node diff plus named netlist nodes converts a heisenbug into a circuit-level explanation, usually within one session.

4.4.4 Verified speculation: the reset-time range check

One piece of correctness machinery runs inside the engine itself. The range-compiled prune tests of Section 4.2.4 are speculation in the technical sense: the hot loop assumes the id space is laid out so that each safety class is one contiguous block. The classification passes therefore recompute the full per-node safety mask from first principles at *every* Reset, as ground truth, and verify every node's mask bits against the bits its id range implies. On any mismatch the engine does not trust the layout: it falls back to safe-degenerate boundaries under which every prune is disabled (and supply filtering still holds), producing a correct, roughly 2× slower simulation plus a loud warning. This is the deoptimization-guard pattern [13] transplanted from runtime compilation to data layout: the optimized form is used only while a machine-checked invariant holds, and the failure mode is performance, never wrongness. In Release builds the verified mask is then freed — the hot path reads only the ranges — making the check literally free at runtime.

4.4.5 Measure-then-keep

The final layer is methodological. Every candidate optimization passes two gates in order: bit-exactness (all checksum gates), then a measured win under interleaved-paired A/B — both binaries prebuilt, alternated base/experiment within each round, medians and paired win counts reported. The interleaving exists because batched A/B (all runs of A, then all of B) was demonstrated untrustworthy: thermal and clock drift across a session biases batched comparisons enough to flip the sign of sub-3% effects, and at least one early "dead end" verdict was later overturned when re-measured paired. Measurements are taken with the clock unlocked; variance is controlled by same-day multi-round round-robin interleaving, a single benchmark process, and median reporting. The cultural consequence is worth recording: with the gates this cheap, the project's default answer to "would X help?" became "build the smallest bit-exact version and measure," and the negative-results catalog of Chapter 9 is the direct product — every entry in it is a measured, checksum-gated rejection rather than an opinion.

4.5 The Behavioral Periphery

4.5.1 Callbacks as fake transistors

The engine needs to observe the chip (to detect frame boundaries, address strobes, test-ROM completion) and to attach behavioral components (memories, video output) at chip pins. Both needs are served by one mechanism inherited from MetalNES [17]: a *callback* is registered by allocating a fresh fake node and adding, for each watched node, a fake transistor whose gate is the watched node, whose c1 is the fake target, and whose c2 is ground. Whenever any watched node changes state, the ordinary propagation machinery pulls the target node into a group and its callback-flagged evaluation enqueues the handler — the watch list is implemented *by* the

simulator rather than beside it, with no per-event polling and no special dispatch in the hot loop beyond the existing HasCallback flag. Handlers run only after the settle reaches quiescence, and re-entrantly: a handler that drives pins triggers a nested settle whose own callbacks drain in turn. This design is why handlers must be attached before Reset — they genuinely extend the netlist — and why the dispatch class cls0 exists.

4.5.2 Memory, clock, and video handlers

Three handler families constitute the entire behavioral surface. The *clock* "handler" is the degenerate case: the board crystal is not simulated; `StepCycle` toggles the resolved `clk` node directly. The *memory* handlers serve the commodity storage parts: the board's two 2 KB SRAMs (CPU work RAM and the PPU's nametable VRAM), the cartridge PRG and CHR ROMs (CHR RAM when the cartridge has none), and an optional battery-RAM region used by test ROMs to report results. A module declares a memory region declaratively; the handler watches the chip-select, write-enable, and address pins, performs a plain array access against an unmanaged byte buffer, and drives the data pins through the ordinary external-drive flags (SetHigh/SetLow) followed by a settle — so from the dies' perspective, a behavioral RAM is indistinguishable from a very fast real one. The *video* handler watches the PPU pixel clock, and on each rising edge reads the horizontal/vertical position counters and the palette index off named internal nodes, translating them through palette RAM into one ARGB framebuffer write. Bit-vector access to named node groups goes through small gather/scatter helpers (`ReadBits` / `WriteBits`) that read the state array branchlessly.

4.5.3 The scope rule

The boundary between switch-level and behavioral is a stated invariant of the project, not an accident of convenience: **everything inside the 2A03 and 2C02 dies is transistor-level, always.** Behavioral components exist only at chip boundaries and replace only parts that were never on the dies — discrete memories, the crystal, the display. No internal register, no OAM cell, no datapath is ever replaced by a hand-written model in this engine; abstraction experiments of that kind were conducted, deliberately, in a separate fork (the S2 investigation, Chapter 9) whose conclusions never modified S1. The rule has teeth in the safety machinery: the turn-off-skip classifier must explicitly exclude the handler-driven data-bus pins from its isolated-leaf class — a node driven by SetHigh/SetLow at the moment its channel opens does not float-hold, and the early divergence that taught this lesson is part of the prune-safety record — so the periphery is not merely scoped but actively accounted for in every soundness argument.

> The model's fidelity claims should be read precisely. The engine is bit-exact with respect to the inherited switch-level semantics [1][15][17] — a discrete, unit-delay-free, two-state-with-charge approximation in which settle order stands in for analog timing and a connection-count proxy stands in for capacitance. It is not a circuit simulator: propagation delays, drive strengths beyond the priority classes, and analog levels are outside the model. Correctness statements throughout this monograph are statements of equivalence to this model and to its ancestor implementations, validated end-to-end by running real software; they are not claims of analog accuracy. Within that scope, the periphery adds one further caveat: behavioral memories respond within a single settle, faster than any real SRAM, which is faithful to the family's established practice [17] and validated by the same end-to-end gates.

The architecture described here is, in summary, a small number of load-bearing decisions — SoA unmanaged layout with access-pattern field splitting, an inline-payload node record, a double-buffered Gauss–Seidel settle whose order is semantics, a composition pipeline that ends in classification and a verified renumbering, and a correctness apparatus that makes bit-exactness a cheap mechanical gate. The next three chapters quantify what was built on top of it: the measured anatomy of the workload and the prune family with its proofs (Chapter 5), cardinality-specialized dispatch (Chapter 6), and self-captured data relayout (Chapter 7); the catalog of measured dead ends follows in Chapter 9. All source and the benchmark package are public [21].

Chapter 5. The Prune Family: Provably-Null Event Suppression at the Source

The previous chapter established the engine's structural facts: a working set small enough to be L1-resident, an event loop whose cost is dominated by chains of dependent loads, and a correctness contract — bit-exactness against full-state checksums — that forbids any transformation whose effect on the output cannot be proven or exhaustively verified. This chapter develops the first of the three optimization families built on that foundation: a set of four *prunes*, P-1 through P-4, that suppress event propagation at the moment of enqueue whenever the suppressed re-evaluation is provably a no-op. The family's premise is simple and, on this engine, measured rather than assumed: the dominant cost of event-driven switch-level simulation is not that each re-evaluation is slow, but that most re-evaluations should never have happened. A re-evaluation whose result equals the value already stored is pure waste — it performs the same random gathers, occupies the same queue slots, and stalls on the same cache lines as a productive one, and then changes nothing. Because each individual stall is already near the memory-latency floor (Chapter 4), the only lever that remains is to do *fewer* of them, exactly, without moving a single output bit.

The chapter is organized as an argument from measurement to mechanism to proof. Section 5.1 quantifies the opportunity and dissects the anatomy of wasted work. Section 5.2 presents P-1, the same-state turn-on prune — deliberately beginning with its failed precursor, because the failure is what taught us where the danger lives. Section 5.3 presents P-2, the turn-off isolation prune, whose suppression happens before the queue and therefore deletes the entire enqueue-pop-resolve chain. Section 5.4 presents P-3 and P-4, the capacitance-dominance un-taint, which we regard as the strongest original claim in this work. Section 5.5 describes the zero-configuration classification pass that makes all four prunes safe without a single hand-listed node. Section 5.6 reports the measured effect, and Section 5.7 delimits, with three independent confirmations, what the prunes cannot reach.

5.1 The Opportunity: Eighty Percent of the Work Changes Nothing

5.1.1 Measuring wasted re-evaluation

The engine settles each half-cycle by draining a double-buffered event list: dirty nodes are popped, each pop dispatches to a resolution path (the O(1) singleton fast paths or the general group walk of Chapter 4), and the resolved value, if changed, propagates new enqueues through the node's gate fanout. We instrumented the pop loop with a DEBUG-only profiler that classifies every pop by whether *any* state actually changed as a result; the event counts are identical in Release builds, so the profile describes the production engine. The headline numbers are stark. On the current engine, a 100,000-half-cycle run of `full_palette.nes` performs roughly 42.85 million pops, of which **80.1% recompute to the value already stored** (measured). Before the prune family landed, the same profiler over 300,000 half-cycles counted 159.6 million pops with

84.4% producing no change, and roughly 88% of that waste concentrated on dynamic-singleton (cls2) nodes — the 10,784 nodes (73.2% of the population) that have pass channels but whose channels are usually all off (measured).

This redundancy is not a defect of the implementation; it is inherent to level-sensitive, event-driven propagation in the selective-trace tradition [4]. A node is re-queued whenever the gate of any adjacent transistor flips, because the flip *may* change the node's conducting group and therefore its resolved value. But usually it does not. A node held low through three parallel pull-down paths does not change when one of them opens; a pull-up node whose single pull-down was already off does not change when a second, redundant pull-down also turns off; a floating storage cell does not change when an unrelated access transistor two channels away toggles. The event system cannot know this at enqueue time — unless we tell it, with a check cheap enough to beat the work it avoids and a safety argument strong enough to preserve bit-exactness.

5.1.2 The anatomy of a no-change pop

The profiler further classifies the no-change pops by the structural class of the node and the resolution branch taken. Table 5.1 gives the breakdown on the current engine (after the prunes; the categories are defined identically before and after).

Table 5.1. Categories of no-change pops, current engine, `full_palette.nes`, per 100,000 half-cycles (measured; DEBUG profiler, event counts identical to Release). Shares are of *all* pops (~42.85M), of which 80.1% are no-change.

Category	Share of all pops	What it is physically
PullUp	42.0%	A depletion-load (pull-up) node re-evaluated after a gate flip on one of its pull-down paths, while another conducting path — or the absence of any — leaves the resolved value unchanged. The re-evaluation must scan the node's ground-channel gate list to discover this.
Supply	38.1%	The analogous case for nodes with direct channels to V_{CC} or ground: one supply path toggles but the OR over the remaining paths, and hence the LUT input, is unchanged.
FloatSingle	7.5%	A driverless node re-evaluated while isolated: its group is itself alone, the flags-OR is empty, and the floating resolution holds the previous state by definition.
FloatMulti	12.1%	A multi-node purely-floating group whose largest-capacitance member already holds the value the tie-break selects — charge sharing that re-derives the stored bit.

The four categories are not equally attackable. The two floating categories are the natural prey of static proofs: whether a node *can* be driven, and whether it *can* win a tie-break, are properties of the netlist, not of the run. The PullUp and Supply categories are the opposite: deciding that a pull-up node will not change requires knowing the live states of its other pull-down gates, which is

exactly the scan the re-evaluation itself performs. This asymmetry shapes the whole chapter: P-1 through P-4 harvest the floating categories (and the enqueue-side redundancy feeding all categories), while Section 5.7 shows — with three independent confirmations — that the PullUp/Supply residue has no static subset and resists even dynamic attack at a profit.

5.1.3 Why suppression must happen at the source

A no-change pop costs far more than its arithmetic. The chain is: an enqueue (a dedup-hash probe and a list append at the producer), a pop (a load from the wave list), a dispatch (a class test), a resolution (at minimum the node's `NodeInfo` line and a scan of its supply gate lists; at worst a group walk over ON channels with its per-member random gathers), and finally the discovery that nothing changed. Measured during the P-5z study (Chapter 9), even the cheapest pop classes cost on the order of 30 cycles, nearly all of it memory latency. Suppressing the event *after* the pop — a cheaper resolution for a wasted event — can at best shave the tail of this chain. Suppressing it at the enqueue site deletes the chain whole: no queue traffic, no dedup-hash pollution, no dispatch, no gathers, and, transitively, no lengthening of the settle wave (the engine averages 12.06 waves per half-cycle; every spurious member of a wave is also a spurious iteration of the drain loop). All four prunes therefore live in `SetNodeState`'s gate-fanout walk — the single site where a state change fans out into future events — and nowhere in the pop path at all.

The constraint that disciplines everything that follows is bit-exactness. As Chapter 4 established, the purely-floating tie-break is not a numerical detail: it *is* the chip's storage mechanism. The 2C02's object attribute memory and palette RAM, and a large population of dynamic nodes in the 2A03, hold their bits as charge whose value is recovered, on every access, by exactly the largest-capacitance-member rule. A prune that mispredicts "no change" on one such node does not cause a small error; it drops a stored bit, and the divergence compounds without bound. Every prune in this family is therefore gated on a per-node *safety proof*, computed statically at load, and every shipped variant was verified against the full-state golden checksums at multiple horizons plus the 10-million-half-cycle sprite-heavy gate.

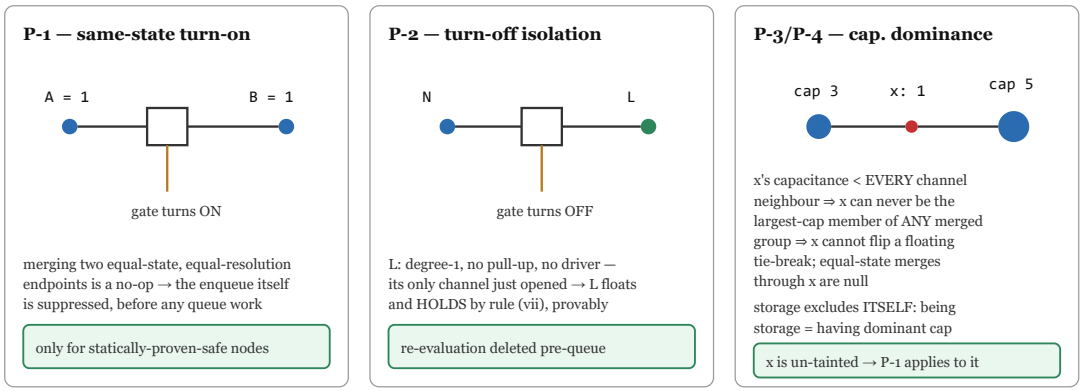

Figure 5.1. The structural proofs behind the three prunes. Left: P-1, the same-state turn-on prune (Section 5.2) — merging two endpoints already equal in state is a no-op; centre: P-2, the turn-off isolation prune (Section 5.3) — a degree-1, driverless leaf must float and hold once its only channel opens; right: P-3/P-4, the capacitance-dominance un-taint (Section 5.4) — a node whose capacitance is strictly below every channel neighbour can never win a floating tie-break. All three are load-time structural proofs; the hot loop tests only a precomputed bit.

5.2 P-1: The Same-State Turn-On Prune

P-1 targets the turn-on case. When a node's resolved value rises, every pass transistor it gates turns ON, and each such channel may merge the conducting groups at its two endpoints. The engine enqueues one endpoint per channel (the walk seeded there traverses the now-ON channel to the other). The observation behind P-1 is that if the two endpoints already hold the same value, the merge of two equal-valued groups usually cannot produce anything new: re-resolving the merged group writes the value its members already hold. The idea — do not schedule a successor whose value provably cannot change — is a standard refinement of selective trace [4] and appears, in equipotential-bypass form, throughout the switch-level literature [3]. What this section contributes is not the idea but the *safety classification* that makes it bit-exact on a Bryant-style charge model [1], and we tell the story in the order it actually happened, because the failure is the load-bearing part.

5.2.1 The precursor: prune-merge and the black screen

The first attempt, roughly a year into the project's optimization branch, was a runtime scheme called *prune-merge*: maintain a group-id per node, and when a transistor turns on between two nodes whose groups hold the same digital value, skip the merge re-evaluation outright — a plain digital-equality test with no further conditions. On the throughput benchmark and the CPU-side test ROMs it appeared spectacular: roughly 1.47× the contemporary baseline, with the CPU executing instructions correctly. It was also wrong. `full_palette.nes`, a PPU test whose reference pattern — a grid of all 64 colors — completes by frame 48, rendered a black screen. The divergence had entered precisely through the floating tie-break: equal-state merges on driverless storage and bus nodes are *not* generally null, because the merged group's resolution is not a

function of the two endpoint values alone. A charge-sharing group can resolve to the previous state of a third member — the largest-capacitance one — and the endpoint-only equality test cannot see it. The stored bits of the PPU quietly rotted, and the picture disappeared.

> **What the black screen taught us about verification.** The unsafe prune-merge passed every CPU-side check we then had. The lesson — that "observably identical on the CPU" and even "passes the instruction tests" are far weaker than bit-exact — is what hardened the project's verification discipline into the form described in Chapter 4: full-state FNV-1a checksums at multiple horizons as the primary gate, at least one visual ROM rendered to a byte-compared image, and, later, a 10-million-half-cycle sprite-heavy gate. Every subsequent prune in this chapter was developed *under* that gate. We report the episode rather than burying it because the methodology is a result: in this domain, a speedup number without a checksum is not data.

5.2.2 The runtime repair that cost more than it saved

The first repair kept the runtime machinery and fixed the semantics: maintain topology-aware group ids, ratified on every walk (a fresh id allocated per resolution, assigned to every member), so that the equality test applies only where the group structure makes it sound. This version was correct — bit-identical checksums, and a 50-frame `full_palette` screenshot whose PNG was byte-identical to the reference, compression artifacts included — but the economics had inverted. The per-walk maintenance (id allocation plus a store per group member, on every one of the millions of walks per second) is paid always; the skip pays off only on the subset of turn-ons that qualify. At the time of the fix the net was a wash (about 1.02×); after the engine's later inlining and layout work made the group walk itself cheap, the maintained ids became a strict liability, measuring **−4.4%** (measured) — slower than not pruning at all. The flag was retired.

The episode generalizes, and the generalization recurs throughout this monograph: on this engine, *maintained runtime facts do not pay*. Any scheme that updates auxiliary state on the hot path so that some later check becomes cheap must amortize its maintenance over events that are individually tens of nanoseconds; the measured maintenance floor of the most ambitious such scheme (P-5, Chapter 9) was around 7% of total runtime — more than the prize. The prune family's eventual form is the structural answer: move every fact that can be moved to load time, where its cost is paid once, and let the hot path consult it for free.

5.2.3 The static rebirth

P-1 in its shipped form (June 2026) asks a different question: not "is this particular merge null?" — a runtime question requiring runtime facts — but "is this *node* one on which an equal-state merge is null *for every group it can ever join*?" — a static question about the netlist, answered once at load and recorded as one bit per node. The hot-path test then collapses to: on turn-on of a channel (c1, c2), skip enqueueing c1 iff c1 is classified safe and `states[c1] == states[c2]`. Two loads and a compare, against live state the enqueue walk has already paid to touch.

Even the static form had to find its safety condition the hard way. The naive static prune — skip on equality everywhere — diverged again (again a black screen). The first theory blamed the ForceCompute nodes (the precharged-bus model in which ground and power contributions cancel; Chapter 4), and was wrong, or rather, incomplete. Per-node state-dump diffing against the unpruned reference localized the earliest divergences to *no-pull-up* nodes in the PPU's object attribute memory and palette RAM — dynamic storage living on the hold-previous tie-break, exactly the mechanism that had killed prune-merge. The shipped classification taints both populations, and the result was **+11.85% (14/14 paired rounds)** at ship time, bit-exact on every gate (measured) — at that date the project's largest single win after the R-1 dispatch path (Chapter 6).

5.2.4 The safety-taint taxonomy

The load-time classifier marks a node *turn-on-unsafe* (it must always be enqueued, equality notwithstanding) if either of two structural conditions holds:

- **No pull-up.** A node without a pull-up can belong to a purely-floating group — one whose flags-OR is empty — whose resolution is the capacitance tie-break over member previous states. For such groups the endpoint-equality test proves nothing (Section 5.2.1). Conversely, a node *with* a pull-up can never be in a purely-floating group at all: its own flag makes the OR non-empty, forcing resolution through the priority LUT. Tainting every no-pull-up node therefore confines the prune to LUT-resolved merges by construction. This single test is what separates combinational logic from charge storage — OAM cells, palette RAM, the dynamic nodes of the 2A03 — without naming any of them.
- **ForceCompute channel component.** The LUT's monotonicity (Section 5.2.5) fails in the presence of the ForceCompute cancellation rule, under which simultaneous ground and power contributions cancel and resolution falls through to weaker flags. A group's resolution is non-monotone if *any* member carries the flag, so the taint must cover every node that could ever share a group with a ForceCompute node. The classifier computes this with a union-find over the channel graph (the c1–c2 edges, i.e., the maximal possible conducting groups, all channels assumed ON) and taints every node whose component contains a ForceCompute member. A node in a ForceCompute-free component can never meet the cancellation rule, no matter which transistors conduct.

Both conditions are deliberately conservative: they over-approximate the dangerous set using only reachability and flags, never simulation. Section 5.4 shows that the first condition is *too* conservative — a provable subset of the no-pull-up population can be re-admitted — and that recovering it is worth more than P-2 and P-4 combined.

One implementation obligation deserves a sentence, because it is the kind of detail on which bit-exactness quietly dies: a pruned node must not have its dedup-hash bit set. The hash marks "already queued this wave"; setting it without appending the node would make the node look queued while never being popped, and a later, *legitimate* enqueue of the same node in the same wave would be silently dropped. The prune therefore guards the enqueue, not the hash update alone — the skip leaves no trace whatsoever.

5.2.5 Soundness

> **Property 5.1 (equal-state merge under driven resolution).** Let t be a pass transistor with channel endpoints a and b whose gate turns ON, and suppose (i) a carries a pull-up, (ii) no node in a's channel component carries ForceCompute, and (iii) `states[a] = states[b] = v` at flip time, with the endpoint groups at quiescence. Then re-resolving the merged group writes v to every member, changes no state, and generates no further events; suppressing the seed is unobservable.

The argument runs through the priority structure of the resolution LUT. At quiescence, every member of a conducting group holds the group's resolved value, so the endpoint states stand for their groups' values. The merged group's flag set is the OR of the two constituent flag sets (the gate flip changes connectivity, not member flags). By condition (i) the merged OR is non-empty — a's pull-up guarantees it — so resolution goes through the LUT; by condition (ii) the cancellation rule cannot fire. Absent cancellation, the LUT is a pure priority function: its output is determined by the single highest-priority flag present (Gnd > Pwr > SetHigh > SetLow > PullUp). The highest-priority flag of an OR of two sets is the higher of the two sets' maxima, hence belongs to one of them; that set's group resolved to v, and since the output depends only on that maximal flag, the merged group also resolves to v. Every member already holds v, so the walk's writes are all no-ops and its follow-on enqueues (which fire only on actual state change) are empty. The suppressed seed could have produced nothing.

Two honest qualifications. First, the quiescence premise in (iii) is an idealization: the prune fires mid-wave, under Gauss-Seidel order, where transient member states exist. The classification removes the cases where mid-wave transients can interact with resolution non-monotonically (floating groups, cancellation); for the remaining LUT-resolved cases the endpoint states the prune reads are exactly the states the suppressed walk would have read, so prune decision and hypothetical walk see one consistent world. Second, we do not claim the argument as a machine-checked proof. It is a proof sketch whose residual gaps are closed by the strongest empirical gate we can construct: full-state checksums at 300k, 400k, and 1M half-cycles, and a 10M-half-cycle sprite gate. That combination — structural argument plus exhaustive bit-level verification — is the standard every member of the family meets.

5.3 P-2: The Turn-Off Isolation Prune

5.3.1 Mechanism and proof

P-2 targets the opposite edge. When a node's value falls, every channel it gates turns OFF, and both endpoints of each channel must normally be re-evaluated, because disconnection can split a group and either fragment may resolve differently. But there is a class of nodes for which the outcome of that re-evaluation is decided at load time. Consider a node with exactly one pass channel, no channel to either supply, no pull-up, no ForceCompute flag, no callback, and no external drive. The instant its single channel opens, such a node is an island: a conducting group

of one, with an empty flags-OR. The resolution of a purely-floating singleton is, by definition, its own previous state — the hold that implements dynamic storage — so the re-evaluation is a guaranteed no-op, every time, regardless of anything else in the machine.

> **Property 5.2 (isolation hold).** Let x be a node with exactly one pass channel, no supply channels, and none of the pull-up, external-drive, ForceCompute, or callback attributes. When the gate of x's channel falls, x's conducting group becomes {x} with an empty flag set; its resolution is its previous state. Re-evaluating x on this event is a no-op, and the static attributes above are decidable at load.

The proof is immediate from the resolution semantics: group membership is the ON-channel reachability set, which for x with its only channel off is {x}; the flags-OR over {x} is empty by the attribute conditions; and the floating rule for a singleton returns `states[x]`. The interesting part is not the proof but where the test is applied: *before the queue*. The turn-off enqueue walk consults one mask bit (in the shipped engine, one register compare — Chapter 7) and simply does not enqueue x. Following Section 5.1.3, this deletes the full enqueue-pop-resolve chain, not merely the resolution; on the pre-P-2 engine the no-change pops attributable to this class measured roughly a quarter of *all* pops, the single largest identifiable block of wasted work. The classifier finds **1,272** such nodes (measured) — passive pass-transistor leaves: the far ends of access transistors, transfer-gate intermediates, and similar driverless appendages of the netlist.

5.3.2 The driven-pin correction

The first cut of the static condition was wrong in an instructive way. The behavioral memory handlers (Chapter 4) drive the RAM/ROM data-bus pins by setting external-drive flags from callbacks — physics the netlist alone cannot see. A pin that is structurally a single-channel driverless leaf is *not* float-held if a handler drives it: once isolated, it resolves to the driven value, and skipping its turn-off re-evaluation freezes a stale bit. The divergence was caught by the checksum gate within 20,000 half-cycles, and per-node diffing identified the culprits as exactly three ROM data pins. The fix is structural, not a patch: the classifier excludes every node appearing in any handler's declared output-pin set, making the handlers' drive capability part of the static model. We report this because it sharpens the methodology claim of Section 5.5: "zero-configuration" means derived from the composed system's structure — netlist *plus* handler interface — not from the netlist alone.

5.3.3 Standing and attribution

The hold-on-isolation behavior itself is bedrock switch-level semantics — it is Bryant's charge retention [1], and IRSIM's incremental evaluator arrives at the same held value when it evaluates such a node [3]. What we have not found in the literature, and claim as a secondary original contribution of this work, is the *lifting* of that runtime evaluation result into a load-time static proof attached to the node, consumed as a pre-queue suppression — the event is never scheduled, rather than scheduled and resolved to a hold. The distinction is between evaluating cheaply and

not evaluating; on a latency-bound engine the second is qualitatively stronger. The shipped gain at its commit was **+1.4%** (interleaved-paired, bit-exact) (measured) — modest alone, but P-2's condition set and mask infrastructure are what P-3 and P-4 build on, and the family's consolidated effect (Section 5.6) is what justifies the shared machinery.

5.4 P-3/P-4: The Capacitance-Dominance Un-Taint

5.4.1 Two nodes that forced a theorem

P-1's no-pull-up taint is sound but blunt: it sacrifices every dynamic node to protect the storage cells among them. The question P-3 asks is whether some no-pull-up nodes can be proven harmless anyway. The empirical trail begins with a failure, in the now-familiar pattern: an early attempt to relax the taint without a guard diverged on exactly *two* nodes in the whole machine — charge-share nodes in the 2A03's ALU/status-register region. Tracing those two nodes through the state dumps showed precisely how an equal-state merge can lie: the merged group went purely floating, and the tie-break elected a member that was neither endpoint, whose previous state differed. The two nodes were heavy — locally large capacitance, which is *why* they won tie-breaks and *why* they functioned as storage. That observation, inverted, is the theorem: a node that can never win a tie-break can never lie.

5.4.2 The dominance property

> **Property 5.3 (capacitance dominance implies tie-break immunity).** Let x be a node with cap(x) < cap(m) for *every* channel neighbour m of x, where cap is the model's capacitance measure and the comparison is strict. Then x is never the largest-capacitance member of any conducting group containing x of size at least two; consequently x never determines the value of any purely-floating resolution. If x additionally carries no resolution flags (no pull-up, no external drive, no ForceCompute) and no supply channels, then x's membership in any group is value-inert — it contributes nothing to the flags-OR and cannot decide the floating tie-break — and suppressing an equal-state turn-on enqueue at x is null.

The proof sketch has two legs. The *membership* leg: any conducting group containing x and at least one other node reaches x through one of x's pass channels, so the group contains at least one channel neighbour of x; by hypothesis that neighbour's capacitance strictly exceeds x's, so x is not the group's maximum. The strictness matters: the engine breaks capacitance ties by first-seen order in the group walk, which depends on BFS seeding and is not statically analyzable, so a node merely *tied* with a neighbour could still win under some walk order. The classifier therefore demands strict dominance by every neighbour, conceding any node with an equal-capacitance neighbour to the taint. The *contribution* leg: a driven group's LUT input is the OR of member flags plus the supply contributions discovered through members' ON supply channels; a node with no flags and no supply channels adds nothing to either, so the merged group's resolved value — driven or floating — is what it would have been without consulting x at all. Combined with P-1's

endpoint equality test, which guarantees x's own state already equals the other endpoint's (blocking the case where x bridges two groups of different value and must itself be rewritten), the suppressed re-evaluation can change neither the group's value nor x's. P-3 applies this to single-channel nodes; P-4 is the same statement generalized to multi-channel nodes, where dominance must hold against *all* neighbours simultaneously.

One caveat belongs in the property's fine print rather than a footnote. The model's "capacitance" is the static connection count inherited from the Visual 6502 family of simulators [15][17] — a topological proxy, not an extracted physical capacitance. This does not weaken the proof: the floating tie-break that defines the engine's (and its ancestors') storage semantics consumes *the same quantity*, so the dominance inequality is exact within the model — the proof is about the simulator's semantics, which is the object the bit-exactness contract is written against. Whether connection count faithfully ranks physical capacitance on the die is a fidelity question about the whole model family, orthogonal to this work, and we make no claim about it.

5.4.3 Storage excludes itself

> **The self-exclusion principle.** The deepest property of the dominance test is who fails it. A storage cell works *by* winning the floating tie-break: holding a bit as charge means being the largest-capacitance member of one's group when the group floats. Locally-dominant capacitance is therefore not incidental to storage — it is what storage physically is, in this model as on the die. The nodes that matter most to protect (OAM cells, palette RAM, dynamic latches, the two ALU nodes of Section 5.4.1) are exactly the nodes that cannot satisfy cap(x) < cap(every neighbour), so they keep P-1's taint *automatically*. The classifier needs no list of registers, no name matching, no per-chip tuning: asked to find the nodes that are safe to prune, it defines safety such that storage disqualifies itself. The un-taint admits **986** nodes (measured) — light pass-transistor appendages whose stored charge is, by the same inequality, never consulted.

We regard Property 5.3 and its use as this work's strongest original claim, with the precise scoping that honesty requires. Every ingredient is classical: the charge-sharing lattice and size-based resolution are Bryant's [1]; automatic transistor-to-gate-level logic extraction and subcircuit pattern recognition via subgraph isomorphism are established EDA practice [5][6] — a library- and structural-pattern-driven family, in contrast to our physics-property-based identification (pull-up flags, capacitance comparisons, channel components); equipotential skipping is folklore atop selective trace [4][3]. What we have not found prior art for is the specific move — an automatically derived, per-node static dominance inequality over the model's capacitance measure, used to prove tie-break immunity and spent on *event suppression* in a bit-exact engine. Our title-level literature sweeps (Chapter 3) found no collision; the verdict from adversarial external review was "plausibly original," and we present it under that label, not a stronger one.

5.4.4 Relation to IRSIM's node sizes

The nearest prior work deserves its own paragraph. IRSIM [3] — the workhorse switch-level simulator of the academic VLSI flow — embodies a practice in which nodes are assigned small integer *sizes* (capacitance classes): a precharged bus might be assigned a larger size than ordinary nodes so that charge sharing resolves in its favor, and storage nodes are distinguished from transient ones, with the size lattice kept deliberately coarse for resolution efficiency. This is genuinely adjacent: it is capacitance-class reasoning about charge-sharing outcomes, in the same Bryant lineage. The differences are the point of our claim. First, direction of derivation: IRSIM's sizes are designer-supplied annotations consumed by the resolver; our inequality is *derived* from the netlist by the classifier, with no annotation. Second, what the fact is spent on: IRSIM's sizes simplify the computation of a resolution that still happens; our dominance proof suppresses the event so that no resolution happens. Third, the guarantee: a coarse size lattice changes (intentionally) the model's semantics; our inequality is a conservative proof *within* an unchanged semantics, gated on bit-exact equivalence to the unpruned engine. We cite IRSIM as the closest neighbour precisely because drawing this boundary sharply is what an originality claim owes its reader.

5.5 The Zero-Configuration Classification Pass

All four prunes are enabled by one load-time pass, and the pass embodies a methodological position worth stating as such: **every safety fact is derived from the composed system's physics — connectivity, flags, capacitance, and the handler interface — and never from names, lists, or profiles**. Nothing in the classifier knows what an OAM cell or a status register is; the same code classifies the 2A03, the 2C02, and the board TTL without per-chip configuration, and would classify any other netlist in the family's format [15][16] unchanged. This matters for more than elegance. A name-based mask would be fragile against the netlists' known naming irregularities; a profiled mask would be unsound by construction (a workload that never exercises a node proves nothing about the node — the engine's own later experience with profile-versus- derived facts, Chapter 7, bears this out); a physics-derived mask is exactly as trustworthy as the structural argument behind it, which is the only currency the bit-exactness contract accepts.

The pass runs at the end of every `Reset()`, before any settle, in three steps over the flattened arrays of Chapter 4 (cold code; its cost is microseconds inside a load measured in seconds):

```
// Step A — P-1 safety taint (mask bit 0: TURN_ON_UNSAFE)
parent := union-find over all c1-c2 channel edges      // maximal possible groups
for each node f with ForceCompute: taint(find(f))
for each live node n (excluding supplies):
    if not PullUp(n) or tainted(find(n)):
        mask[n] |= TURN_ON_UNSAFE

// Step B — P-2 isolation class (mask bit 1: TURN_OFF_SKIP)
driven := union of every memory handler's output-pin set   // Section 5.3.2
for each live node n not in driven:
    if channelCount(n) == 1 and gndChannels(n) == 0 and pwrChannels(n) == 0
       and none of {PullUp, ForceCompute, Callback}:
        mask[n] |= TURN_OFF_SKIP

// Step C — P-3/P-4 dominance un-taint (clears bit 0)
for each live node n not in driven:
    if flags(n) = none and no supply channels and channelCount(n) >= 1
       and cap(n) < cap(m) for EVERY channel neighbour m:    // strict
        mask[n] &= ~TURN_ON_UNSAFE
mask[VCC] |= TURN_OFF_SKIP                                   // supply-skip fold:
mask[GND] |= TURN_OFF_SKIP                                   // supplies are never re-evaluated
```

Three engineering notes. First, the two facts were consolidated into a single bit-packed byte per node (bit 0 turn-on-unsafe, bit 1 turn-off-skip) so that the hot path reads one array rather than two and future prunes add a bit rather than an array; the consolidation alone measured +0.64% (measured) — on this engine, even the *number* of hot arrays is a measurable quantity. Second, the final line folds the supply guard into the same mechanism: marking V_{CC} and ground as turn-off-skip subsumed the historical explicit supply tests in the enqueue walk, unifying its two unrolled legs (+1.4–2.1% at its commit, measured). Third, the mask as an array eventually vanished from the hot path altogether: Chapter 7 describes how a class-major renumbering makes each prune class a contiguous id block, turning every mask read into a register compare against three boundaries (A=460, S=1275, B=7532), with the freshly recomputed mask retained as the ground truth against which the ranges are verified at every reset — and a safe-degenerate fallback (prunes disabled, correctness preserved) if verification ever fails, in the spirit of deoptimization guards [13]. The block populations give the census directly: 457 nodes are skip-and-unsafe, 815 skip-and-safe — together P-2's 1,272 — and 6,257 more are no-skip-and-safe; in total 2,258 nodes carry a prune bit, and roughly half the live node population is turn-on-prunable.

5.6 Measured Effect

Two measurement frames apply, and we report both because they answer different questions. At ship time, each prune was gated individually by interleaved-paired A/B against its immediate predecessor (Chapter 8 describes the protocol): **P-1 +11.85%; P-2 +1.4%; P-3 +5.96%; P-4 +1.71%; mask consolidation +0.64%** (all measured, all bit-exact at their commits). Retrospectively, the nine-round same-day ablation ladder of Chapter 8 rebuilds the historical commits and measures them under one protocol; there the stage containing P-1 steps **+26.4%**

over its predecessor and the stage containing P-2/P-3/P-4 plus the consolidation steps **+7.2%** (measured). The two frames need not agree numerically — ladder stages are cumulative commits and so bundle each prune with the unrelated micro-work that landed alongside it in history, and the runs are separated by days of engine evolution and a .NET version — and we flag the bundling honestly: the +26.4% step is an upper bound on P-1's isolated effect under the ladder's conditions, while +11.85% is the contemporaneous isolated measurement. Under either frame, the prune family is the largest single contributor to the engine's cumulative +94.6% (Chapter 8).

Table 5.2. The prune family: conditions, populations, and ship-time gains (interleaved-paired medians, bit-exact; measured).

Prune	Suppresses	Static condition (per node)	Population	Gain
P-1	turn-on enqueue when endpoint states are equal	safe iff pull-up present and channel component ForceCompute-free	~half of live nodes safe	+11.85%
P-2	turn-off enqueue entirely	single channel, no supply channels, no pull-up/FC/callback, not handler-driven	1,272 nodes	+1.4%
P-3	(extends P-1 to no-pull-up nodes)	flagless, supply-free, cap strictly below the single neighbour's	986 nodes un-tainted	+5.96%
P-4	(multi-channel generalization)	as P-3 with cap strictly below *every* neighbour's		+1.71%

At the event level, the effect is direct deletion. Comparing the profiler's censuses immediately before P-2 and after P-4 (300,000 half-cycles, `full_palette`): total pops fall from 159.6M to 126.1M — **33.6M re-evaluations, about 21% of all pops, deleted** — and the no-change share falls from 84.4% to 80.3% (measured). The composition shifts as the theory predicts: the prunes harvest the floating categories and the enqueue-side redundancy, leaving the residue progressively dominated by the PullUp and Supply categories that Section 5.7 shows to be structural. Note the arithmetic of diminishing percentages: deleting only no-change pops lowers the no-change *share* slowly even as it removes a fifth of all work, because the denominator shrinks too — the honest way to read the table is in absolute pops, not in the waste percentage.

Table 5.3. Event- and instruction-level effect of the prune stages (measured: profiler at 300k hc; ETW PMC at 1M hc, ladder stages S1 = before P-1, S2 = + P-1, S3 = + P-2/3/4 + consolidation).

Metric	before P-1 (S1)	+ P-1 (S2)	+ P-2/3/4 (S3)
RecalcNode pops / 300k hc	159.6M (pre-P-2 census)		126.1M
no-change share of pops	84.4%		80.3%
instructions retired (B / 1M hc)	128.7	116.0	105.7
IPC	2.08	2.38	2.35
branch mispredicts (MPKI)	3.37	2.74	2.52
L1d misses (MPKI)	30.6	24.5	25.8

The hardware counters (Chapter 8 gives the full ladder) corroborate the mechanism claim. The prunes delete *instructions*, not merely abstract events: retired instructions fall 18% across the two prune stages alone, on the way to the full ladder's −33% (155.6B to 104.4B per million half-cycles, S0 to S7), while IPC holds in the 2.1–2.4 band — the engine does proportionally less work rather than the same work faster. Branch mispredicts improve by a quarter across the prune stages (each suppressed event is also a suppressed chain of hard-to-predict dispatch branches), and the per-instruction L1d miss rate *improves* at S2 even as total misses fall — evidence that the deleted pops were no cheaper in memory behavior than the surviving ones. Every one of the 72 ladder runs, and every ship-time A/B round, passed the full-state checksum gate.

5.7 What the Prunes Cannot Reach

After P-1 through P-4, 80.1% of the surviving pops still recompute an unchanged value (Table 5.1), and the residue is dominated by the PullUp (42.0%) and Supply (38.1%) categories. We close the chapter by delimiting the family's boundary: this residue is, on the evidence, structurally irreducible by static means — not "we have not found the prune yet" but "there is no static fact there to find." The claim rests on three independent confirmations, obtained by different methods at different times.

- **The mechanism argument, adversarially reviewed.** A PullUp-category pop re-evaluates a pull-up node after one of its pull-down gates flips; whether the value changes depends on the OR of the node's *other* pull-down gate states — live state, different per event, with no per-node invariant. Any predictor must read those gate states, which is precisely the scan the O(1) fast path already performs as its whole cost. Suppressing the pop therefore requires doing the pop's work, a self-defeating bargain. We submitted this analysis, together with the project's accumulated dead-end ledger, to deliberately adversarial external review with instructions to find the flaw; the review concurred that no cheap predictor exists for this category.
- **The exhaustive static census.** If any statically-classifiable subclass of the residue existed, it would appear as a population in the classifier's terms. We built the diagnostic (the P-2b

census, extending P-2's conditions toward single-channel *pull-up* nodes) and measured the candidate population at **0.04%** of pops (measured) — three orders of magnitude below the smallest shipped prune's payoff, and consistent with zero exploitable structure. The prune was closed without implementation, on population alone.

- **The dynamic-fact experiments.** The residue *can* be predicted dynamically — by maintaining, per node, facts about its currently-conducting paths — and Chapter 9 recounts the two full attempts. The maintained form (P-5) measured a gross prize of +13.76% and a net of −6.84%: the maintenance floor, ~7% of runtime, exceeds the prize, the same economics that killed prune-merge in Section 5.2.2. The zero-maintenance revival (P-5z) split the residue into a leg whose fact could be derived for free at the suppression site — sound, bit-exact, and measured performance-neutral (+0.16%/−0.51%) — and the valuable pull-up leg, whose required fact provably binds to the value at last resolution time: reading live state instead diverges (measured), so soundness demands a maintained snapshot, returning to the maintenance floor. The boundary is closed from both sides.

> **Chapter finding.** Of the ~84% of node re-evaluations that recompute an unchanged value, the fraction removable by load-time proof has been removed: four prunes, 2,258 masked nodes, ~21% of all re-evaluations deleted, ship-time gains of +11.85%, +1.4%, +5.96%, and +1.71%, all bit-exact. The remainder is protected by the same physics that makes the model worth simulating: the residue's value depends on live conducting-path state (combinational re-confirmation) or on resolution-time charge snapshots (storage), neither of which admits a static certificate. Further event-count reduction on this engine therefore requires either paying a maintenance tax the events cannot amortize (Chapter 9) or leaving the bit-exact contract altogether — which is no longer this engine. The remaining headroom must come from elsewhere: from resolving the surviving events faster (Chapter 6) and from the memory system itself (Chapter 7).

Chapter 6. Cardinality-Specialized Dispatch

The prune family described earlier in this monograph attacks the event *count*: it deletes re-evaluations whose outcome is provably unchanged before they ever enter the queue. This chapter attacks the complementary quantity, the cost of each event that survives. In the engine's inherited form — Visual 6502's `chipsim.js` [15] by way of MetalNES [17] — every pop pays the full price of generality: clear the scratch state of the previous walk, seed a breadth-first search over currently-ON pass transistors, accumulate a flags-OR across the discovered members, resolve through the priority table, and write the result back to every member. That generality is demanded by the semantics, which must handle a conducting group of any size and shape. It is not demanded by the data. On a die-derived NMOS netlist the conducting groups are overwhelmingly tiny, and their size is often *provable* — sometimes statically, at load time, and sometimes dynamically, from a handful of byte reads at the moment the event fires. Cardinality-specialized dispatch is the systematic exploitation of those proofs: classify every node into a dispatch class at load, and at runtime prove that the active channel-connected component (CCC) has size exactly one or exactly two before resolving it inline, falling back to the unmodified general walk the instant the proof fails.

We state the intellectual position up front, because this chapter contains the least novel material in the monograph and the honesty requirement binds. Recognizing that a node with no pass channels forms a permanent singleton CCC is implicit in Bryant's switch-level partitioning [1] and explicit in COSMOS, which compiled such components into Boolean code outright [2]. Determining the *currently active* subnetwork dynamically is the founding idea of IRSIM [3]. What we add is an implementation architecture — cardinality tests hoisted into the dispatch decision so that the general machinery is never even set up for the common case — together with a careful enumeration of the bit-exactness obligations that make the size-2 inline path a semantics-preserving rewrite rather than an approximation. The measured payoffs justify the chapter's length: the dynamic-singleton path is the single largest step in the nine-stage ablation ladder of Chapter 8 (+18.7%), and the pair path is the last (+7.0%).

6.1 The Observation: Group-Size Distribution and Dispatch Classes

The case for specialization is empirical, so we begin with the measured shape of the work. All figures in this section are event counts from the DEBUG-build profilers, which run the identical algorithm to the Release build and therefore produce identical event populations; the workload is `full_palette.nes` over 100,000 master-clock half-cycles (hc). The engine performs approximately 42.85 million pops in that window — about 418 per half-cycle — where a *pop* is one node drawn from the current settle-wave list and re-evaluated. Of those pops, 60.4% are resolved by the fast paths described in Sections 6.2 and 6.3 without ever touching the group-walk machinery, and 39.5% (16.94 million) proceed to an actual group walk. The walks themselves are minuscule: 77.1% of them build a group of exactly two nodes, 16% reach three nodes, and only 5.7% reach four or more. The breadth-first frontier rarely advances at all — the mean BFS depth

(hops from the seed through ON transistors to the farthest member) is 1.13, the 99th percentile is 3, and the maximum ever observed is 14. Expressed against the total event stream rather than against walks, size-2 groups alone account for 30.5% of *all* pops.

This distribution is a direct imprint of the silicon. A 1980s NMOS chip is mostly depletion-load gate outputs — nodes pulled up by a load device and pulled down through enhancement transistors to ground — which have no pass-transistor channel to any other signal node and therefore can never share a CCC with one. The pass-transistor structures that do exist (transmission gates into latches, register-file access transistors, bus multiplexers) are mostly one transistor deep: a storage node behind a single access device, a bus leg behind a single driver. Long conducting chains exist — the maximum-depth-14 walks are real, and the wide bus groups are precisely the structures the engine must get right — but they are rare events, not the steady state. The engineering consequence, which recurs as a verdict throughout this monograph's negative results, is that any scheme whose per-walk overhead is amortized over *large* walks is betting against the data: the bit-parallel BFS experiment (156× slower, Chapter 9) failed for exactly this reason. The profitable direction is the opposite one — make sizes one and two as close to free as the semantics permit.

The dispatch architecture is a one-byte-per-node classification, `cls`, computed once per `Reset()` alongside the prune masks and consumed at the top of every pop. Table 6.1 gives the class definitions and their measured populations.

Table 6.1. Dispatch classes, their load-time definitions, and measured populations (full_palette.nes; node shares over live nodes, pop shares over all pops per 100k hc).

Class	Definition (load time)	Nodes	Runtime handling
cls1 — static singleton	No pass channel to any signal node; none of {callback, ForceCompute, supply} flags	3,929 (26.7%)	O(1) LUT resolve (Section 6.2), unconditionally
cls2 — dynamic-singleton candidate	Has ≥1 pass channel; none of the excluded flags	10,784 (73.2%)	Runtime cardinality test (Sections 6.3–6.4); fall back to BFS on failure
cls0 — general	Carries a callback or ForceCompute flag, or supply resolution	~16 (≈0.1%; ≈0.4% of pops)	Always the general group walk

Table 6.2. Measured event-level shape of the work per 100k hc (DEBUG profilers; event counts identical to Release).

Quantity	Measured value
Total pops	≈42.85M (≈418 per hc)
Pops resolved without the group walk	60.4%
Pops reaching a group walk	16.94M (39.5%)
Walk size distribution	size-2: 77.1%; size-3: 16%; size-4+: 5.7%
Size-2 walks as share of all pops	30.5%
BFS depth (hops from seed)	mean 1.13; p99 = 3; max = 14
No-change pops (all causes)	80.1% (structural PullUp/Supply residue 42.0% + 38.1%)

Two remarks on Table 6.1. First, the class populations explain why this family matters at all: nearly three quarters of the chip's nodes are cls2 candidates, and — because most pass-transistor gates are OFF at any given instant, a fact already visible in the 80.1% no-change rate — the candidates' runtime test succeeds far more often than it fails. Second, cls0 is vanishingly small. The sixteen general-path nodes are the behavioral-memory callback anchor nodes and the eight ForceCompute bus nodes; their combined 0.4% pop share is the reason every attempt to specialize *them* died of population starvation (Section 6.6).

6.2 cls1: Static Singletons

The static singleton class is the known technique, and we credit it as such. A node whose transistor incidence contains no channel to another signal node — every transistor touching it is either a gate connection (which never merges CCCs) or a channel straight to a supply rail — forms a CCC of exactly one node, permanently, by the structure of the netlist alone. This is the trivial stratum of Bryant's switch-level partitioning [1], and COSMOS went further than we do, compiling such components out of the simulator entirely into levelized Boolean code [2]. We deliberately do not compile (the project's compiled backends were uniformly slower, for instruction-cache reasons quantified in Chapter 9); instead we keep the singleton interpretive but reduce its evaluation to its information-theoretic floor.

For a singleton group $\{n\}$, the general resolution — flags-OR over members, then the 256-entry priority LUT — collapses to the following: OR the node's own flag byte (pull-up, plus any runtime external-drive SetHigh/SetLow flags) with a Gnd bit if any of its gate-to-GND transistors is ON and a Pwr bit if any gate-to-VCC transistor is ON; index the LUT; store. The gate states are read with a short OR-all loop over the node's inline payload — branchless by construction, since states are 0/1 bytes — and the entire evaluation touches one `NodeInfo` cache line, the handful of gate state bytes, and the LUT.

```
procedure ResolveSingleton(n):              // cls1 always; cls2 after the all-OFF proof
    f ← Flags[n]                            // PullUp | SetHigh | SetLow; Gnd/Pwr/FC/callback
                                            //   excluded from the class at load time
    anyG ← OR over gnd-gates g of n : state[g]
    anyP ← OR over pwr-gates g of n : state[g]
    f ← f | (anyG << GndBit) | (anyP << PwrBit)
    if f ≠ 0:  SetNodeState(n, LUT[f])
    // f = 0 ⇒ purely floating singleton ⇒ hold previous value.
    // For a one-node group the largest-capacitance member IS n, so "hold previous"
    // is a guaranteed no-op: skip the store and the downstream enqueue walk entirely.
```

Two details of the implementation carry the exactness argument. First, the flag byte is read fresh on every call rather than folded into a precomputed default: a cls1 node can acquire and shed external-drive flags at runtime (the clock pin and the handler-driven pins do), and reading live flags lets those ride the LUT priority exactly as they would through the general path. Second, the class membership does *not* require a pull-up. The original classification did, on the intuition that a pull-up guarantees a driven resolution; but the purely-floating singleton resolves, by the general path's own floating rule, to the previous state of its largest-capacitance member — which for a one-node group is the node itself. "Hold previous" on a singleton is a structural no-op, and the fast path reproduces it by simply not storing. Dropping the pull-up requirement widened static coverage from 23.1% to 26.7% of live nodes and was verified checksum-identical. We note the pattern because it recurs: the exactness argument is always made against the general path's *behavior on the specialized population*, not against an idealized model, and several percentage points of coverage repeatedly hid behind requirements that were sufficient but not necessary.

6.3 R-1: The Dynamic Singleton

cls1 exhausts what is provable from structure alone. The remaining 73.2% of nodes have at least one pass channel, so their CCC *can* grow — but whether it grows *now* depends only on the present states of the gates controlling those channels, and those are a few byte reads away. R-1 is the observation that the first step of the BFS, performed before any of the BFS's setup costs are paid, is a complete cardinality proof for the most common case.

Property 6.1 (dynamic singleton). Let n be a cls2 node and suppose that at the moment of evaluation every transistor whose channel connects n to another signal node has its gate OFF. Then the conducting group of n is exactly $\{n\}$, and the inline resolution of `ResolveSingleton(n)` is byte-identical, in its effect on all engine state, to the general path `ComputeNodeGroup(n)` followed by member writeback.

The proof is short because the walk's own definition does the work. The BFS admits a new member only by traversing a channel whose gate is currently ON, starting from members already admitted. With every channel of the seed OFF, the frontier cannot leave the seed; the group is $\{n\}$. (The netlist's channel endpoints c_1/c_2 are interchangeable in NMOS and the per-node channel list

covers both orientations, so there is no asymmetric escape.) For the bit-identity claim we enumerate what the general path would have done. The scratch-state clear touches only the previous group's members and leaves no observable trace. Admitting the seed clears its pending-pop dedup flag — which the pop loop clears anyway for the node being popped. The flags-OR over a one-member group is the seed's own flag byte plus the Gnd/Pwr scan of the seed's supply gates — precisely `ResolveSingleton`. The floating case holds the previous value, which Section 6.2 showed is a no-op for singletons. The member writeback is a single `SetNodeState(n, v)`, identical in both forms, and the candidate class excludes callback and ForceCompute nodes at load time, so the general path's callback-enqueue epilogue is vacuous on this population. Every observable consequence coincides.

The runtime test fails — some channel gate is ON — by jumping to the general walk having mutated nothing, so failure costs only the scan itself. The scan reads gate ids and states from the node's inline payload, one cache line in the common case; nodes whose transistor incidence overflows the inline payload fall back to a terminated-list scan of the shared transistor array. A small but measured engineering note: flattening this dispatch with a direct `goto` into the fallback, rather than accumulating a boolean and merging control flow, was worth roughly +0.5% (four interleaved-paired batches, bit-exact) — at these per-event costs, the shape of the JIT's exit code is visible in the wall clock.

```
procedure RecalcNode(n):                    // top of every pop
    switch cls[n]:
      case 1:  ResolveSingleton(n); return
      case 2:
          for each (gate, other) in channels(n):
              if state[gate] ≠ 0:  goto BFS // group may grow (B1 intercepts here, §6.4)
          ResolveSingleton(n); return       // all gates OFF ⇒ group = {n}  (Property 6.1)
    BFS:
      v ← ComputeNodeGroup(n)               // the unmodified general walk
      for each m in group:  SetNodeState(m, v)
      enqueue callbacks of group members
```

R-1 is the largest single step in the ablation ladder: +18.7% in median throughput (67,955 to 80,657 hc/s, stage S0→S1, nine-round round-robin, all runs checksum-gated). The performance counters identify the mechanism unambiguously as *work deletion* rather than improved efficiency on the same work: retired instructions fell from 155.6 to 128.7 billion per million half-cycles (−17.3%), while IPC was essentially flat (2.11 to 2.08) and the L1d miss rate per kilo-instruction actually *rose* from 27.3 to 30.6 — the deleted BFS bookkeeping was well-pipelined, miss-free work, so the residual instruction stream is slightly denser in misses. The wall-clock gain tracks the instruction deletion almost exactly, which is what one expects when the eliminated work was not itself stalled.

We frame the credit honestly. IRSIM's central idea is that the active subnetwork is determined dynamically, at event time, rather than fixed by static partitioning [3]; in that lineage, R-1 contributes no new model — it is an *early-out execution filter*, the hoisting of the walk's first frontier expansion into the dispatch decision so that the walk's setup (scratch clears, group buffer,

flag accumulator) is never instantiated for the 60%-of-pops case where the answer is already determined. The claim is an implementation architecture and its measured consequences, nothing more.

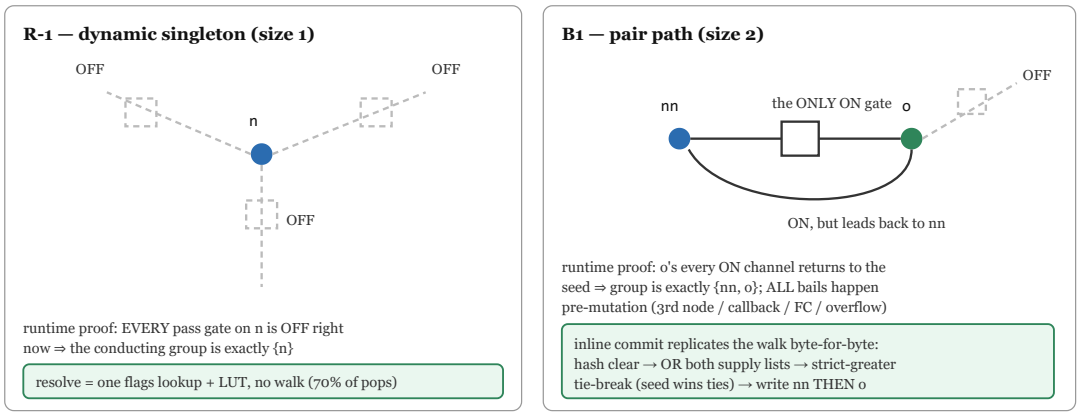

Figure 6.1. The runtime proofs of the two cardinality-specialized paths. Left: the R-1 dynamic singleton (Section 6.3) — every pass gate is OFF at this instant, so the component must be {n}; right: the B1 pair path (Section 6.4) — the only conducting gate leads to o, and every conducting channel of o folds back to the seed, so the component must be {n, o}; all bail conditions are checked before any mutation, and the inline resolution replicates the walk's writes byte for byte.

6.4 B1: The Pair Path

After R-1, the singleton population is handled, and Table 6.2 says exactly one further class is large enough to matter: size-2 groups, 77.1% of all surviving walks and 30.5% of all pops. These are the chip's access transistors in action — one ON channel connecting the popped node to one neighbour. The general walk handles such a group with the full apparatus: scratch clear, seed admission, frontier expansion (which discovers one neighbour and then terminates), flags accumulation, value resolution, two-member writeback, callback sweep. B1 resolves it inline at the point in the cls2 scan where the first ON gate is discovered — but only after *proving* the group is exactly the pair $\{n, o\}$, with every proof obligation discharged before any engine state is mutated.

Property 6.2 (pair). Let n be a cls2 node whose channel scan finds an ON gate to neighbour o, and suppose: (i) $o \neq n$; (ii) no other channel gate of n is ON; (iii) o is inline-resident and carries neither the callback nor the ForceCompute flag; and (iv) every ON channel of o leads back to n. Then the conducting group of n is exactly $\{n, o\}$, and the inline resolution below is byte-identical in its effect on all engine state — including the pending-pop dedup hash, the value, the write order, and the next wave's enqueue order — to `ComputeNodeGroup(n)` followed by member writeback.

The cardinality part of the proof mirrors Property 6.1: the BFS frontier from n can cross only n's single ON channel, reaching o; from o it can cross only ON channels of o, all of which, by (iv), return to the already-admitted n. The group is closed at $\{n, o\}$. The substance of B1, however, is

not this observation — it is the list of things the general walk does *besides* computing a value, each of which is observable and each of which the inline path must replicate exactly. We enumerate them, because the engineering discipline they represent — treat the legacy path's side effects as the specification, byte for byte — is the chapter's main methodological content.

1. **Pre-mutation bails.** All four conditions of Property 6.2 are tested before anything is written. A self-channel ($o = n$) bails; a second ON gate on the seed bails — even when that gate's channel leads to the *same* neighbour o, a case where the group would still be the pair but the proof would need to reason about parallel channels, and we chose the cheaper, conservative test; an overflow or callback/ForceCompute neighbour bails; an ON channel of o to any third node bails. Because no mutation precedes the last test, the fallback re-executes the general walk on pristine state and is exact by construction. This guard-then-commit shape is the same discipline as a deoptimization guard [13]: the specialized path must prove its own applicability and surrender losslessly.
2. **The member dedup-hash clear.** When the BFS admits a member, it clears that member's pending-pop flag, cancelling any scheduled re-evaluation of it later in the *same* wave — its value has just been resolved as part of this group. The inline path must perform the identical `dedupHash[o] ← 0`. This clear is load-bearing: omitting it leaves o to pop again within the wave and re-resolve after n's write, which diverged in testing. It is the kind of side effect that is invisible in a value-centric reading of the algorithm and fatal in a bit-exact one.
3. **Both-node supply scans.** The group's flag word is the OR of both members' flag bytes plus a Gnd bit if *any* gate-to-GND transistor of *either* member is ON, and likewise for Pwr. OR-accumulation is order-free, so the inline path's straight-line OR-all over both supply lists produces the same word as the walk's early-breaking accumulation, despite visiting in a different order.
4. **The exact floating tie-break.** If the flag word is zero the pair is purely floating, and the general rule — the largest-capacitance member's previous state, with *strict* comparison and first-seen winning ties — must be reproduced in its exact form. For the pair this is `cap[o] > cap[n] ? state[o] : state[n]`: strictly greater, seed wins ties, which is precisely `GetNodeValue` iterating over the group buffer [n, o]. We emphasize, as elsewhere in this monograph, that this tie-break is not a numerical detail: it is the chip's storage mechanism, and a ≥ in place of a > is a different machine.
5. **Write order.** The walk writes members in group-buffer order: seed, then neighbour. Each `SetNodeState` appends that node's gate-dependents to the *next* wave's list, and within-wave pop order is Gauss-Seidel and observable semantics (Chapter 2). The inline path therefore writes `SetNodeState(n, v)` then `SetNodeState(o, v)` — the same order, hence the same next-wave enqueue order, hence the same downstream pop order.
6. **Callback and ForceCompute exclusion.** The seed's cls2 classification already excludes both flags on n; bail (iii) excludes them on o. Consequently the general path's callback-enqueue epilogue is vacuous on the admitted population, and the ForceCompute Gnd+Pwr cancellation — which the LUT does encode, but which interacts with group membership in

ways we declined to re-verify inline — never arises. Conservative exclusion was cheaper than proof.

```
// inside the cls2 channel scan, at the FIRST gate found ON, channel (gate, o):
if o = n:  goto BFS                                  // (i)  self-channel
for each later (gate', ·) in channels(n):
    if state[gate'] ≠ 0:  goto BFS                   // (ii) second ON seed gate — conservative
if overflow(o) or Flags[o] ∩ {Callback, FC} ≠ ∅:  goto BFS   // (iii)
for each (gate', x) in channels(o):
    if state[gate'] ≠ 0 and x ≠ n:  goto BFS         // (iv) ON path to a third node
// committed: group is exactly {n, o}; nothing has been mutated yet
dedupHash[o] ← 0                                     // BFS member clear (obligation 2)
f ← Flags[n] | Flags[o]
anyG ← OR over gnd-gates of n, o : state[·]
anyP ← OR over pwr-gates of n, o : state[·]
f ← f | (anyG << GndBit) | (anyP << PwrBit)        // (obligation 3)
v ← (f ≠ 0) ? LUT[f]
            : (cap[o] > cap[n] ? state[o] : state[n])  // strict; seed wins ties (obligation 4)
SetNodeState(n, v);  SetNodeState(o, v)              // walk's write order (obligation 5)
```

Verification proceeded through every gate the project maintains: the DEBUG event-count check — total pops are unchanged at 42,848,807 per 100k hc, confirming B1 is a pure rewrite of how events are resolved, not a change in which events occur — the three Release full-state checksums at 300k/400k/1M hc, the self-test netlists, and the 10-million-half-cycle sprite-heavy Super Mario Bros. checksum gate. The conservative bails leave coverage on the table by design: of the 30.5% of pops whose walks are size-2, the pair path captures 8.34 million pops per 100k hc (19.5% of all pops); the remainder bail on conditions (ii)–(iv) and take the general walk as before. We did not pursue the residual: each relaxation (parallel channels, overflow neighbours, ForceCompute pairs) adds proof surface for a population fragment, and the family's kill criteria (Section 6.6) had by then hardened.

6.5 Measured Effects

Table 6.3 collects the dispatch family's measured effects. R-1's +18.7% was discussed in Section 6.3. B1 enters the ablation ladder as stage S7: +7.0% in median throughput over S6 (123,545 to 132,243 hc/s, same-day interleaved; best run 135,828), and about +8.9% after separating the attribution from the adjacent housekeeping commit. Measurements are taken with the clock unlocked; variance is controlled by same-day multi-round round-robin interleaving, a single benchmark process, and median reporting.

Table 6.3. Measured effects of the dispatch family (medians; ablation ladder 9 rounds × 8 stages, 400k hc, checksum-gated; counters from the 1M-hc PMC pass).

Effect	Measurement
R-1 (ladder S0→S1)	+18.7% (67,955 → 80,657 hc/s)
R-1 instruction deletion	155.6 → 128.7 B instr/1M hc (−17.3%); IPC 2.11 → 2.08
B1 (ladder S6→S7)	+7.0% (123,545 → 132,243 hc/s); about +8.9% after separating the attribution
B1 instruction deletion	111.3 → 104.4 B instr/1M hc (−6.2%); IPC 2.41 → 2.34
B1 event coverage	8.34M pops/100k hc (19.5% of pops) resolved inline; total pops unchanged

The counter profile of S7 repeats R-1's signature at smaller scale: instructions retired fall 6.2% while IPC dips slightly and the L1d miss rate is flat (12.5 to 13.0 MPKI) — again deletion of well-pipelined machinery, not a cache effect. But B1's deleted instructions are not uniformly distributed across the pipeline's slack: a substantial fraction of the removed work — the group buffer maintenance, the scratch clears, the indirect member iteration — sits on the *dependent-load chain* that Chapter 8 identifies as the engine's true limiting resource.

One framing distinction is worth making precise. The prunes of the preceding chapter reduce the event count: pops simply do not happen. Dispatch leaves the event stream untouched — the pop-count invariance check is part of B1's verification — and reduces the constant per event. The two families therefore compose multiplicatively and do not compete for the same prize, with one subtle interaction: by deleting the cheapest, most predictable pops, the prunes *raise* the average cost of the survivors, which is part of why the dispatch family's later steps remained profitable on an engine that had already absorbed +60% from pruning.

6.6 The Kill List

The dispatch family was closed after B1, and we document the closure with the same evidentiary standard as the adoptions, because the negative space is where most of the family's exploration actually happened. Every candidate below was either measured on the live engine or killed by an analytic argument that itself rests on measured populations. Table 6.4 summarizes; prose follows for the two that carry general lessons.

Table 6.4. Dispatch-family candidates killed with measured populations (full_palette.nes, 100k hc; "—" = killed analytically before a build was warranted).

Candidate	Measured population	Verdict
cls3: callback-singleton fast path	6 nodes; 67,968 pops = 0.16% of pops	Structurally sound (the callback fake-transistors verified to sit in the supply buckets), but population-dead; 100% of these pops are no-change — the value is constantly 0 and the only effect is the callback enqueue.
ForceCompute admission to cls2	8 nodes; 101,120 pops = 0.24% of pops	Bit-exact in principle (the LUT already encodes the FC Gnd+Pwr cancel), but all eight are high-fanout overflow bus nodes whose channels are usually ON — they would fall through to the BFS regardless.
Enqueue-time dispatch hints in the dedup-hash byte	—	Unsound in the profitable direction: a later pop in the same wave can switch a gate ON after the hint was frozen, so a stale "fast" hint computes a wrong value. The safe direction only adds BFS falls; maintaining hint truth is the counter-fast-path dead end (measured negative).
Partial-scan reuse on BFS fallthrough	mean first-ON gate index < 2	The "saved" work is re-reading L1-hot lines that feed a predicted branch — off the carried dependency chain; no gain available.
Epoch-based dedup-hash clearing	—	An 8-bit epoch wraps every ~21 settle calls, producing false "already queued" verdicts — divergence; a possible explanation for the stores it would save is that they are already store-buffer-hidden (not separately verified).
Per-class queues / within-wave reordering / batched pops	—	Within-wave order is Gauss-Seidel and observable semantics; any reordering is a different simulator by specification, not an optimization.
Wave-local group memoization	BFS depth p50 = 1	Group membership is a function of live gate states; there is no repeated-large-walk population to amortize against, and validity tracking is the maintained-runtime-fact floor (the P-5 lesson, Chapter 9).
Branchless duplicate-tolerant enqueue	enqueue branch ~97.7% one-sided	A branch predicted at 97.7% costs nothing to remove; duplicates would also break the wave list's capacity guarantee.

Two of these merit elaboration. The *enqueue-time hint* idea — when a gate change enqueues a node, record whether the change was a turn-on or a turn-off, so the eventual pop can skip its own re-scan — fails for a reason that generalizes: in a Gauss-Seidel wave, information recorded at enqueue time is a snapshot, and any pop executed between the snapshot and the consuming pop can invalidate it. The profitable direction of the hint (trust it and skip work) is therefore exactly

the unsound direction. This is the same live-state-versus-snapshot distinction that killed the P-5 dominant-bypass family (Chapter 9), surfacing in miniature; the engine's only trustworthy facts at pop time are the ones it re-derives at pop time, and the only affordable re-derivations are the O(1) proofs of Sections 6.3–6.4.

The *within-wave reordering* entry deserves its bluntness. Sorting pops by class, batching singletons, or deferring walks would all improve locality and branch coherence on paper — and every one of them changes which values later pops in the same wave observe, because within-wave order is semantics in this model. There is no "mostly equivalent" here; the floating tie-break and the Gauss-Seidel visibility rules make the wave order part of the machine's definition. We verified this the only way available — divergence — early in the project, and the rule has been absolute since.

A final candidate from an adjacent family reinforces the chapter's selection criterion: encoding the `cls` byte itself as id-range compares (the same transformation that made the prune masks free) measured neutral (−0.28%), because the class load feeds a well-predicted branch rather than sitting on the carried dependency chain. Together with Table 6.4 this fixes the three-condition checklist that all later proposals in this engine were screened against: a profitable target must be *static* (provable without runtime truth maintenance), *hot* (a population measured in whole percents of pops, not fractions), and *on the carried chain* (the work deleted must be latency-bearing, not pipeline filler). R-1 and B1 satisfy all three; everything else in the family fails at least one, with the measurements above as the evidence.

With B1 shipped, the family's end state is as follows: roughly 80% of all pops resolve without the general group machinery — 60.4% as singletons on a chain of about five dependent loads, a further 19.5% on the inline pair path — cls0 in its entirety is 0.4% of pops, and the residual general walks are the genuinely irregular structures (buses, deep pass chains) for which the BFS is the right tool. What remains of the engine's time is the structural no-change residue (80.1% of pops, with no static subset — confirmed three independent ways) and the dependent-load latency of the minimized resolution itself, which is the subject of the relayout chapters and of the ceiling analysis that closes the monograph [21].

Chapter 7. Self-Captured Data Relayout

The optimizations of the preceding chapters — the dynamic-singleton fast path, the P-1 through P-4 enqueue prunes, the clause reordering — all operate on the event schedule: they delete work the event-driven engine would otherwise perform. This chapter describes a different kind of intervention, one that leaves the event schedule untouched and instead rewrites the *identifier space* of the netlist. At load time, the engine computes a permutation of node ids and rebuilds every data structure under the permuted numbering. The permutation serves two distinct objectives. The first, *range-prune*, sorts nodes so that each prune class occupies one contiguous id block, converting the hottest lookup table in the engine into a pair of register compares — a 4.2% ladder step. The second, the *self-captured first-touch key*, orders nodes within each block by the order in which the production settle cascade actually first touches them, a trace the engine captures from itself during load — a further 5.0% ladder step.

The chapter is also a story about how performance dead ends are indexed. One month before these techniques shipped, node renumbering had been measured in this project, found worthless, and recorded in the dead-end catalogue. Nothing about that measurement was wrong; what was wrong was the inference drawn from it — that the *tool* was dead, rather than the *objective* it had been asked to serve. Section 7.1 reconstructs that episode, because the diagnostic chain that re-opened the case is, we believe, more transferable than the techniques themselves. Sections 7.2 and 7.3 describe the two mechanisms; Section 7.4 presents the experiment that isolates what the locality key actually buys (order, not density); and Section 7.5 situates the construction relative to the inspector-executor and trace-driven-layout literature [7, 8, 9, 10, 11, 12].

7.1 The Dead End That Came Back

7.1.1 Classic RCM, and why it failed honestly

Reverse Cuthill-McKee (RCM) renumbering reduces the bandwidth of a sparse graph: vertices that are adjacent in the graph receive nearby identifiers, so that data belonging to neighbors shares cache lines. The hypothesis that motivated trying it here was natural. The engine's group walk — the per-event BFS over ON pass transistors that delimits a channel-connected component [1] — jumps from a node to its channel neighbors, and the enqueue walk in `SetNodeState` jumps from a gate node to the channel endpoints of the transistors it controls. If those neighbors lived on the same cache lines, the walks should miss less. Measured in May 2026, the result was approximately 1.04× during the boot transient and approximately 0.98× in steady state: nothing, within noise. The verdict entered the project's dead-end catalogue as "renumbering is dead."

The diagnosis attached to that verdict was already correct, and it is worth stating precisely because it predicted everything that followed. The engine's hot working set — the 1-byte state array, the 16-byte `NodeInfo` records of the currently active nodes, the wave queues — is on the order of 15 KB and is resident in the L1 data cache during steady-state simulation. RCM addresses a *capacity* problem: it reduces the number of distinct cache lines a traversal touches so that they

fit in, or stream through, a finite cache. When the traversal's working set already fits, there is no capacity problem to solve, and bandwidth reduction is an answer to a question the machine is not asking.

7.1.2 Three independent confirmations

One month later, three experiments ran back-to-back, all attacking the engine's memory behavior from different directions, all gated by the bit-exactness protocol (golden checksums) and measured with the interleaved-paired methodology. Their joint outcome is summarized in Table 7.1.

Table 7.1. The three latency-hiding experiments of 2026-06-10 (C# engine, full_palette, interleaved-paired). Each attacks memory behavior; each returns nothing or worse. All measured [21].

Experiment	What it tests	Result
Shrink the wave queues from `int` to `ushort` (−60 KB of streamed data)	whether streamed queue traffic costs anything	−0.07% (neutral)
Software prefetch of the next pops' `NodeInfo` records	whether there is memory latency left to hide	−0.3% to −1.1%
Pure co-activity locality renumber: cache-line density of the per-half-cycle hot set improved by 45%, near the packing ideal	whether even a near-perfect locality win moves wall-clock	−0.36% (neutral)

The third row deserves emphasis because it is the strongest possible form of the negative result. The experiment did not merely fail to find a good layout; it *found* one — an instrumented co-activity profile reduced the number of distinct `NodeInfo` cache lines touched per half-cycle by 45%, close to the packing-theoretic ideal for the measured activity — and that near-perfect locality improvement still produced no wall-clock gain. Locality, as an objective, was not under-optimized; it was fully optimized and worth nothing.

> **Finding.** Three independent experiments confirm one fact: the engine's per-event cost floor (≈20 ns per pop at the time of measurement) is not cache-miss-bound. It is *dependent-load-chain* bound. Each event is a short serial chain — load a node's record, load what it points to, branch on the result — and the cost is the chain's *length* in dependent loads, each individually an L1 hit. Moving bytes closer together changes where data lives; it does not shorten chains.

7.1.3 The objective flip

If the bound is chain length, the only lever is deleting links. This reframing transforms the renumbering machinery from a failed optimization into an enabling mechanism, because there is a class of chain links that a permutation can delete outright: loads whose only purpose is to

answer a *static* question about a node. A static per-node fact — one fixed at load time and constant for the life of the netlist — does not need to be fetched from a table indexed by node id. It can be encoded in the node id itself: sort the nodes so that all nodes for which the fact is true occupy a contiguous id interval, and the table lookup becomes a comparison between the id (already in a register, because the loop is iterating over ids) and a constant (also in a register). The load disappears from the chain; nothing replaces it.

The May verdict and the June success are therefore both correct, and the resolution of the apparent contradiction is the central methodological lesson of this chapter: **dead ends are indexed by objective, not by tool.** "Renumbering is dead" was a true statement about the objective renumbering had been asked to serve (locality, on an engine with no locality problem) and a false statement about a question nobody had yet asked it (encoding static facts as id ranges, on an engine whose bottleneck is precisely the loads such encoding deletes). The general form, for any engine that is dependent-chain-bound rather than miss-bound: every static per-element fact currently answered by a lookup table is a candidate to be answered by the element's *position* instead — sort once at load, compare forever at run time.

7.2 Range-Prune: Class-Major Renumbering

7.2.1 The hottest table in the engine

The prune family described in the preceding chapter is implemented through a per-node mask, `PruneMask`, a 15 KB byte array with two semantically distinct bits: bit 0 marks a node as *turn-on-unsafe* (the P-1 structural safety taint — no-pull-up nodes and ForceCompute channel components, minus the cap<all-neighbours subset un-tainted by P-3/P-4), and bit 1 marks it as *turn-off-skip* (the P-2 degree-1 driverless leaf class, plus the supply rails after the supply-skip fold). Both bits are static: they are computed once from the netlist's structure at `Reset` and never change during simulation.

The consumer of this mask is the single hottest loop in the engine: the enqueue walk inside `SetNodeState`. Every time a node's state flips, the engine iterates the packed transistor list of the devices that node gates — read four `ushort` entries at a time as one unaligned `ulong` — and for each device decides whether to enqueue its channel endpoints into the next settle wave. At roughly 42.85 million pops per 100k hc, with multiple endpoint checks per pop, this decision executes hundreds of millions of times per simulated second. Before the renumbering, each decision read `PruneMask[c]` for an endpoint id *c* that is, from the cache's perspective, a random index into the 15 KB array. The array is L1-resident, so the load always hits — but it is still a 4–5-cycle dependent load feeding a conditional branch, sitting on the critical chain of the hottest loop. By the analysis of Section 7.1, it was exactly the kind of link worth deleting; it was, in fact, the *only* deletable link the loop contained.

7.2.2 Four blocks, two compares

The two prune bits define four classes, and the class-major renumbering assigns each class one contiguous id block. Ids 0 (reserved), 1, and 2 (the power and ground rails) are pinned; all remaining nodes are sorted with the prune class as the major key. The measured boundaries on the lowered NES netlist are $A = 460$, $S = 1275$, and $B = 7532$:

```
ids:  [3, A)           [A, S)          [S, B)           [B, end)
      skip ∩ unsafe    skip ∩ safe     no-skip ∩ safe   no-skip ∩ unsafe

turn-off skip(c)    ⇔  c < S            // supply ids 1,2 < 3 ≤ S ride the same compare
turn-on  unsafe(c)  ⇔  c < A || c ≥ B   // c is never a supply id on this path
```

Two details of the block ordering are load-bearing. First, both skip blocks are placed below both no-skip blocks, so the turn-off-skip test collapses to a single compare, $c < S$. Because the supply rails occupy ids 1 and 2, below the first block's start at 3, the same compare also answers "is this endpoint a supply rail?" — the historical explicit guard `c2 != ngnd && c2 != npwr`, and its successor the supply-skip mask fold, are both subsumed by the one compare for free. Second, the no-skip ∩ unsafe block is placed *last*, because the fake handler nodes created after the renumber (callback targets for the behavioral memories) are always of exactly that class — no pull-up, hence unsafe; callback, hence no-skip — and so they extend the open-ended final block without disturbing any boundary. On the measured boundaries, the blocks hold 457, 815, and 6,257 nodes respectively, with the remainder of the ≈14.7K-node id space (plus the appended handler nodes) in the final block.

Within each block, nodes need a secondary ordering key; since locality is worth approximately nothing (Section 7.1.2), the class-major sort sacrifices nothing by overriding adjacency. The initially shipped secondary key was a blind static approximation of signal flow: a BFS from the master clock node along the edges the enqueue walk actually traverses (from a node to the channel endpoints of the transistors it gates), ignoring gate states. Unreached nodes sink to their block's tail in original relative order. Section 7.3 replaces this key with a measured one.

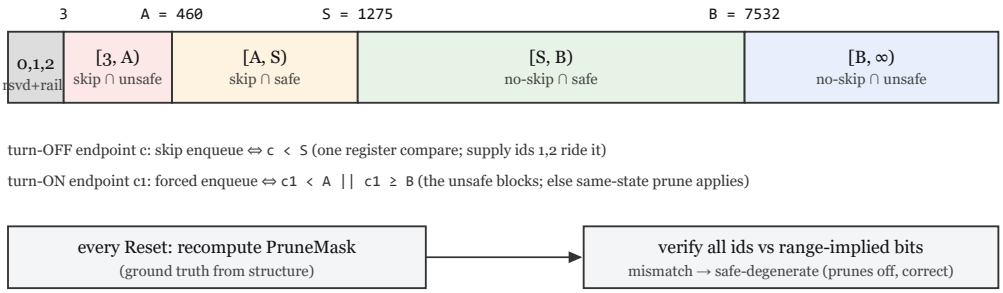

Figure 7.1. The class-major id space. Each prune class occupies one contiguous block ([3, *A*) skip∩unsafe, [*A*, *S*) skip∩safe, [*S*, *B*) no-skip∩safe, [*B*, ∞) no-skip∩unsafe; measured boundaries *A*=460, *S*=1275, *B*=7532), so the static prune facts consumed by the hot enqueue walk become register compares against the block boundaries: turn-off skip ⇔ $c < S$; turn-on unsafe ⇔ $c < A \,||\, c \geq B$. The supply rails (ids 1, 2) ride the skip compare; post-renumber handler nodes append to the final block.

7.2.3 Never trust the layout: verification and the safe-degenerate fallback

A wrong block boundary would not crash; it would silently mis-prune. The failure modes range from subtle (suppressing a re-evaluation that mattered, corrupting state divergence-style) to grotesque (treating the power rail as enqueueable and group-walking VCC). The design therefore refuses to trust the layout. The mask computation is retained, unchanged, and runs at every `Reset` as *ground truth*: the same structural classification that previously produced the runtime table now produces a reference against which the claimed ranges are verified, node by node, before the first settle wave is allowed to run:

```
// at every Reset, after the mask is recomputed from the netlist (ground truth):
ok = (3 ≤ A ≤ S ≤ B)
for nn in [3, NodeArrayCount):
    implied = nn < A ? skip|unsafe : nn < S ? skip : nn < B ? 0 : unsafe
    if (PruneMask[nn] & 3) != implied: ok = false
if (!ok):  A = ∞; S = 3; B = ∞    // safe-degenerate: prunes off, supply guarded
```

On mismatch, the engine falls back to *safe-degenerate* boundaries: $S = 3$ makes the skip compare true only for the supply rails (the mandatory guard), and $A = B = \infty$ treats every node as turn-on-unsafe. This disables the P-1/P-2/P-3/P-4 prunes entirely — the engine performs more no-op re-evaluations and runs slower — but it remains bit-exact-correct on *any* numbering, including identity numbering, hand-built selftest netlists, and any future netlist whose structure violates an assumption baked into the sort. This is the deoptimization-guard pattern of Hölzle, Chambers, and Ungar [13] transplanted from compiled code to data layout: the optimized representation is used only while a cheap runtime check certifies the invariant it depends on, and the system degrades to a slower, trivially correct representation the moment the check fails. The hot path compiles only the compare form — there is no mode branch and no flag; the degenerate case is

reached purely by the boundary values. After successful verification, the Release build frees the now-dead 15 KB mask array; since untouched memory costs nothing in cache, this is hygiene rather than speed.

7.2.4 Measured effect

In the ablation ladder of Chapter 8, the range-prune step (S4→S5, commit `51e046d`) moves the median from 112,887 to 117,671 hc/s, a +4.2% step. One caveat must be stated plainly: the ladder's S5 stage also moves the runtime from .NET 10 to .NET 11, so the step conflates the layout change with the runtime upgrade. The isolated, same-runtime, interleaved-paired measurement at ship time was +3.56% (19/20 and 11/12 paired wins across two campaigns) — at the time, the largest hot-path win since the P-1 prune itself.

The hardware counters of Chapter 8 corroborate the mechanism. The S4→S5 step cuts L1d misses from 26.3 to 13.2 MPKI — the largest single L1d movement in the entire ladder — and the AMD-native counter pass shows that L1d *accesses* per 1M hc fall from 75.6 billion at S0 to 44.1 billion at S5: the loads are not missing less, they are *gone*, which is exactly what deleting a load per endpoint check predicts. (Retired instructions tick up slightly at S5, 104.3→107.8 billion per 1M hc, and IPC dips from 2.31 to 2.25 — the win is in the chain structure, not in instruction count, a point Section 7.4 develops.) L1i MPKI rises from 0.209 to 0.284 at the same step; we attribute this to the runtime move and code-layout churn rather than to the data relayout, but the ladder cannot separate the two, and we flag it as an honest confound.

7.2.5 The boundary of applicability

The closing generalization of Section 7.1.3 — every static per-element lookup is a candidate for position-encoding — was put to the test on the next candidate on the list, and the result bounds the technique honestly. The fast-path dispatch reads a per-node class byte (the `IsPureLogic` / cls1-versus-cls2 discriminator) in the pop loop; since the class is static, the same trick applies, and a cls-range variant that folded the class into additional id blocks was built and measured the following day. It was neutral: −0.28%, 7/20 paired wins, and it was reverted. The diagnosis completes the model rather than contradicting it. The class-byte load feeds a *predicted branch* — the out-of-order core speculates past it, so the load's latency is off the critical path and deleting it buys nothing. The `PruneMask` load, by contrast, sat on the *carried* dependence chain of the enqueue walk, where every cycle of latency is a cycle of wall-clock. The resulting checklist for the technique is three-fold: the fact must be *static* (or the layout is invalidated), the site must be *hot* (or the gain is unmeasurable), and the deleted load must lie *on the carried chain* (or speculation already hides it). Range-encoding pays only when all three hold; on this engine, the prune masks were the one table that satisfied all three.

7.3 The Self-Captured First-Touch Key

7.3.1 From a static guess to a measured order

The blind clk-BFS secondary key of Section 7.2.2 is a static guess at the order in which the settle cascade visits nodes. During the range-prune experiments, an instrumented variant that used a *measured* co-activity profile as the secondary key reached +5.12% in early tests — clearly better than the blind key, but operationally unattractive: it required dumping a profile file from one run and loading it into subsequent runs, with all the staleness, mismatched-ROM, and configuration-drift hazards that offline profiles carry [12, 20]. The question that produced the final form was: if the engine can measure itself, why is there a file? The shipped design makes the engine its own profiler. At load, it builds itself twice, capturing between the two builds the true first-touch order of its own production cascade. The structure is a three-pass load (Figure 7.2):

1. **Pass 0 — classify.** Build the netlist under identity ids, run `Reset`, and let the ordinary structural classification compute the `PruneMask`. Capture each node's two prune-class bits; these are the major sort key for both subsequent passes.
2. **Pass 1 — rebuild, warm, capture.** Rebuild class-major with the temporary blind clk-BFS secondary key. Because the rebuilt layout's ranges pass the Reset-time verification of Section 7.2.3, the event prunes are *on*: the cascade being observed is the real, pruned production schedule, not an unpruned approximation. Warm the chip for 1,024 hc to get past the reset transient, then run 32,768 hc through a *cold instrumented copy* of the settle loop that records, for each node, the sequence number of its first pop. Translate the captured order back to identity ids through the pass-1 permutation.
3. **Pass 2 — final build.** Rebuild with the pass-0 class bits as the major key and the captured first-touch order as the secondary key (never-touched nodes sink to their block's tail in original relative order). This is the layout the production run executes under.

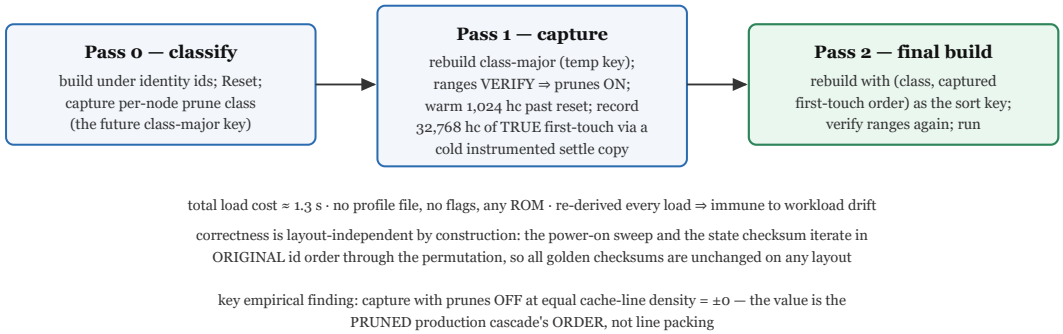

Figure 7.2. The three-pass self-capturing load. Pass 0 classifies prune classes under identity ids; pass 1 rebuilds class-major with a temporary static key, warms 1,024 hc, and captures 32,768 hc of true first-pop order through a cold instrumented copy of the settle loop (prunes verified and ON, so the observed cascade is the production schedule); pass 2 rebuilds with the captured order as the within-block key. Total load cost ≈1.3 s; the hot loop of the final build is untouched.

7.3.2 The cold instrumented settle copy

The capture instrument is worth describing because of what it deliberately is *not*. It is not a flag, a callback hook, or a conditional branch added to `ProcessQueue` — any of which would tax the hot loop of the final build to pay for a measurement that happens only at load. Instead, the capture pass drives the chip through a separate, cold copy of the settle loop that mirrors the production loop's double-buffered wave drain and clock toggle exactly, plus one line:

```
for hc in 0..32768:
    toggle clk; EnqueueNode(clk)
    while next-wave count != 0:              // mirrors ProcessQueue's wave drain
        swap wave buffers and dedup hashes
        for nn in wave:
            if RecalcHash[nn] != 0:
                if order[nn] == UNSEEN: order[nn] = seq++     // the capture
                RecalcNode(nn); RecalcHash[nn] = 0
    InvokeCallbacks(); Time++
// translate back through the pass-1 permutation:
identityOrder[orig] = order[perm[orig]]
```

The copy runs only during pass 1 and is torn down with it; the production `ProcessQueue` is byte-for-byte unaffected. The failure mode of this duplication is also bounded by construction: if the instrumented copy ever drifts semantically from the production loop, the damage is confined to a *worse locality key* — a performance regression, never a correctness defect — because the class blocks, the range verification, and all correctness machinery are independent of the secondary key.

7.3.3 Why correctness is layout-independent by construction

A renumbering scheme that required re-validating bit-exactness against behavioral test ROMs on every layout change would be operationally untenable. The engine instead maintains a structural argument that *any* permutation produces the same physical event sequence.

Property 7.1. The simulation's observable behavior is invariant under node renumbering. The event-driven core has exactly one id-order-dependent site: the power-on `RecomputeAllNodes` sweep, which enqueues every node once and whose enqueue order seeds the first settle's within-wave (Gauss-Seidel, observable) order. That sweep — like the full-state checksum — iterates in *original* id order through the stored permutation. Every subsequent wave's order is determined by the cascade itself, which is a function of the netlist's connectivity, not of node ids. A renumbered run therefore replays the identical physical event sequence and reproduces the same golden checksums as an identity-numbered run.

This property was not assumed; it was audited (the search for id-order-dependent sites is what surfaced the power-on sweep as the unique one) and is continuously re-confirmed: every layout variant in this chapter's experiments — blind key, profile key, captured key, prunes-off captures — reproduced all three golden checksums (300k/400k/1M hc) and the 10M-hc sprite-heavy SMB1 gate. The verification-plus-fallback of Section 7.2.3 covers the one place where layout does carry semantics (the range-encoded prune facts), closing the loop.

7.3.4 Cost, drift immunity, and measured effect

The entire three-pass load costs approximately 1.3 s, once, at startup — two netlist builds plus 33,792 hc of instrumented warm-up, against simulation sessions that run minutes to hours. The recurring alternative it replaced (the profile file) carried a subtler cost: staleness. An offline profile binds the layout to the workload, engine version, and configuration that produced it; the self-capture is immune to that drift *by construction*, because the key is re-derived from the actual chip, actual ROM, and actual engine binary on every load. There is no file format, no flag, and no way to feed the engine a wrong profile, since the profile no longer exists as an artifact.

A word on the capture window's representativeness. At 714,732 hc per video frame, the 32,768-hc trace observes under five percent of a single frame — it cannot, and does not attempt to, sample the workload's full behavioral diversity. What it captures is the *first-touch* order, which is dominated by the early cascade structure: the clock fan-out, the phase generators, the register and bus paths that fire within the first few thousand half-cycles of any workload. Nodes the window never reaches sink to their block's tail and contribute nothing either way. The empirical question is whether an order learned in this early window generalizes; the answer is that the +6.17% measured at 400k hc persisted unchanged through the 1M-hc checksum runs and the 10M-hc sprite-heavy SMB1 gate — the early-cascade order is evidently a stable property of the netlist's activity structure rather than of the particular boot sequence observed.

Measured: the ladder step S5→S6 (commit `3e4a571`) is +5.0% median (117,671→123,545 hc/s). The isolated paired measurement was **+6.17% (20/20 paired wins)** over the blind static key — and, decisively for the design, the self-captured key beat the *loaded offline profile* head-to-head by +4.94% (16/16) in a same-executable comparison. Online self-profiling beating stale offline profiling is the classic dynamic-versus-static PGO outcome, and it justified deleting the file mode entirely (commit `a553d38`).

7.4 Order, Not Density

The natural reading of the +6.17% would be that the captured key packs co-active nodes onto fewer cache lines — the locality story, finally vindicated. The data refute this reading, and the refutation is the most instructive measurement in the chapter. The instrument is a cache-line-density counter: the number of distinct `NodeInfo` cache lines touched per half-cycle, averaged over the run. Four secondary keys were compared under it (Table 7.2).

Table 7.2. The four-key experiment: `NodeInfo` cache-line density versus wall-clock gain (C#, full_palette, interleaved-paired, all bit-exact; measured [21]). Density and gain do not correlate; the prunes-OFF capture is the controlled contrast.

Secondary key	Lines/hc	Wall-clock vs. blind key
Blind clk-BFS (static)	118.4	baseline (+0%)
Offline co-activity profile	109.5	+1.55%
Self-capture with prunes **OFF**	110	±0.0%
Self-capture with prunes **ON** (shipped)	105.9	+6.17%

The controlled contrast is between the last two rows. A first-touch order captured with the event prunes *disabled* achieves essentially the same cache-line density as the profile key (110 vs. 109.5 lines/hc) and gains *nothing*. The same capture mechanism, run with the prunes enabled — observing the schedule the production engine will actually execute — gains +6.17% at a density (105.9) only marginally better. Density is necessary but nothing like sufficient; the active ingredient is the order's *lineage*. The key is worth +6.17% when, and only when, it is a faithful image of the pruned production cascade's first-pop sequence.

The hardware counters say the same thing from the other side. The range-prune step S5 halves L1d MPKI (26.3→13.2); the self-capture step S6 then adds its +5.0% with *no further miss reduction at all* (13.2→12.6 MPKI, marginal). Whatever S6 buys, it does not buy it by missing less. Nor does it buy it by doing less: retired instructions per 1M hc actually *rise* across the step (107.8→111.6 billion — the event counts are identical by construction, so this is code-generation churn), while IPC climbs from 2.25 to 2.39. A change that executes more instructions, misses the same, and finishes 5% sooner is spending its gain entirely on latency — the same dependent chains, traversed in an order the machine tolerates better. Combined with Section 7.1's finding that the engine is dependent-chain-bound, the picture is consistent: the gain rides on the *order* in which an L1-resident structure is walked, not on how much of it is walked, nor on how many instructions the walk retires.

> **Microarchitectural hypotheses — labelled as such.** We can state where the gain does *not* come from (cache misses, line density) with measurement behind us; where it *does* come from, we can only hypothesize, because the machine exposes no counter that resolves it (this machine's LLC counters are non-functional, and load-to-use latency distributions are not observable from software). Three non-exclusive candidates: (i) *prefetcher stream training* — when ids follow first-touch order, the cascade's accesses approximate ascending address order, which next-line and stride prefetchers recognize, converting some dependent-load latency into prefetched hits-under-miss; (ii) *memory-level parallelism* — adjacent ids for consecutively scheduled nodes let the out-of-order window's overlapping loads land on lines already in flight; (iii) *micro-window spatial density* — correlated access patterns within short temporal windows, the regime exploited by spatial memory streaming [14], reward layouts that match the schedule even when aggregate density is unchanged. All three are consistent with the signature "wall-clock moves, miss counters do not"; none is confirmed. We flag them as hypotheses, not findings.

7.5 Relation to Inspector-Executor and Trace-Driven Layout

Nothing in this chapter's toolbox is without ancestry, and we state the lineage plainly. The overall shape — run an inexpensive *inspector* over the irregular computation's actual access pattern, then reorganize data (and/or iterations) before the *executor* runs — is the inspector-executor paradigm developed for irregular scientific computation: the CHAOS runtime library [7], Ding and Kennedy's run-time data and computation reorganization [8], and Strout, Carter, and Ferrante's composition framework for run-time reorderings [9]. Ordering data by a traced access sequence is likewise established: cache-conscious structure layout and Chilimbi's stream-based abstractions [10, 11], profile-driven data placement [12], and profile-guided data layout in deployed toolchains [20]. Our verification-plus-fallback discipline borrows its pattern from dynamic deoptimization [13]. Against that background, the construction here is a *variant*, and its specific differences are five:

- **In-process and self-contained.** The simulator is its own inspector: no compiler support, no instrumentation toolchain, no profile artifact, no separate training run. The inspector is a cold copy of the executor's own inner loop, and the entire cycle — classify, capture, rebuild — completes inside one 1.3 s load.
- **An event-driven trace, not an array-access stream.** The captured object is the first-pop order of a pruned discrete-event settle schedule [4] over a switch-level netlist [1], not the index stream of an array traversal; the prunes-ON/prunes-OFF contrast of Section 7.4 shows the trace's *lineage* — its fidelity to the production schedule — is the entire payload, a distinction the locality-objective framing of [8, 10, 11, 12] does not surface.
- **A full id-space rebuild serving a non-locality objective.** The permutation rewrites the netlist's entire identifier space, and its primary purpose is semantic encoding — position-as-predicate, deleting lookups (Section 7.2) — with locality demoted to the secondary key. The

literature's reorganizations optimize placement for locality; here the measured locality value was approximately zero, and the relayout pays anyway.

- **A verified fallback.** The layout's semantic load (the range-encoded prune facts) is certified against independently recomputed ground truth at every Reset, with a safe-degenerate, still-bit-exact fallback on mismatch — a deoptimization guard [13] applied to a data layout rather than to compiled code.
- **Bit-exactness as a hard gate.** Every layout in this chapter reproduced full-state checksums; correctness is layout-independent by audited construction (Property 7.1), not by testing alone.

Each ingredient, taken alone, is known. The combination — an event-driven simulator that re-derives its own memory layout from its own production trace at every load, uses the permutation to encode its hottest static predicates as register compares, certifies the encoding against recomputed ground truth, and degrades safely when certification fails — is, to our knowledge, not described in the prior literature, and we offer it as this chapter's contribution, with the dead-end-indexing lesson of Section 7.1 as its methodological companion.

Chapter 8. Evaluation

This chapter quantifies everything the preceding design chapters claimed. Its centerpiece is an eight-stage ablation ladder, measured in a single day on a single machine, in which each historical commit of the engine is rebuilt from source and benchmarked under an identical protocol; the ladder is then re-examined under hardware performance counters and placed against the other public simulators of the same netlist family [15, 16, 17, 18, 19]. We give the methodology unusual space — more than is customary for a systems evaluation — because one of the project's most instructive findings is itself *methodological*: a change that cut data-cache misses by 67% turned out to buy only single-digit wall-clock improvement. That result would have been silently misreported under a less paranoid protocol, and it reshapes how the engineering results of Chapters 5–7 should be interpreted. Throughout, every number is a measurement taken in this project; where a result is negative, conflated, or weaker than its first publication suggested, we say so.

8.1 Environment and methodology

8.1.1 Machine, workload, and verification gates

All measurements were taken on one machine: an AMD Ryzen 7 3700X (Zen 2 microarchitecture, 8 cores / 16 threads, 32 KB L1 data and 32 KB L1 instruction cache per core, 512 KB private L2 per core, 32 MB shared L3) running Windows 11. The C# engine runs on .NET 10 or .NET 11 depending on the stage under test — each historical commit is built against the runtime it shipped with, a choice we return to in Section 8.2. Unless otherwise noted, all data in this chapter were collected on 2026-06-12.

The workload is `full_palette.nes` (a community open-source NES test ROM displaying the full NES palette as a static screen, from the public nes-test-roms collection), a NROM test ROM that exercises the full rendering pipeline of the 2C02 alongside continuous 2A03 execution. Throughput runs simulate 400,000 master-clock half-cycles (hc); hardware-counter runs simulate 1,000,000 hc to amortize sampling startup. Exactly one benchmark process runs at a time, with no other foreground load on the machine. Per stage we report the median across rounds, with the per-stage best given as an ancillary figure; we never report a single run.

Every run — without exception — is gated on the full-state FNV-1a checksum over all node states: 0x9174E19D961CB6E5 at 400,000 hc for the throughput runs and 0x6D4CCBCE2E9CD599 at 1,000,000 hc for the counter runs (the 300,000 hc golden 0x794A43ABDF169ADA and a 10,000,000 hc sprite-heavy *Super Mario Bros.* gate guard adoption decisions throughout the project's history). The gate serves two purposes. First, it enforces the bit-exactness contract of Chapter 2 at evaluation time: a stage that produced a fast but wrong simulation would be excluded automatically. Second, and more mundanely, it guards against silently misbuilt binaries

when historical commits are reconstructed — a stale artifact or a wrong-runtime build either reproduces the golden state trajectory or it does not. All 72 ladder runs of Section 8.2, all attribution runs of Section 8.3, and all counter runs of Section 8.4 passed their gates.

8.1.2 Rebuilding history: worktrees and commit identities

The ablation ladder does not re-implement earlier versions of the engine; it rebuilds them. Each stage is a real commit from the repository history [21], checked out into an isolated git worktree and compiled from source. This removes an entire class of ablation artifacts — "the baseline with feature X disabled" is notoriously unrepresentative of the code that actually existed before feature X — at the cost of one conflation we discuss honestly in Section 8.2.2: a commit may carry incidental changes beside the named technique. The stage commits, in ladder order, are: a80dab4~1 (S0, the baseline fork), a80dab4 (S1, the R-1 dynamic-singleton path), ed8c457 (S2, the P-1 same-state prune), 6bdc25b (S3, the P-2/P-3/P-4 prunes plus mask consolidation), 00ab4fa (S4, clause reordering and the supply-skip fold), 51e046d (S5, the range-prune renumbering, which also moved the project to .NET 11), 3e4a571 (S6, the self-captured first-touch key), and 2749105 (S7, the B1 pair path).

8.1.3 Variance-control protocol

Modern desktop processors do not run at one frequency: the 3700X opportunistically boosts well above its 3.6 GHz base clock, with the achieved frequency depending on temperature, recent load history, and power-management state. All measurements in this chapter are taken with the clock unlocked, and variance is controlled by protocol: exactly one benchmark process at a time; all stages of an experiment interleaved *round-robin per round* within a single day, so that each round executes every stage once in rotation and slow thermal drift debits every stage equally; medians across rounds; a full-state checksum gate on every run; and historical commits rebuilt verbatim in isolated worktrees (Section 8.1.2). For effects below roughly 1%, we use the paired interleaving described in Section 8.6. Threads are left unpinned in the ladder for uniformity across historical stages (the opt-in affinity tool postdates several of them); its measured effect on variance is reported in Section 8.6.

8.2 The ablation ladder

8.2.1 Results

Table 8.1 and Figure 8.1 present the ladder: nine rounds over eight stages — 72 runs, all checksum-gated — at 400,000 hc per run, measured in a single session on 2026-06-12. Each row adds one technique family to the previous row; the "step" column is the median-to-median improvement over the preceding stage.

Table 8.1. The ablation ladder: 9 rounds × 8 stages, round-robin interleaved, 400,000 hc per run, all 72 runs gated on the full-state checksum (2026-06-12). "TFM" is the target framework each commit shipped with. Throughputs in half-cycles per second.

Stage	Commit	Adds	TFM	Median hc/s	Best	Step
S0	a80dab4~1	baseline fork (static fast path, SoA layout)	net10	67,955	69,677	—
S1	a80dab4	+ R-1 dynamic singleton	net10	80,657	82,806	+18.7%
S2	ed8c457	+ P-1 same-state prune	net10	101,964	103,818	+26.4%
S3	6bdc25b	+ P-2/P-3/P-4 + mask consolidation	net10	109,841	113,571	+7.2%
S4	00ab4fa	+ clause reorder + supply-skip fold	net10	112,887	114,833	+2.8%
S5	51e046d	+ range-prune (class-major renumber; + .NET 11)	net11	117,671	119,464	+4.2%
S6	3e4a571	+ self-captured first-touch key	net11	123,545	126,781	+5.0%
S7	2749105	+ B1 pair path	net11	132,243	135,828	+7.0%

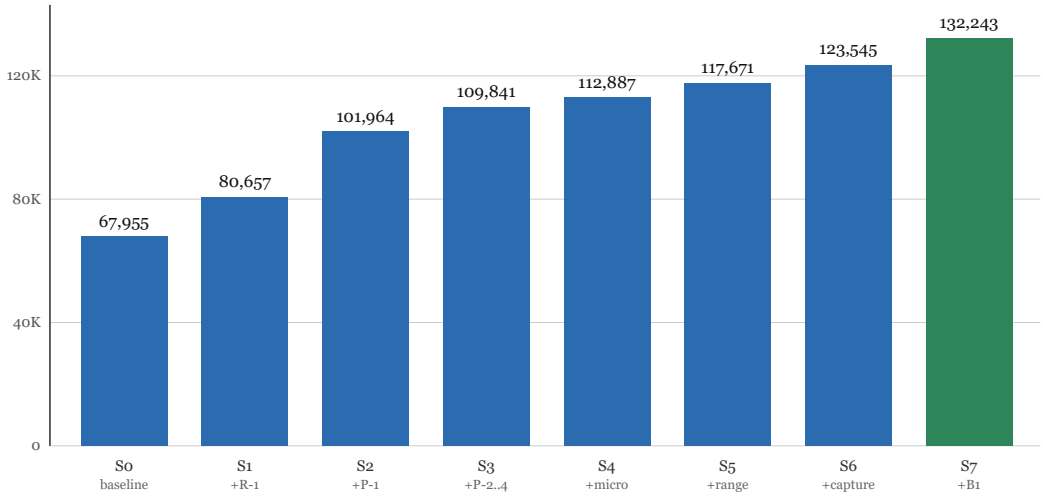

Figure 8.1. The ablation ladder: median throughput per stage (hc/s, 400,000 hc runs, 9 rounds round-robin, 2026-06-12). Every bar is bit-exact: all 72 runs reproduce the golden full-state checksum.

> **Cumulative result.** S0 → S7 is +94.6% in median throughput — the engine nearly doubled — on the same machine, the same day, under the same protocol, with every run reproducing the golden checksum. No stage trades correctness for speed; the bit-exactness contract holds across the entire ladder.

8.2.2 Stage-by-stage commentary

S0 — the baseline is not naive. The 67,955 hc/s starting point already incorporates the unmanaged structure-of-arrays layout, the double-buffered wave lists with deduplication hashing, and the static fast path for cls1 nodes — the 3,929 nodes (26.7% of the netlist) with no pass channels at all, whose resolution never requires a group walk. S0 is, in family terms, already faster than every external simulator we measured (Section 8.5). The ladder therefore measures the techniques of this monograph against a competent event-driven engine, not against a strawman.

S1 — R-1 dynamic singleton, +18.7%. R-1 extends the singleton fast path from the statically isolated cls1 nodes to the 10,784 cls2 nodes (73.2% of the netlist) whose pass transistors merely happen to be all OFF at the moment of evaluation: a single check proves the conducting group is $\{n\}$ and the resolution collapses to an O(1) table lookup. The magnitude is explained by the event profile: roughly 70% of all node re-evaluations resolve as singletons, so R-1 converts the most common case of the hottest loop from a BFS-plus-LUT walk into a handful of instructions. The counter data (Section 8.4) adds a nuance: S1 *lowers* IPC slightly (2.11 → 2.08) and raises per-instruction miss rates, while cutting retired instructions by 17% — fewer instructions remain, and the survivors are more latency-bound. The trade is decisively positive, but it foreshadows the regime the rest of the ladder operates in.

S2 — P-1 same-state prune, +26.4%, the largest single step. P-1 suppresses, before enqueue, the turn-on events whose two channel endpoints already agree in state, under the structural safety taint of Chapter 5 (no-pull-up nodes and ForceCompute channel-components are excluded). Its magnitude makes sense against the waste profile: 80.1% of all queue pops resolve to no state change, so any sound mechanism that deletes a slice of provably-null events removes not just the resolution but the pop, the deduplication-hash traffic, and the downstream enqueue attempts that each dead event drags along. We note for honesty that P-1's naive form — without the taint — diverged visibly (a black screen within frames): the prune family's value is inseparable from its safety analysis, since the floating tie-break that the taint protects *is* the chip's storage mechanism [1].

S3 — P-2/P-3/P-4 and mask consolidation, +7.2%. This stage adds the turn-off isolation prune for degree-1 driverless leaves (P-2), the capacitance-dominance un-taint that recovers provably tie-break-immune nodes for P-1 (P-3), its multi-channel generalization (P-4), and the consolidation of the safety arrays into one bit-packed mask. At their individual adoption commits these measured +1.4%, +5.96%, +1.71%, and +0.64% respectively — a multiplicative composition of roughly +10%, against the ladder's +7.2%. We report both numbers and trust the ladder: the adoption measurements were taken on different days against moving baselines, and same-day round-robin composition is the authoritative protocol. Together the P-family deletes approximately 21% of all node re-evaluations at zero hot-path cost, all decisions being precomputed at load time.

S4 — micro-optimizations, +2.8%. Two changes share this step: reordering the conjuncts of the hot `SetNodeState` condition by measured selectivity (a profiler showed the leading clause was true ~97% of the time and thus nearly useless as a gate), and the supply-skip fold, which

marks the power and ground rails so the turn-off walk's two channel endpoints become a uniform unrolled loop. Each measured near +1–2% at adoption. This is the ladder's reminder that conventional micro-tuning still pays — but at a fraction of what the structural techniques deliver.

S5 — range-prune, +4.2%. The class-major renumbering of Chapter 7 reorders the node id space so that the prune classes become contiguous id blocks (boundaries A = 460, S = 1275, B = 7532 on this netlist), turning the per-event safety lookups from dependent memory loads into register comparisons against constants. The counter signature of this stage is dramatic — L1d misses fall by half per instruction (Section 8.4) because class-major layout also densifies the hot set — yet wall-clock moves only +4.2%, the first strong evidence for the latency-bound interpretation developed below.

> **Conflation caveat.** S5 is the one stage whose commit carries a second change: the move from .NET 10 to .NET 11. The ladder honors each commit's shipping runtime, so the +4.2% step bundles technique and runtime. The technique's own adoption measurement — paired and interleaved against its immediate parent — was +3.56%, which bounds the runtime upgrade's contribution to a small residual. We did not re-litigate the split further; readers should treat the S4 → S5 step as "range-prune plus runtime," with the technique responsible for most of it.

S6 — self-captured first-touch key, +5.0%. The three-pass load (classify; rebuild and warm; capture 32,768 hc of true first-pop order through a cold instrumented copy of the settle loop; rebuild again) re-derives the node id space from the production cascade's own access order, at ~1.3 s of load cost. Its counter signature is the inverse of S5's: essentially no further miss reduction, yet a wall-clock gain larger than S5's. Order, not density, is the active ingredient — the central finding of the relayout work, and one that distinguishes it from the cache-conscious-layout literature whose objective function is the miss count [10, 11, 12].

S7 — B1 pair path, +7.0%. B1 extends cardinality specialization from groups of size one to groups of size exactly two: when a node has exactly one ON gate and the neighbor's ON channels all lead back to the seed, the pair {seed, neighbor} is resolved inline, replicating the general walk's member-hash clear, supply scans, tie-break form, and write order byte for byte. The population justifies the gain: 77.1% of all group walks are size-2, and the B1-eligible subset covers 19.5% of all pops. After separating the attribution from the adjacent housekeeping commit, the effect is about +8.9% (Section 8.3).

8.3 Attribution methodology

A ladder built from historical commits inherits history's untidiness: between two named stages may sit a housekeeping commit unrelated to either technique, and if that commit had its own performance effect, the ladder would silently attribute it to the named technique. The general rule we adopted: a ladder step is only attributed to a named technique after every interposed commit has been either shown neutral by direct interleaved measurement or absorbed into the step's

description. This is laborious — it roughly doubles the measurement budget around contested steps — but it is the only defense against the most common failure mode of ablation studies, which is not bad statistics but bad *provenance*. Applying this procedure to the single commit interposed between S6 and S7: direct interleaved measurement shows it to be neutral, and with the attribution separated, B1's effect is about +8.9%.

8.4 Hardware-counter study

8.4.1 Counter methodology

We profiled every ladder stage under Windows ETW performance-monitoring-counter sampling at 1,000,000 hc per run, with sampling intervals of 1,048,576 events for instructions and cycles and 65,536 for cache misses and branch mispredicts; event totals are reconstructed as samples × interval. Sampling adds roughly 5% overhead, so the deliverable of this section is *ratios* — MPKI, IPC, instructions per simulated half-cycle — not absolute throughput, which Table 8.1 already provides uncontaminated. Table 8.2 and Figure 8.2 give the per-stage profile; Table 8.3 gives a second pass on three stages using AMD-native cache events (data-cache accesses, instruction-cache fetches and misses), which yield access counts and miss ratios rather than per-instruction rates and serve as an independent cross-check.

Table 8.2. Hardware-counter profile per ladder stage (ETW PMC sampling, 1,000,000 hc per run, 2026-06-12). MPKI = misses per thousand retired instructions; the final column is total instructions retired per 1M simulated half-cycles, in billions.

Stage	IPC	L1i MPKI	L1d MPKI	Branch-mispredict MPKI	Instr. retired (B / 1M hc)
S0	2.11	0.169	27.3	3.27	155.6
S1	2.08	0.192	30.6	3.37	128.7
S2	2.38	0.187	24.5	2.74	116.0
S3	2.35	0.194	25.8	2.52	105.7
S4	2.31	0.209	26.3	2.72	104.3
S5	2.25	0.284	13.2	2.87	107.8
S6	2.39	0.274	12.6	2.62	111.6
S7	2.34	0.292	13.0	2.68	104.4

Table 8.3. AMD-native cache-event pass on three stages (1,000,000 hc per run): absolute access counts and miss ratios, independent of the per-instruction normalization of Table 8.2.

Stage	L1d accesses (B)	L1d miss ratio	L1i fetches (B)	L1i miss ratio
S0	75.6	5.6%	3.8	0.59%
S5	44.1	3.2%	3.2	0.74%
S7	40.7	3.3%	4.6	0.63%

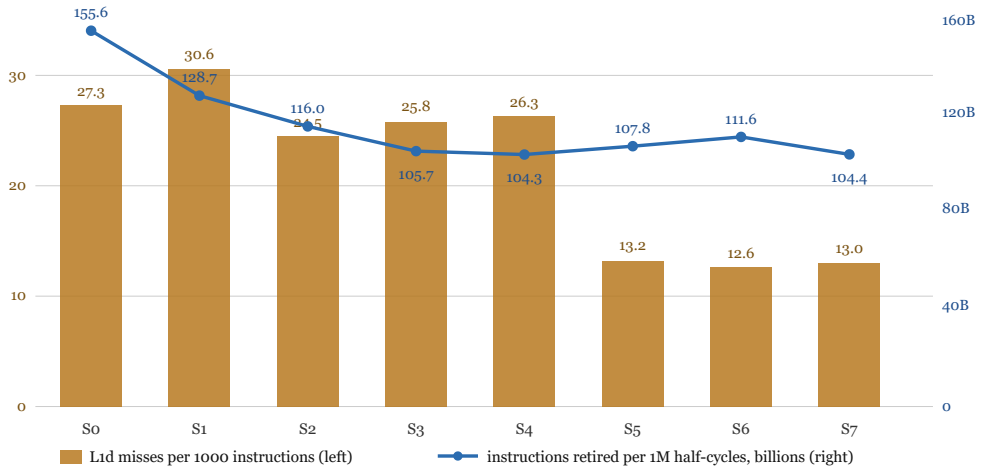

Figure 8.2. Counter trajectories across the ladder: L1d misses per thousand instructions (bars, left axis) against total instructions retired per 1M hc (line, right axis). The miss rate halves at S5 while wall-clock moves +4.2%; the instruction count falls monotonically by a third across the ladder.

8.4.2 Finding 1: instruction fetch is free — why interpretation beats compilation here

At every stage, L1i misses sit between 0.169 and 0.300 MPKI — and the native pass confirms the picture in absolute terms: of 3.2–4.6 billion instruction fetches per million half-cycles, only 0.59–0.74% miss the 32 KB instruction cache. The entire settle loop — dispatch, fast paths, group walk, resolution LUT, both prune walks, and as of S7 the inline pair path — fits comfortably in L1i. Code footprint does grow as fast paths are added (i-fetch MPKI nearly doubles from S0 to S7), but from a base so low that it never approaches relevance.

> **Finding 1.** The interpreter pays effectively nothing for instruction delivery. This is the counter-level explanation for a result that initially surprised us at the systems level: every compiled or ahead-of-time variant of this simulator lost, and lost badly — generated-code backends 3–6× slower, compiled cone evaluation 45–84× slower (Chapter 9). Compiled switch-level simulation in the COSMOS tradition [2] trades data-driven dispatch for straight-line code volume; on a flat 26.8K-transistor netlist that trade explodes the code working set precisely where this engine pays nothing, while the data-side costs — which the counters show are the real ones — remain untouched.

8.4.3 Finding 2: a 67% miss cut buys single digits — the engine is latency-bound, not miss-bound

The S4 → S5 transition halves the per-instruction L1d miss rate, 26.3 → 13.2 MPKI. Compounded with the ladder-long instruction reduction, absolute misses fall by roughly two thirds: 27.3 MPKI × 155.6 B instructions at S0 is about 4.25 billion misses per million half-cycles, versus about 1.42 billion at S5 — a −67% absolute cut, independently corroborated by the native pass (75.6 B accesses at 5.6% versus 44.1 B at 3.2%, i.e. 4.2 B versus 1.4 B misses). The wall-clock response to this two-thirds reduction in cache misses: +4.2%.

Conversely, S5 → S6 — the self-captured first-touch key — moves the miss counters almost not at all (13.2 → 12.6 MPKI) and delivers +5.0%, a *larger* gain than the stage that halved the miss rate. The order-versus-density experiment of Chapter 7 showed the same dissociation at the source level: a captured key applied with the prunes disabled reaches the same cache-line footprint per half-cycle (110 versus 105.9 NodeInfo lines/hc) yet yields ±0.0%, while the same capture taken with the prunes enabled yields +6.17%. Equal density, different order, opposite outcomes.

> **Finding 2.** This engine is bound by the *latency of dependent-load chains*, not by the *count* of cache misses. Misses that overlap with other work are cheap; what costs is the serial chain — node id → NodeInfo → transistor list → neighbor state — whose links cannot be issued until their predecessors resolve, and which hardware prefetching cannot anticipate because the address sequence is data-dependent (the streaming-predictor literature explicitly targets the complementary, spatially correlated case [14]). The optimizations that pay are those that *delete links from the chain* (range-prune replaces a mask load with a register compare; B1 replaces a walk with inline endpoint reads) or that schedule the surviving chain in the order the pruned cascade will actually traverse it (the self-captured key). This inverts the objective function of the classical cache-conscious layout literature [10, 11, 12] and of profile-guided data placement [20], which optimize miss counts; our relayout machinery is structurally an inspector–executor [7, 8, 9], but what the inspector must capture is the execution *order*, not the affinity graph.

8.4.4 Finding 3: the prunes delete instructions, not just events

Total retired instructions fall monotonically from 155.6 to 104.4 billion per million half-cycles across the ladder — a 33% reduction — while IPC holds in a band of 2.08–2.41. Absolute branch mispredicts fall further still: from roughly 509 million per million half-cycles at S0 (3.27 MPKI × 155.6 B) to roughly 280 million at S7, a −45% cut. The event-suppression machinery is thus visible at the architectural level as genuine work deletion: P-1 through P-4 remove about 21% of re-evaluations before they are enqueued, and each deleted event takes its pop, its resolution, its hash probes, and its mispredicted branches with it. This is Ulrich's activity-exclusive principle [4] pushed one step further: classical event-driven simulation refuses to evaluate what has not changed; the prune family refuses to *enqueue* what provably cannot change.

8.4.5 IPC across the ladder, and the LLC limitation

The IPC column rewards reading stage by stage. S1 dips IPC (2.11 → 2.08) while cutting instructions 17% — R-1 removes easy, well-predicted walk instructions and leaves a denser residue of chain-bound ones; a per-instruction miss-rate *rise* (27.3 → 30.6 MPKI) accompanies an absolute miss *fall*, a reminder that per-instruction normalization misleads when the denominator moves. S2 lifts IPC to 2.38 on the strength of the mispredict cut (3.37 → 2.74 MPKI): the deleted no-change pops were precisely the poorly predicted ones. S5 dips IPC to 2.25 *despite halving misses* — consistent with the latency-bound reading: the misses removed were not the stalls that gated retirement. The band's overall level, 2.1–2.4 IPC on a Zen 2 core for a pointer-chasing interpreter, indicates the fast paths keep the pipeline substantially fed; this is not an engine idling at 0.5 IPC waiting on DRAM.

> **Limitation.** This machine exposes no functioning last-level-cache counter source under our ETW configuration, so we cannot directly report L3 or DRAM traffic. Two indirect arguments bound the omission: the engine's hot data set is approximately 15 KB, comfortably L1-resident, and the native-event pass shows L1d miss ratios of 3.2–5.6% feeding a 512 KB private L2 — leaving little plausible traffic beyond L2. We report the gap rather than paper over it.

8.5 The family comparison

AprVisual belongs to a small family of simulators built on the same die-derived netlists [15, 16]. On 2026-06-08 we compiled and measured the public members headless on this machine, same ROM, same unit (whole-console master-clock half-cycles per second) — none of these original projects ships an hc/s throughput measurement of its own, so we experimentally instrumented their source to emit output in the same unit (details in Section 2.4) — to place the engine against its relatives rather than against incommensurable published figures. Table 8.4 gives the result.

Table 8.4. The netlist family on one machine, one ROM, one unit (external members measured 2026-06-08; AprVisual is the S7 engine of Table 8.1). perfect6502 is included for completeness but is not rankable on this axis: it simulates a lone 6502 with no PPU and no charge model, a different unit of work.

Simulator	Language / lineage	Throughput	Relative to AprVisual
VisualNes [19]	C++, literal chipsim.js port	~24K hc/s	~5.6× slower
perfect6502 [18]	C, optimized 6502-only rewrite	~29K (6502-only unit)	not comparable
MetalNES [17]	C++, the direct ancestor	~54K hc/s	~2.5× slower
AprVisual [21]	C#	~136K hc/s (current record)	—

Two readings of this table matter. First, the 2.5× over MetalNES is the honest measure of this monograph's contribution, because MetalNES is the engine we ported: the gap is exactly the ladder of Table 8.1 plus the baseline-fork work, with no algorithmic-semantics differences to confound it. A managed-language implementation beating its optimized C++ ancestor by 2.5× — while adding a bit-exactness regime the ancestor does not have — empirically rebuts a widespread impression in performance-engineering practice, for which we found no formal source: that managed languages are unsuited to a pointer-chasing core of this kind. Second, the absolute scale: 136K hc/s is an equivalent silicon master clock of about 68 kHz (the master clock ticks every second half-cycle), a 6502 core clock of about 5.7 kHz (hc/24), and a PPU pixel clock of about 17 kHz (hc/8). Against the real console's 42,954,552 hc/s, the gap is approximately 316×. We state this plainly: real-time switch-level simulation of the NES remains far out of reach on one core, and Chapter 9 argues from measured negative results that no software route we tried — compilation, GPU offload, bit-parallelism, threading — closes it.

8.6 Measurement hygiene

We close with the measurement practices the project converged on, because several of this chapter's results are smaller than the noise floor of a casual benchmarking setup and would be unreproducible without them.

Variance control without frequency distortion. The opt-in `--pin` mode (thread affinity to one physical core, high process priority, EcoQoS disabled — frequency deliberately untouched) halves run-to-run variance, coefficient of variation 1.5% → 0.94%, with bit-exact output. It is off by default so that published leaderboard numbers [21] reflect an unprivileged process, and it was off in the ladder for cross-stage uniformity; it is the right tool for adoption-time A/B where its variance reduction directly buys statistical power.

Thermal honesty. The same binary on the same machine spreads about 10% between a cool morning and a heat-soaked afternoon. Cross-day comparisons of raw throughput are therefore meaningless at the effect sizes this chapter deals in; every comparative claim here is same-day, and the ladder's stages were additionally interleaved round-robin so that intra-session drift debits all stages equally rather than whichever happened to run last.

Paired interleaving for sub-1% effects. Batched A/B — all runs of A, then all runs of B — embeds time-correlated drift directly into the contrast and repeatedly misled us. The adopted protocol prebuilds both binaries, alternates base/experiment order each round, and reports medians, trimmed means, and the paired win count (reported throughout this monograph as "k/n"). It earned its keep concretely: an early fast-path variant read as ambiguous under batching and resolved decisively under pairing.

The discipline in summary. Unlocked frequency; same-day round-robin interleaving within a round; one benchmark process at a time; medians reported; paired interleaving for sub-1% effects; a checksum gate on every run. None of these steps is novel; the discipline of applying all of them, every time, is what gives the ladder of Table 8.1 its claim to be believed.

Chapter 9. Negative Results and the Single-Core Boundary Map

The preceding chapters reported the techniques that worked: the dispatch fast paths, the event prunes, the identifier-space transformations, and the cumulative +94.6% they produced. This chapter reports everything that did not work, and argues that the failures are not residue but the project's primary scientific product. An optimization ledger that records only wins describes a trajectory; one that records the losses, with their populations, mechanisms, and the conditions under which each was measured, describes a *boundary* — the shape of the region within which a bit-exact, event-driven, single-core switch-level simulator of this netlist can be made faster, and beyond which it cannot. We present that boundary map in seven parts: why we treat negative results as first-class artifacts (§9.1); the abstraction routes and the reducibility paradox (§9.2); the parallelization attempts and their four structural walls (§9.3); the maintained-state floor, told through the most instructive single failure in the project (§9.4); a census of the smaller dead ends with the measured populations that closed them (§9.5); a catalogue of the recurring anti-patterns behind them (§9.6); and the resulting boundary statement (§9.7).

9.1 Why negative results are first-class here

The project's documented outcome is, by its own framing, a falsifiable negative result: under the stated contract — full Bryant switch-level semantics [1], per-node bit-exactness against three golden checksums plus a 10-million-half-cycle sprite-heavy gate, one CPU core — real-time simulation of this netlist is unreachable, and the routes by which one might have hoped to reach it are individually closed by measurement. A claim of that form is only as good as the discipline behind it. Two properties of our methodology are what make the negatives in this chapter trustworthy rather than anecdotal.

First, every negative is a *measured* negative. Each failed technique was either implemented and benchmarked under the same interleaved-paired protocol as the wins (alternating builds within each round, medians over many rounds, checksum-gated every round), or was killed by a population measurement taken before implementation — a profiler count showing that the events the technique would have saved are too few, too cheap, or too diffuse to pay for it. We distinguish three grades of negative throughout: a measured loss under stated conditions (e.g., the IR interpreter's −2.5%); a structural argument backed by measurement (e.g., the cone-compression census showing no coarser granularity exists); and a soundness failure, where the technique does not merely lose but diverges from the golden model (e.g., live-state derivation of the pull-up pin in §9.4, or under-settling in §9.5).

Second, every negative carries its scope, because the project has been burned by overgeneralizing them. The "ceiling" of the engine was itself declared four times — at roughly 80K, 100K, 120K, and 127K half-cycles per second — and overturned four times, most decisively when the R-1 dynamic-singleton path added +18.7% to a configuration we had pronounced exhausted. Several

sub-3% negatives measured under a batched A/B protocol were later found suspect when one of them (a branchless fast-path scan) flipped to a small win under interleaved pairing; and a renumbering "dead end" flipped to +3.6% when its objective was changed from cache locality to deleting dependent loads (Chapter 7). The honest form of a negative result is therefore not "X is impossible" but "X, in this form, on this engine, loses by this much, for this mechanism" — and the mechanism is the reusable part.

9.2 The abstraction routes

The natural response to a slow interpreter is to raise the abstraction: build an intermediate representation, levelize it, compile it. This is the lineage of COSMOS [2], which compiled Bryant's switch-level model into Boolean evaluation programs and won decisively over interpreted simulation on the synchronous MOS circuits of its era. We attempted every rung of that ladder on this netlist. Every rung was slower than the event-driven interpreter, and the reasons compose into a single structural statement.

9.2.1 The IR interpreter: correct, covered 23% of live nodes, and −2.5%

The first attempt was deliberately conservative: extract the cleanest pure-logic nodes into per-node truth tables, dispatch those through an IR evaluator, and leave everything else on the switch-level path. The extraction was bit-exact — the whole-NES checksum was unchanged — and covered 3,390 nodes, 23% of the live netlist. Measured interleaved-paired, it lost: −2.54% median, zero of nine rounds won.

The mechanism is the important part. The nodes clean enough to extract into truth tables are precisely the nodes the switch-level engine already resolves through its O(1) singleton fast path: a handful of instructions and one lookup in the 256-entry priority LUT. The IR evaluation — read the gate states, pack an index, chase a gate list, read a table — is *heavier* than the thing it replaced. Meanwhile the nodes the engine resolves slowly (multi-node conducting groups, charge-sharing buses) are exactly the ones that cannot be expressed as a Boolean function of fixed inputs and so cannot be lifted into the IR at all. The abstraction layer therefore wraps the already-minimal work in overhead while leaving the expensive work untouched. A clean, correct IR, on this workload, is pure cost.

9.2.2 Compiled evaluation: 3–6× to 84× slower, and the i-cache inversion

The next rung is compilation. Two independent efforts reached it. The earlier (S4-era) ahead-of-time batch backends — C# code emission, an LLVM path, bit-sliced and dense variants — were all measured 3–6× slower than the event-driven engine, for an algorithmic reason that no code quality can fix: a batch backend re-evaluates the order of 14.7K nodes per half-cycle when only a few hundred change. The compiler is being asked to out-execute a 30–150× redundancy in work performed, and it cannot.

The later (Escape-1) effort measured the oblivious route end to end on the extracted-logic model. Interpreted oblivious evaluation — sweep all ~6,000 extracted nodes to a fixed point, ~6.5 relaxation iterations per half-cycle because the bidirectional pass-transistor network cannot be levelized — ran at 539 µs per half-cycle against the golden engine's 12.0 µs: 45× slower, performing ~39,000 node evaluations per half-cycle where the event-driven engine performs a few hundred. Compiling that same sweep with Roslyn into straight-line code made it *worse*: 1,004 µs per half-cycle, 84× slower. The compiled sweep unrolls ~6,000 node evaluations into roughly 700 KB of machine code streamed 6.5 times per half-cycle — more than an order of magnitude beyond the 32 KB L1 instruction cache — so the processor front-end stalls re-fetching code while the execution units idle. Compilation of an oblivious sweep is self-defeating precisely because obliviousness means the code footprint scales with the whole circuit.

This "i-cache inversion" — compiled slower than interpreted — is the single most diagnostic data point on the abstraction ladder, and the 2026-06-12 hardware-counter ablation (Chapter 8) supplies its converse. Across all nine measured stages of the event-driven engine's own history, L1 instruction-cache misses sit between 0.169 and 0.300 per thousand instructions: the fully-inlined settle loop fits the instruction cache with two orders of magnitude to spare. The event-driven interpreter is a tiny program over data; the compiled sweep is an enormous program over the same data. On a workload whose data footprint (~15 KB hot set) is already cache-resident, trading a resident code loop for a streaming code footprint is strictly a loss.

Could a *selective* compiler have won — compiling only the hot multi-node groups, as a tracing JIT compiles hot paths? We profiled for exactly this before writing a backend, and the profile killed it pre-implementation. Over 300K half-cycles (181.4M re-evaluations on the engine of that date), 69.5% of events took the O(1) fast path that compilation cannot improve; the remaining 30.5% — the group walks — were spread over 4,752 distinct nodes with a nearly flat distribution: the hottest single node accounted for 0.14% of group-walk work, the top 10 for 1.2%, the top 50 for 6.0%, and the top 200 for only 21.6%. Covering even half the group-walk work would require compiling thousands of node-specific bodies, recreating the 700 KB footprint piecemeal; and with conducting groups averaging 2.25 nodes (now 1.13–1.4 on the pruned engine), there is almost nothing inside a group for straight-line code to save. Both prerequisites of a tracing-compilation win — concentration and depth — are absent.

9.2.3 Cone compression: the 1.1× census

If dense evaluation loses and per-node compilation has nothing to grab, the remaining abstraction is to keep event-driven sparsity but coarsen the event unit: replace node-level events with combinational-cone-level events, bounded at latch and bus boundaries, hoping for an order-of-magnitude fewer, atomic events. We measured the available compression directly rather than building the scheduler first. Condensing the combinational nodes by channel connectivity yields 2,737 cones with a mean size of 2.0 nodes; replaying a trace, 90 node-level events condense to 81 cone-level events — a structural compression of 1.1×. Widening the cone definition to include gate connectivity does not produce intermediate granularity: the netlist collapses into one giant 5,349-node cone, and firing it is oblivious evaluation again. There is no knee in between.

> **The granularity finding.** A combinational cone in this netlist averages 2.0 nodes; the event-driven engine's mean conducting group is 1.13–1.4 nodes. The interpreter already operates at the netlist's natural minimum granularity. There is no coarser reusable structural unit to extract, and therefore no abstraction layer that can batch work the interpreter is doing piecemeal — the work is not piecemeal by implementation accident but by circuit structure.

9.2.4 The reducibility paradox and its resolution

The sharpest way to state the abstraction wall is as a paradox. The Escape-1 pipeline proved, fully automatically and falsifiably, that ~98.9% of the chip's dynamic activity is reducible to Boolean logic plus registers, with only ~1.1% genuinely analog (charge-sharing buses, dynamic storage); an idealized Amdahl ceiling of 88× sits on top of that result. Automatic transistor-to-gate-level logic extraction, and subcircuit recognition via subgraph isomorphism and pattern matching, are established practice [5][6]; our physics-property-based identification (pull-up flags, capacitance comparisons, channel components) and its verify-then-enable discipline differ in using the golden simulator as an oracle rather than library or structural-pattern matching. Yet every evaluation strategy built on the extracted logic — interpreted, compiled, cone-batched — was slower than the switch-level engine it abstracts, by factors from 1.0 to 84. Reducibility did not translate into speed.

The resolution has three legs, all measured. First, the reducible fraction is reducible *per node* but not *levelizable*: NMOS pass transistors are bidirectional (which side is "input" is decided at runtime by which side drives), and the two-phase transparent latches make the transparent paths cyclic, so 94% of the dependency graph is one bidirectional SCC and the extracted logic is structurally condemned to ~6.5 relaxation iterations. Even an idealized dense sweep at 2.5 cycles per node costs ~97,500 CPU cycles per half-cycle — about twice the event-driven engine's budget — before any implementation inefficiency. Second, the event-driven engine's per-event cost is already at the memory-latency floor (Chapter 8): there is no interpretation overhead for a compiler to remove, in the sense of Ulrich's original argument that simulating only activity is the dominant saving [4] — the engine descends from exactly that tradition through chipsim.js [15] and MetalNES [17], and the activity rate is ~3% of nodes per half-cycle. Third, the i-cache counters above show the interpreter pays nothing for being a loop, while every compiled variant pays heavily for being a program the size of the circuit.

For completeness, the GPU data point: a single simulated instance, mapped to the GPU as one workgroup (the settle wave is a serial dependency chain, so one instance cannot use more), ran 10.7× slower than the CPU engine while occupying roughly 1/76 of the device. Many independent instances would be the natural GPU shape — but multi-instance throughput was never the project's goal, and it does nothing for the latency of one bit-exact instance.

Table 9.1. The abstraction routes, measured. "Golden" is the event-driven switch-level engine of the date of each comparison; all variants that ran were validated bit-exact (or behaviorally exact on covered nodes, for the Escape-1 model) before timing.

Route	Measured outcome	Failure mechanism
Hybrid IR interpreter (truth-table nodes)	−2.5% (0/9 rounds)	extractable nodes are exactly the O(1) fast-path nodes; the IR wraps minimal work in overhead
AOT batch backends (C# emit, LLVM, bit-sliced)	3–6× slower	re-evaluates ~14.7K nodes/hc when hundreds change; codegen cannot beat 30–150× work redundancy
Selective/macro-block codegen	killed by profile, pre-implementation	no concentration: top 200 nodes = 21.6% of group-walk work; groups average ~2 nodes
Oblivious sweep, interpreted	45× slower (539 vs 12.0 µs/hc)	~39,000 node-evals/hc (6,000 nodes × 6.5 iterations) vs hundreds of events
Oblivious sweep, compiled (Roslyn)	84× slower (1,004 µs/hc)	~700 KB straight-line code × 6.5 sweeps ≫ 32 KB L1i; front-end starvation
Macro-event / cone granularity	1.1× structural compression	cones mean 2.0 nodes ≈ the conducting group; no coarser unit exists
GPU, single instance	10.7× slower	serial settle chain ⇒ one workgroup ⇒ ~1/76 of the device utilized

9.3 Parallelization

Two parallel decompositions were implemented and measured; both lost by an order of magnitude or more, and the post-mortems converge on four structural walls that any parallelization of this engine must scale.

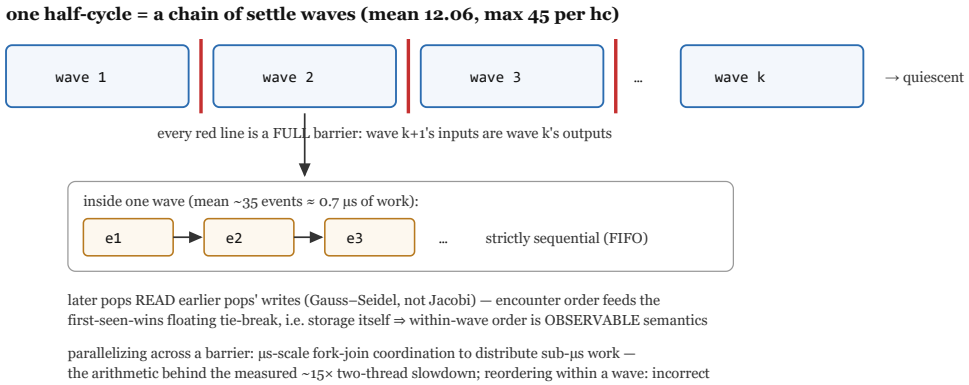

Figure 9.1. Settle waves are barriers; within-wave order is semantics. Each half-cycle drains a mean of 12.06 waves (max 45), and every wave boundary is a full synchronization point; within a wave events execute strictly FIFO, later pops reading earlier pops' writes (Gauss–Seidel), and encounter order feeds the first-seen-wins floating tie-break — i.e. storage itself. Parallelizing across a barrier spends microseconds of fork-join coordination to distribute sub-microsecond work (the measured ~15× two-thread slowdown); reordering within a wave changes the simulation outright.

The first attempt was the natural one: the NES contains two large chips, so step the 2A03 and 2C02 on two threads (fork-join per settle wave). Measured: 15× *slower* than single-threaded. The second attempt changed the axis entirely: keep one thread but make the group walk data-parallel, replacing the per-event BFS with a Ligra-style dense frontier sweep over bit-vectors, evaluated on an 8K-node / 16K-transistor PPU subset. The algorithm was correct — bit-identical checksums — and 156× slower. Two different decompositions, two different failure magnitudes, one shared root: both assumed the graph's size implies exploitable bulk, when the operative quantity is the *per-step work distribution*, and that distribution is brutally small.

The four walls, with their measured dimensions:

1. **Event grain vs. communication grain.** A processed event costs roughly 70–82 CPU cycles — about 20 ns at 4 GHz. Moving one cache line between cores costs on the order of 40–80 ns — an order-of-magnitude estimate from published microarchitecture data, not separately measured on this machine. At that estimate, any cross-core sharing of node state costs roughly two to four events' worth of time *per line moved*, before any synchronization primitive is counted. The unit of work is smaller than the unit of communication, which inverts the economics of distribution: shipping an event to another core costs more than executing it.

2. **Settle waves are barriers.** The double-buffered settle loop drains a mean of 12.06 waves per half-cycle (max 45), and each wave boundary is a full synchronization point: wave k+1's inputs are wave k's outputs. At ~418 events per half-cycle, the mean wave holds ~35 events — roughly 0.7 µs of work — between barriers. A fork-join across that grain pays microseconds of coordination to distribute sub-microsecond work, which is the arithmetic behind the 15× loss. The under-settling experiment of §9.5 closes the obvious loophole: the deep waves cannot be truncated or approximated, so the barrier count is not negotiable.

3. **Within-wave order is observable semantics.** Inside a wave the engine is Gauss–Seidel: later events in the same wave read the writes of earlier ones, and the floating-capacitance tie-break — the chip's actual storage mechanism — is sensitive to that order. Splitting a wave across cores changes the interleaving and therefore the simulated state; it is not a performance transformation but a semantics change. (This was confirmed the cheap way: a within-wave reordering experiment on one thread already diverged the checksum.)
4. **There is no thin cut.** 94% of the dependency graph is a single bidirectional SCC, and the CPU and PPU interact through shared board nodes continuously, so neither a static spatial partition nor a chip boundary yields independent work between synchronizations. The "two chips" of attempt one are electrically one machine at wave granularity.

What would have to change for parallelism to become viable is consequently well-defined, and each option breaks the contract. Amortizing synchronization over many waves requires waves to be independent, which they are not by construction. Decoupling the chips requires bounding their interaction lag, which trades bit-exactness for an approximation window. Optimistic parallel discrete-event simulation — speculate ahead, roll back on conflict — is the classical escape, but with ~35-event waves and a 94% SCC the conflict rate can be expected to be near-total (not validated by implementation), and the checkpoint state is the whole node array. And the bit-parallel route fails not at synchronization but at occupancy: when 77% of group walks visit two nodes and the mean BFS depth is 1.13, a 256-bit dense frontier does 100–200× the necessary work per step. None of these is an engineering shortfall; all four walls are properties of the netlist plus the semantics contract.

9.4 The maintained-state floor

The largest open cost in the engine, after the shipped prunes, is re-evaluating a node when one of its drivers disconnects, only to find it unchanged because another driver still holds it. The structural residue numbers (§9.5) say 80.1% of all event pops are such no-changes. The most instructive failure in the project — pursued three separate times over five days, each time to a sharper conclusion — is the attempt to skip them using a per-node runtime fact. We tell it in full because it terminates in a quantitative law rather than an anecdote.

9.4.1 P-5: a correct, firing optimization that lost to its own bookkeeping

The idea (P-5, the "dominant-driver bypass") was to record, per node, the single transistor whose conduction currently determines its value; on a turn-off of gate g, endpoint c can be skipped if its recorded dominant driver exists and is not g — a different conductor holds it, so the disconnect cannot change it. The first implementation was provably sound for its firing cases, verified bit-exact at 300K and 1M half-cycles, and genuinely pruned: with the bookkeeping kept but the skip disabled, the decomposition measured the skip's benefit at +3.4%. The full feature measured −12.05% (0 of 16 rounds), because the maintenance — capturing the dominant driver at every resolution, turning a deliberately branchless supply scan into a branchy count-and-capture, and writing a 29 KB array at ~117M writes per frame — cost −15.41% on its own.

9.4.2 The rebuild: a bigger prize behind the same toll

A from-scratch reconstruction (branch `dominant-bypass` [21]) replaced the 16-bit dominant-gate id with a cheaper, broader 1-bit predicate — "this node's current value is held by its *own* ground / power / pull-up regardless of pass connections" — and re-measured the decomposition. The prize was far larger than the first form suggested: the bypass alone is worth +13.76%, suppressing roughly 40% of node re-evaluations. The toll did not shrink: maintenance −15.6%, plus a storage tax with no free option. Packing the pinned bit into the 1-byte state array makes its write free but forces a mask onto every hot conduction read (−4.98% floor, net −8.36%); giving it its own array removes the mask but adds a ~50M-write random stream (net −6.84%, the best achieved). Fusing the capture into the group walk — which already scans member supplies — measured neutral: the walk visits far more members than the candidate set that benefits.

Table 9.2. The P-5 cost decomposition (rebuild form, vs. the P-4 baseline, interleaved-paired, all variants bit-exact). The benefit is real and large; the floor under it is larger.

Term	Measured	What it is
Skip benefit (bypass alone)	+13.76%	re-evaluations deleted; ~40% of pops suppressed
Maintenance (capture at every resolution)	−15.6%	branchy capture + a new random write stream on a latency-bound loop
Storage tax, option (i): bit packed into NodeStates	−4.98%	a mask on every hot conduction read; net −8.36%
Storage tax, option (ii): separate 1-byte array	net −6.84% (best)	no read mask, but the flag becomes its own ~50M-write stream
Maintenance fused into the group walk	neutral	captures for all walked members ≈ the targeted re-scan it replaced

9.4.3 P-5z: deleting the maintenance entirely — and the leg that provably cannot be freed

The third pass (branch `dominant-bypass-2`) attacked the floor itself, observing that part of the pinned predicate decomposes into *static structure* × *live state* — and the live state, `NodeStates`, is maintained by the engine anyway. For the 6,338 nodes with exactly one own ground channel in a clean component, the skip can be derived at the skip site with zero bookkeeping: skip turn-off endpoint *c* iff `NodeStates[ProbeGate[c]] != 0` — that ground channel's gate is ON, ground outranks every other contribution in the priority LUT, and the disconnect cannot change the node, transients included. A read-only table, no writes, no new maintained state. It passed every gate we have, including the 10M-half-cycle SMB1 run, and suppressed 5.3M re-evaluations per 100K half-cycles — 13.6% of all pops.

It measured **neutral**: +0.16% median, 10/20 paired (a range-encoded variant, −0.51%). The arithmetic is exact and merciless. The turn-off walk performs 9.4 endpoint tests per fired skip (50.1M tests buying 5.3M skips), and the pops this class avoids are the engine's *cheapest* — single-

ground leaf resolutions of roughly 30 cycles — so break-even requires the test to cost under ~3 cycles, which is precisely the price of its two branch-feeding loads. A sound, free-to-maintain skip of 13.6% of events bought nothing, because suppression count is not the objective function; suppressed *cycles* are.

The valuable leg ended differently. Sixty-four percent of the rebuild's +13.76% mass comes from the pull-up case — "PullUp pins the node high." Deriving *that* from live state **diverged the checksum**: the soundness proof for the pull-up pin binds to the node's value *at its last resolution*, and mid-settle transients can momentarily violate a live-state read between resolutions. The maintained pinned bit of §9.4.2 was never an implementation convenience — it *is* the resolution-time snapshot the proof requires. The profitable leg cannot be freed from maintenance even in principle.

> **The maintained-state floor.** Three independent attempts — an active-driver counter (−6.0%), a dominant-driver id (−15.4% maintenance), and a pinned bit in two storage encodings (−7% combined floor) — converge on one law: on this latency-bound engine, any skip predicate that requires a per-resolution runtime fact costs more to maintain than the re-evaluations it deletes, at every encoding tried; and the one leg whose predicate decomposes into static structure × live state reaches only the engine's cheapest events and nets zero. The general lesson, paid for precisely: *live state is not the state at last resolution* — proofs about a node's held value bind to the snapshot taken when it was last resolved, and keeping that snapshot is the irreducible toll.

> An adversarial review of the rebuild branch also found two *latent* soundness holes its clean checksums had never exercised: ForceCompute-component mates break the pinned-high proof (that set is the 2C02 sprite/OAM bus, quiescent in `full_palette`), and a staleness window existed inside the member loop. The branch's passing checksums were survivorship, not proof — which is why the sprite-heavy SMB1 run was promoted into the permanent gate set. A negative-result record must be auditable for exactly this failure mode: a "bit-exact" claim is only as strong as the diversity of the workloads behind it.

9.5 Smaller dead ends, with their populations

Around the major walls sits a perimeter of smaller closed questions. Each was closed not by intuition but by a measured population or a measured loss; we record both, because the populations are what prevent the questions from being reopened.

Table 9.3. The smaller dead ends. "Population" is the measured share of nodes, transistors, or events the technique could have touched; a technique whose population is cold or near-zero is closed regardless of its per-event merit.

Dead end	Measured population / result	Verdict and mechanism
Netlist constant-folding beyond the shipped lowering	76 foldable classes / 111 transistors (0.41%) at fixpoint; series-fusible 0; duplicate/parallel devices 0; total remaining <0.8% — all cold (no events)	closed by census: no events ⇒ no gain, independent of soundness
Weak/depletion transistor flag	population zero — the flag is false on every row of both raw 2A03/2C02 netlists [16]; pull-ups are segment attributes instead	model surface, not a missed optimization
Dead-end (unobserved-leaf) skip	diagnostic flagged 38% of BFS work as "no gates + unobserved"; all three skip variants broke the CPU; truly closed subset <1%	false positive: leaf state still flows out via the group walk; "no gate fan-out" ≠ "no observer"
Counter fast path (maintained active-driver counts)	−6.0%	the write path (every transistor flip) runs ~10× more often than the read it saves; an instance of §9.4's law
DFS group walk (replacing BFS)	−2.28% (17/20), bit-exact	order is irrelevant (the group is a set); BFS wins only because the result buffer doubles as the queue, making it free — DFS pays for a stack
Software prefetch in the pop loop; 16-bit queues; pure-locality renumbering	negative / neutral / neutral (density −45%, wall-clock ±0)	the hot set is L1-resident; the bound is dependent-load *latency*, not miss count — addresses become known too late for prefetch [14] to hide anything
Settle-depth capping ("approximate the deep tail")	cap 33 (abandons ~0.58% of settles): diverges within 1,000 hc; cap 8: CPU crashes in frame 1 (PC → $0040/BRK)	soundness failure, not a graceful trade — settle depths 7–45 are the netlist's real critical path
Pruning the structural residue (no-change PullUp/Supply pops)	80.1% of pops are no-change (PullUp 42.0% + Supply 38.1%); static single-channel subset 0.04%	closed three independent ways: profiler census, the P-5 family (§9.4), and an external-consult audit — the predicate is runtime state by nature
Estimator-guided reductions (memory cells, bus fabric, clock-phase waste)	measured 17.3–25.0% / 3.8% / 0.3% vs. consulted predictions of 30–40% / 7% / 20–35%; mean BFS depth 1.13 ⇒ "no giant BFS exists"	every predicted lever shrank by 2–100× when measured; the memory share is real but behavioralizing it breaches per-node bit-exactness

Three of these deserve a sentence of elaboration. The *dead-end-skip* entry is the project's canonical false positive: a 38% population looked like the largest untouched prize on the books, and all three implementations corrupted CPU execution, because a node with no gate fan-out still propagates state outward through the bidirectional channel walk — the observation relation in a pass-transistor netlist is not the gate-driven fan-out relation. The *settle-cap* entry closes the "approximate computing" door with unusual finality: the deep settle tail is not numerical refinement that can be truncated at small error, but the logical critical path of the circuit; under-settling does not degrade, it derails, with divergence in under a thousand half-cycles and a hard crash at aggressive caps. And the *structural residue* entry is the boundary's largest single face: four-fifths of all event work is re-evaluation that confirms no change, it is the direct price of event-driven local ignorance (the engine cannot cheaply know that a sibling pull-down still holds the node), and every mechanism for buying that knowledge — counters, ids, pinned bits, static subsets — has been priced and lost. The waste and the speed are the same phenomenon seen from two directions.

9.6 The anti-pattern catalogue

Stepping back from individual results, five failure shapes recur often enough across the project's thirteen months to deserve names. Each is stated with its single clearest project instance; most have two or more.

1. **The state-caching fallacy.** "Maintain a small runtime fact so the hot path can skip work." The fact's write path almost always runs more often than the read path it accelerates, and on a latency-bound loop a new random-access write stream is among the most expensive things one can add. Instances: the active-driver counter (−6.0%), the entire P-5 family (§9.4). The diagnostic question that would have killed both early: *how often is the fact written, versus how often is the skip taken?*

2. **The micro-branch trap.** "This test is only a compare — it's free." A guard executed on a hot walk is paid at the walk's frequency, not the skip's; P-5z executed 9.4 endpoint tests per fired skip, and two branch-feeding loads per test exactly consumed the ~30-cycle pops being saved. The shipped range-prune (Chapter 7) is the constructive counterexample: it won precisely by converting such tests into register compares against constants, deleting the loads rather than adding them.

3. **The small-N SIMD delusion.** "The graph has 14.7K nodes and 26.8K transistors — surely the work is wide enough to vectorize or bit-slice." The operative width is the per-step work distribution, and here it is 1.13–2.25 nodes. The bit-parallel BFS (156× slower, bit-identical) is the pure case: a 256-lane frontier sweep in which, 77% of the time, two lanes carry information. Four separate dead ends in the project's ledger share this root; the checklist item is to measure the *walk-size distribution* before assuming bulk.

4. **Compiler micromanagement.** "Take control away from the JIT and it will go faster." Native AOT compilation measured −5.5% in our implementation; a possible explanation is the loss of runtime dynamic profile-guided optimization (not separately verified). Forcing

aggressive inlining grew a method past the JIT's own inlining thresholds and measured −6%, consistent with the destruction of a downstream inline cascade. These effects live in the compiler/microarchitecture interaction, not in the algorithm — they can only be measured per configuration, never derived from first principles.

5. **The fine-grained parallelism illusion.** "Two chips, two threads." The per-sync work quantum (one ~35-event wave, ~0.7 μs) is smaller than the cost of distributing it, and the semantics forbid enlarging the quantum (§9.3). The 15× slowdown of the cleanest possible two-thread split is the canonical instance; the lesson generalizes to any decomposition whose synchronization period is fixed by the model rather than chosen by the implementer.

The common ancestor of all five is misdiagnosis by plausibility: reasoning from the structure that is easy to see (graph size, chip count, "wasted" work share) rather than the distribution that governs (events per wave, nodes per walk, writes per skip). Every shipped win in this project was preceded by a profiler census; every entry in this chapter that was implemented before being censused lost.

9.7 The boundary statement

We can now state what the engine's current figure means and what surrounds it. The measured peak is ~136K half-cycles per second (ablation stage S7, best 135,828; median 132,243) on a Zen 2 core — roughly 5.4 seconds of wall-clock per simulated frame, a factor of ~316 from the real chip's 42,954,552 hc/s. Within the family of public transistor-level NES/6502 simulators measured on the same machine, workload, and unit, this is the fastest by 2.5× over its own ancestor (MetalNES [17] ~54K, VisualNes [19] ~24K; perfect6502 [18] simulates the 6502 alone and is not unit-comparable; none of the original projects ships an hc/s measurement — these figures come from our experimental instrumentation of their source, detailed in Section 2.4) — and the family comparison is the fair scope of the claim, not a general superlative.

Property 9.1 (the single-core boundary). Under the conjunction of three constraints — (i) full switch-level semantics with the floating-capacitance tie-break [1], per-node bit-exact against the golden checksums; (ii) event-driven evaluation; (iii) one CPU core — the measured engine sits at a Pareto frontier: every abstraction of the evaluation model measured 1.0–84× slower (§9.2); every parallel decomposition measured 15–156× slower (§9.3); every maintained-state event prune measured at or below break-even, with the profitable leg provably requiring the maintained snapshot (§9.4); and the remaining no-change work (80.1% of pops) has no static, bit-exact-prunable subset above 0.04% (§9.5). Relaxing any one constraint changes the artifact, not the artifact's speed.

What *would* move the number is therefore specific. First, memory latency and clock frequency: the bound is dependent-load latency on an L1-resident working set, so higher-IPC, lower-load-latency cores can be expected to translate hardware-generation improvements directly into throughput. Second, residual identifier-space and dispatch refinements of the kind Chapter 7

shipped — measured in single percents, accumulating, and bounded well short of an order of magnitude. What will *not* move it, on the evidence of this chapter: more cores, under these semantics; compilation, at any granularity tried or profiled; and abstraction, under the per-node bit-exact contract — the 98.9% reducibility result is real and is the right foundation for a *different* artifact (a verified gate-level model for analysis), but it cannot make *this* artifact faster.

The boundary map is the deliverable. A future implementer of a switch-level simulator for a flat, bidirectional, two-phase pass-transistor netlist of this scale inherits, from this chapter, not only the knowledge that the event-driven interpreter is the winning shape — Ulrich's activity principle [4], surviving four decades of hardware change — but the measured reasons each alternative loses, the populations that close the smaller questions, the five shapes of self-deception to test for, and the precise legal moves left on the board. That is a different and, we argue, more durable contribution than the engine's rank on any leaderboard [21].

Chapter 10. Conclusions and Future Work

This monograph set out to answer a narrow question honestly: how fast can a bit-exact, event-driven switch-level simulation of an entire commercial console — the NES's 2A03 CPU, 2C02 PPU, and board glue, some 14.7K nodes and 26.8K NMOS transistors after lowering — be made to run on one core of a commodity processor, in a managed language — chosen in part to put to an empirical test a widespread impression in performance-engineering practice, for which we could find no formal source, that managed languages are unsuited to this kind of metric-chasing core — without surrendering a single bit of the model's observable semantics? The answer, measured rather than argued, is roughly 136,000 master-clock half-cycles per second on a Zen 2 core: about 0.3% of silicon speed, about 2.5× the optimized C++ ancestor from which the engine descends [17], and about 5.6× a literal port of the original JavaScript simulator [15, 19]. The remaining ~316× gap to real time is, on the evidence of this project's own negative results, not closable along this route. This closing chapter restates what was contributed and at what measured magnitude, extracts the lessons we believe travel beyond this codebase, states the threats to validity plainly, and bounds the future work by the same negative results that bound the engine.

10.1 Summary of Contributions

Chapter 1 listed five contributions. We restate them here with the measured magnitudes attached, in the order the ablation ladder of Chapter 8 attributes them. All figures below are same-day, round-robin-interleaved medians on one machine, every run gated on the full-state checksum, with stages rebuilt from the actual historical commits.

1. A prune family with static safety proofs (P-1 through P-4). The engine suppresses an enqueue when the eventual re-evaluation is provably a no-op, using only load-time structure plus the two endpoints' live states. The family rests on a structural safety taint — nodes without pull-ups and ForceCompute channel-components are excluded because their floating tie-break *is* the chip's storage mechanism — and on the project's strongest originality claim, the capacitance-dominance un-taint: a node whose capacitance is strictly below all of its channel neighbours can never be the largest-capacitance member of any group containing a neighbour, hence can never win Bryant's charge-share tie-break [1], hence equal-state merges through it are provably null. Measured: the family deletes ~21% of all node re-evaluations; on the ladder it contributes the +26.4% step (P-1) and a further +7.2% (P-2/P-3/P-4 with mask consolidation), and across the full ladder retired instructions fall by 33% (155.6B to 104.4B per million half-cycles) while IPC holds between 2.1 and 2.4. We position this within the activity-suppression lineage that begins with Ulrich [4]; the pre-queue, statically-proved form appears to be the new element.

2. Cardinality-specialized dispatch (R-1 and B1). Two runtime proofs that the active channel-connected component has size one (all pass gates currently off) or exactly two (one ON gate whose far side leads only back to the seed), each resolved inline while replicating the general walk's order-sensitive semantics byte for byte. Measured: R-1 is the single largest step on the ladder, +18.7%; B1 contributes +7.0% (S6→S7, same-day interleaved; about +8.9% after

separating the attribution from the adjacent housekeeping commit). We call these variants, not inventions: IRSIM already sized active subnetworks dynamically [3], and our contribution is the early-out execution filter and the demonstration of its bit-exactness obligations.

3. Self-captured first-touch data relayout. A three-pass load: classify every node into prune classes; rebuild the id space class-major so the hot loop's safety tests become register range compares against three boundaries (verified against recomputed ground truth at every reset, with a safe-degenerate fallback in the deoptimization-guard pattern [13]); then capture 32,768 half-cycles of the production cascade's true first-pop order through a cold instrumented copy of the settle loop and rebuild once more with that order as the locality key. Measured: the class-major range form contributes +4.2% and the captured key a further +5.0% on the ladder; head-to-head against a file-loaded measured profile the captured key wins +6.17%. The mechanism sits squarely in the inspector-executor and trace-driven-layout lineages [7, 8, 9, 10, 11, 12, 20]; the new element is that the simulator captures its *own* cascade in-process at every load — no profile files, any ROM, immune to workload drift by construction — and the empirically surprising part is *why* it works, which is lesson two below.

4. Zero-configuration safety classification by physics. Storage, bus, and memory nodes are never identified by name. Pull-up flags, channel-component union-find, driven-pin exclusion, and capacitance comparison identify them by what they physically are; storage excludes itself from unsafe pruning because large capacitance on a floating node is what storage *is* in this technology. We present this as implementation discipline rather than a research claim — automatic transistor-to-gate-level logic extraction and structural-pattern-driven subcircuit recognition form an established family in the literature [5, 6]; our identification rests on physical properties (pull-up flags, capacitance comparisons, channel components) rather than library patterns — but it is the reason the prune family required no per-chip configuration when extended from the 2A03 to the 2C02 and the board.

5. A falsifiable performance boundary map. The nine-stage ablation with hardware counters (Chapter 8), the measured failure of every abstraction and parallelization route (Chapter 9), and the maintained-runtime-fact floor (the P-5 family: +13.76% gross prize, −6.84% best net, with the sound zero-maintenance leg measuring neutral and the valuable leg provably requiring a resolution-time snapshot rather than live state). The map's headline structural facts: the mean conducting group is 1.13–1.4 nodes; 80.1% of pops are no-change and the dominant PullUp/Supply residue has no static subset (three independent confirmations); 94% of the dependency graph is one bidirectional strongly-connected component; and within-wave order is observable semantics.

> **Headline result.** Cumulative S0→S7: +94.6% median throughput on one core, bit-exact at every stage, with the counters showing the engine is dependent-load-*latency*-bound, not miss-*count*-bound: the relayout cuts L1d misses from 27.3 to 13.2 MPKI while wall-clock moves single digits, and L1i misses never exceed 0.30 MPKI — the counter-level explanation for why interpretation beats compilation in this regime.

Table 10.1. Contributions restated with measured magnitudes (same-day ablation, 400k hc, medians). "Variant" status follows the prior-art audit of Chapter 3.

Contribution	Measured magnitude	Status vs. prior art
P-1 same-state turn-on prune (+ safety taint)	+26.4% ladder step	Plausibly original (pre-queue static suppression)
P-2 isolation prune; P-3/P-4 capacitance-dominance un-taint	+7.2% ladder step; family deletes ~21% of re-evaluations	P-3/P-4 strongest claim; nearest neighbour is IRSIM sizing practice [3]
R-1 dynamic singleton	+18.7% ladder step	Variant (cardinality early-out over dynamic CCC sizing [3])
B1 pair path	+7.0% ladder step (≈+8.9% after separated attribution)	Variant (runtime template matching)
Class-major renumber + range-prune	+4.2% ladder step	Engineering on [10, 12]; deopt guard per [13]
Self-captured first-touch key	+5.0% ladder step; +6.17% vs. loaded profile	In-process self-capture appears new; lineage [7, 8, 9, 11, 20]
Boundary map + negative results	~136K hc/s peak; gap to real time ≈316×	Methodological contribution

10.2 Lessons That Generalize

10.2.1 Event suppression needs proofs, not heuristics

Every suppression mechanism that shipped carries a structural argument over the resolution semantics; every one that relied on a plausible heuristic failed, and failed loudly. The first, naive form of P-1 produced a black screen: it pruned equal-state merges through floating storage nodes whose tie-break outcome depended on exactly the merge being suppressed. The zero-maintenance variant of the dominant-bypass family had one leg whose proof was sound (and measured profit-zero) and one leg whose obvious formulation — read the live state of the dominating node — diverged within 100k half-cycles, because the underlying proof binds to the value at *resolution time*, not to the current value. Under-settling, the most tempting approximation of all, derails the simulated CPU within a thousand half-cycles even when only the deepest 0.58% of settle chains are truncated. The general lesson for event-driven simulators whose state mechanism is a tie-break: there is no "approximately safe." The compensating mercy is that failures in such systems are catastrophic and immediate rather than subtle, which makes a checksum gate an effective and cheap falsifier — every candidate suppression in this project was accepted or destroyed by it within minutes.

10.2.2 Layout value can lie in order, not density

The literature on data layout for irregular codes is organized around miss counts [8, 10, 11, 12, 14], and our own counters show why that framing can mislead. The class-major relayout cut the L1d miss rate by half (27.3 to 13.2 MPKI; AMD-native miss ratio 5.6% to 3.2%) for a wall-clock gain of +4.2%; the self-captured key then added +5.0% while leaving the miss counters essentially flat. The controlled comparison is sharper still: a captured key with prunes disabled achieves the same cache-line footprint as the winning key (110 vs. 105.9 NodeInfo lines per half-cycle, against 118.4 for the blind key) and gains *nothing*, while the capture taken with prunes enabled gains +6.17%. Equal density, different lineage, opposite outcomes. The value of the layout is that it matches the *order* of the pruned production cascade — shortening and overlapping the dependent-load chains the out-of-order core must traverse serially — not that it packs lines. For latency-chain-bound pointer code with an L1-resident hot set, optimizing the named metric of the layout literature is a category error; the objective function must be the serial chain.

10.2.3 Measurement discipline is a first-class deliverable

Three practices made the numbers in this monograph trustworthy. First, measurements are taken with the clock unlocked, with variance controlled by same-day multi-round round-robin interleaving, a single benchmark process, and median reporting: this machine's throughput varies ~10% between a cool and a heat-soaked day, so cross-stage comparisons that are not interleaved within one session measure weather. Second, checksum gating of every benchmark run (all 72 ladder runs passed), which converts every performance experiment into a regression test. Third, attribution by historical rebuild: when the S6→S7 step looked too large, separating the attribution from the adjacent housekeeping commit isolated B1 at about +8.9%, with fully separated distributions. Affinity pinning with EcoQoS disabled halves run-to-run variance (cv 1.5% to 0.94%) and is the hygiene tool we do endorse.

10.2.4 Honesty about variants is cheap and pays

The prior-art audit concluded that our resolution semantics are functionally MOSSIM II without the X state [1], that the group walk is what the literature calls CCC evaluation, and that R-1, B1, and the range-prune are variants of known practice [2, 3, 13]. Stating this plainly cost nothing and bought focus: effort concentrated on the two claims the audit could not match (the pre-queue prune and the dominance un-taint) and on the boundary map, instead of on re-deriving and over-claiming forty years of switch-level engineering. We commend the exercise — a verdict table against the literature, written before the paper — to any project of this kind.

10.3 Threats to Validity

One machine. Every number in this monograph was measured on a single AMD Ryzen 7 3700X (Zen 2). The ladder's *ordering* of techniques is likely robust; the step sizes are facts about this Zen 2 part, and the latency-bound diagnosis should be re-verified on cores with different load-to-use latencies and window sizes.

One workload family. Throughput and event statistics come from `full_palette.nes`, with a 10M-half-cycle sprite-heavy Super Mario Bros. run as the adversarial correctness gate. Both exercise one console, two die-derived NMOS netlists, and NROM cartridges only. The structural findings — group-size distribution, the 80.1% no-change fraction, the single giant SCC — are properties of these netlists; we expect, but have not measured, similar shapes on other Visual6502-family dies [15, 16].

Counter methodology. The per-stage hardware profile uses ETW PMC sampling with ~5% overhead; we therefore report ratios (MPKI, miss ratios, IPC), not absolute throughput, from those runs. This machine exposes no functioning last-level-cache counter source, so L3/DRAM behaviour is inferred from the ~15 KB hot set and the 3.2% L1d miss ratio rather than observed. The inference is comfortable but it is an inference.

Bit-exactness is checksum-attested, not proven. Full-state FNV-1a checksums at 300k/400k/1M half-cycles plus the 10M SMB1 gate constitute strong evidence over the inputs tested; they are not a proof of equivalence over all programs. The under-settling experiment shows divergence in this system is fast and loud, which raises our confidence that the gates would catch real errors, but the limitation stands.

10.4 Future Work, Honestly Bounded

The analysis-value pivot. The most defensible next investment is not speed. The same extraction machinery that measured the chip as ~98.9% logic-reducible supports a gate-level *view* of the running silicon: named-register watchpoints (the netlists already name `a0..a7`, `pcl0..pcl7`, and kin [16]), Φ1/Φ2 clock-phase maps, bus-contention diagnostics, and Verilog/Graphviz export of the extracted logic. None of this competes with the bit-exact engine; all of it is value the engine's fidelity uniquely enables, and it is where the project's marginal hour now pays.

A behavioral-OAM fork, outside the contract. Measured structural profiling attributes ~25% of all event pops to memory-cell traffic (OAM and palette precharge, clock-driven). Replacing those cells with behavioural arrays would delete interior nodes and therefore violates the bit-exact, full-state-checksum contract that defines this engine — it is the abstraction style we explicitly disallowed. As a separately-labelled fork it is legitimate future work, with a hard ceiling set by that event share: roughly 1.3×, no more. We state the ceiling now so the fork cannot later be oversold.

Hardware moves the floor. The counters say the engine spends its time on dependent-load latency over an L1-resident working set, with instruction fetch effectively free. That is the one regime in which future hardware helps without any software change: shorter load-to-use latency, deeper out-of-order windows, and higher sustained IPC translate directly. Conversely, the measured negatives — threading 15× slower, GPU 10.7× slower, compilation 3–6× slower, bit-parallelism 156× slower — say that more cores, wider vectors, and accelerators do not, because there is no batchable regularity in a 1.13-node mean group and the per-event budget (~20 ns) is smaller than a cross-core cache-line transfer.

Re-measuring our own ceilings. Every throughput ceiling this project published — 80K, 100K, 120K, 127K half-cycles per second — subsequently fell, and each fell to a measurement, not to an argument. The structural arguments that remain (the no-change residue with no static subset; the giant SCC; order-as-semantics) are of a different kind: they are reasons routes are closed, verified from three independent directions, not extrapolated trend lines. Our final position is accordingly split: we assert the ~316× real-time gap is unreachable *via this route* with confidence, and we assert any particular hc/s number is the end with none. The standing policy — accumulate small, checksum-gated, paired-measured wins; pre-dismiss nothing as too small — remains in force.

10.5 Availability

Everything this monograph measures is public [21]. The repository at `github.com/erspicu/AprVisual` contains the C# engine (`src/AprVisual.S1/`), the deprecated experimental forks that produced the negative results of Chapter 9, the self-contained benchmark packages, and the measurement scripts. The companion site at `erspicu.github.io/AprVisual` hosts the article series and the public leaderboard with the uploaded runs cited here. The raw data behind Chapter 8 — the per-run ablation ladder and the per-stage counter summaries — ships with the manuscript sources (`MD/paper/data/`), together with the per-stage commit identifiers and golden checksums needed to rebuild and re-gate every rung of the ladder on other hardware. The die-derived netlists themselves are the property of their extractors [15, 16] and are referenced, not vendored, under their upstream licences. We hope the boundary map is used the way it was built: as a set of falsifiable claims, each attached to a command line that can prove it wrong.

Acknowledgments

During the preparation of this work — the engine, the measurements, and this manuscript — the author used generative AI tools, principally Anthropic's Claude (via the Claude Code environment), to assist in implementing and refactoring the C# engine code, orchestrating the benchmark and hardware-counter measurement runs, analyzing results, surveying related work, and drafting and polishing the text of this monograph in both its English and Traditional Chinese editions. After using these tools, the author thoroughly reviewed, tested, and edited the content as needed — every accepted engine change is gated by the bit-exactness checksums of Appendix A.3, and every performance number is a measurement whose raw data is reproduced in Appendices B and C — and takes full responsibility for the content, scientific accuracy, and originality of this publication.

This statement constitutes the disclosure required by arXiv's policy on generative AI language tools (significant use of text-generation tools must be reported in the work) and the acknowledgments disclosure required by the IEEE submission policies (PSPB Operations Manual) for any subsequent journal or conference submission. For the section-level identification the IEEE policy asks for: AI assistance at the level described above — full-text drafting from author-directed source material and measured data, followed by author review and editing — applies to *all* chapters and appendices of this monograph, and the SVG figures were likewise AI-drafted from the measured data and reviewed by the author; no part of the manuscript is exempt. Consistent with both policies, no AI system is an author or co-author of this work; authorship and accountability rest solely with the human author.

This work stands on the shoulders of the open silicon-archaeology community: the Visual 6502 project [15], Quietust's Visual 2A03/2C02 netlists [16], and the MetalNES [17], perfect6502 [18], and VisualNes [19] simulators.

References

1. R. E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Transactions on Computers* C-33(2), 1984.
2. R. E. Bryant, D. Beatty, K. Brace, K. Cho, T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," *Proc. 24th Design Automation Conference (DAC)*, 1987.
3. A. Salz and M. Horowitz, "IRSIM: An Incremental MOS Switch-Level Simulator," *Proc. 26th Design Automation Conference (DAC)*, 1989.
4. E. G. Ulrich, "Exclusive Simulation of Activity in Digital Networks," *Communications of the ACM* 12(2), 1969.
5. M. Boehner, "LOGEX — an Automatic Logic Extractor from Transistor to Gate Level for CMOS Technology," *Proc. 25th Design Automation Conference (DAC)*, 1988.
6. M. Ohlrich, C. Ebeling, E. Ginting, L. Sather, "SubGemini: Identifying SubCircuits using a Fast Subgraph Isomorphism Algorithm," *Proc. 30th Design Automation Conference (DAC)*, 1993.
7. J. Saltz, R. Ponnusamy, S. D. Sharma, B. Moon, Y.-S. Hwang, M. Uysal, R. Das, "A Manual for the CHAOS Runtime Library," University of Maryland technical report, 1995.
8. C. Ding and K. Kennedy, "Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time," *Proc. PLDI*, 1999.
9. M. M. Strout, L. Carter, J. Ferrante, "Compile-time Composition of Run-time Data and Iteration Reorderings," *Proc. PLDI*, 2003.
10. T. M. Chilimbi, M. D. Hill, J. R. Larus, "Cache-Conscious Structure Layout," *Proc. PLDI*, 1999.
11. T. M. Chilimbi, "Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality," *Proc. PLDI*, 2001.
12. B. Calder, C. Krintz, S. John, T. Austin, "Cache-Conscious Data Placement," *Proc. ASPLOS-VIII*, 1998.
13. U. Hölzle, C. Chambers, D. Ungar, "Debugging Optimized Code with Dynamic Deoptimization," *Proc. PLDI*, 1992.
14. S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, A. Moshovos, "Spatial Memory Streaming," *Proc. ISCA*, 2006.
15. The Visual 6502 project — JavaScript simulator (chipsim.js) and the die-shot-derived 6502 netlist. visual6502.org.
16. Quietust, Visual 2A03 and Visual 2C02 — die-shot-derived netlists of the NES CPU and PPU.
17. iaddis, MetalNES — transistor-level NES-001 simulation. github.com/iaddis/metalnes.
18. M. Steil et al., perfect6502 — switch-level simulation of the 6502 in C. github.com/mist64/perfect6502.
19. VisualNes — C++ port of the Visual 2A03/2C02 simulators. (GitHub.)
20. M. R. Haghighat and D. C. Sehr, "Profile-guided data layout," U.S. Patent 7,143,404, granted 2006.
21. AprVisual — the engine, the benchmark package and the public leaderboard: github.com/erspicu/AprVisual; companion articles: erspicu.github.io/AprVisual.

Appendix A. Reproducibility

A.1 Machine and invocation

All measurements in this monograph were taken on one machine: AMD Ryzen 7 3700X (Zen 2, 8C/16T, 32 KB L1d + 32 KB L1i per core), Windows 11, .NET 10/11 as listed per stage. Workload: `full_palette.nes` (NROM; a community open-source NES test ROM displaying the full NES palette as a static screen, from the public nes-test-roms collection), whole-console simulation. The benchmark entry point is:

```
dotnet run -c Release --project src/AprVisual.S1 -- --benchmark full_palette.nes --bench-hc 400000
```

Clock was unlocked; variance was controlled by protocol (see §8.1): exactly one benchmark process at any time, same-day round-robin interleaving across stages, medians reported. The optional `--pin` switch (core affinity + High priority + EcoQoS off; frequency untouched) halves run-to-run variance and is bit-exact, but the headline numbers were taken without it for leaderboard comparability.

A.2 Stage-to-commit map

Table A.1. The ablation stages of Chapter 8: repository commits, runtimes, and what each adds. Historical commits were rebuilt unmodified in isolated git worktrees.

Stage	Commit	Runtime	Adds
S0	`a80dab4~1`	.NET 10	baseline fork (static fast path)
S1	`a80dab4`	.NET 10	R-1 dynamic singleton
S2	`ed8c457`	.NET 10	P-1 same-state turn-on prune
S3	`6bdc25b`	.NET 10	P-2/P-3/P-4 + PruneMask consolidation
S4	`00ab4fa`	.NET 10	clause reorder + supply-skip fold
S5	`51e046d`	.NET 11	range-prune (class-major renumber)
S6	`3e4a571`	.NET 11	self-captured first-touch key
S6b	`a553d38`	.NET 11	file-mode removal (attribution control; neutral)
S7	`2749105`	.NET 11	B1 pair path

A.3 Golden checksums (bit-exactness gate)

Table A.2. Full-state FNV-1a checksums every accepted change must reproduce. Checksums iterate the ORIGINAL node-id order through the layout permutation, so they are layout-independent by construction.

Gate	Workload	Horizon	Checksum
primary	full_palette.nes	300,000 hc	`0x794A43ABDF169ADA`
ladder gate	full_palette.nes	400,000 hc	`0x9174E19D961CB6E5`
long-horizon	full_palette.nes	1,000,000 hc	`0x6D4CCBCE2E9CD599`
sprite-heavy	Super Mario Bros.	10,000,000 hc	`0x7DE79A1BD3D8536F`

All 72 ablation runs (9 rounds × 8 stages) were checksum-gated at 400k hc; a run that fails its checksum is discarded as a measurement of a different program.

Appendix B. Raw ablation data

The full 9-round × 8-stage matrix behind Table 8.1 and Figure 8.1 (half-cycles per second, 400,000 hc per run, unlocked clock, round-robin order within each round). Medians of each column are the stage results quoted throughout. The data file is preserved at `MD/paper/data/ablation_runs.csv` in the repository [21].

Table B.1. Raw ablation matrix: 9 rounds × 8 stages (hc/s, 400,000 hc per run, all checksum-gated).

Round	S0 _preR1	S1_R1	S2_P1	S3 _P234	S4 _micro	S5 _range	S6 _capture	S7_B1
1	64,587	78,969	101,064	108,448	112,887	117,671	121,473	133,007
2	66,705	80,586	101,381	109,055	113,232	118,141	124,899	135,828
3	68,392	79,857	101,158	111,875	113,265	116,197	125,534	134,211
4	67,949	78,808	100,659	110,155	111,769	115,168	121,455	128,329
5	65,712	81,606	102,790	109,346	112,038	113,701	122,647	129,117
6	68,049	81,493	103,818	109,841	110,496	119,464	126,512	128,354
7	67,955	82,806	103,737	109,640	112,942	118,976	120,614	135,174
8	69,677	80,657	102,210	112,177	111,297	119,323	126,781	128,747
9	68,729	81,716	101,964	113,571	114,833	115,142	123,545	132,243
Median	**67,955**	**80,657**	**101,964**	**109,841**	**112,887**	**117,671**	**123,545**	**132,243**

Appendix C. Raw hardware-counter data

ETW PMC sampling (PerfView collection, xperf extraction), 1,000,000 hc per stage, samples filtered to the simulator process. Sampling intervals: 1,048,576 events/sample for instructions and cycles; 65,536 events/sample for cache-miss and branch-misprediction sources. Estimated event counts are samples × interval; per-1M-hc normalization and MPKI ratios derived from these are tabulated in §8.4. The data file is preserved at `MD/paper/data/pmc_summary.csv` [21].

Table C.1. Raw PMC samples (1,000,000 hc per stage; estimated events = samples × sampling interval).

Stage	Counter source	Samples	Estimated events
S0_preR1	IcacheMisses	402	26,345,472
S0_preR1	InstructionRetired	148,400	155,608,678,400
S0_preR1	TotalCycles	70,311	73,726,427,136
S0_preR1	BranchMispredictions	7,773	509,411,328
S0_preR1	DcacheMisses	64,872	4,251,451,392
S1_R1	IcacheMisses	378	24,772,608
S1_R1	InstructionRetired	122,781	128,745,209,856
S1_R1	TotalCycles	59,113	61,984,473,088
S1_R1	BranchMispredictions	6,618	433,717,248
S1_R1	DcacheMisses	60,180	3,943,956,480
S2_P1	IcacheMisses	331	21,692,416
S2_P1	InstructionRetired	110,642	116,016,545,792
S2_P1	TotalCycles	46,584	48,846,864,384
S2_P1	BranchMispredictions	4,845	317,521,920
S2_P1	DcacheMisses	43,312	2,838,495,232
S3_P234	IcacheMisses	313	20,512,768
S3_P234	InstructionRetired	100,840	105,738,403,840
S3_P234	TotalCycles	42,978	45,065,699,328
S3_P234	BranchMispredictions	4,058	265,945,088
S3_P234	DcacheMisses	41,681	2,731,606,016
S4_micro	IcacheMisses	333	21,823,488
S4_micro	InstructionRetired	99,463	104,294,514,688
S4_micro	TotalCycles	43,060	45,151,682,560
S4_micro	BranchMispredictions	4,322	283,246,592
S4_micro	DcacheMisses	41,825	2,741,043,200
S5_range	IcacheMisses	467	30,605,312

S5_range	InstructionRetired	102,824	107,818,778,624
S5_range	TotalCycles	45,798	48,022,683,648
S5_range	BranchMispredictions	4,728	309,854,208
S5_range	DcacheMisses	21,779	1,427,308,544
S6_capture	IcacheMisses	467	30,605,312
S6_capture	InstructionRetired	106,416	111,585,263,616
S6_capture	TotalCycles	44,445	46,603,960,320
S6_capture	BranchMispredictions	4,468	292,814,848
S6_capture	DcacheMisses	21,363	1,400,045,568
S6b_filerm	IcacheMisses	510	33,423,360
S6b_filerm	InstructionRetired	106,170	111,327,313,920
S6b_filerm	TotalCycles	44,094	46,235,910,144
S6b_filerm	BranchMispredictions	4,567	299,302,912
S6b_filerm	DcacheMisses	21,168	1,387,266,048
S7_B1	IcacheMisses	465	30,474,240
S7_B1	InstructionRetired	99,571	104,407,760,896
S7_B1	TotalCycles	42,572	44,639,977,472
S7_B1	BranchMispredictions	4,277	280,297,472
S7_B1	DcacheMisses	20,768	1,361,051,648

Limitations: sampling-based attribution; the Windows generic `DcacheAccesses` / `CacheMisses` sources were silent on this part — an AMD-native second pass (`DCAccess` / `ICFetch` / `ICMiss`) supplied the access-count denominators quoted in §8.4; no functioning last-level-cache counter was available on this machine.