剪除不可變者:靜態安全證明、基數特化派發與自我捕捉資料重排

—— 整機遊戲主機的位元精確開關級模擬(技術專論)

黎映微(Ying-Wei Li)
Independent Researcher, Taiwan
baxermux@gmail.com

摘要

開關級模擬把晶片當成它的電晶體來執行 —— 是唯一能逐位元重現 pass-transistor 匯流排、 動態電荷儲存與預充行為的軟體模型,因此始終是 1980 年代 NMOS 晶片晶粒級保存的參考工具。 它也以慢著稱:每個事件都要透過指標追逐式的圖走訪重新解析一個通道相連元件(CCC),而受控 語言向來被認為不適合這種核心。我們提出 AprVisual:一個以 C# 實作的 NES 整機模擬器(2A03 + 2C02,約 14.7K 節點 / 26.8K 電晶體),解析語意在功能上等同 MOSSIM II。在單顆 Zen 2 核心上它持續達到**每秒約 13.6 萬個主時脈半週期** —— 等效約 68 kHz 的矽主時脈,約為其 C++ 直系祖先(MetalNES)的 2.5 倍、chipsim.js 逐字移植版 的 5.6 倍 —— 且全程**位元精確**(全狀態 checksum 驗證)。速度來自三個家族: (1) *可證明無效的事件壓制* —— 載入期結構證明(含電荷分享 tie-break 免疫的電容支配 論證),在入列之前刪掉約 21% 的重算;(2) *基數特化派發* —— 執行期證明 active CCC 大小 ≤ 2,就地解析;(3) *自我捕捉資料重排* —— 同進程的 inspector-executor 遍,每次載入由生產級聯的初次觸碰順序重新導出節點 ID 空間。九階段同日消融加硬體計數器 逐級量化,並證明引擎受限於延遲鏈而非 miss 數:重排砍掉 67% 的 L1d miss 而牆鐘只動個位數; L1i miss 全程僅 0.2~0.3 MPKI —— 這正是「直譯在此勝過編譯」的體系結構原因。

關鍵字:開關級模擬;事件驅動模擬;通道相連元件(CCC);事件壓制;資料佈局與快取局部性;位元精確;硬體保存;NES

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

閱讀說明。本專論是本專案量測紀錄的長篇版本。 書中每一個效能數字都是在第 8 章所述的機器上實際量測所得(原始數據完整收錄於附錄); 負面結果與正面結果以同等嚴謹呈現,屬於既有文獻的技術一律如實標註。本工作的工程、量測與 寫作過程廣泛使用了生成式 AI 工具;依循 arXiv 與 IEEE 對 AI 輔助稿件的指引,完整揭露聲明見〈致謝〉。 人類作者是本工作的唯一作者,並對其內容、科學準確性與原創性負完全責任。

目錄

第 1 章 簡介

本專書談的是如何讓一台電腦最慢、但最可信的模型跑得夠快而足以實用,同時始終不讓它變成另一個模型。研究對象是 AprVisual:一個位元精確、事件驅動的開關級模擬器,模擬完整的任天堂紅白機(Nintendo Entertainment System)——2A03 CPU、2C02 PPU 與主機板上的 TTL 膠合邏輯,lowering 之後約 14.7K 個電氣節點與 26.8K 顆 NMOS 電晶體——以 C# 寫成 [21]。在一顆市售桌上型 CPU 的單一核心上,它持續達到約每秒 136,000 個主時脈半週期,等效約 68 kHz 的矽主時脈,約為其 C++ 直系祖先吞吐量的 2.5 倍、原始 JavaScript 模擬器逐字移植版的 5.6 倍,皆於同一台機器、同一工作負載下量得(所有數字皆為實測;見 1.5 節)。促成這個數字的每一項優化,都經過全狀態 checksum 驗證的閘門;而每一項未通過閘門的優化——或通過了閘門卻仍然賠上時間的優化——都以與成功項目同等的嚴謹度回報。本書的成果可以在兩個層次上閱讀:一是針對特定模擬器的工程紀實,二是一張以量測繪成的地圖,標示事件驅動開關級模擬的單核效能邊界究竟落在哪裡。

1.1 為什麼是開關級,以及為什麼位元精確不可妥協

這些晶片沒有暫存器轉移層級(register-transfer level,RTL)的描述。2A03 與 2C02 是 1980 年代初期的商用 NMOS 設計,唯一具權威性的紀錄就是矽晶片本身;我們模擬的網表(netlist)是由 Visual 6502 計畫以及 Quietust 的 Visual 2A03 / Visual 2C02 工作從晶粒照片復原而來——多邊形層級的線段定義、電晶體定義,以及一份部分的名稱對照表 [15, 16]。這種復原得到的不是邏輯設計,而是一張實體連接圖:哪些擴散區彼此相觸、哪些多晶矽閘極跨過哪些通道、哪些節點帶有空乏式上拉。凡是疊加在這個層級之上的模型都是一種詮釋;就實務而言,開關級模型是仍然算得上「謄寫」的最高抽象層級。

Bryant 的開關級模型 [1] 把晶片視為一組儲存電荷的節點,由雙向開關相連,衝突由訊號強度與節點電容解決。這個模型最關鍵的性質——也是它存在的根本理由——在於*儲存是湧現的,而非宣告的*。一個與所有驅動源斷開的節點不會變成未知值;它會保持自身電荷。一群彼此相連的浮接節點以電荷分享(charge sharing)決定狀態,我們的引擎將其實作為文件化的 tie-break:純浮接群組取其電容最大成員的前一狀態(嚴格比較,平手時先見者勝)。模擬器中沒有任何資料結構被標記為「暫存器」或「RAM 細胞」。暫存器、閂鎖與記憶體之所以出現,是因為網表裡恰好存在某些組態,使停駐在浮接節點上的電荷得以存活到下一次被讀取為止。

2C02 的精靈記憶體(OAM)是最典型的例子。它不是一塊 RAM 巨集:它是約 2,300 顆浮接電容,經由 pass transistor 寫入,純粹以電荷形式保存。它的容量、它的時序、以及它眾所周知的脆弱性,都是物理的後果——開關級模型重現這些後果,而 RTL 模型必須以行為式註記手寫出來——一旦如此,模型記錄的就是建模者的信念,而不是晶片本身。同樣的道理也適用於遍布兩顆晶粒的設計慣用法:雙向 pass-transistor 匯流排、預充電加條件放電邏輯、以驅動強度決勝的匯流排競爭,以及 PPU 中把位址與資料訊號多工到同一批實體節點上的做法。RTL 的核心假設——單向訊號流、宣告式狀態元件、不帶強度的值——在這塊矽晶片上是錯的,而且錯在承重之處。對保存(preservation)這個使用情境而言——交付物是*關於這件工件實際行為的證據*,而不是一個方便的重新實作——開關級模擬因此不是諸多選項之一;它是唯一能讓這些有趣行為「湧現」而非「被斷言」的軟體層級 [1, 15]。

這也是位元精確之所以不可妥協、而非僅僅可取的原因。在傳統的效能專案裡,一項略微改變結果的優化可以拿來和它的加速做交換。但在這裡,浮接 tie-break *就是*晶片的儲存機制,而波內 Gauss-Seidel 求值順序是可觀察語意:改動其中任何一者都不是讓模擬劣化,而是模擬出另一台機器。我們是用慘痛的方式驗證了這件事。一個對 settle 迴圈深度設上限的實驗——只放棄最深的不到百分之一的 settle 活動——在不到 1,000 個半週期內就發散,而更激進的上限在第一個畫格內就讓 CPU 的程式計數器出軌(皆為實測;本書的負結果目錄會再回到這個例子)。不存在優雅降級的區間:被模擬的機器要嘛正常運行,要嘛以與網表臭蟲無法區分的方式崩潰。因此,本研究中的每一項優化都必須通過單一、機械化的正確性定義才獲准採用:

> **性質 1.1(位元精確閘門)。** 一項引擎變更為*位元精確*,若且唯若:在參考工作負載上,所有節點狀態的全狀態 FNV-1a checksum 於 300k、400k 與 1M 半週期處皆不變(golden 值 0x794A43ABDF169ADA、0x9174E19D961CB6E5、0x6D4CCBCE2E9CD599),且通過一個 10M 半週期、精靈密集的 *Super Mario Bros.* 閘門。

在本書全文中,「快」一律指「在性質 1.1 約束下的快」。凡是討論會放寬此性質的技術——行為式記憶體、錐抽象(cone abstraction)、settle 重構——我們都明確將其視為對模型的偏離,並且在本引擎中予以拒絕。

1.2 精確性的代價

湧現式儲存的代價是按事件支付的。引擎屬於 Ulrich 選擇性追蹤(selective trace)原則 [4] 譜系下的事件驅動式:只有輸入發生變化的節點才會被重新求值。一般而言,每次對某節點重新求值,都必須先找出當下經由導通(ON)pass transistor 與它相連的節點集合——即此刻的通道相連元件(CCC),我們稱這個過程為群走訪(group walk)——再解析該群組:把成員的驅動旗標做 OR,透過一張 256 項的優先序查找表分類(接地勝過供電、供電勝過外部驅動、外部驅動勝過上拉、上拉勝過保持),或者,若群組為純浮接,則套用電容 tie-break。狀態變化被寫回後,變化節點的閘極鄰居與通道鄰居會被入列到下一個安定波(settle wave);波一波波排空,直到網表靜止,時脈才會再次翻轉。

這個工作負載的結構由網表決定,而網表並不友善。在目前引擎上實測,一次 100k 半週期的執行約進行 42.85M 次節點重新求值——每半週期約 418 次——其中僅 39.5% 真正需要群走訪。走訪本身極小:77.1% 的群組恰好只有兩個成員,BFS 平均深度 1.13,第 99 百分位是 3,觀測到的最大值是 14。導通群平均 1.13–1.4 個節點。安定波平均每半週期 12.06 個,最多 45 個。換句話說,模擬器的「圖演算法」實際上是數千萬條每條只有兩三次載入的相依指標追逐序列,穿插著佇列維護;而全部重新求值中有 80.1% 最終完全沒有狀態變化——這是一種結構性殘餘,主要由上拉節點(42.0%)與鄰近供電的節點(38.1%)構成:它們被鄰居喚醒,解析之後仍是原來的值。

這個形狀立下了一道傳統加速手段無法穿越的地板。群走訪的每一步載入的位址都依賴前一次載入;引擎的關鍵路徑是一條相依載入鏈,作用在一個小到足以常駐 L1 資料快取的工作集上——這意味著瓶頸是載入到使用的*延遲*(load-to-use latency),不是頻寬,不是算術吞吐量,而且——如同我們評估章節中的硬體計數器直接顯示的——甚至不是 miss 數。這裡沒有大型而規則的計算可以批次化、向量化或卸載到加速器,而我們並非憑信念接受這一點:在本專案自身的實測歷史中,IR 直譯器

比事件驅動引擎慢 2.5%,AOT 程式碼產生慢 3–6 倍,單實例 GPU 移植約慢 10.7 倍,位元平行 BFS 形式約慢 156 倍,把兩顆晶片拆到兩條執行緒約慢 15 倍。這些失敗每一個都有同一個根本原因:事件驅動引擎已經運作在網表的自然粒度上,而那個粒度就是幾奈秒的序列式指標追逐。

還有一個預設值得點名:一個廣泛存在於效能工程實務中的一般印象(我們未找到其正式文獻出處)——受控語言(managed language)不適合這種指標追逐的核心。本書以實測檢驗這個印象。引擎的熱狀態是非託管的陣列結構(structure-of-arrays)儲存——1-byte 的節點狀態、含 12-byte 行內 payload 的 16-byte 節點紀錄、16-bit 的電晶體表、帶去重 hash 的雙緩衝波清單——內層迴圈不做邊界檢查。在這個基底上,C# 引擎在同一台機器、同一工作負載下超越了同家族的原生 C++ 實作(見 1.5 節);本書全程以實測說明,在這類延遲受限的核心上,語言執行環境本身並不是決定性因素。

1.3 問題陳述與方法

本書處理的問題可以狹義地陳述如下:*給定固定的開關級網表與固定的可觀察語意(解析順序、優先序格、電荷 tie-break),在性質 1.1 的約束下,於單顆市售核心上最大化每秒持續半週期數,並以量測——而非論證——刻劃剩餘邊界落在何處*。這個狹義陳述很重要。我們不改變模型,不做近似,不做平行化(我們量測過為什麼不做),也不接受未經 checksum 閘門的速度主張。在這個框架內,1.2 節中引擎的實測剖面恰好容許三條進攻路線,本書回報的工作便圍繞它們組織。

家族一:證明無效,然後壓制。如果 80.1% 的重新求值結論是「沒有變化」,那麼最便宜的重新求值就是根本不曾入列的那一次。困難在於,「這個事件不可能改變任何東西」是一個關於電荷分享模型的語意主張,而它的天真版本錯起來會破壞儲存:我們第一版的同態(same-state)剪枝產生了黑畫面,追查後發現是浮接的動態節點——剪枝悄悄繞過了它們「保持前值」的 tie-break。因此,存活下來的家族(P-1 到 P-4)把每一條壓制規則都綁上一個載入期的*結構安全證明*:一套汙染(taint)分類找出該規則的健全性論證成立的節點群體(有上拉支撐的節點;位於 ForceCompute 通道元件之外的節點),而一個電容支配論證則對可證明永遠贏不了浮接 tie-break 的節點解除汙染(un-taint)——因為它們的電容嚴格小於每一個通道鄰居。這些剪枝合計在進入佇列之前就刪掉約 21% 的節點重新求值,且位元精確。

家族二:證明很小,然後特化。群組大小分布告訴我們:幾乎每一次走訪找到的都只有一或兩個節點。因此,派發層在投入一般走訪之前,先花幾個週期嘗試一個便宜的基數證明:一個所有 pass 閘極當下皆為 OFF 的節點,自己就是自己的群組,可在 O(1) 解析(R-1 動態單例);一個恰有一個 ON 閘極、且其鄰居的 ON 通道全都通回種子的節點,構成一個就地解析的雙節點群組(B1 成對路徑)——逐位元組複製一般路徑的順序敏感細節:成員 hash 清除、兩次供電掃描、tie-break 形式、寫入順序,且所有回退分支都在任何變動發生之前先行判定。另有一個含 3,929 個節點(26.7%)的靜態類別完全沒有 pass 通道,從不走訪。這些都是已知想法的特化,1.4 節會謹慎地說明這一點;它們在這裡的價值,是把基數當作派發時*逐事件證明*的對象所換得的實測收益。

家族三:自我導出佈局。壓制與特化之後剩下的,是相依載入鏈本身;僅存的槓桿是資料住在哪裡、以及熱迴圈為了做出任何決定必須載入什麼。引擎在每次載入時分三遍重建自己的節點 id 空間:先把節點分類到各剪枝類別並以類別為主鍵重編號,使家族一的逐節點安全查表塌縮成對三個 id 邊界的暫存器比較;接著暖機,透過 settle 迴圈的一份冷儀器化副本捕捉 32,768 個半週期內生產級聯的真實初次彈出(first-pop)順序,再以該順序為 locality 鍵重建一次。這套分類在每次 reset 都重新推導並對基準真相驗證,不符時退回安全退化版——即去優化防護(deoptimization guard)模式

[13]。其血統來自不規則計算重排序的 inspector–executor 傳統 [7, 8, 9] 與軌跡導向資料佈局 [10, 11, 12, 20];而本研究新增的部分在於這個組態——事件驅動模擬器在同進程內捕捉自身級聯,沒有 profile 檔、不假設工作負載——以及關於它*為何*有效的實證發現(是順序,不是快取行密度)。

在三個家族之下還有第四個元素,我們把它當作一級工件而非鷹架看待:正確性基礎設施。三個時間點上的 golden checksum,外加一個長時程、精靈密集的閘門;用於定位發散點的逐節點狀態傾印差分(黑畫面剪枝正是靠它診斷出來的);以及一套量測紀律——同日輪替交錯、中位數、針對 1% 以下效應的成對排序——若沒有這套紀律,本書大約一半的百分比都會是熱噪音。在一個近似會災難性失效、而非優雅降級的模型裡,驗證機制不是貢獻之上的額外開銷;它是任何貢獻得以成為「主張」的前提。

1.4 貢獻

我們在陳述貢獻時附上明確的誠實標籤,因為與本研究相鄰的文獻已有五十年之深,而我們的若干機制最恰當的描述是已知想法的變體。解析語意本身不是我們的:在功能上,引擎實作的是去掉 X 態的 Bryant MOSSIM II 模型 [1],群走訪就是 CCC 求值,浮接 tie-break 就是他的節點尺寸格(node-size lattice)。派發與壓制機制坐落在該模型之上;各項的最近鄰先行研究會逐項註明出處。

(1) 帶靜態安全證明的剪枝家族(P-1 到 P-4)。當後續的重新求值可被證明為 no-op 時,我們僅憑靜態結構加上兩端點的即時狀態,就把入列本身壓掉。P-1,同態(same-state)入列剪枝,誠實地說是一個變體:端點相等的旁路精神上承襲 IRSIM 的等電位處理 [3] 與廣義的選擇性追蹤 [4];我們新增的是結構安全汙染分類學——精確刻劃等態合併在何時仍不安全的分類(無上拉節點;ForceCompute 通道元件),而那正是天真版本破壞儲存之處。P-2,針對度 1 無驅動葉節點的關斷隔離剪枝,把在 IRSIM 類模擬器中屬於執行期結果的事實,提升為在入列*之前*套用的載入期證明,省下整條入列–彈出–解析鏈;我們評估它可信地具原創性,但屬次要。P-3 與 P-4 是這個家族最強的主張:一個電容支配論證——電容嚴格小於其所有通道鄰居的節點,永遠不可能成為它所加入的任何群組中電容最大的成員,因此永遠不可能決定一次浮接 tie-break,因此經過它的等態合併可證明無效。最接近的先行實務是 IRSIM 為解析效率所做的人工節點尺寸粗化 [3];我們的是一條自動推導的不等式,花在事件壓制上,且是精確而非近似。合併量測,這個家族刪掉約 21% 的重新求值,並貢獻了消融階梯上最大的單一階差。

(2) 基數特化派發(R-1 與 B1)。誠實標籤:工程上的變體,不是新理論。active 子網路的動態尺寸判定是 IRSIM 的地盤 [3];靜態 CCC 抽取與編譯是 COSMOS 的 [2]。我們的版本是疊在這個想法之上的早退執行過濾器:在派發時證明群組恰為 $\{n\}$(所有 pass 閘極皆 OFF)或恰為 {種子, 鄰居}(一個 ON 閘極,所有回程通道皆關閉),然後就地解析,同時完全複製一般走訪的可觀察語意。實測價值很大:R-1 值 +18.7%;B1 在消融階梯上為 +7.0%(S6→S7,同日交錯;與緊鄰的整理性 commit 分離歸因後約 +8.9%)。

(3) 自我捕捉初次觸碰重排。即 1.3 節的三遍載入:類別為主鍵重編號把剪枝中介資料化為 id 區間的暫存器比較,加上在同進程內捕捉生產級聯的初次彈出順序作為 locality 鍵,每次 reset 驗證並配有去優化式回退 [13]。每個組件都有血統——inspector–executor 重排序 [7, 8, 9]、軌跡導向與快取意識佈局 [10, 11, 12]、編譯器實務中的 profile 導引佈局 [20]——我們一一註明;但我們評估這個綜合體(事件驅動模擬器在同進程內捕捉自身事件級聯,每次載入皆然、零組態、構造上免疫於 profile 過期)是我們與我們的先行研究諮詢都找不到先例的。隨之而來的實證發現,我們相信是新

的,且有硬體計數器佐證:這個鍵的價值在於*已剪枝級聯的順序*,不在快取行密度。在剪枝停用下取得的捕捉達到同等的快取行密度卻毫無收益(±0.0%);同一捕捉在剪枝開啟下收益 +6.17%;把 L1d miss 砍半的那一階段只買到個位數的牆鐘時間,而買到 +5% 的那一階段幾乎不動 miss 計數器。

(4) 零組態安全方法論(以實作呈現,不作理論主張)。上述每一項安全分類——哪些節點是儲存、哪些是匯流排織構、哪些合併對 tie-break 免疫——都是從物理推導,從不依靠名字:上拉旗標、pass 通道上的 union-find、電容比較、驅動腳排除。儲存會自己把自己排除在不安全的剪枝之外,因為大電容在物理上*就是*儲存的本質。從電晶體層級網表自動抽取閘級邏輯、以子圖同構做子電路識別,是成熟的 EDA 實務 [5, 6];相對於這一系以元件庫/結構圖樣比對為導向的自動辨識,我們基於物理性質的辨識用作事件壓制的健全性基底、施加在一份沒有任何人工註記的晶粒衍生網表上,定位為支撐貢獻 (1)–(3) 的方法論。

(5) 一張可證偽的單核開關級模擬效能邊界地圖。負結果也是貢獻,以與成功項目同等的精確度陳述:每一條抽象化路線(IR、AOT/codegen、錐壓縮)與每一條平行化路線(GPU、位元平行、每晶片執行緒)實測皆更慢,並指認了背後機制;維護執行期事實類優化約 7% 的維護地板,由 P-5 系列實驗以公分級精度測繪,包括解釋「為什麼有利可圖的變體無法便宜地做到健全」的語意區分——即時狀態 vs 解析時快照;settle 不足(under-settling)的災難性(而非優雅)失效;以及這一切底下的結構事實:依賴圖的 94% 是單一個雙向強連通元件,而無變化殘餘沒有任何靜態子集(經三種獨立方式確認)。一個帶硬體效能計數器的九階段同日消融量化了每一個被採用的階段。這張地圖在最樸素的意義上可證偽:每一道邊界都是一筆量測,未來的硬體或更好的想法都可能推翻它——事實上,我們自己早先發表的好幾個「天花板」,正是在本專案內被後來的量測推翻的。

1.5 主要結果

以下所有數字皆於同一台機器量得(AMD Ryzen 7 3700X,Zen 2,Windows 11;.NET 10/11 依階段而定),工作負載為 `full_palette.nes`(社群開源的 NES 測試 ROM,顯示完整 NES 調色盤的靜態畫面,取自公開的 nes-test-roms 測試集),同一時間僅一個 benchmark 行程,各階段在同一場次內輪替交錯,使熱漂移均勻落在每個階段;報告中位數,且每一跑都通過性質 1.1 的 checksum 閘門。核心結果是消融階梯:八個歷史引擎階段從其實際 commit 重建,並於同一天並列跑分。從每秒 67,955 半週期的基線分叉出發,被採用的技術複合累積到中位數 132,243(最佳 135,828)——累計 **+94.6%**——其中最大的幾階是 R-1 動態單例(+18.7%)、P-1 同態剪枝(+26.4%)、P-2/P-3/P-4 剪枝補完(+7.2%)、類別為主鍵的區間剪枝重排(+4.2%)、自我捕捉初次觸碰鍵(+5.0%),以及 B1 成對路徑(+7.0%)。

> **主要結果**。單顆 Zen 2 核心上約 136K 半週期/秒——等效約 68 kHz 的矽主時脈——在性質 1.1 下位元精確,且整個 +94.6% 的同日消融在每一階段都經過 checksum 閘門。距離即時(42,954,552 hc/s)的剩餘差距約為 316 倍。

表 1.1. 開關級 NES/6502 家族於同一台機器、同一工作負載、同一單位下的對照(2026-06-08 量測;AprVisual 已更新為目前數字)。原始專案皆無內建 hc/s 量測,我們對其原始碼做了實驗性儀器化以取得同單位數字(詳見 2.4 節)。perfect6502 僅模擬 6502,列出供參考脈絡,不參與排名。

模擬器	範圍	語言	吞吐量(hc/s)	相對值
VisualNes [19]	整機 NES	C++(chipsim.js 逐字移植)	約 24K	0.18×
MetalNES [17]	整機 NES	C++(優化版;我們的祖先)	約 54K	0.40×
AprVisual [21]	整機 NES	C#	**約 136K**	**1.0**×
perfect6502 [18]	僅 6502	C	約 29K(單位不同)	—

硬體計數器的故事可以用一段話講完,而它反轉了一篇快取意識優化論文照理會採用的敘事框架。縱觀整個階梯,每百萬半週期的已執行(retired)指令數下降三分之一(155.6B 降至 104.4B)——剪枝刪的是工作,不只是佇列項——而 IPC 維持在 2.1 到 2.4 之間。重排階段把 L1d miss 從 27.3 砍到 13.2 MPKI,牆鐘時間卻只移動 +4.2%;自我捕捉階段收益 +5.0%,miss 計數器卻幾乎不動;而 L1i miss 在每一階段都停留在 0.17–0.30 MPKI,證實全內聯的直譯器迴圈塞得進指令快取——這就是所有編譯變體落敗的計數器層級解釋(編譯讓程式碼足跡爆炸的地方,正是直譯器分文不付之處)。引擎在一個常駐 L1 的工作集上受限於相依載入*延遲*,而非 miss *數*;決定性的佈局勝利來自已剪枝級聯的順序,而非快取行的緊密排列。

最後是誠實的距離:在這台硬體上要達到即時,需要每秒 42.95M 個半週期,而目前的引擎還差約 316 倍。本書沒有任何地方主張這個差距能在單核上補上;第 9 章的邊界地圖以量測論證它補不上——同時指出,本專案先前發表的每一個天花板都是倒在量測之下、而非倒在論證之下,而這正是提出任何此類主張時恰如其分的不安註腳。

1.6 本書架構

第 2 章建立背景:開關級模型及其解析語意、晶粒衍生網表及其資料格式,以及從 chipsim.js 經 MetalNES 到本引擎的血統 [15, 16, 17]。第 3 章沿四條脈絡綜覽相關研究——開關級學派 [1, 2, 3]、事件壓制與選擇性追蹤 [4]、儲存抽取 [5, 6]、執行期資料重組 [7–12, 20]——並對我們的機制各是哪些已知想法的變體給出明確判定。第 4 章描述引擎本身:非託管的陣列結構佈局、settle 迴圈、各派發類別,以及正確性基礎設施。第 5 至 7 章深入展開三個家族——剪枝家族及其安全證明(第 5 章)、基數特化派發(第 6 章)、自我導出佈局(第 7 章)——每一章都涵蓋機制、健全性論證與實測效果。第 8 章完整呈現實驗評估:消融階梯、硬體計數器剖面、順序對密度研究,以及量測衛生方法論。第 9 章從負結果組裝出邊界地圖。第 10 章以「我們相信哪些東西能遷移到本工件之外、哪些誠實地說只屬於它」作結。引擎、benchmark 套件與公開排行榜皆可在專案儲存庫取得 [21]。

第 2 章 背景:開關級模型、NES 與模擬器家族

本章組裝起本專著其餘部分所立足的四塊背景。首先是*語意模型*:本書討論的每一個模擬器——包括我們的在內——執行的都是一個離散的開關級(switch-level)模型,其本質內容由 Bryant 於 1984 年形式化 [1],而我們在開頭就明白指出我們原封不動繼承了該模型的哪些部分。其次是*對象*:任天堂娛樂系統(NES)的兩顆 NMOS 晶片,其電路慣用手法(比例式(ratioed)邏輯、pass-transistor 網路、預充匯流排、動態電荷儲存)正是迫使模擬必須下到開關級的那些行為。第三是*演算法*:模擬器家族每一代共享的事件驅動 settle 迴圈,我們會在一個小型演練範例上逐步走過它,讓後面的章節可以指名引用它的各個步驟。第四是*家族*本身:從 Visual 6502 計畫的 JavaScript 模擬器、經 MetalNES、到 AprVisual 的譜系,包括每一代各自加上了什麼、以及本工作落在哪裡。最後一節固定術語與單位——特別是主時脈半週期(hc),本書每一個吞吐量數字都以這個單位表示。

2.1 Bryant 開關級模型

2.1.1 雙向開關構成的網路

閘級抽象把數位電路視為布林運算子的有向圖:每個閘有輸入、有輸出,以及一個從前者到後者的函數。對 1970 年代末的 NMOS 晶片而言,這個抽象不只是有損,而是性質上就錯了。MOS 電晶體不是邏輯閘;它是一個*開關*,其通道在導通時對稱地連接兩個節點。那個時代的設計者有系統地利用這種對稱性:單顆 pass 電晶體就實作一個多工器臂、一個暫存器檔案埠、或一條匯流排連接,而電流——因此資訊——在其中流動的方向,取決於周圍電路在那一瞬間的需求。這類結構不存在靜態的輸入/輸出指派,因此若沒有晶粒照片所不攜帶的設計知識,就無法從中抽取出閘級網表。

Bryant 提出的開關級模型 [1],是在保持離散的前提下、仍對這個現實保持忠實的最弱抽象。電路是一組*節點*(電氣網)由被建模為電壓控制開關的*電晶體*相連:閘極節點的邏輯狀態決定兩個通道端點之間的通道是否導通。節點狀態取自集合 {0, 1, X},其中 X 代表未知或無效的電平。計算的進行不是逐閘、而是*逐元件*:在任一瞬間,導通中的通道把電路的非電源軌節點劃分成**通道相連元件**(channel-connected components,CCC)——透過目前 ON 的電晶體彼此可達的極大節點集合——而元件中每個節點的穩態值,由連接到該元件的一切共同決定。由於這個劃分取決於閘極狀態,而閘極狀態又隨模擬進行而改變,元件一般必須動態發現。這種動態發現——在我們的引擎裡是逐事件的廣度優先*群走訪*(group walk)——是本書每一個模擬器的演算法核心,也就是文獻所稱的 CCC 求值(CCC evaluation)。

Bryant 的模型還固定了元件聯合值*如何*解析,而模型的價值正在於此。有兩個物理現象必須被離散地捕捉。第一個是*比例式驅動*(ratioed drive):當一個元件同時連到一條強路徑(例如經一顆 ON 的增強式(enhancement)電晶體接地)與一條弱路徑(空乏式(depletion)上拉負載)時,強路徑獲勝——這正是 NMOS 反相器在其上拉仍導通時產生邏輯 0 的方式。第二個是*電荷儲存*(charge storage):一個完全沒有驅動路徑的元件並不會取任意值;它保有電荷,其節點維持先前的狀態,而當先前狀態不一致的節點被合併時,電容較大者支配較小者。

2.1.2 強度、尺寸與電荷分享

為了讓這兩個現象都能在不做類比計算的前提下可判定,Bryant 的形式化為每顆電晶體指定一個離散的*導通強度*(conductance strength)、為每個節點指定一個離散的*電容尺寸*(capacitance size),各自取自一個小的有序集合 [1]。通道相連元件的穩態回應由訊號上的一個排序定義:經較強電晶體通往電源軌的路徑壓過經較弱電晶體的路徑;任何被驅動的路徑壓過儲存電荷;而在儲存電荷之間,節點尺寸最大者支配。同強度對立訊號之間的衝突、以及歷史未知的電荷,產生 X。模型的中心定理是:這種離散解析與底層 RC 網路的定性行為一致——這正是允許用查表與圖走訪取代類比模擬的依據。

電荷分享值得強調,因為它是隨手移植最容易弄壞的部分,也因為在我們的對象晶片裡它承載著真實的資訊。當一條導通路徑在一個浮接節點與一個較大的被驅動或帶電結構之間形成時,較小節點的電荷被淹沒:離散地說,合併後的元件取電容最大參與者所對應的值。反過來,當一顆 pass 電晶體斷開、把一個節點孤立起來時,該節點無限期保持其最後的值——模型沒有衰減。動態 MOS 設計同時依賴這個行為的兩半:動態閂鎖*就是*一個刻意被孤立的節點,而預充匯流排*就是*一場刻意安排的電荷分享之爭,預充的尺寸設計決定了它依設計要贏還是要輸。一個錯誤解析浮接元件的模擬器,產生的不是稍微不對的波形;它產生的是一台暫存器會忘記自己內容的機器。

Bryant 的模型催生了一系列學術模擬器,其設計軸線預示了本專著的許多內容。MOSSIM II [1] 是模型本身的參考事件驅動實作。COSMOS [2] 展示了逐元件解析可以被*編譯*:靜態抽取通道相連元件,每個元件一次性翻譯成布林求值程式碼,模擬就變成執行那些程式碼——以模型的一般性換取吞吐量。IRSIM [3] 則走增量路線:一個帶線性 RC 時序的事件驅動模擬器,只重建並重新解析每個事件周圍的*活躍*子網路,並有明確的電荷保持規則。這些系統共享的事件驅動紀律源自 Ulrich 的「只模擬活動」原則 [4]:在一個每個時間步幾乎沒有東西改變的大網路裡,工作量必須正比於改變了的東西,而不是存在的東西。這三個系統都會在第 3 章作為對照我們技術定位的先行研究再次出現;在這裡它們的作用是固定詞彙。

2.1.3 我們的解析語意:MOSSIM II 減去 X 態

我們現在陳述 AprVisual——以及在實作細節之外,visual6502 譜系中的每一個模擬器——實際執行的解析語意,並對其出處刻意直白。**這個模型在功能上就是 Bryant 的模型,移除 X 態。**節點狀態是二元的。每個節點帶一小組旗標:`Gnd` 與 `Pwr` 標記兩條電源軌;`SetHigh` / `SetLow` 標記外部接腳驅動;`PullUp` 標記空乏式上拉負載;`ForceCompute` 標記某些帶有特殊競爭規則的主機板層級匯流排節點。通道相連元件的解析是一個兩階段函數:所有成員的旗標 OR 在一起,所得位元組索引一張 256 項優先序表。

> **性質 2.1(解析函數)**。令 F 為通道相連元件全體成員旗標的逐位元 OR(電源軌貢獻其旗標但從不加入走訪)。元件的新值由第一條符合的規則決定:(i) 若 F 同時含 `ForceCompute` 以及 `Gnd` 與 `Pwr` 兩者,則先消去兩個電源軌旗標再繼續;(ii) `Gnd` $\to 0$;(iii) `Pwr` $\to 1$;(iv) `SetHigh` $\to 1$;(v) `SetLow` $\to 0$;(vi) `PullUp` $\to 1$;(vii) 否則元件為純浮接,所有成員取電容代理值嚴格最大之成員的前一狀態,平手時最先被走訪遇到的成員獲勝。

與 Bryant 的格(lattice)之對應是直接的,但被粗化了。強度順序「電源軌 > 外部驅動 > 上拉 > 儲存電荷」是他的導通強度階層的一個固定四級實例——這已足夠,因為這些晶片所用的 NMOS 製程恰好只有一種強元件與一種弱負載。浮接 tie-break 是他的節點尺寸比較,以節點的靜態連接數作為電容代理:網表不帶任何抽取出的電容值,而附著的通道端點數正是這個家族從其 JavaScript 祖先 [15] 繼承下來的拓撲替代品。在我們的引擎裡,整條優先序階梯被預先計算成一張以 OR 後旗標位元組為索引的 256 項查找表(MetalNES 的貢獻,2.4.3 節),於是解析的第一階段是一次圖走訪、第二階段是一次載入。

```
// 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);
```

刪除 X 失去了什麼?三件事,而每一件都以結構手段而非語意手段補償。未知的初始電荷無法表示;這個家族代之以一個決定性的開機程序:把每個節點入列一次,並把整個網路安定到一個可重現的固定點,此後所有電荷都有了確定的歷史。驅動衝突無法產生「無效」;規則 (ii) 決定性地讓接地獲勝,這對電源軌之爭符合 NMOS 的電氣現實,而在它唯一出錯的地方——對立的常開路徑相遇的主機板層級匯流排節點——由規則 (i) 的 `ForceCompute` 互消修補。振盪無法靠 X 傳播被標記;它在操作層面由 2.3.1 節的 settle 迴圈結構所界定,而我們指出在這些網表上安定實際上至多 45 個波就收斂(實測;2.3.1 節)。

我們堅持「我們在功能上*就是* MOSSIM II 減去 X 態」這個表述、而不採用更柔軟的措辭,有兩個塑造整本專著的理由。第一是歸屬上的誠實:我們的解析語意沒有任何新東西,讀到我們的優先序表或浮接 tie-break 的讀者,應該立刻被告知他們讀的是經 chipsim.js 過濾後的 Bryant 模型 [15, 1]。後面各章主張的每一項貢獻,都是*在一個四十年未變的語意模型之上,關於排程、壓制、證明與記憶體佈局*的貢獻——從不是模型本身的貢獻。第二個理由是方法論上的:我們的核心正確性契約——位元精確(bit-exact,2.5 節)——之所以有意義,正因為語意是固定的、已知的、且跨實作共享的。一個「改良」模型的模擬器沒有可供位元精確對照的黃金參考;一個凍結模型的模擬器,則可以把任何位元組層級的發散——在 14.7K 節點、一千萬個半週期之中的任何一處——按定義視為缺陷。凍結是這套方法的基礎。

2.2 電晶體層級下的 NES

2.2.1 兩顆客製晶片

任天堂娛樂系統(1983 年的 Famicom 之 1985 年外銷型態)圍繞兩顆客製 NMOS 晶片建成。**2A03** 是 CPU 元件:一顆授權的 MOS 6502 核心、十進位修正電路被停用,與音訊處理單元(兩個方波聲道、三角波、雜訊與一個取樣聲道)、一個精靈 DMA 引擎、以及控制器 I/O 埠共用同一塊晶粒。**2C02** 是圖像處理單元(PPU):它擷取背景與精靈圖樣資料,從其晶粒上的物件屬性記憶體(OAM)對

目前掃描線評估至多 64 個精靈,合成一幅 256×240 的畫面,並直接產生視訊訊號時序。兩顆晶片由單一主時脈驅動,經一套多工匯流排織構(bus fabric)通訊;CPU 把主時脈除以 12、PPU 除以 4,因此這台機器自然的全域時間基準比任一晶片的指令或像素粒度都細(2.5 節)。

兩顆晶粒都已被拍攝,其完整的電晶體級網表由 Quietust [16] 依 Visual 6502 計畫為原版 6502 建立的方法 [15] 從照片中抽取:逐層拍攝晶粒,為擴散層、多晶矽與金屬層描出多邊形,再把多邊形堆疊編譯成節點與電晶體。其結果不是依文件畫出的電路圖,而是以圖的形式呈現的矽本身,包含每一個設計上的特異之處、每一個未見諸文件的行為,以及——對本書至關重要——每一個其功能只存在於電荷層次的結構。

值得停下來說明這樣一份網表*不是*什麼。它不帶任何意圖標籤:沒有「這是一個暫存器」、沒有「這條匯流排是預充的」、沒有時脈域註記(節點名稱檔確實為一些內部訊號命名,但名字是文件、不是結構,而我們的方法論拒絕依賴它們)。它不帶任何抽取出的電氣參數:沒有電容,也沒有可作為強度使用的電晶體幾何。它是一張二部連通圖加上一份上拉標記——而模擬器所做的一切都必須僅從這些推導出來。正是這種簡樸,使得後面各章「以物理而非名字辨識結構」的紀律既必要又可能。

2.2.2 NMOS 電路慣用手法

四種 1970 年代末 NMOS 設計的慣用手法支配著這兩顆晶粒,每一種都對應到 2.1 節模型的一個特定特徵。

比例式增強/空乏邏輯。基本的 NMOS 閘是一顆常導通的空乏式上拉負載壓在一個增強式下拉網路之上。邏輯 0 的產生不是靠切斷高側,而是靠*壓過*它:下拉路徑的尺寸設計讓它贏得這場拉鋸。在模型裡這就是規則順序——性質 2.1 中 `Gnd` 位階高於 `PullUp`——而在網表資料裡它呈現為逐節點的上拉旗標、而非顯式的負載電晶體:我們量測到兩顆晶片網表中電晶體記錄的第七欄(「weak」)在每一列都是 false,因為抽取過程已把所有空乏式負載折疊成逐段(per-segment)的上拉標記。

Pass-transistor 網路。多工器、暫存器讀寫埠與內部匯流排開關都由裸 pass 電晶體構成。它們就是 2.1.1 節的雙向結構,也是通道相連元件必須逐事件發現的原因:哪些節點構成一個元件,取決於*此刻*哪些閘極是高電位。這種動態性在運轉中機器上的實測形狀十分醒目,且在全書反覆出現:平均導通元件只含 1.13–1.4 個節點,而在優化後引擎中抵達一般解析器的元件走訪裡,77.1% 恰好有兩個成員。這兩顆晶片是一片開關之海,但在任一瞬間幾乎全部都是斷開的。

預充動態匯流排。長的內部匯流排在一個時脈相位被預充到高電位、在另一個相位有條件地放電,以一種有時序的電荷分享紀律取代緩慢的比例式上拉。模擬器並不特別建模這件事;它從規則 (vi)–(vii) 與時脈中*湧現*。然而其成本是結構性的:無論匯流排值是否改變,預充流量每個相位都會重現,而我們量測到運轉中引擎裡大約四分之一的節點重新求值,屬於這種時脈驅動的記憶體與匯流排織構活動。多個優化章節會回到這個數字。

動態電荷儲存。暫存器、管線閂鎖、尤其是 PPU 的精靈記憶體,都把位元儲存為刻意孤立節點上的電荷,由雙相時脈刷新。2C02 的 OAM 是極端案例:它不是一塊 RAM 巨集,而是大約 2,300 顆浮接的儲存電容。在模型裡,這些位元每一個都活在規則 (vii) 裡——純浮接最大電容 tie-break *就是*晶片的儲存機制。這個單一事實驅動了本書的大半:它是 tie-break 順序屬於可觀察語意的原因(2.3.5 節),是擾動浮接解析的近似會災難性而非優雅地失效的原因,也是剪枝各章的電容支配(capacitance

dominance)安全證明之所以必須存在的原因。我們指出,在平坦電晶體網表中辨識儲存結構本身就是一個經典的 EDA 問題 [5, 6];這裡不尋常的是,模擬器必須隱式地、在每個事件上、永遠把它們的*行為*做對。

2.2.3 網表交換格式

網表以 Visual 6502 計畫的三檔案 JavaScript 格式發佈 [15, 16],摘要見表 2.1。`segdefs` 檔列出矽多邊形:每筆記錄給出多邊形所屬的節點、一個 pull 標記(`'+'` 為上拉、`'-'` 為下拉)、層索引,以及多邊形座標。`transdefs` 檔把電晶體列為(閘極、通道端點 1、通道端點 2)三元組,附包圍盒與幾何欄位;2A03/2C02 變體再附加前述的第七個 weak/空乏布林欄。`nodenames` 檔把人類可讀的名字映射到節點編號——電源軌、時脈、外部接腳,以及(對這兩顆晶片而言)許多內部訊號,包括 CPU 架構暫存器的逐位元名稱。

表 2.1. Visual 6502 / Quietust 網表交換格式 [15, 16]。

檔案	記錄	內容	備註
`segdefs`	多邊形	節點 id、pull 標記(`'+'` / `'-'`)、層、頂點列表	pull 標記承載空乏式負載;我們同時保留 `'+'` 與 `'-'`
`transdefs`	電晶體	name, gate, c1, c2, bbox, geometry [, weak]	c1/c2 可互換(雙向通道);weak 欄在兩顆晶片實測逐列皆為 false
`nodenames`	名稱 → id	例如 `vcc`、`clk0`、`res`、`a0..a7`、`pcl0..pcl7`	名稱可能帶 `/` `#` `~` `_` 前綴;2C02 把其 `ab` / `db` 匯流排多工到共用節點上

這個格式有兩個性質影響後續工作。第一,通道端點是無序的:c1 與 c2 可互換,因為元件是對稱的,任何在引擎資料結構中強加方向的做法,都必須是正規化步驟、而非語意主張。第二,格式的連通性是逐晶片的;要組出一台主機需要一個組成層,而這個家族直到 MetalNES 才取得它。

2.2.4 主機板層級組成與行為式記憶體

一台主機不只是它的兩顆客製晶片。NES-001 主機板再加上 2 KB 的 CPU 工作 RAM、2 KB 的視訊 RAM、卡匣(在我們模擬的無 mapper NROM 組態下:程式 ROM 與字元 ROM),以及諸如 CPU 與卡匣之間位址閂鎖之類的 TTL 膠合邏輯。AprVisual 繼承 MetalNES 的組成機制 [17]:系統由模組定義檔描述,每個檔宣告其接腳、子模組實例、模組間連接、逐節點上拉、`ForceCompute` 節點集合,以及附掛在該模組上的任何行為式記憶體。載入時,每個模組實例獲得一段私有的節點 id 範圍,跨模組連接則實作為一顆連接兩節點的常開電晶體。

那些常開的連接電晶體在語意上是 no-op——兩個由永久導通通道相連的節點就是同一個電氣網——我們載入器的 *lowering* 遍會把它們合併掉,熔合 441 個節點與 530 顆電晶體。本書所有數字所指的、組成並 lowering 後的整機圖,含大約 14.7K 個節點與 26.8K 顆電晶體。其載入期計算的結構普查為:3,929 個節點(26.7%)完全沒有 pass-transistor 通道、永遠不可能加入多節點元件;10,784 個節點(73.2%)有通道、因而是動態元件候選;另有約 16 個節點的殘餘帶著行為式 callback。這三個母體就是優化各章的派發類別 cls1、cls2 與 cls0(2.5 節)。

記憶體是這個家族唯一刻意偏離電晶體保真度的地方。把 4 KB 的主機板 SRAM 加上各 ROM 模擬成電晶體,只會把網表規模放大數倍而沒有任何保存上的收益——主機板上的記憶體不同於晶粒上的 OAM,是普通的市售零件。MetalNES 引入、而我們保留了*行為式記憶體處理器*(behavioral memory handlers):在相關匯流排節點上註冊的 callback(機制上是一顆在其節點解析時觸發的「假電晶體」)直接實作 RAM 與 ROM 陣列語意 [17]。這條邊界有原則:兩顆客製晶粒上的一切都在開關級模擬,包括每一個動態儲存節點;主機板上原本是插槽式市售晶片的一切則是行為式的。性質 2.1(i) 的 `ForceCompute` 競爭規則正是為同一條邊界而存在——某些因這種組成而使兩條電源軌都可達的匯流排節點,需要那個互消才能像真實的開集極織構那樣解析。

2.3 每一代共享的演算法

2.3.1 事件、波與 settle 迴圈

這個家族的所有模擬器都是 Ulrich 意義下的事件驅動 [4]:工作量正比於狀態變化。外部刺激的單位是主機板時脈節點的一次翻轉——一個主時脈半週期,簡記 *hc*——之後引擎必須把後果傳播到網路靜止為止。傳播結構是一個雙緩衝工作清單,而它的排空就是本書一切量測對象的最內層迴圈。

```
// 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
```

這個迴圈的幾個細節攸關全局。兩份清單是*以波為單位緩衝*的:在排空目前清單時入列的節點落到下一份清單上,於是傳播以廣度優先的世代推進——我們把每一次排空稱為一個**安定波**(settle wave)。一個逐節點 hash 在波內對入列去重,而群走訪會清掉它吸收的每個節點的 hash 項,從而取消該節點在目前波中待處理的彈出:一個剛剛作為別人元件的一部分被解析過的節點,不可以在一瞬間之後又被多餘地重新求值。閘極轉變的入列是不對稱的:閘極轉高使其通道導通,所以入列一個端點就夠了(走訪會穿過通道找到另一端);閘極轉低可能把一個元件一分為二,所以兩個端點都必須各自重新解析。

在真實工作負載上,這個迴圈的實測形狀如下(所有數字為組成後主機跑標準測試 ROM、每 100k hc 統計):每半週期約 418 次節點求值(「彈出」,pop);每半週期平均 12.06 個安定波、觀測最大值 45;元件走訪的廣度優先深度平均 1.13 層、第 99 百分位 3、最大 14。這些數字描繪的圖像——每半週期數千次微小、淺層、受延遲所限的圖探測,任何地方都沒有大而規則的結構——是本書每一

個優化都必須在其上作戰的地形,也是對我們稍後以負結果回報的每一個批次化、編譯與平行化策略的量化反駁。我們也先於方法論各章指出:安定深度並非鬆弛——把波數設上限的實驗,在不到一千個半週期內就偏離了參考。安定分布的深尾是這份網表真實的關鍵路徑,不是假象。

2.3.2 群走訪與解析

群走訪(group walk)是 2.1.1 節的動態 CCC 發現,特化到手上的事件。走訪以被彈出的節點為種子,維護一份成長中的成員清單(兼作 BFS 佇列)與一個成員旗標的 OR 累加器。對每個成員,引擎掃描該節點的通道端點記錄:對每顆通道觸及該成員的電晶體,若其閘極節點目前為高,遠端端點就加入元件——除非它是電源軌,此時走訪不跨越、而是把該軌的旗標 OR 進累加器。成員資格以逐節點標記去重,因此走訪的時間線性於元件所關聯的電晶體記錄數。佇列排空後,累加的旗標位元組通過性質 2.1 的 256 項表;若它完全為空,浮接分支就掃描成員清單找出最大連接數,並採用該成員的前一狀態。

解析出的值接著經由 settle 迴圈的狀態變化過濾器寫回*每一個*成員:已經等於該值的成員不被觸碰(關鍵地,也不再產生任何事件);改變了的成員觸發上述閘極側的入列,為下一個波播種。這種「走訪後廣播」的結構意味著一次彈出可以一次安定多個節點;反過來說,一個節點的值也可能被別處播種的走訪改寫——當後面的章節推理哪些入列可被證明為多餘時,這兩個事實都很重要。

值得再說一次這些元件在實務上有多小。這份網表上的平均導通元件含 1.13–1.4 個節點;在抵達一般解析器的走訪中,77.1% 恰好兩個成員、16% 三個、只有 5.7% 四個以上。模型允許任意大的元件——開機安定也確實短暫產生過大元件——但穩態下的機器壓倒性地是一台單例與成對的機器。優化各章的派發類別架構(cls1/cls2 快速路徑、成對路徑)不過就是把這個分布當真。

2.3.3 浮接 tie-break 就是儲存機制

性質 2.1 的規則 (vii) 值得單獨一小節,因為它是引擎裡後果最重大的一行。當一個元件完全沒有旗標——沒有電源軌、沒有外部驅動、沒有上拉——模型宣告它為電荷隔離,並把其最大電容成員的前一狀態指派給它。這兩顆晶粒上的每一個動態閂鎖、每一個管線暫存器位元、每一個 OAM 細胞,都*只*透過這條規則儲存它的位元。系統中沒有任何閂鎖原語:儲存是解析函數作用於刻意孤立節點時的湧現性質。

由此有三個後果。第一,tie-break 在細部的正確性沒有商量餘地:任何擾動浮接解析中哪個成員獲勝的改動,即使只在少數幾個冷僻節點上,都會破壞架構狀態——我們的專案史上就有一次黑畫面退化,被追查到正是這樣一個對無上拉節點的擾動。第二,這條規則的輸入是可觀察語意的一部分:嚴格大於的比較、先見者勝的平手順序、以及因此走訪遇到成員的順序,在任何對走訪的改寫中都必須逐位元組保持(正因如此,後面某章的成對路徑(pair path)優化帶著一份明確的義務清單)。第三,而且是建設性的:由於儲存節點在*物理上*就是大電容參與者——擁有支配性的電容正是讓一個節點能用作儲存的原因——對電容代理的結構不等式可以證明某些重新求值不可能改變任何值。這個觀察成長為電容支配剪枝,本工作最強的原創性主張;在這裡我們只指出,使它成為可能的正是規則 (vii)。

2.3.4 演練範例:一個事件穿過動態閂鎖

我們現在在一個刻意極簡的電路上,把演算法沿一個事件走過一遍。*這個電路是為本節而作的說明性虛構,不是任一網表的節錄*,但它正是真實晶粒上實例化了數千次的慣用手法——一個 NMOS 反相器餵入一個 pass-transistor 動態閂鎖。

```
          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
```

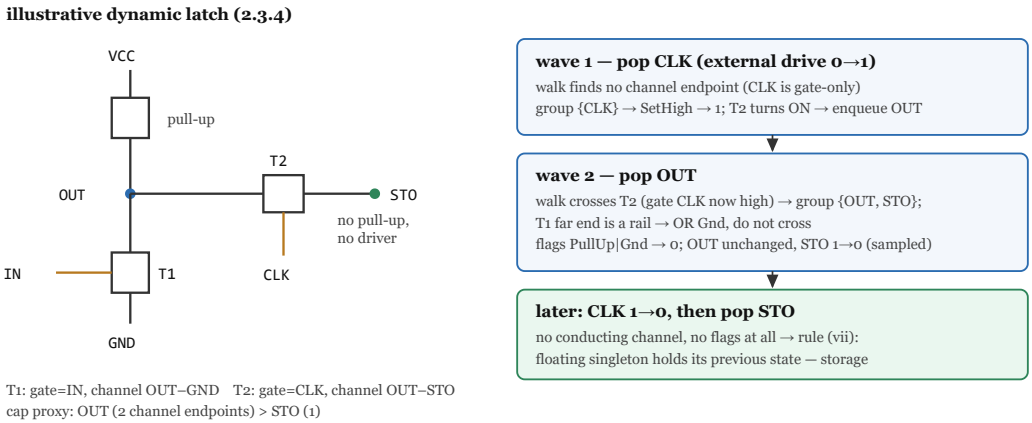

圖 2.1. 2.3.4 節的演練範例:一個 NMOS 反相器餵入 pass-transistor 動態閂鎖。左:電路與電容代理;右:CLK 上升沿事件逐波展開,以及 CLK 下降後對 STO 的彈出如何落入規則 (vii) 的浮接保值分支——儲存由此湧現。

設初始狀態為:IN 被驅動為高(`SetHigh`),故 T1 導通;CLK 被驅動為低,故 T2 斷開;OUT = 0(其元件經規則 (ii) 解析:`PullUp` 與 `Gnd` 的 OR 是接地獲勝——比例式反相器在運作);STO = 1,一個先前取樣、由電荷保持的位元。連接數代理給 OUT(兩個通道端點)比 STO(一個)更大的電容。事件:外部驅動器把 CLK 從低翻到高。

驅動變化把 CLK 的旗標從 `SetLow` 改寫為 `SetHigh` 並把 CLK 入列;settle 迴圈啟動。**波 1** 彈出 CLK。以 CLK 為種子的群走訪找不到任何通道端點——在這個電路裡 CLK 只接閘極——所以元件是 {CLK},OR 後的旗標是 `SetHigh`:規則 (iv) 解析為 1。寫回使 CLK 0→1,於是掃描 CLK 的閘極列表:T2 導通,而導通情況入列一個通道端點,OUT。**波 2** 彈出 OUT。從 OUT 出發的走訪檢查 OUT 的通道:T2 的閘極(CLK)現在為高,所以 STO 加入元件;T1 的閘極(IN)為高,但其遠端端點是接地軌,所以走訪不跨越——它把 `Gnd` OR 進累加器。元件是 {OUT, STO},旗標為 `PullUp | Gnd`;規則 (ii) 解析為 0。寫回:OUT 維持 0(無變化、無事件);STO 改變 1→0。STO 在此不作任何閘極,所以沒有東西入列,下一份清單為空,網路在兩個波、兩次彈出、一次雙成員走訪之後靜止——正是實測分布的眾數事件形狀。閂鎖已透明地取樣了 NOT(IN)。

現在讓時脈下降:CLK 1→0。關斷情況把 T2 的*兩個*端點都入列,因為通道現在可能把它們分開。下一個波彈出 OUT:其元件只有 {OUT}(T2 斷開),旗標 `PullUp | Gnd`,值 0,無變化。它彈出 STO:走訪找不到任何導通通道、也完全沒有任何旗標——STO 沒有上拉、沒有驅動、也不是電源軌——所以規則 (vii) 的浮接分支在這個單例元件上執行,回傳 STO 自己的前一狀態 0。什麼都沒變;機器再度靜止。

第二個事件才是有教益的那個。位元存活下來:從現在直到 T2 下次導通,STO 在沒有任何驅動者的情況下保持 0——若 IN 此刻改變、OUT 擺到高,STO 不受影響,因為 OUT 的活動止於斷開的通道。這就是從規則 (vii) 湧現的動態儲存,正如 2.3.3 節所承諾。而對 STO 的那次彈出,可證明是一次浪費:一個唯一通道剛剛斷開、自身又沒有驅動的節點,*必然*浮接並保值——重新求值它永遠不可能改變任何東西。乘上整顆晶粒、乘上每一個時脈邊緣,這類有保證的 no-op 在全部工作中佔可量測的比例,而在它們入列之前就壓制掉其中的若干類——附上載入期的結構性安全證明——正是剪枝各章的主題。

2.3.5 波內順序是可觀察語意

settle 迴圈有一個性質必須謹慎陳述,因為它約束本書的每一個優化:**波內的求值順序是語意的一部分**。狀態陣列就地更新,所以同一個波中較晚的彈出會讀到較早彈出寫下的值——這個疊代是 Gauss–Seidel,不是 Jacobi。一個波的入列順序固定了下一個波的彈出順序;彈出順序決定哪個節點為某元件的走訪播種;種子與鄰接順序決定走訪遇到成員的順序;而遇到順序餵入浮接 tie-break 的先見者勝條款——那就是儲存。這條鏈上沒有任何一處有鬆弛。

這不是理論上的講究。這個家族的各參考實作全都執行同一套 FIFO 紀律,這正是它們能逐位元一致的原因;而我們自己的實驗證實了逆否命題——波內重排的嘗試改變了模擬結果,並以語意無效、而不僅是無利可圖之名被放棄。對工程的後果是一條我們將反覆援引的硬規則:*優化可以刪除可證明不可能改變狀態的工作,也可以用任何能逐位元組重現走訪寫入的方法解析元件,但它永遠不可以重排剩下的工作*。把一個波內的工作平行化被同一個論證排除——獨立於其(同樣被量測、同樣致命的)同步成本。

> Gauss–Seidel 順序、FIFO 波紀律與先見者勝 tie-break,是從 chipsim.js [15] 繼承下來的偶然取捨,不是物理定律;換一個參考順序就會定義另一個——同樣站得住腳的——固定點。但保存性質的計畫必須選定一套語意並凍結之,而這個家族凍結了這一套。本書中的位元精確永遠意指「相對於那個被凍結順序的位元精確」。

2.4 模擬器家族

在這些網表上執行這個模型的模擬器構成一棵單一的家族樹,見圖 2.2。一段 JavaScript 是共同的祖先;各分支在範圍、對該祖先的忠實程度上各不相同,而——一如第 1 章所預告、評估各章所量化——最逐字的後裔與我們之間的吞吐量差距約為五倍半。

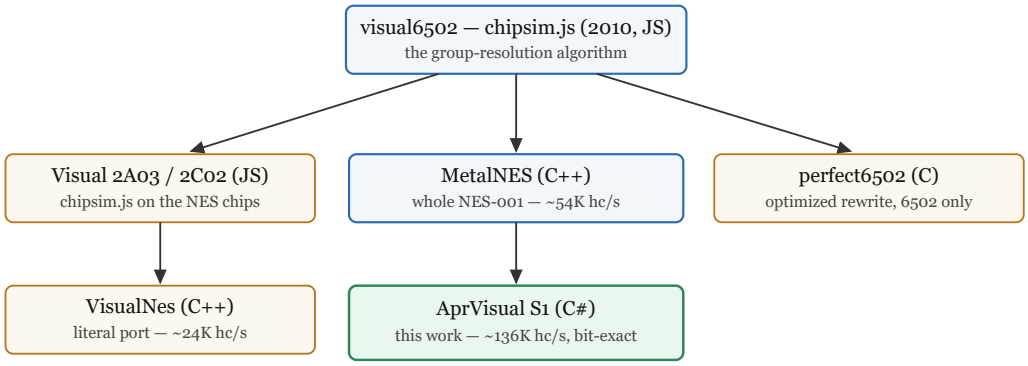

圖 2.2. 模擬器家族:Visual 6502 計畫的 chipsim.js [15] 與 Quietust 的 2A03/2C02 網表 [16] 衍生出 VisualNes [19](逐字移植)、perfect6502 [18](僅 6502 的優化重寫)與 MetalNES [17](重新工程化的整機組成),AprVisual 由後者衍生 [21]。

2.4.1 visual6502 與網表

Visual 6502 計畫 [15] 確立了整個文類:從晶粒照片抽取網表,然後用幾百行 JavaScript——`chipsim.js`——在瀏覽器裡讓它動起來,實作的正是 2.3 節的事件驅動安定、群走訪與優先序解析。這個模擬器的目的在於解說與保存,不在於快;它的遺產是其*語意*——包括上面討論的每一個偶然的順序決定——成為這個家族事實上的參考模型。Quietust 的 Visual 2A03 與 Visual 2C02 [16] 把同一套抽取方法應用到兩顆 NES 晶片,並以同一格式發佈網表,帶兩個值得注意的增補:transdefs 的第七欄(2.2.3 節),以及把 CPU 內部暫存器逐位元命名的節點名稱檔——對除錯是方便,而我們的方法論刻意拒絕依賴它。

2.4.2 三個後裔

接著,三個彼此獨立的專案把這段 JavaScript 帶離瀏覽器,而它們不同的企圖心使它們碰巧構成一場關於實作品質的對照實驗。**VisualNes** [19] 是把兩個晶片模擬器逐行移植成 C++ 並接成一台主機;它如此忠實地保留了 JavaScript 的資料結構與低效之處(包括群走訪中一個二次方的成員資格測試),以致它在本書中充當*未優化基準線*——演算法被翻譯但未被工程化時的成本。**perfect6502** [18] 是相反的一極:一個真正優化過的 C 重寫,帶位圖狀態儲存與預計算的相依清單,但範圍只到裸 6502,且——對我們的目的而言是決定性的——沒有浮接元件的電荷保持模型。對單獨一顆 6502 而言這個簡化還能存活;對一台視訊晶片把精靈記憶體存成孤立電荷(2.2.2 節)的主機而言它是出局的,這也是為什麼沒有任何整機 NES 分支從它長出來。

MetalNES [17] 是重新工程化的後裔、我們的直系祖先:對完整 NES-001 主機板的 C++ 電晶體級模擬。它在我們參考機器上的實測吞吐量(約 54K hc/s,對比 VisualNes 的約 24K;2.4.4 節)量化了在演算法不變的前提下,有紀律的原生工程能買到什麼——大約是逐字移植版的兩倍。

2.4.3 MetalNES 加上了什麼

MetalNES 的貢獻定義了 AprVisual 的出發平台,我們在此明確致謝。第一,*整系統組成*:2.2.4 節的模組定義機制,把兩份晶片網表加上主機板 TTL 組裝成一台被模擬的主機——更早的家族成員沒有一個模擬過系統。第二,*256 項解析表*:把性質 2.1 的優先序階梯預先計算成一張以 OR 後旗標位元組為鍵的查找表,取代最內層迴圈中的分支瀑布,並在旗標分類學裡區分「被驅動為高」與「保持為高」。第三,*行為式記憶體處理器*:讓市售主機板記憶體免於電晶體模擬的 callback(「假電晶體」)機

制。第四,*ForceCompute*:用於組成後匯流排節點的電源軌互消規則。MetalNES 也朝一個我們刻意不跟進的方向延伸——在電壓層級建模視訊與音訊 DAC 的類比輸出階梯——而它欠缺兩樣 AprVisual 必須補上的東西:它只保留 segdefs 的 `'+'` 上拉標記,且它沒有位元精確方法論、也沒有 checksum 紀律。

2.4.4 AprVisual 在家族中的位置

AprVisual 始於對 MetalNES wire core 的逐函數 C# 移植——移植刻意保守,使起點是一個已知量——然後沿著本專著其餘部分所記錄的兩條軸線分岔。第一條軸線是*驗證*:固定半週期數上的全狀態 checksum、跨專案歷史凍結的 golden 值,以及一道精靈密集的一千萬半週期閘門(2.5 節)。第二條軸線是*模型不變下的效能*:後面各章的派發類別、剪枝家族、重編號與佈局技術,每一項都以驗證軸為閘門。表 2.2 給出這個家族的實測名次——同一台機器、同一個 ROM、同一個單位,全部由我們自行編譯並測試,而非引用各作者的數字。

表 2.2. 同機、同 ROM、同單位下實測的家族(主時脈半週期每秒;2026-06-08 量測,AprVisual 的目前數字為 2026-06-12)。perfect6502 一列使用 6502 時脈半週期——粗一個數量級——因此不可與整機各列排名比較。

模擬器	範圍	語言	與 chipsim.js 的關係	吞吐量
VisualNes [19]	整機 NES	C++	逐字逐行移植	約 24K hc/s
MetalNES [17]	整機 NES	C++	重新工程化的後裔	約 54K hc/s
perfect6502 [18]	僅 6502	C	優化重寫,無電荷模型	約 29K(6502 hc;單位不同)
AprVisual [21]	整機 NES	C#	MetalNES 移植 + 本書的計畫	約 136K(最高紀錄)

表中 VisualNes、MetalNES 與 perfect6502 的數字並非各專案的原生輸出——這些原始專案皆不內建 hc/s 之類的吞吐量測;為了同單位比較,我們對各專案原始碼做了實驗性修改,加入與本書相同定義的半週期計數與計時,並在同一台機器、同一工作負載下量測(第 8 章)。

把這張表讀成一條進度條——未優化的演算法 24K、有紀律的原生工程約 54K、本工作約 136K——它分開了三個常被混為一談的貢獻:演算法、其實作品質、以及建立其上的優化計畫。最後兩列之間的差距是本書的主題;懸在全部之上的剩餘差距也是故事的一部分,我們在此為校準陳述一次:真實的矽以每秒 42,954,552 個半週期運轉,所以目前的引擎離即時大約 316×。本專著的評估各章將以實測的負結果而非直覺論證:對忠實執行這個模型的單一核心而言,這個殘餘差距是結構性的。

2.5 術語與單位

全書中,效能以**主時脈半週期每秒(hc/s)**表示。一個 *hc* 是主機板時脈節點的一次翻轉——引擎的基本刺激步(2.3.1 節),也是組成後系統所擁有的最細時間基準。主機所有的時脈都是它的固定整數分頻,所以 hc/s 的吞吐量可直接換算成「被模擬的矽跑多快」;表 2.3 彙整全書各處使用的換算。

表 2.3. NTSC NES 的時間基準換算。R 表示以 hc/s 為單位的實測吞吐量。

量	關係	實機數值
主時脈半週期(hc)	主機板 `clk` 節點的一次翻轉	42,954,552 hc/s
主時脈	hc / 2	21.477 MHz
PPU 像素時脈	hc / 8	≈5.37 MHz
CPU 時脈	hc / 24	≈1.79 MHz
一個視訊畫格	714,732 hc	60.0988 畫格/s
等效畫格率	R / 714,732	—
與即時的距離	42,954,552 / R	—

選這個單位是為了跨家族的誠實:它計數整機模擬器實際執行的工作、與哪顆晶片正忙無關,並且暴露以每「cycle」計的數字所掩蓋的單位不匹配(表 2.2 的 perfect6502 註腳即現成的例子)。在目前的約 136K hc/s 下,引擎每秒約畫出五分之一個畫格;即時的要求是表 2.3 的 42.95M hc/s。

位元精確(bit-exact),自此作為技術名詞使用,意指以下具體協定。引擎對完整的節點狀態陣列計算一個 FNV-1a 雜湊;一個組態若在 300k、400k 與 1M 半週期重現被凍結的 golden checksum(分別為 0x794A43ABDF169ADA、0x9174E19D961CB6E5 與 0x6D4CCBCE2E9CD599),並另外通過一道在精靈密集商業工作負載(Super Mario Bros.)上的一千萬半週期閘門,即為位元精確。由於雜湊涵蓋每一個節點,一致意味著整個被模擬晶粒狀態的逐節點、逐快照相等,而不僅是架構可見暫存器的相等;又由於語意被凍結(2.1.3 節),任何地方的任何發散按定義都是缺陷。本書回報的每一個優化,都先通過這個協定,然後才考慮其效能。

最後是全書使用的引擎內部詞彙:一次**彈出**(pop)是從 settle 工作清單取出的一次節點求值;一個**安定波**(settle wave)是雙緩衝清單的一次排空;**群走訪**(group walk)是 2.3.2 節的逐事件 BFS(即文獻的 CCC 求值 [1]);而**派發類別**依 2.2.4 節的結構普查劃分節點——**cls1** 是靜態單例(沒有 pass 通道:元件可證明永遠是 $\{n\}$),**cls2** 是動態單例(dynamic singleton)候選(通道存在,但每當其所有閘極瞬時皆 OFF 時,元件「此刻」就是 $\{n\}$),**cls0** 是必須永遠走一般路徑的 callback/ForceCompute 殘餘。這些名字在每個優化章節反覆出現;其母體——分別為節點的 26.7%、73.2% 與不到一個百分點——是 2.3.1 節的動態量測所懸掛其上的靜態骨架。

本書沿用專案內部發展出的命名;為避免與既有文獻脫節,表 2.4 給出這些名稱與 EDA 及系統文獻慣用術語的對應。正文首次引入各機制時,亦會就地重述對應的出處。

表 2.4. 本書用語與文獻慣用術語的對應。

本書用語	文獻慣用術語	出處
群走訪(group walk)	通道相連元件(CCC)求值	[1, 2]
安定波(settle wave)	事件驅動模擬中事件佇列的一輪排空;與 selective trace 同源	[4]
旗標 OR + 優先序 LUT 解析	開關級驅動強度格(strength lattice)解析	[1]
浮接 tie-break(最大電容者保值)	電荷分享(charge sharing)解析、電容強度排序	[1]
動態單例(R-1)、成對路徑(B1)	動態 active-subnetwork 尺寸偵測上的基數特化早退	[3]
同態入列剪枝(P-1)、關斷隔離剪枝(P-2)	事件壓制(event suppression)的載入期結構證明形式	[3, 4]
電容支配解除汙染(P-3/P-4)	最近鄰為 IRSIM 的 maxnode 尺寸粗化實務(差異見 3.2 節)	[3]
類別為主鍵重編號 + 區間比較(區間剪枝)	inspector–executor 式檢查/執行分離下的資料重排	[7–9]
自我捕捉初次觸碰鍵	追蹤驅動資料佈局(trace-driven layout)、首次觸碰重排	[8, 10–12]
安全回退(範圍驗證失敗 → 剪枝關閉)	去優化防護(deoptimization guard)	[13]

第 3 章 相關研究

本專著回報的每一項優化,在被允許稱為貢獻之前,都先經過一輪對抗式的先行研究對映:我們指示一個外部模型去*反駁*新穎性而非確認它,把每項技術對映到其最接近的具名先例,倖存下來的「疑似原創」項目再以標題層級搜尋查核——查詢的建構方式刻意讓碰撞(若存在)必然浮現。工作原則是:我們寧可發現自己重新發明了四十年前的東西,也不要宣稱虛假的新穎性。本章即是這項演練的文字紀錄。全章使用固定的三級判定詞彙:**已知(KNOWN)**(標準文獻;我們引用之,不作任何主張)、**變體(VARIANT)**(概念已知;我們的形式或工程作法有值得陳述的差異),以及**疑似原創(PLAUSIBLY ORIGINAL)**(對抗式對映加標題層級搜尋皆未找到先例——這是關於已執行搜尋的陳述,不是不存在的證明)。

> 讓以下每一項比較有意義的前提是:我們的解析模型**在功能上就是 Bryant 的 MOSSIM II** [1] 減去 X(未知)態。優先序解析與最大電容浮接 tie-break 即是他的離散強度/節點尺寸格(lattice)形式化,經由 Visual 6502 計畫的 `chipsim.js` [15] 繼承而來。引擎所稱的「群走訪」就是文獻所稱的**通道相連元件(CCC)求值**。我們先承認這一點;之後主張的差異,是*建立在*該模型之上的差異,而非模型本身的差異。

3.1 開關級模擬

兩條截然不同的譜系在本研究中匯流:1980 年代的學院派開關級學派,以及 2010 年代的晶粒照片(die-shot)逆向工程譜系——後者大體在文獻之外重新實作了前者的核心演算法。我們先概述兩者,再把我們的派發層級技術放到它們面前對照。

3.1.1 古典學派:MOSSIM II、COSMOS、IRSIM

Bryant 的開關級模型 [1] 是本專著一切內容的語意基礎。它把 MOS 網路抽象為攜帶離散狀態的節點與作為開關的電晶體,並且——對我們而言至關重要地——形式化了*電荷分享*(charge sharing):當一組節點經由 ON 電晶體相連而無任何驅動者在場時,解析後的狀態由離散節點尺寸的格決定,以最大電容成員為支配者。我們的 256 項優先序 LUT(接地勝過電源,電源勝過外部驅動,外部驅動勝過上拉,上拉勝過保持)與最大電容浮接 tie-break,都是這套形式化的直系後裔。唯一的語意減項是 X 態:晶粒照片譜系捨棄三值模擬,改採決定性的二態鬆弛(relaxation),這也是為什麼波內 Gauss-Seidel 順序在我們的引擎中成為可觀察語意(第 4 章)。

在 Bryant 之前,Ulrich [4] 闡明了所有事件驅動模擬的奠基洞見:只模擬活動。一個大部分狀態在大部分時間都靜止的網路,只應為會改變的部分付出代價。我們的事件佇列、我們的安定波(settle wave),以及最終整個剪枝家族(第 3.2 節)都源自這個原則;這些剪枝可以讀作在開關級語意下,對 Ulrich 的問題「什麼才算活動?」一次比一次更銳利的回答。

COSMOS [2] 走的是編譯路線:網路前處理器(Anamos)靜態地把電晶體網路切分成通道相連元件,推導每個元件的布林穩態行為,並把結果編譯成可執行碼,完全消去解譯式的走訪。COSMOS 對我們有兩重意義。第一,它的靜態 CCC 抽取是我們 cls1 派發類別的具名先例(第 3.1.3 節)。第二,它的靜態子圖樣板比對——辨識傳輸閘(transmission gate)與閂鎖(latch)並為其產生特化程式碼——

是我們的 B1 成對路徑必須對照衡量的先例。我們在此僅指出而不展開:我們自己在這份網表上對編譯式與 oblivious 求值的實測經驗一律是負面的(比事件驅動引擎慢 3–6 倍;第 9 章);COSMOS 的取捨是真實的,但在一份導通群平均僅 1.13–1.4 個節點的網表上並不划算。

IRSIM [3] 是精神上最接近的祖先:一個帶線性 RC 時序的增量式、事件驅動開關級模擬器,其求值作用在「目前 ON 電晶體」構成的*動態*子網路上,而非靜態切分。有三個 IRSIM 的想法在第 3.2 節反覆出現,作為我們各項技術的最近鄰:它的動態 active subnetwork、它對被隔離節點的電荷保持行為,以及它用於電荷分享解析的節點尺寸粗化實務。

3.1.2 晶粒照片譜系

第二條譜系始於 Visual 6502 計畫 [15]:它從晶粒照片導出 MOS 6502 的電晶體級網表,並以 JavaScript 實作了二態的 Bryant 式鬆弛(`chipsim.js`)。Quietust 把這套方法論延伸到 NES 晶片組,產出本計畫所使用的 Visual 2A03 與 Visual 2C02 網表 [16]。該演算法其後被多次重新實作:perfect6502 [18],6502 模擬的 C 移植;VisualNes [19],2A03/2C02 模擬器的 C++ 移植;以及 MetalNES [17],以 C++ 把兩顆晶片與 NES-001 主機板的 TTL 邏輯組合成整機模擬。MetalNES 是我們的直系祖先:我們的引擎始於對其 wire 層核心的 C# 移植。

這條譜系由工程驅動,與其所承載演算法的學院派學派大體上互不相聞;本專著的目的之一就是把兩者重新接上。為了校準,我們在同一台機器、同一個 ROM、同一個單位(每秒主時脈半週期數,可適用處皆為整機)上量測了這個家族——這些原始實作皆無內建的 hc/s 量測,我們對其原始碼做了實驗性儀器化以取得同單位數字(詳見 2.4 節):VisualNes 持續約 24K hc/s,MetalNES 約 54K,本文所述引擎約 136K;perfect6502 量得約 29K,但其單位為僅 6502,無法與整機數字直接排名 [21]。相對直系祖先的約 2.5 倍,即是第 5–8 章累積的主題。

3.1.3 我們的派發技術位居何處

引擎透過數個派發類別之一解析每個事件(第 4 章),而每個類別在文獻中都有其位址。

cls1,靜態單例,屬於已知(KNOWN)。一個沒有任何 pass-transistor 通道的節點,永遠不可能加入多節點導通群;僅憑靜態結構,其解析即為 O(1)。這是 MOSSIM II 與 COSMOS [1, 2] 所實踐之靜態 CCC 抽取的退化情形——COSMOS 更進一步,把這類單向邏輯直接編譯成布林閘。在我們的網表中,26.7% 的節點(3,929 個)落入此類。我們引用先例,不作任何主張。

R-1,動態單例,屬於變體(VARIANT)。在事件被處理的當下,若入射到該節點的每個 pass 閘*此刻*都是 OFF,則導通群恰為單例 $\{nn\}$,可以 O(1) 解析,且與完整走訪逐位元相同。其底層概念——「相關的網路是建立在目前 ON 開關上的動態網路」——正是 IRSIM 的動態 active subnetwork [3]。我們的新穎性在實作層級而非理論層級:把執行期的*基數證明*用作*早退派發*,一條完全繞過泛用鬆弛機制的 O(1) 行內路徑。我們採用的誠實定位是「把動態尺寸判定用作早退執行過濾器」。在第 8 章的消融中量得,這一步本身值 +18.7%——本計畫最大的單一步——我們把它讀作關於現代記憶體階層的證據,而非新模擬理論的證據。

B1,成對路徑,屬於帶動態變化的變體(VARIANT)。當恰有一個入射閘為 ON,且鄰居的 ON 通道全部繞回種子時,該群可證明為 {種子, 鄰居};接著就地解析這一對,逐位元組複製泛用走訪的順序敏感語意(成員 hash 清除、兩次供電掃描、tie-break 形式、寫入順序),且每個回退點都在任何修改之前取用。具名先例是 COSMOS/Anamos 的靜態樣板比對 [2],它在編譯期辨識傳輸閘與閂鎖子圖。

我們的差異在綁定時機:這是*在 active 通道圖上的執行期拓撲樣式比對*。理論上這不過是把群大小為二的 BFS 做迴圈展開;我們把它呈現為有效的工程,而非理論。在第 8 章的消融階梯中,這一步量得 +7.0%(S6→S7,同日交錯);與緊鄰的整理性 commit 分離歸因後約 +8.9%。

3.2 跨抽象層級的事件壓制

剪枝家族(P-1 到 P-4;第 6 章)壓制處理過程可被證明為 no-op 的事件。壓制可證明無效的工作是模擬領域最古老的想法之一;讓我們家族成員與眾不同的是*證明了什麼*、*何時證明*、*以及證明被花在哪裡*。本節從 IRSIM 開始沿抽象堆疊向上走,把每個剪枝精確安放。

3.2.1 選擇性追蹤、等電位旁路與 P-1

Ulrich 的選擇性追蹤(selective trace)[4] 確立了「無變化不必傳播」。IRSIM 應用了同一直覺的開關級實例:當一顆電晶體在等電位的節點之間導通時,這個連接什麼也不改變,工作可以旁路 [3]。我們的 P-1——當兩個通道端點持有相等狀態時壓制導通入列——是這個標準想法的變體(VARIANT),我們也如此標示它。

我們認為值得陳述的,是*在離散電荷分享下,同態(same-state)合併何時仍不安全的分類學*。在 Bryant 式二態模型裡,浮接 tie-break 不是邊角案例:它*就是*晶片的儲存機制,而一次同態合併仍可能改變後續浮接解析中由哪個成員的電容支配。我們第一版天真實作的 P-1 交付出來的是黑畫面——發散的節點恰恰是活在浮接 tie-break 上的無上拉動態儲存節點(OAM 與調色盤 RAM 細胞)。生產版把這種不安全編碼為載入期的結構性汙染(taint):無上拉節點與 ForceCompute 通道元件被排除在剪枝之外。貢獻是這套分類學,而非旁路本身;促成它的那個 bug,則是文獻自身的怪癖被實證化。

3.2.2 最近鄰:IRSIM 的節點尺寸實務 vs 電容支配(P-3/P-4)

P-3 與 P-4 承載本專著最強的主張,因此其最近鄰值得最謹慎的處理。語意基底仍是 Bryant 的節點尺寸格 [1]:在純浮接群中,最大電容成員的先前狀態獲勝。P-3/P-4 在載入期、全自動地推導出一條嚴格支配不等式:一個無驅動節點,若其電容嚴格低於其*所有*通道鄰居,則它永遠不可能成為任何它所加入之群的最大電容成員,因此永遠不可能影響浮接 tie-break,因此經過它的同態合併可證明為 no-op——即使該節點就坐落在儲存機制之上。這個證明把 P-1 安全汙染所排除節點中的一個子集解除汙染(un-taint),而解除汙染被花在事件壓制上——量得 +5.96%(P-3)與 +1.71%(P-4),位元精確。

標題層級的搶先掃描沒有發現直接碰撞,但確實浮現一個必須引用的最近鄰:IRSIM 的節點尺寸(`maxnode`)實務 [3]——使用者手動指定粗略的離散尺寸(高電容節點如預充匯流排給較大尺寸、一般節點給預設值、儲存節點可選最小尺寸),以簡化並加速電荷分享解析。相似性是真實的(兩者都利用節點間的電容排序),因此差異必須被明確劃出。兩者在四個軸上不同。*作者身分:*IRSIM 的尺寸由使用者手工指定;我們的不等式由抽取出的電容自動推導。*知識論地位:*尺寸指定是一種粗化——由使用者背書的近似;我們的是一個證明,且結果相對未剪枝引擎位元精確。*機制:*IRSIM 的尺寸餵進解析函式本身;我們的不等式從不觸碰解析——它把關入列。*目的:*解析簡化 vs 事件壓制。關係形式(對*所有*通道鄰居的支配)有一個令人欣慰的推論:重量級儲存細胞會自己排除自己——擁有局部支配的電容,正是讓一個節點成為儲存的原因,所以這個證明永遠不會在危險之處觸發。

3.2.3 P-2:從執行期結果到載入期證明

IRSIM 的電荷保持規則確保被關斷隔離的節點保有其電荷——但在 IRSIM 中這是一個*執行期求值結果*:事件被處理、隔離被發現、狀態被保持 [3]。P-2 做了反向操作:對一個靜態上度為 1 且無驅動的節點,其唯一通道關斷的結果在載入期即可判定——該節點必然隔離並保值,永遠如此。因此事件在*進入佇列之前*就被壓制,省下整條入列→彈出→解析鏈,而不只是下游的傳播。標題層級掃描沒有發現碰撞(只有主題相鄰的標題),我們把 P-2 作為疑似原創(PLAUSIBLY ORIGINAL)的次要主張提出:機制上謙遜,但代表了這個家族的招牌動作——把執行期的發現提升為載入期的結構證明,並比文獻更早地在管線中把它花掉。

3.2.4 其他抽象層級的壓制

開關級之上與之下都存在類比的壓制機制,而這些區別值得劃清,因為它們界定了我們的方法論能主張什麼。Fast-SPICE 模擬器對潛伏(latent)子電路旁路數值積分——那是在連續模型上的動態、執行期偵測,並明確以精確度作為交換;我們的遮罩則是靜態、載入期且位元精確的。RTL 合成與模擬利用可觀察性 don't-care 與時脈閘控(clock gating)——但 RTL 有顯式的暫存器基元,所以「這個儲存元件不可能改變」在語法上唾手可得;我們則必須從一片平坦的電晶體之海中推斷隱式狀態。技術上最接近的是 EDA 的自動抽取/圖樣識別傳統:LOGEX [5] 從電晶體層網表自動抽取閘級邏輯,SubGemini [6] 以子圖同構在網表中識別子電路——兩者皆屬程式庫/結構圖樣比對導向的自動抽取家族,是有近四十年歷史的既定實務。我們的「以其物理而非其名字辨識記憶體」遍(上拉旗標、通道元件 union-find、電容比較、驅動腳排除;零手列節點)改以物理性質而非結構圖樣比對來辨識,屬於同一傳統的另一個取徑,不主張任何演算法新穎性。差異在應用:抽取在歷史上回答的是*求值或抽象什麼*;我們回答的是*丟掉什麼*——它為一個事件淘汰(event-culling)系統的安全類別提供參數。依對抗式諮詢的建議,我們把它呈現為支撐性方法論(第 6 章),而非貢獻。

3.3 資料佈局與局部性

第三個家族——類別為主鍵重編號、區間編碼的剪枝謂詞,以及自我捕捉初次觸碰鍵(第 7 章)——坐落在一個完全不同的文獻裡:執行期資料重組與快取感知(cache-conscious)佈局。每個元件都有清楚的譜系;我們將論證,這個綜合(synthesis)沒有。

3.3.1 Inspector-Executor 譜系

在執行期檢視一個不規則計算的存取模式、並在執行優化排程之前重排資料(及/或疊代)的範式,就是 inspector-executor 模型:由 CHAOS 執行期程式庫 [7] 確立,由 Ding 與 Kennedy [8] 延伸到面向快取的資料與計算重組,並由 Strout、Carter 與 Ferrante [9] 給出編譯期的組合框架。我們的三遍載入可一眼認出是 inspector-executor:分類遍與儀器化捕捉遍構成 inspector;最終重建是資料重排;生產運行是 executor。Ding 與 Kennedy 的初次觸碰打包(first-touch packing)[8]——依 inspector 觀察到的首次存取順序重排陣列元素——是我們初次觸碰鍵在演算法上最接近的祖先,我們也如此具名。

有三個差異把我們的實例與這條譜系分開。第一,*檢視的對象*:不是不規則平行迴圈巢的索引陣列,而是離散事件模擬器的事件級聯——一個只在動態中存在、只有靠運行引擎本身才能觀察到的物件,這也是為什麼 inspector 是 settle 迴圈的一個冷的儀器化副本,而不是對輸入的分析。第二,*行程拓撲*:整個循環在進程內、自我封閉,每次載入都從網表重新導出、約 1.3 秒完成,沒有任何 profile 檔;

過時(staleness)——離線 profile 導引方法的經典失效模式——在構造上即被排除,而線上鍵在同一可執行檔的對決中,可量測地以 +4.94% 勝過同形式的離線載入 profile(16/16 配對勝)。第三,*重排的對象*:模擬器的整個節點 ID 空間,其在置換下的語意不變性先以構造方式確立、再加以驗證(第 3.3.3 節)。

3.3.2 軌跡導向與快取感知佈局

一條平行的譜系從配置器、垃圾回收器與編譯器的角度處理佈局:快取感知的結構佈局與切割 [10]、用於參照局部性的熱資料流(hot data stream)抽象 [11]、由 profile 驅動的編譯器導向資料放置 [12],以及 profile 導引資料佈局的編譯器層級專利形式 [20]。這些工作確立了:應由時間性的存取行為、而非靜態結構,來支配放置——這也是我們的前提。差異同樣在層級與機制:那些系統使用由配置器或編譯器消費的離線或取樣 profile 來放置堆積物件或結構欄位;我們則在載入期從模擬器自身的生產級聯重建其 ID 空間。

我們的一項實證發現使這個文獻變得更銳利,而不只是把它實例化:捕捉到的鍵之價值在於已剪枝生產級聯的*順序*,不在快取行密度。以引擎內建儀器計數每半週期觸碰的熱結構快取行:盲目結構鍵觸碰 118.4 行/hc;離線 profile 鍵 109.5(牆鐘 +1.55%);在剪枝關閉下取得的真捕捉達到相等密度(110)卻只得 ±0.0%;同一捕捉在剪枝開啟下——即生產引擎實際將執行的順序——達到 105.9 與 +6.17%(20/20)。等密度、不同順序、相反結果。快取感知佈局文獻優化的是密度代理指標;我們的數據說,在這裡密度是必要而不充分。

3.3.3 去優化防護模式

圍繞重排的正確性體制是一個已知模式的直接應用,我們引用它而不主張它。Hölzle、Chambers 與 Ungar 的動態去優化 [13] 確立了這樣的紀律:在防護(guard)之下執行激進優化的程式碼,當假設被違反時,語意透明地回退到基準線。我們的剪枝遮罩在每次 reset 時由基準真相重新計算;由區間導出的位元會與之核驗;任何不符時,引擎退化到一個安全邊界——剪枝關閉、供電不變式保留、正確性維持。這是把去優化防護(deoptimization guard)模式從 JIT 編譯器移植到模擬器的載入路徑。

3.3.4 預取器與記憶體層級平行性

等密度之下,順序為什麼會重要?硬體層面的解釋來自預取(prefetching)與記憶體層級平行性(memory-level parallelism)文獻:空間記憶體串流(spatial memory streaming)[14] 記載預取的有效性取決於與程式碼行為相關聯、反覆出現的存取*序列*,而不僅僅是哪些快取行駐留。與生產排程相符的佈局,讓 settle 迴圈面對近乎單調的存取序列,得以訓練面向串流的預取器並讓載入佇列保持滿載;一個等密度但非生產順序的佈局,則在每個半週期內以打亂的序列觸碰同樣的熱行。我們把這提出為「順序 vs 密度」結果的可信機制,並附帶警語:我們並未直接隔離預取器計數器。

3.3.5 學術文獻之外的實務類比

有三種工程實務是本研究中區間編碼這一半誠實的精神孿生。現代遊戲引擎的實體-元件-系統(ECS)archetype 儲存,依元件擁有權把實體排進連續記憶體區塊,使熱迴圈得以省去逐實體的成員檢查——與我們的類別為主鍵重編號一一對應:靜態謂詞變成連續 ID 區間,熱迴圈查表變成暫存器比較。資料庫的叢集索引(clustered index)與範圍分割(range partitioning)以宣告式實現同一招:

以 (class, captured-order) 叢集,類別謂詞就變成連續掃描。複製式垃圾回收器則是重建本身的類比:複製順序(回收器的走訪)決定了共同存取物件的空間局部性,而我們的重建遍相當於一次重定位走訪,其順序是量測得來而非由可達性導出。

以上這些——連同前述的學術譜系——沒有任何一個是*在進程內捕捉事件驅動模擬器自身的級聯作為佈局鍵*。遊戲引擎依靜態元件集合排序;資料庫依宣告的鍵;回收器依可達性;inspector-executor 系統依迴圈索引流;profile 導引編譯器依其他運行的離線軌跡。我們的鍵是生產級已剪枝級聯實測的首次彈出順序——是系統動態的產物,不是其結構的產物。基於結構的順序在這個工作負載上可證地失敗:從穩定快照出發的規則式 BFS 量得*劣於*盲目結構鍵(174 vs 118.4 行/hc),因為雙向 pass-transistor 網路的拓撲並不對映到它的執行流。那道鴻溝——由動態導出 vs 由結構導出的順序、且完全在被優化的進程內部執行——正是我們在第 3.5 節主張的那個特定綜合。

3.4 判定矩陣

表 3.1 濃縮了這份對映。方法:每一列的判定先由對抗式諮詢產出(指示一個外部模型尋找最強的反駁先例),再以標題層級的搶先掃描在 IEEE Xplore、ACM Digital Library 與 Google Scholar 上查核,所用查詢集合的設計目的是讓碰撞(若存在)必然浮現(例如「switch-level simulation」與「charge sharing」「event suppression」的組合,或「inspector-executor」與「event-driven simulation」的組合)。判定受限於那些搜尋的投入與召回;它們是盡職調查的陳述,不是不存在的證明。完整紀錄——主張措辭、搜尋鍵、逐項引文——保存在計畫儲存庫中 [21]。

表 3.1. 先行研究判定矩陣。KNOWN(已知)= 標準文獻,引用而不主張;VARIANT(變體)= 概念已知,我們的形式如文中所述有別;PLAUSIBLY ORIGINAL(疑似原創)= 對抗式對映加標題層級搜尋皆未找到先例。

我們的技術	最接近的具名先行研究	判定	一行差異
cls1 靜態單例派發	靜態 CCC 抽取;編譯式布林元件 [1, 2]	已知(KNOWN)	不作主張;標準實務,已引用。
R-1 動態單例	IRSIM 動態 active subnetwork [3]	變體(VARIANT)	把執行期基數證明用作 O(1) 早退派發,而非新的尺寸理論。
B1 成對路徑	COSMOS/Anamos 靜態樣板比對 [2]	變體(VARIANT,帶動態變化)	在 active 通道圖上的執行期拓撲比對;語意上是展開的大小 2 走訪,逐位元組複製。
P-1 同態 (same-state) 入列剪枝 + 安全汙染	選擇性追蹤 [4];IRSIM 等電位旁路 [3]	變體(VARIANT)	新增「離散電荷分享下同態合併何時仍不安全」的分類學,編碼為載入期結構汙染。
P-2 關斷隔離剪枝	IRSIM 電荷保持(一種執行期求值結果)[3]	疑似原創 (PLAUSIBLY ORIGINAL,次要)	把保持結果提升為載入期靜態證明;在佇列之前壓制,省下整條入列→彈出→解析鏈。
P-3/P-4 電容支配解除汙染 (un-taint)	Bryant 節點尺寸格 [1];IRSIM 節點尺寸(`maxnode`)實務 [3];儲存抽取 [5]	疑似原創 (PLAUSIBLY ORIGINAL,最強)	自動推導的支配不等式證明 tie-break 免疫,花在位元精確的事件壓制上——相對於為簡化解析而手動做尺寸粗化。
基於物理的記憶體辨識	狀態元件抽取 [5, 6];Fast-SPICE latency bypass;RTL ODC/時脈閘控	已知(KNOWN)技術,新應用	把抽取花在事件淘汰的安全類別上(「丟掉什麼」),作用於平坦網表中的隱式狀態;以方法論呈現,不作主張。
區間編碼剪枝謂詞(ID 區間比較)	索引空間分割實務;ECS archetype;快取感知佈局 [10, 12]	變體(VARIANT,實務)	文獻中無單一具名來源;為重排系統的元件,不獨立主張。
自我捕捉初次觸碰重排	Inspector-executor [7, 8, 9];軌跡導向佈局 [10, 11, 12];profile 導引佈局專利 [20]	疑似原創 (PLAUSIBLY ORIGINAL,綜合)	在進程內、自我封閉地捕捉事件驅動模擬器自身的已剪枝級聯作為佈局鍵;其價值已證明在順序、不在密度。
驗證 + 安全退化回退	動態去優化防護 (deoptimization guard)[13]	已知(KNOWN)	把 JIT 防護模式直接應用於模擬器的載入路徑;已引用。

3.5 新穎性主張的範圍

上述對映留下恰好兩個頂層主張與一個次要主張仍然成立,我們在此以我們準備好捍衛的形式陳述之。

主張 3.1(電容支配 tie-break 免疫;P-3/P-4)。在 Bryant 式離散電荷分享模型中,靜態推導的嚴格不等式「無驅動節點的電容小於其每一個通道鄰居的電容」是 tie-break 免疫的充分條件:該節點永遠不可能決定一個純浮接群的解析。這個證明安全地讓同態事件壓制得以在動態儲存節點的一個子集上位元精確地啟用,而最重的儲存細胞會憑著讓它們成為儲存的同一套物理把自己排除在外。

主張 3.2(自我捕捉軌跡重排)。一個在進程內、自我封閉的 inspector-executor 遍,在每次載入時由生產級已剪枝級聯的初次觸碰順序重新導出開關級模擬器的節點 ID 空間——把靜態安全謂詞區間編碼為連續 ID 區塊,並以基準真相驗證加安全退化回退來防護——就我們搜尋所及,作為一個綜合並無先例;且其實測價值在於級聯的順序而非快取行密度。

主張 3.3(次要;P-2)。靜態辨識度 1 的無驅動葉節點並在進入佇列之前壓制其斷接事件,把 IRSIM 式的執行期電荷保持提升為載入期證明,刪除的是完整的事件處理鏈,而不只是傳播本身。

同等重要的是我們明確*不*主張什麼。我們不主張解析模型:它是 Bryant 的 [1],經 Visual 6502 計畫 [15] 繼承而來,而我們在每一處定位中都說「功能上即 MOSSIM II」。我們不主張 CCC 求值、靜態 CCC 抽取或事件驅動的選擇性追蹤 [1, 2, 4]。我們不為 R-1 或 B1 主張理論新穎性:它們分別是 IRSIM 動態子網路與 COSMOS 樣板比對的工程變體,而諮詢方建議的第三個主張——「基數派發架構」——被刻意降格為變體定位,未作為貢獻提出。我們不把基於物理的記憶體辨識當作演算法貢獻來主張;它是四十年前的抽取技術換個地方部署,在本專著中以方法論的身分出現。我們不主張去優化防護模式 [13]、inspector-executor 範式 [7] 或初次觸碰打包 [8];我們的重排主張嚴格限定在那個綜合與「順序而非密度」的發現。我們不主張表 3.1 中任何判定能證明先行研究不存在:標題層級掃描界定了搜尋投入的上限,而它們確實浮現的那一次近似碰撞(IRSIM 的 `maxnode` 實務)在第 3.2.2 節中是被劃清差異、而非被打發掉。最後,我們不主張超出量測範圍的一般性:所有主要數字都來自單一網表家族與單一主機微架構(Zen 2),輔以社群排行榜數據點 [21]。

第 4 章 引擎架構與正確性基礎設施

本章描述引擎實際執行時的樣貌:熱狀態的記憶體佈局、排空它的事件迴圈、由原始網表產生它的建構管線,以及——刻意給予同等份量的——讓第 5–7 章的優化攻勢得以成立的正確性基礎設施。演算法內容在設計上是保守的:解析語意沿用 Visual 6502 家族 [15][16][17],在功能上是 Bryant 開關級(switch-level)模型 [1] 的一個限制版本(第 3 章鋪陳這個對應)。屬於本引擎自身的,是資料的組織方式、要求每一個轉換都必須對該語意可證明或可驗證地位元精確(bit-exact)的紀律,以及把「位元精確」從一句願景變成機械化閘門的整套裝置——golden checksum、逐節點 dump-diff,以及對排列(permutation)有所自覺的上電程序。全章引用的數字皆量測於第 8 章描述的參考機器與工作負載;在此引用僅為說明結構選擇的動機。

4.1 資料佈局

4.1.1 非託管堆積上的 structure-of-arrays

引擎是單一個靜態類別,其熱狀態完全存放於非託管記憶體,於每次 `Reset()` 透過 對齊配置包裝函式一次配置,並於單次掃除中釋放。選擇 structure-of-arrays(SoA)分解而非 物件圖,是最根本的佈局決策。自然的物件導向編碼——一個持有自身狀態、旗標與鄰接清單的 `Node` 物件——在引擎中僅作為*建構期*表示法存活(4.3 節);模擬開始之前,內層迴圈會碰到的一切都被攤平成以節點 id 索引的平行陣列。理由就是快取感知佈局文獻 [10][12] 的標準論證:引擎的行為由數百 KB 工作集上的相依載入主導,因此欄位按 *存取模式*而非按*實體*共置,決定了每個事件要碰幾條 cache line。

這個分解細到個別欄位都由存取模式驅動。每節點的邏輯值是 `NodeStates` 中的 一個 byte——它是引擎中被讀取最頻繁的陣列,因為群走訪期間的每一次閘極測試、入列期間的 每一次同態(same-state)比較都會讀它。每節點描述子 `NodeInfo`(4.1.2 節)持有 求值期間讀取的旗標與通道鄰接。有兩個欄位,天真的設計會放進 `NodeInfo`,卻正因存取模式不同而被拆成獨立的「冷」陣列:電容代理值 `NodeConnections` 只有純浮接 tie-break 會讀(不到 1% 的群走訪會走到),而節點 閘極扇出清單的索引 `NodeTlistGates` 只有 `SetNodeState` 在寫回時 會讀——每次狀態改變、每個群組成員一次,而非每次 BFS 造訪一次。把它們排除在 16-byte 熱紀錄之外,使每條 cache line 可容納的熱紀錄數量加倍。這就是 Chilimbi 等人 [10] 意義下的結構拆分(structure splitting),只是拆分邊界由實測存取頻率而非欄位大小決定。

4.1.2 16-byte 的 NodeInfo 與行內/溢位聯集

每個節點的求值期描述子是一個顯式佈局(explicit-layout)的 16-byte 結構——每條 64-byte cache line 裝四個。前四個 byte 依序是:旗標 byte(上拉、外部驅動、供電、 ForceCompute、callback——即參與群組解析的位元)、`Inline` 判別子、行內通道對 數量,以及一對打包的半位元組(nibble)計數,分別記錄接地/接電源通道數。其餘十二個 byte 是一個聯集(union),由判別子選擇兩種解讀之一:

- **行內形式**(約 96% 的節點):六個 16-bit 欄位直接存放節點完整的通道鄰接——先是 通往一般節點的 pass 通道之(閘極, 另一端)對,再來是通往接地之通道的閘極 id,最後是 通往電源之通道的閘極 id。凡 $2 \cdot \text{pairs} + \text{gnd} + \text{pwr} \le 6$ 的節點皆符合資格。

- **溢位形式**(約 4% 的節點——匯流排、時脈扇出樹):三個 32-bit 索引,指向共享的 攤平鄰接陣列(4.1.3 節),每個通道桶各一。

兩種形式互斥,且每一次熱存取都以判別子把關,因此這個聯集是安全的。行內形式的重點在於 消除一次相依的指標追逐:對絕大多數節點而言,派發測試「這個節點的所有 pass 閘極都是 OFF 嗎?」(R-1 動態單例檢查,4.2.2 節)、O(1) 的單例解析,以及 BFS 擴張步驟,全都能在 描述子載入已經帶進來的*那一條* cache line 內完成。在一條受記憶體延遲限制、平均導通群 只有 1.13–1.4 個節點的走訪上,一次相依載入與兩次之間的差距是一階效應;行內 payload 這項 改動(工程日誌中名為 S2-A)是單項佈局收益中較大的幾個之一。附帶的好處是,行內節點完全 不再把通道子清單放進共享鄰接陣列,縮小了其餘溢位走訪需要遍歷的陣列。

把通道分桶為*一般*/*接地*/*接電源*三類,本身就具語意,不只是整理上的 方便。遠端是供電節點的通道永遠不會讓群組長大——它唯一的作用是貢獻 Gnd 或 Pwr 旗標—— 因此 BFS 擴張迴圈只疊代(閘極, 另一端)對,而供電接觸由一次帶提早跳出的平坦掃描偵測 (快速路徑上則是無分支 OR)。這是「供電節點絕不可進入群組或佇列」這個觀察在佈局層級的 編碼。

4.1.3 攤平鄰接與閘極扇出清單

所有放不進行內的鄰接都存放在 `TransistorList`:單一個由 16-bit 節點 id 組成、 以零終止子清單組織的陣列;索引 0 是哨兵零,使得值為 0 的「空子清單」參照一讀就讀到 終止符。有兩個表示法決策至關重要。其一,id 採 16-bit:lowering 後的系統約 14.7K 個節點, 遠低於 65,536,而元素寬度減半把當時引擎最熱的陣列從 697 KB 砍半到 350 KB—— 一次純粹的工作集縮減、毫無演算法內容,在這個工作負載上卻比多數指令層級的技巧更值錢。 其二,陣列尾端至少帶四個零填充,使得每次疊代以單一 64-bit 載入讀取*兩個*(c1, c2)對 的迴圈(4.2.4 節)永遠不會因為越過最後一個終止符的過量讀取而出錯;填充讀起來就是終止符, 無害。

每個節點——不論是否行內——都另外參照一條*閘極扇出*子清單:所有以此節點為 *閘極*的電晶體之(c1, c2)端點對。這份清單是引擎的事件傳播邊集——恰好在節點狀態 真的改變時被讀取,用來決定哪些通道端點必須重新求值——且對所有節點都保留在共享陣列中, 因為它的存取模式(寫回時的循序突發)與描述子的(隨機單行探測)不同。

4.1.4 波佇列、去重 hash,與邊界檢查的缺席

待決事件結構是一對雙緩衝佇列,每個緩衝各配一個成員資格 byte 陣列:`RecalcList/RecalcListNext`(32-bit id,波內只追加)與 `RecalcHash/RecalcHashNext`(每節點一個 byte,0/1)。「Hash」是沿襲自祖先實作 [15][17] 的歷史命名;在本引擎中它是直接索引的旗標陣列,而它從 `int` 窄化為 `byte`(每個緩衝 58 KB → 14 KB)同樣是工作集考量。這個旗標一次承擔三項職責: 入列時的 O(1) 去重、彈出時的「本波仍待處理」測試,以及——對語意至關重要的——群走訪 *吸收*它所造訪的每個成員之待決求值的機制(4.2.3 節)。群走訪的暫存狀態遵循相同模式: `_groupBuf`(16-bit id,29 KB)同時兼任結果清單與 BFS 工作佇列,而 `_inGroup`(每節點一個 byte,14 KB)是走訪的去重旗標,在兩次走訪之間做稀疏 清除——只清前一個群組的條目。

熱迴圈中沒有邊界檢查,不是因為它們被壓制,而是因為這個表示法讓它們根本無從表達: 每個熱陣列都是指向非託管記憶體的裸指標,執行期沒有任何陣列物件可供檢查。安全論證是 結構性的,而非逐次存取的。索引或來自建構管線(在組成期對節點數做一次性驗證),或來自 清單本身(零終止,節點 0 保留);id 依構造即為 16-bit,陣列依節點數定大小;唯一刻意的 子清單外讀取就是有填充保護的

64-bit 雙對載入。我們直白地指出:這是在一個封閉、建構期已 驗證的索引空間內,以記憶體安全換取速度——對模擬器核心而言是站得住腳的交換,而其殘餘 風險正是 4.4 節的正確性基礎設施所要捕捉的對象:讓它以發散的形式現形,而不是讓毀損悄悄 溜過。

表 4.1. lowering 後系統規模(約 14.7K 個節點 id,含處理常式 假節點)下的引擎熱陣列。大小依工程筆記所載;每半週期的*活躍*集遠小於此 (約 15 KB,L1 常駐——第 8 章)。

陣列	元素	約略大小	角色
`NodeStates`	byte	15 KB	邏輯值 0/1;被讀取最多的陣列
`NodeInfos`	16-byte 結構	235 KB	旗標 + 行內通道 payload 或溢位索引
`TransistorList`	ushort	≤350 KB	攤平的零終止子清單(溢位通道;所有節點的閘極扇出)
`NodeConnections`	int	58 KB	冷:浮接 tie-break 用的電容代理值
`NodeTlistGates`	int	58 KB	冷:各節點閘極扇出子清單的索引
`RecalcList` (×2)	int	各 58 KB	雙緩衝波佇列(目前/下一)
`RecalcHash` (×2)	byte	各 14 KB	各緩衝的佇列成員資格/待決旗標
`_groupBuf`	ushort	29 KB	群組成員清單 ≡ BFS 工作佇列
`_inGroup`	byte	14 KB	群走訪去重旗標(稀疏清除)
`IsPureLogic`	byte	15 KB	每節點派發類別(cls0/cls1/cls2)
`FlagsToState`	byte	256 B	旗標 OR → 解析值優先序 LUT

4.2 熱迴圈

4.2.1 StepCycle 與 settle 迴圈

模擬時間的單位是主時脈半週期(hc):主機板 `clk` 節點的一次翻轉。 `StepCycle` 以無分支方式翻轉時脈節點的外部驅動旗標(時脈永遠在翻轉,所以 不需要方向分支)、將其入列,然後呼叫 `ProcessQueue` 傳播至靜止:

```
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
```

雙緩衝定義了*安定波(settle wave)*:由波 k 中的狀態變化觸發的求值會被追加 到下一個緩衝,在波 k+1 執行。然而在波內,語意是 Gauss–Seidel 而非 Jacobi: `RecalcNode` 立即寫入解析後的狀態,因此同一波中較晚求值的節點會看到較早求值 節點的更新值,而吸收了某成員的群走訪會清掉該成員的旗標,取消它在*目前*這一波中仍 待決的求值。我們要強調——因為它約束了本書中的每一個優化——波內順序因此是**可觀察 語意**,不是實作細節:浮接群組的 tie-break(4.2.3 節)讀取先前狀態,所以兩種排空順序 可以正當地產生不同的(且都在物理上站得住腳的)結果。位元精確正是對這個順序定義的。 任何重排彈出順序、拆分波、或延遲寫回的轉換,改變的是模型本身,而不只是排程;波重構的實測失敗在第 9 章與其他負結果一併討論。

在參考工作負載上,一個半週期平均以 12.06 個波安定(觀測到的最大值為 45),每半週期 約 418 次求值(實測)。Release 組建執行 settle 迴圈時*沒有*任何疊代上限:此網表在 兩個真實工作負載中觀測到的最深安定約 45 波;迴圈內防護的實測代價為 +2.77%(冷的診斷 區塊抑制了完全內聯迴圈的程式碼生成);另一項獨立研究(第 9 章)則確立了安定不足是 災難性的而非優雅的——即使只截斷最深的 0.58% 安定,也會在一千個半週期內發散。Debug 組建保留一道 128 遍的絆線(tripwire),純粹作為開發輔助。

4.2.2 RecalcNode 中的派發

每次求值都從 `IsPureLogic` 載入一個 byte 的類別開始,該陣列於 Reset 時填妥 (4.3.3 節):

```
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
```

類別 1(*靜態單例*,3,929 個節點,26.7%)涵蓋古典的 NMOS 閘輸出:只有通往供電的 通道、沒有任何通往一般節點的 pass 通道的節點。其導通群可證明永遠是 $\{nn\}$,因此求值 化約為把其接地通道與接電源通道閘極的狀態 OR 進旗標字,再查一次 LUT——即 `RecalcNodeFast`,一個無迴圈、分支極少的解析,逐 byte 重現一般路徑的結果, 包括浮接情形(空旗標 ⇒ 保持前值 ⇒ 完全不寫)。類別 2(*動態單例候選*,10,784 個 節點,73.2%)涵蓋*確實有* pass 通道、但不帶任何排除旗標的節點;對這些節點,單例 性質在執行期透過掃描(通常是行內的)閘極清單來測試,而這個掃描同時兼任 B1 雙節點行內 解析的進入點。類別 0(約 16 個節點,佔求值的 0.4%)是一切必須無條件走一般走訪的節點 —— callback 目標與帶 ForceCompute 旗標的節點。這套派發及其實測經濟學是第 6 章的主題;此處 只記錄它的架構形狀:一個由 byte 陣列驅動的三向分支,最稀有的類別放在最後解析。

4.2.3 群走訪與解析

一般路徑就是文獻中的通道相連元件(CCC)求值 [1][2][3],限制在*目前導通*的子圖上: 從種子出發、跨越 ON 的 pass 電晶體做廣度優先走訪,沿途對成員累積旗標 OR,最後以 256 項優先序表解析。

```
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)
```

有三個機制值得評述。其一, `_groupBuf` 同時充當結果清單與 BFS 佇列,使佇列 形同免費:沒有獨立的工作清單配置、push 或 pop,只有一個讀索引追著寫索引跑。這正是把 BFS 換成 DFS 的實驗量得 −2.28% 的原因——在此兩種順序語意上可互換(群組是集合,解析對 OR 而言與順序無關),但 DFS 需要一個堆疊,而 BFS 白白拿到。其二,成員旗標清除 (`curHash[other] := 0`)是承重的語意:被吸收進群組的節點已經*作為該群組的 一部分*被求值,若放任它過期的待決條目在同一波稍後彈出,就會讓它對著寫入後的狀態被 重新求值。其三,走訪本體被編譯為一個單元:引擎在走訪期間把靜態陣列指標與群組計數/旗標 提升為區域變數(實測 +3.2%),而整條鏈——派發、走訪、解析、寫回——內聯進 `ProcessQueue` 的波排空迴圈;該迴圈則刻意*不*強制內聯進它眾多的呼叫端 (強制內聯時實測 −1.4%)。

解析本身就是承襲自家族 [15][17] 的旗標 OR → LUT 方案:256 項的表由一個純優先序函數 一次建好——同時含 Gnd 與 Pwr 的 ForceCompute 群組兩者互消;其後 Gnd > Pwr > SetHigh > SetLow > PullUp > 保持——因此每群組的成本是一次 byte 載入。純浮接分支 (OR 為空)是晶片的儲存機制,不是錯誤情形:被斷開的 pass 閘極隔離的動態節點以受困電荷 保住自身的值,建模為「電容嚴格最大成員的前值,平手由先見者勝」,電容以總連接數為代理。 在這個工作負載上,浮接分支被不到 1% 的走訪走到,這正是 `NodeConnections` 可以是冷陣列的原因;但它的語意汙染了每一個觸碰求值順序的優化,因為它回傳的值取決於 成員的*先前*狀態。實測分布(第 6 章):77.1% 的走訪建出大小為 2 的群組,16% 為大小 3,5.7% 為大小 4 以上;BFS 深度平均 1.13,p99 = 3,最大 14。換句話說,整套走訪機具是為 一個幾乎永遠極小的導通元件而存在——這個觀察既催生了第 6 章的派發類別,也解釋了第 9 章每一個批次化方案的失敗。

4.2.4 SetNodeState 與入列走訪

狀態寫回是事件傳播之處,也是第 5 章剪枝家族施力之處。其結構是:同值提早返回、存入, 然後對節點的閘極扇出清單——此節點作為閘極的那些電晶體——做一次依新值特化的走訪,因為 閘極變高與閘極變低的傳播義務不同:

```
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
```

兩條腿之間的不對稱是物理性的。導通中的閘極把通道兩端*合併*為一個導通群,因此 入列任一端點即足夠——群走訪會沿著現已 ON 的通道走到另一端——且建構管線會把任何供電 端點正規化到 c2,使 c1 永遠是合法的入列目標。關斷中的閘極則是*斷開*,兩端必須各自 就自己這一側殘餘的驅動者獨立重新解析。與邊界 A、S、B 的不等式比較, 是剪枝安全類別的區間編譯形式(關斷跳過 ⇔ id < S;導通剪枝不安全 ⇔ id < A 或 id ≥ B);其推導、證明義務與驗證是第 7 章的主題——在架構層級,要緊的 事實是熱迴圈的安全詮釋資料已被編譯進 id 空間本身,因此這個走訪完全*不*讀任何逐節點 的遮罩陣列。扇出清單以單一 64-bit 載入一次吞掉兩個(c1, c2)對(實測約 +1.2%):不同於 群走訪的隨機聚集——其指令負擔藏在記憶體延遲停滯之下——這條循序走訪對吞吐量敏感,把 迴圈分支與載入次數砍半是划算的。同一個雙對技巧在群走訪內部只用於長的(溢位節點)鄰接 掃描,在那裡實測 +1.4%——在短走訪上則被否決,因為它反而吃虧;「相同的微優化會隨情境 正負翻轉」這一課在第三部反覆出現。

4.3 組成管線

4.3.1 剖析與模組組成

輸入工件是 Visual 6502 家族的網表 [15][16]: `segdefs` (矽多邊形,引擎只取用 其中的逐節點上拉註記)、 `transdefs` (NMOS 電晶體,為閘極/c1/c2 節點三元組, 其弱元件欄在 2A03/2C02 資料集中一律為 false——空乏式負載改以上拉線段編碼),以及 `nodenames` (名稱 → 節點 id,含 2A03 的內部暫存器)。在兩顆晶粒周圍,系統以 MetalNES 式 [17] 的模組定義檔組裝:模組宣告接腳、子模組實例、上拉清單、ForceCompute 清單、行為式記憶體區域,以及*連線(connections)*——而連線的組成規則是本引擎最 優雅的簡化:**連線就是一顆常開電晶體**,即閘極接在電源節點上的元件。實例化為每個 實例配置一塊新的節點 id 區塊,跨模組邊界解析接腳與萬用字元名稱參照,並為整塊主機板輸出一份平坦的電晶體清單:2A03、2C02、主機板膠合邏輯與卡匣接線成為單一一張均勻的 開關級圖,晶片間通訊沒有任何特例處理。

4.3.2 Lowering

平坦網表接著通過一道保持行為的 lowering,是網表前處理傳統 [2] 的一個刻意極簡的對應物。 共執行三個轉換,全部依構造即可證成。(1)*靜態群組合併*:僅由一顆常開、非弱元件 相連的兩個一般節點,永遠是同一個電氣節點;union-find 把每個這樣的等價類塌縮到一個代表 節點。合併後節點的電容代理值取該類的*最大值*——恰好是若類內成員保持獨立、浮接 tie-break 會在其中選出的那個值——因此連這個邊角情形都與未 lowering 的模型位元相同。 (2)*死元件移除*:閘極接地的電晶體永遠不可能導通,逕行刪除。(3)*緻密化*: 倖存者重新緊密編號(id 0–2 保留給哨兵與兩個供電節點),已消化的連線留下的自迴圈被刪除, 重複的(gate, c1, c2)三元組去重。在組成後的 NES 上,合併塌縮了 441 個節點、移除了 530 顆電晶體,得到本書各處模擬的約 14.7K 節點/約 26.8K 電晶體系統。Lowering 同時*定義* 了 golden 節點編號:4.4 節的每一個 checksum 都以這個 id 空間陳述,而與未 lowering 模型 的比較(以旗標保留的診斷 A/B,實測 −3.7%)則透過在舊 id 空間計算的映射 checksum 進行。

4.3.3 分類遍、重編號與 Reset

處理常式掛載完成之後(4.5 節)——這必須先於 `Reset()`,因為 callback 會 新增假節點與假電晶體,而 Reset 是按最終節點數來決定每個熱陣列的大小—— `Reset()` 執行建構期到執行期的交接:配置 SoA 陣列;由純優先序函數建出 256 項 LUT;把受控節點圖走 一遍,輸出行內 payload 與攤平子清單(沿途把每個節點的通道分桶為一般/接地/接電源); 設定上電狀態(上拉節點起始為高,其餘為低/浮接);蓋上供電與 ForceCompute 旗標;最後 執行各分類遍:派發分類器(cls0/1/2)、剪枝安全汙染遍(對通道元件做 union-find 以傳播 ForceCompute 汙染,外加無上拉汙染),以及關斷跳過遍——後者同時執行 4.4.4 節描述的 區間驗證。

完整的生產載入是這套流程的*三遍*版本:第 0 遍在恆等 id 下組成並分類,取得每個 節點的剪枝類別;第 1 遍以類別為主鍵的排列加上臨時的結構排序鍵重建,把晶片暖機越過重置 暫態(1,024 hc),並透過 settle 迴圈的一份冷儀器化副本,在 32,768 hc 內捕捉生產級聯的 真實初次求值順序;第 2 遍以類別區塊與捕捉到的 locality 鍵做最終重建。整個載入耗時約 1.3 s,且每次載入都從實際晶片重新導出——沒有任何會過期的 profile 檔。這個自我捕捉佈局 的設計與實測經濟學另闢專章處理;此處要緊的是它帶來的正確性義務,下文隨即處理:引擎必須 *可證明地*對自身的節點編號無動於衷。上電本身以一次把所有節點入列、讓整顆晶片安定 一次的掃掠收尾,接著是忠實的重置序列——拉起主機板重置線、跑 192 個半週期、再放開—— 而不是任何抄捷徑的初始化。

4.4 把正確性當作基礎設施

引擎的優化攻勢只接受讓模擬保持位元相同的轉換。這條政策唯有在「位元相同」測起來便宜、騙不過去、且失敗時能提供診斷的前提下才有意義。本節描述這套裝置;我們視它為與它把關出的加速同等地位的貢獻,因為第 9 章的每一個死路都是由這套機制——而非由目測——識別出來的, 而且好幾個看似合理的優化中潛伏的健全性漏洞也是被它抓到的。

4.4.1 Golden checksum 與 SMB1 閘門

主要閘門是對完整 `NodeStates` 陣列做的 FNV-1a 64-bit 雜湊——是全狀態指紋 而非取樣軌跡,因此兩次執行若在同一時間索引上相符,則約 14.7K 個節點值每一個都位元相同。 三個 golden 值在遞增深度上錨定參考工作負載(300k、 400k 與 1M 半週期: `0x794A43ABDF169ADA` 、

`0x9174E19D961CB6E5` 、 `0x6D4CCBCE2E9CD599`),而本書中的每一次 benchmark 執行——含消融階梯全部 72 跑——都理所當然地受 checksum 閘門把關。

單一工作負載,無論量測得多深,都是倖存者偏差的風險:一個轉換可能恰好在該工作負載 靜置不動的電路上不健全。這不是假設性的。對一個已放棄優化(維護事實旁路,第 9 章)的 對抗式審查,找到了參考工作負載永遠暴露不出來的潛伏漏洞,因為精靈 OAM 匯流排在其中基本 上閒置;它的 golden 是倖存者偏差。對策是第四道閘門:對一款精靈密集的商業遊戲 (Super Mario Bros.)跑 1000 萬半週期,讓 OAM 與精靈求值資料路徑——正好是落在浮接 tie-break 上的動態儲存電路——被鍛鍊達數分鐘的模擬時間。這個組合刻意異質:一個量得深的 合成全螢幕測試 ROM,加上一個量得久的真實程式。

4.4.2 依構造而得的佈局獨立性

4.3.3 節的自我捕捉重編號在每次載入時都會把引擎的整個 id 空間重新排列。要讓 golden checksum 維持意義——也要讓重編號後的執行還能與恆等編號的歷史相比——模擬就必須在這個 排列下不變。引擎靠構造、而非靠運氣達成這一點:

> **性質 4.1.** 對任何固定保留 id 的節點 id 排列, 模擬出的事件序列與所有 golden checksum 皆不變,因為:(i) 引擎中唯一依賴 id 順序的計算—— 在第一次安定之前把所有節點入列的上電掃掠,其順序會餵進浮接保持前值 tie-break——是透過 該排列以*原始* id 順序疊代;且 (ii) checksum 同樣透過該排列以原始 id 順序對 `NodeStates` 取雜湊。引擎中所有其他順序(波追加順序、群走訪順序、扇出清單 順序)皆由結構清單誘導,而這些清單重建時保留其相對順序。

這是查核過的,不是假設出來的:上電掃掠在原始碼中被記載為唯一依賴 id 順序的位置,而 這個性質持續受到鍛鍊,因為每一次生產執行都是重編號後的執行,卻對照著恆等編號下記錄的 checksum 檢驗。同一套紀律對正當改變節點集的轉換(例如 lowering)給出*映射 checksum* 規則:閘門變成透過該轉換的映射、在舊 id 空間計算的 checksum,與未轉換的參考值相比—— 即倖存狀態的位元精確,並把重新定基(re-baselining)明示出來,而非默默接受。

4.4.3 逐節點 dump-diff

checksum 是一位元的神諭;診斷需要座標。互補的工具會在選定時刻傾印完整的逐節點狀態 向量(以原始 id 順序,並從 `nodenames` 附上名稱),再對兩次執行逐節點做差異 比對。它最具決定性的一次出場,是第一次同態(same-state)剪枝嘗試的事後剖析:天真版剪枝 通過了短程執行,然後發散成黑畫面,而 dump-diff 把最早發散的節點定位到 OAM 與調色盤 RAM 細胞——其值流經浮接 tie-break 的無上拉動態儲存——推翻了一個起初看似合理但錯誤的理論 (ForceCompute 交互作用),並直接催生出讓該剪枝變得健全的結構性安全汙染。這個通用模式 值得明說:在這套方法論裡,一個失敗的 checksum、加上逐節點差異比對、加上具名的網表節點, 通常能在一個工作階段之內把一個 heisenbug 轉化為電路層級的解釋。

4.4.4 驗證過的投機:Reset 期的區間檢查

有一件正確性機具就跑在引擎本身之內。4.2.4 節的區間編譯剪枝測試是技術意義上的投機(speculation):熱迴圈假設 id 空間的佈局使每個安全類別恰為一個連續區塊。因此分類遍在 *每一次* Reset 都從第一性原理重新計算完整的逐節點安全遮罩作為基準真相,並逐節點 驗證遮罩位元與其 id 區間所蘊涵的位元一致。一旦有任何不符,引擎就不信任該佈局:回退到 安全退化邊界,使所有剪枝失效(供電過濾仍然成立),產出一個正確、但約慢 2× 的模擬,外加 一聲響亮的警告。這是去優化防護(deoptimization guard)模式 [13] 從執行期編譯移植到資料 佈局:只有在機器查核的不變式成立時才使用優化形式,而失敗模式只是效能、永遠不是錯誤。 在 Release 組建中,驗證過的遮罩隨後即被釋放——熱路徑只讀區間——使這個檢查在執行期 是字面意義上的免費。

4.4.5 先量測、再保留

最後一層是方法論上的。每個候選優化依序通過兩道閘門:位元精確(所有 checksum 閘門), 然後是在交錯配對(interleaved-paired)A/B 下的實測勝出—— 兩個二進位皆預先建好,每輪之內輪替基準/實驗,回報中位數與配對勝場數。之所以交錯,是 因為批次式 A/B(先跑完所有 A、再跑所有 B)被證明不可信:整個場次間的熱漂移 與時脈漂移,足以讓 3% 以下效應的正負號翻轉,而且至少有一個早期的「死路」判決在改用配對 重測後被推翻。量測一律不鎖頻,以同日多輪 round-robin 交錯、單一 benchmark 進程、中位數 回報控制變異。文化上的後果值得記錄:當閘門便宜到這個地步,專案對「X 會有幫助嗎?」的 預設回答變成「做出最小的位元精確版本然後量」,而第 9 章的負結果目錄正是其直接產物—— 其中每一條都是經過量測、受 checksum 把關的否決,而不是一個意見。

4.5 行為式周邊

4.5.1 以假電晶體實作的 callback

引擎需要觀察晶片(偵測影格邊界、位址選通、測試 ROM 完成),也需要在晶片接腳掛上 行為式元件(記憶體、視訊輸出)。兩種需求由一個承襲自 MetalNES [17] 的機制一併滿足: 註冊一個 *callback* 的方式,是配置一個新的假節點,並為每個受監看節點加一顆假 電晶體——其閘極是受監看節點、c1 是假目標節點、c2 接地。每當任何受監看節點改變狀態, 普通的傳播機具就會把目標節點拉進一個群組,其帶 callback 旗標的求值再把處理常式排入—— 監看清單是*由*模擬器本身實作,而非掛在它旁邊,沒有逐事件輪詢,熱迴圈中除了既有的 HasCallback 旗標外也沒有任何特殊派發。處理常式只在安定到達靜止之後執行,且可重入: 驅動接腳的處理常式會觸發一次巢狀安定,其自身的 callback 再依序排空。這個設計正是處理 常式必須在 Reset 之前掛載的原因——它們是貨真價實地擴充了網表——也是派發類別 cls0 存在 的原因。

4.5.2 記憶體、時脈與視訊處理常式

三個處理常式家族構成全部的行為式表面。*時脈*「處理常式」是退化情形:主機板的 石英振盪器不被模擬;`StepCycle` 直接翻轉解析後的 `clk` 節點。 *記憶體*處理常式服務那些市售儲存零件:主機板上兩顆 2 KB SRAM(CPU 工作 RAM 與 PPU 的 nametable VRAM)、卡匣的 PRG 與 CHR ROM(卡匣沒有 CHR ROM 時則為 CHR RAM), 以及測試 ROM 用來回報結果的選配電池 RAM 區域。模組以宣告式方式宣告記憶體區域;處理常式 監看晶片選擇(chip-select)、寫入致能與位址接腳,對一塊非託管 byte 緩衝區做普通的陣列 存取,再透過尋常的外部驅動旗標(SetHigh/SetLow)驅動資料接腳、接著安定一次——因此從 晶粒的視角看,行為式 RAM 與一顆非常快的真實 RAM 無

從區別。*視訊*處理常式監看 PPU 像素時脈,在每個上升緣從具名內部節點讀出水平/垂直位置計數器與調色盤索引,經調色盤 RAM 轉換成一次 ARGB 影格緩衝區寫入。對具名節點群的位元向量存取,經由小型聚集/散佈輔助函式（`ReadBits` / `WriteBits`）以無分支方式讀取狀態陣列。

4.5.3 範圍規則

開關級與行為式之間的邊界是專案明文宣告的不變式,不是便宜行事的偶然:**2A03 與 2C02 晶粒內部的一切,永遠是電晶體層級**。行為式元件只存在於晶片邊界,且只替代從來不在晶粒 上的零件——分立記憶體、石英振盪器、顯示器。在本引擎中,沒有任何內部暫存器、任何 OAM 細胞、任何資料路徑曾被手寫模型取代;那一類抽象化實驗是刻意放在一個獨立分叉(S2 調查,第 9 章)中進行的,其結論從未改動 S1。這條規則在安全機具中是有牙齒的:關斷跳過分類器 必須把處理常式驅動的資料匯流排接腳明確排除在其孤立葉類別之外——在通道斷開的那一刻被 SetHigh/SetLow 驅動的節點並不會浮接保值,而教會我們這一課的早期發散就記錄在剪枝安全 檔案裡——所以周邊不只是被劃定範圍,更在每一個健全性論證中被主動納入考量。

> 本模型的擬真度主張應當被精確地解讀。引擎是對承襲而來的開關級語意 [1][15][17] 位元精確——那是一個離散、無單位延遲、帶電荷的二態近似,以安定順序代替類比 時序、以連接數代理代替電容。它不是電路模擬器:傳播延遲、優先序類別之外的驅動強度、 類比準位都在模型之外。本書通篇的正確性陳述,是對這個模型及其祖先實作的等價性陳述,並以 執行真實軟體做端到端驗證;它們不是類比精確度的主張。在這個範圍之內,周邊再添一條告誡: 行為式記憶體在單一次安定之內就回應,比任何真實 SRAM 都快——這忠於家族的既定實務 [17], 並由同一套端到端閘門驗證。

總結而言,此處描述的架構是少數幾個承重決策——按存取模式拆分欄位的 SoA 非託管佈局、 行內 payload 的節點紀錄、順序即語意的雙緩衝 Gauss–Seidel 安定、以分類與驗證過的重編號 收尾的組成管線,以及一套讓位元精確成為便宜機械閘門的正確性裝置。接下來三章量化在其上 建造的成果:工作負載的實測解剖與剪枝家族及其證明(第 5 章)、基數特化派發(第 6 章)、自我捕捉資料重排(第 7 章);實測死路的目錄則見第 9 章。所有原始碼與 benchmark 套件皆公開 [21]。

第 5 章 剪枝家族:在源頭壓制可證明無效的事件

前一章確立了引擎的結構性事實:小到足以常駐 L1 的工作集、成本由相依載入鏈主宰的事件迴圈,以及一份正確性契約——以全狀態 checksum 為準的位元精確——它禁止任何「對輸出的影響無法被 證明或被窮盡驗證」的變換。本章發展建立在該基礎之上的三個優化家族中的第一個:一組四個 *剪枝*(prune),P-1 到 P-4,只要被壓制的重算可被證明是 no-op,就在入列的當下壓制事件 傳播。這個家族的前提很簡單,而且在本引擎上是量測而非假設出來的:事件驅動開關級模擬的主要 成本,不在於每次重算很慢,而在於大多數重算根本不該發生。一次結果等於既存值的重算是純粹的 浪費——它執行與有生產力的重算相同的隨機聚集(gather)、佔用相同的佇列槽位、停滯在相同的 cache line 上,然後什麼都沒有改變。由於每一次個別停滯都已逼近記憶體延遲地板 (第 4 章),剩下唯一的槓桿就是精確地*少做*幾次——而且不能移動任何一個輸出位元。

本章的組織是一條從量測、到機制、到證明的論證。5.1 節量化機會並解剖被浪費工作的構造。5.2 節呈現 P-1——同態(same-state)入列剪枝——刻意從它失敗的前身講起,因為正是那次失敗 教會我們危險住在哪裡。5.3 節呈現 P-2——關斷隔離剪枝——其壓制發生在佇列之前,因此刪除了 整條入列—彈出—解析鏈。5.4 節呈現 P-3 與 P-4——電容支配解除汙染(un-taint)——我們視之為本研究最強的原創主張。5.5 節描述讓四個剪枝全數安全、卻不需手列任何一個節點的零組態分類遍。 5.6 節報告實測效果;5.7 節則以三項獨立佐證劃定剪枝所不能及的範圍。

5.1 機會:八成的工作什麼都沒改變

5.1.1 量測被浪費的重算

引擎以排空一份雙緩衝事件清單的方式安定(settle)每個半週期:髒節點被彈出,每次彈出派發到 一條解析路徑(O(1) 單例快速路徑,或第 4 章的一般群走訪),解析出的值若有改變,便經由該節點 的閘極扇出傳播新的入列。我們在彈出迴圈裝上一個僅 DEBUG 的剖析器,依「結果是否真的改變了 *任何* 狀態」分類每一次彈出;事件數在 Release 組建中完全相同,所以這份剖面描述的就是 生產引擎。頭條數字相當鮮明。在目前的引擎上,一次 100,000 半週期的 `full_palette.nes` 執行約進行 42.85M 次彈出,其中 **80.1% 重算出的值等於既存值** (實測)。在剪枝家族落地之前,同一個剖析器在 300,000 半週期上數得 159.6M 次彈出,其中 **84.4%** 沒有產生任何變化,而那些浪費約有 88% 集中在動態單例(cls2)節點——即那 10,784 個(佔母體 73.2%)擁有 pass 通道、但通道通常全部關閉的節點(實測)。

這種冗餘不是實作的缺陷;它是選擇性追蹤(selective trace)傳統 [4] 之下位準敏感 (level-sensitive)、事件驅動傳播所固有的。只要相鄰任何一顆電晶體的閘極翻轉,節點就會被 重新排入佇列,因為這次翻轉*可能*改變該節點的導通群、進而改變其解析值。但通常並不會。 一個經由三條並聯下拉路徑保持為低的節點,在其中一條斷開時不會改變;一個唯一下拉本來就關閉 的上拉節點,在第二條冗餘下拉也關閉時不會改變;一個浮接的儲存細胞,在兩個通道之外某顆 不相干的存取電晶體切換時不會改變。事件系統在入列當下無從得知這些——除非我們告訴它:用 一個便宜到勝過它所省下工作的檢查,加上一個強到足以保住位元精確的安全論證。

5.1.2 無變化彈出的解剖

剖析器進一步依節點的結構類別與所走的解析分支,分類無變化彈出。表 5.1 給出目前引擎上的 細分(剪枝之後;各類別在前後的定義完全相同)。

表 5.1. 無變化彈出的類別,目前引擎, `full_palette.nes` ,每 100,000 半週期(實測;DEBUG 剖析器,事件數與 Release 相同)。占比以*全部*彈出(約 42.85M)為分母,其中 80.1% 為無變化。

類別	占全部彈出比例	其物理意義
PullUp	42.0%	一個空乏負載(depletion-load,即上拉)節點,在其 某條下拉路徑上的閘極翻轉後被重算,而另一條導通路徑——或根本沒有任何導通路徑——使解析值 維持不變。重算必須掃描該節點的接地通道閘極清單才能發現這一點。
Supply	38.1%	對於有直接通道連到 V_{CC} 或接地之節點的 類似情形:一條供電路徑切換,但其餘路徑的 OR——也就是 LUT 的輸入——並未改變。
FloatSingle	7.5%	一個無驅動節點在被隔離時遭到重算:其群組只有 自己,旗標 OR 為空,而浮接解析依定義保持前一狀態。
FloatMulti	12.1%	一個多節點的純浮接群組,其最大電容成員早已 持有 tie-break 將選出的值——電荷分享只是重新導出已儲存的位元。

這四個類別並非同等可攻。兩個浮接類別是靜態證明的天然獵物:一個節點*能否*被驅動、 *能否*贏得 tie-break,是網表的性質,不是這一次執行的性質。PullUp 與 Supply 類別恰好 相反:要判定一個上拉節點不會改變,需要知道其其餘下拉閘極的即時狀態——而那正是重算本身 所做的掃描。這種不對稱形塑了整章:P-1 到 P-4 收割浮接類別(以及餵養所有類別的入列端冗餘), 而 5.7 節以三項獨立佐證顯示:PullUp/Supply 殘餘不存在靜態子集,連有利可圖的動態攻擊也 抵擋得住。

5.1.3 為什麼壓制必須發生在源頭

一次無變化彈出的成本遠高於其算術。整條鏈是:一次入列(生產端的去重 hash 探測與清單附加)、一次彈出(從波清單載入)、一次派發(類別測試)、一次解析(至少要讀該節點的 `NodeInfo` cache line 並掃描其供電閘極清單;最壞時要在 ON 通道上做一次群走訪, 連帶每成員的隨機聚集),最後才發現什麼都沒有變。在 P-5z 研究(第 9 章)期間的量測顯示, 即使最便宜的彈出類別也要約 30 個週期,幾乎全是記憶體延遲。在彈出*之後*壓制事件——替 浪費的事件換上較便宜的解析——頂多削掉這條鏈的尾巴。在入列現場壓制則整條鏈一併刪除:沒有 佇列流量、沒有去重 hash 汙染、沒有派發、沒有聚集,並且遞移地不再拉長安定波(settle wave) (引擎平均每半週期 12.06 個波;波裡每個多餘成員,同時也是排空迴圈的一次多餘迭代)。因此 四個剪枝全都住在 `SetNodeState` 的閘極扇出走訪裡——狀態變化向未來事件扇出的 唯一現場——完全不在彈出路徑上。

約束後續一切的紀律是位元精確。如第 4 章所確立,純浮接 tie-break 不是個數值細節:它 *就是*晶片的儲存機制。2C02 的物件屬性記憶體(OAM)與調色盤 RAM,以及 2A03 中為數眾多 的動態節點,都以電荷保存位元;每次存取時,其值正是由「最大電容成員」規則復原。一個剪枝若 在這類節點上誤判「沒有變化」,造成的不是小誤差;它會弄丟一個已儲存的位元,而發散會無界地 複利擴大。因此這個家族的每個剪枝都以逐節點的*安全證明*把關,該證明在載入期靜態算出; 而每個上線變體都在多個時間長度的全狀態 golden checksum、外加 1000 萬半週期的精靈密集門上 通過驗證。

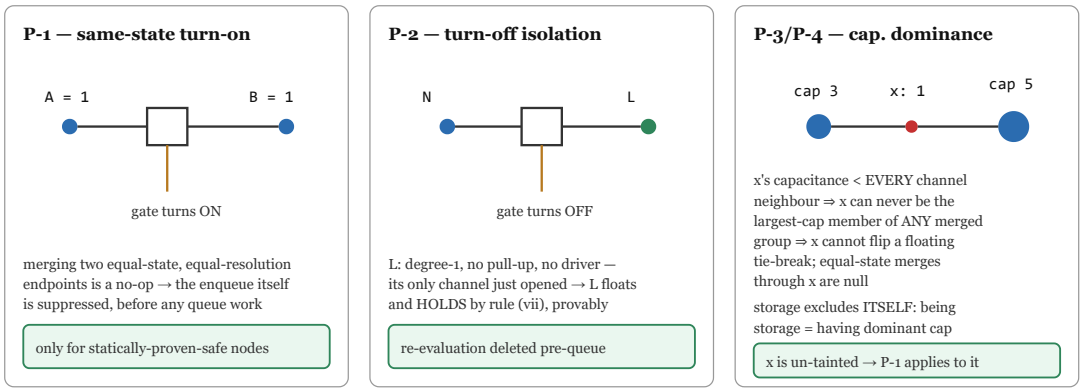

圖 5.1. 三個剪枝的結構證明示意。左:P-1 同態入列剪枝(5.2 節)——兩端點狀態相同的導通合併是 no-op;中:P-2 關斷隔離剪枝(5.3 節)——度 1 無驅動葉節點在唯一通道斷開後必然浮接保值;右:P-3/P-4 電容支配解除汙染(5.4 節)——電容嚴格小於所有通道鄰居的節點永遠贏不了浮接 tie-break。三者皆為載入期結構證明,熱迴圈只測一個預計算位元。

5.2 P-1:同態(same-state)入列剪枝

P-1 針對導通(turn-on)情形。當一個節點的解析值上升,它所閘控的每顆 pass 電晶體都會轉為 ON,而每條這樣的通道都可能把其兩個端點上的導通群合併起來。引擎每條通道入列一個端點(從該處 播種的走訪會沿著剛轉為 ON 的通道走到另一端)。P-1 背後的觀察是:若兩個端點早已持有相同的值, 兩個等值群組的合併通常不可能產生任何新東西——重新解析合併後的群組,寫回的正是成員們早已 持有的值。這個想法——不要排程一個其值可證明不會變的後繼——是選擇性追蹤 [4] 的標準精煉, 並以等電位旁路(equipotential bypass)的形式遍見於開關級文獻 [3]。本節的貢獻不在想法本身, 而在讓它於 Bryant 式電荷模型 [1] 上位元精確的*安全分類*;我們依事情實際發生的順序講這個故事,因為失敗才是承重的那一段。

5.2.1 前身:prune-merge 與黑畫面

第一次嘗試大約在專案優化分支進行一年時,是一個叫 *prune-merge* 的執行期方案:為每個 節點維護一個群組 id,當一顆電晶體在兩個其群組持有相同數位值的節點之間導通時,直接跳過合併 重算——單純的數位等值測試,沒有任何附加條件。在吞吐 benchmark 與 CPU 側測試 ROM 上它看起來 十分驚人:約為當時基準線的 1.47×,CPU 也正確執行指令。但它同時是錯的。`full_palette.nes` ——一個 PPU 測試,其參考圖樣(全部 64 色的格狀畫面)在第 48 影格前完成——畫出來的是黑畫面。發散正是從浮接 tie-break 進來的:在無驅動的儲存與匯流排 節點上,同態合併*並非*一般性地無效,因為合併後群組的解析並不是兩個端點值單獨決定的 函數。一個電荷分享群組可以解析成第三個成員——最大電容那個——的前一狀態,而只看端點的 等值測試看不見它。PPU 裡儲存的位元悄悄腐壞,畫面就此消失。

> **黑畫面教會我們的驗證課**。不安全的 prune-merge 通過了我們當時擁有的每一項 CPU 側 檢查。這個教訓——「在 CPU 上觀察起來相同」、甚至「通過指令測試」都遠弱於位元精確——把 專案的驗證紀律淬鍊成第 4 章描述的形態:以多個時間長度的全狀態 FNV-1a checksum 為首要關卡、 至少一個視覺 ROM 渲染成逐位元組比較的影像,以及後來加上的 1000 萬半週期精靈密集門。本章 其後的每個剪枝都是在那道關卡之下開發的。我們選擇報告這段插曲而非埋掉它,因為方法論 本身就是一項結果:在這個領域裡,沒有 checksum 的加速數字不算數據。

5.2.2 代價超過收益的執行期修補

第一次修補保留了執行期機制、修正了語意:維護拓撲感知的群組 id,並在每次走訪時批准生效 (每次解析配發一個新 id,指派給每個成員),使等值測試只適用於群組結構讓它健全的地方。這個 版本是正確的——checksum 逐位元相同,而且 50 影格的 `full_palette` 截圖之 PNG 與 參考檔逐位元組相同、連壓縮痕跡都一致——但經濟學已經反轉。每次走訪的維護成本(id 配發, 加上每個群組成員一次寫入,每秒數百萬次走訪、次次都付)是恆常支出;跳過的回報卻只發生在 符合條件的導通子集上。修補當時淨值打平(約 1.02×);等到引擎後來的內聯與佈局工作把 群走訪本身變便宜之後,被維護的 id 就成了純然的負債,實測 **−4.4%**——比完全不剪枝還慢。該旗標就此退役。

這段插曲可以一般化,而且這個一般化在本書中反覆出現:在這顆引擎上,*被維護的執行期事實 不划算*。任何在熱路徑上更新輔助狀態、好讓後續某個檢查變便宜的方案,都必須把維護成本攤提 在單個只有數十奈秒的事件上;此類方案中最具野心的一個(P-5,第 9 章)實測維護地板約佔總 執行時間 7%——比戰利品還多。剪枝家族的最終形態就是結構性的解答:把所有能搬的事實搬到 載入期,讓成本只付一次,熱路徑則免費查閱。

5.2.3 靜態重生

上線形態的 P-1(2026 年 6 月)問的是另一個問題:不是「這一次合併是否無效?」——一個需要 執行期事實的執行期問題——而是「這個*節點*是否屬於『對它可能加入的每一個群組,同態合併 皆無效』的那種節點?」——一個關於網表的靜態問題,在載入時回答一次,記成每節點一個位元。 於是熱路徑測試塌縮為:在通道 (c1, c2) 導通時,若且唯若 c1 被分類為安全且 `states[c1] == states[c2]`,就跳過 c1 的入列。兩次載入加一次比較,而且讀的是 入列走訪本來就已付費觸碰的即時狀態。

即便是靜態形態,其安全條件也是吃過苦頭才找到的。天真的靜態剪枝——凡等值處處跳過—— 再次發散(又一次黑畫面)。第一個理論怪罪 ForceCompute 節點(接地與電源貢獻互消的預充匯流排模型;第 4 章),而它是錯的——或者說,不完整。以逐節點狀態 dump 與未剪枝參考做差分,把 最早的發散定位到 PPU 物件屬性記憶體與調色盤 RAM 裡的*無上拉*節點——活在保持前值 (hold-previous)tie-break 上的動態儲存,正是當年殺死 prune-merge 的同一機制。上線的分類 把這兩個母體都標為汙染,結果是上線當時 **+11.85%(14/14 成對輸)**, 在每一道關卡上位元精確(實測)——在當時是專案中繼 R-1 派發路徑(第 6 章)之後的最大單筆 勝利。

5.2.4 安全汙染(safety-taint)分類學

載入期分類器在以下兩個結構條件任一成立時,把節點標為*導通不安全*(turn-on-unsafe; 無論等值與否都必須入列):

- **無上拉**。沒有上拉的節點可能屬於一個純浮接群組——旗標 OR 為空的群組——其解析是 對成員前值做電容 tie-break。對這類群組,端點等值測試什麼也證明不了(5.2.1 節)。反之, *有*上拉的節點根本不可能落在純浮接群組裡:它自身的旗標讓 OR 非空,迫使解析走優先序 LUT。因此把所有無上拉節點標為汙染,就從構造上把剪枝限制在 LUT 解析的合併。光是這一個測試, 就把組合邏輯與電荷儲存——OAM 細胞、調色盤 RAM、2A03 的動態節點——區隔開來,而不必點名 其中任何一個。
- **ForceCompute 通道元件**。LUT 的單調性(5.2.5 節)在 ForceCompute 互消規則面前失效:該規則下,同時出現的接地與電源貢獻彼此抵消,解析落到較弱的旗標。只要*任何*成員帶著 這個旗標,群組的解析就是非單調的,所以汙染必須涵蓋一切可能與 ForceCompute 節點同群的節點。 分類器用一個建立在通道圖(c1–c2 邊,亦即假設所有通道皆 ON 時的最大可能導通群)上的 union-find 計算這件事,並把元件中含有 ForceCompute 成員的每個節點標為汙染。位於不含 ForceCompute 之元件中的節點,無論哪些電晶體導通,都永遠碰不上互消規則。

兩個條件都刻意保守:它們只用可達性與旗標、絕不用模擬,來過度近似危險集合。5.4 節將顯示 第一個條件*過於*保守——無上拉母體中有一個可證明的子集能被重新接納——而把它撈回來的 價值,超過 P-2 與 P-4 加總。

有一項實作義務值得專門一句,因為位元精確往往就悄悄死在這種細節上:被剪掉的節點不可以 設定其去重 hash 位元。這個 hash 標記的是「本波已入列」;若設定了它卻沒有真的把節點附加進 佇列,節點看起來像已入列、實際上永遠不會被彈出,而同一波中該節點後來一次*正當的*入列 就會被無聲丟棄。因此剪枝把關的是整個入列動作,而非單獨的 hash 更新——這次跳過不留下任何 痕跡。

5.2.5 健全性

性質 5.1(受驅動解析下的同態合併)。設 t 為 一顆通道端點為 a 與 b 的 pass 電晶體,其閘極轉為 ON,並假設 (i) a 帶有上拉,(ii) a 的通道 元件中沒有任何節點帶 ForceCompute,且 (iii) 翻轉當下 `states[a] = states[b] = v`, 且兩端點群組處於靜止(quiescence)。則重新解析合併後的群組會把 v 寫入每個成員、不改變任何 狀態、也不產生任何後續事件;壓制這顆種子是不可觀察的。

論證沿著解析 LUT 的優先序結構展開。在靜止態下,導通群的每個成員都持有該群的解析值,因此端點狀態可以代表各自群組的值。合併後群組的旗標集合是兩個組成旗標集合的 OR(閘極翻轉改變的 是連通性,不是成員旗標)。由條件 (i),合併後的 OR 非空——a 的上拉保證了這一點——所以解析 會走 LUT;由條件 (ii),互消規則不可能觸發。沒有互消時,LUT 是純粹的優先序函數:其輸出由 在場的單一最高優先序旗標決定 (Gnd > Pwr > SetHigh > SetLow > PullUp)。兩個集合之 OR 的最高優先序旗標,是 兩集合各自最大值中較高的那個,因此必屬於其中一個集合;該集合的群組解析為 v,而既然輸出只 取決於那個最大旗標,合併後的群組同樣解析為 v。每個成員都已持有 v,所以走訪的寫入全是 no-op,其後續入列(只在實際狀態改變時觸發)為空。被壓制的種子本來就不可能產生任何東西。

兩點誠實的保留。第一,(iii) 中的靜止前提是一種理想化:剪枝在波中、在 Gauss-Seidel 順序下 觸發,當時存在暫態的成員狀態。分類已移除波中暫態可能與解析發生非單調互動的情形(浮接群組、互消);對剩下的 LUT 解析情形,剪枝讀到的端點狀態正是被壓制的走訪原本會讀到的狀態,因此 剪枝

決策與假想中的走訪看見的是同一個一致的世界。第二,我們不宣稱這個論證是機器檢核過的 證明。它是一份證明概述,其殘餘縫隙由我們所能構築的最強實證關卡補上:300k、400k 與 1M 半週期的全狀態 checksum,以及一道 10M 半週期的精靈門。這個 組合——結構性論證加上窮盡的位元層級驗證——是這個家族每個成員都達到的標準。

5.3 P-2:關斷隔離剪枝

5.3.1 機制與證明

P-2 針對另一側的邊緣。當節點的值下降,它閘控的每條通道都轉為 OFF,而每條通道的兩個端點 通常都必須重算,因為斷開可能把一個群組劈開,而任一碎片都可能解析出不同的值。但有一類節點,其重算結果在載入期就已注定。考慮一個恰有一條 pass 通道、沒有通往任一供電的通道、無上拉、 無 ForceCompute 旗標、無 callback、也無外部驅動的節點。它唯一的通道一斷開,這個節點就成了 孤島:只有一個成員的導通群,旗標 OR 為空。純浮接單例的解析依定義就是它自己的前一狀態—— 實作動態儲存的那個「保持」——所以這次重算保證是 no-op,每一次都是,無論機器其他部分發生 什麼。

> **性質 5.2(隔離保持)**。設 x 為一個恰有一條 pass 通道、沒有供電通道、且不具上拉、外部驅動、ForceCompute 或 callback 任何屬性的節點。 當 x 之通道的閘極下降時,x 的導通群成為 {x},旗標集合為空;其解析即其前一狀態。在此事件上 重算 x 是 no-op,且上述靜態屬性皆可在載入期判定。

證明可由解析語意直接得出:群組成員資格是 ON 通道可達集合,對唯一通道已關閉的 x 而言就是 {x};由屬性條件,{x} 上的旗標 OR 為空;而單例的浮接規則回傳 `states[x]`。有趣的不是證明本身,而是測試套用的位置:*在佇列之前*。關斷入列 走訪查閱一個遮罩位元(在上線引擎中是一次暫存器比較——第 7 章),然後乾脆不把 x 入列。依 5.1.3 節,這刪掉的是整條入列—彈出—解析鏈,而不只是解析;在 P-2 之前的引擎上,可歸因於 這一類的無變化彈出實測約佔*全部*彈出的四分之一,是可指認的最大單塊浪費。分類器找到 **1,272** 個這樣的節點(實測)——被動的 pass 電晶體葉節點:存取電晶體的遠端、傳輸閘的 中繼點,以及網表上類似的無驅動附屬節點。

5.3.2 受驅動腳位的修正

靜態條件的第一版錯得很有教育意義。行為式記憶體 handler(第 4 章)透過 callback 設定外部 驅動旗標來驅動 RAM/ROM 資料匯流排腳位——這是光看網表看不見的物理。一個在結構上是單通道 無驅動葉節點的腳位,若有 handler 在驅動它,就*不是*浮接保持:一旦被隔離,它解析成被 驅動的值,跳過它的關斷重算就會把一個過期位元凍結住。這次發散在 20,000 半週期之內被 checksum 關卡攔下,逐節點差分指認出元兇恰好是三支 ROM 資料腳位。修法是結構性的,不是補丁: 分類器排除出現在任何 handler 宣告之輸出腳位集合中的每個節點,讓 handler 的驅動能力成為 靜態模型的一部分。我們報告這件事,因為它讓 5.5 節的方法論主張更銳利:「零組態」的意思是 從組合系統的結構——網表*加上* handler 介面——推導而來,不是只從網表。

5.3.3 定位與歸屬

隔離即保持的行為本身是開關級語意的基石——它就是 Bryant 的電荷保持 [1],而 IRSIM 的增量求值器在求值這類節點時也會得到同樣的保持值 [3]。我們在文獻中沒有找到、並主張為本研究次要 原創貢獻的,是把那個執行期求值結果*提升*為附著於節點的載入期靜態證明,並以佇列前壓制 的形式消費——事件從未被排程,而不是被排程後解析成一次保持。區別在於「便宜地求值」與 「不求值」;在延遲綁定的引擎上,後者在性質上更強。其 commit 當時的上線增益為 **+1.4%**(交錯成對,位元精確)(實測)——單獨看不大,但 P-2 的條件集合與遮罩 基礎設施正是 P-3 與 P-4 的地基,而整個家族的合併效果(5.6 節)才是共用機制的正當性所在。

5.4 P-3/P-4:電容支配解除汙染(un-taint)

5.4.1 逼出一條定理的兩個節點

P-1 的無上拉汙染健全但鈍:為了保護其中的儲存細胞,它犧牲了所有動態節點。P-3 問的是: 有沒有一些無上拉節點終究可以被證明無害?實證軌跡以一次失敗開場,模式如今已經熟悉:早期 一次沒有護欄就放寬汙染的嘗試,在整台機器上恰好於*兩個*節點發散——2A03 ALU/狀態暫存器 區域的電荷分享節點。沿著狀態 dump 追蹤這兩個節點,精確展示了同態合併如何說謊:合併後的 群組變成純浮接,tie-break 選出了一個兩個端點以外的成員,而它的前一狀態不同。這兩個節點 很「重」——局部電容大,而那正是它們*為何*贏得 tie-break、也*為何*能充當儲存。 把這個觀察反過來,就是定理:一個永遠贏不了 tie-break 的節點,永遠不可能說謊。

5.4.2 支配性質

> **性質 5.3(電容支配蘊含 tie-break 免疫)**。設 x 為一節點,對 x 的*每一個*通道鄰居 m 皆有 cap(x) < cap(m),其中 cap 為模型的電容 度量,且比較為嚴格不等。則 x 永遠不會是任何包含 x、大小至少為二的導通群中的最大電容成員; 從而 x 永遠不會決定任何純浮接解析的值。若 x 另外不帶任何解析旗標(無上拉、無外部驅動、無 ForceCompute)且無供電通道,則 x 在任何群組中的成員身分都是值惰性(value-inert)的——它對 旗標 OR 毫無貢獻、也無法裁決浮接 tie-break——於是壓制 x 處的一次同態導通入列即為無效。

證明概述有兩條腿。*成員資格*腿:任何包含 x 與至少一個其他節點的導通群,必經 x 的 某條 pass 通道抵達 x,因此該群至少含有 x 的一個通道鄰居;由假設,該鄰居的電容嚴格大於 x 的, 所以 x 不是群中的最大者。嚴格性是關鍵:引擎以群走訪中的先見順序裁決電容平手,而那取決於 BFS 播種、無法靜態分析,所以一個只是與鄰居*平手*的節點,在某些走訪順序下仍可能獲勝。 因此分類器要求被每一個鄰居嚴格支配,凡有等電容鄰居的節點一律讓給汙染。*貢獻*腿:受驅動 群組的 LUT 輸入,是成員旗標的 OR 加上經由成員 ON 供電通道發現的供電貢獻;一個無旗標、無 供電通道的節點對兩者皆無所增,所以合併後群組的解析值——無論受驅動或浮接——都與完全不 諮詢 x 時相同。再結合 P-1 的端點等值測試——它保證 x 自身的狀態已等於另一端點的狀態(堵住 x 橋接兩個不同值群組、自身必須被改寫的情形)——被壓制的重算既改不了群組的值,也改不了 x 的值。P-3 把這套用在單通道節點;P-4 是同一敘述向多通道節點的推廣,此時支配必須對*所有* 鄰居同時成立。

有一項但書應寫進性質的細則,而非腳註。模型的「電容」是承襲自 Visual 6502 模擬器家族 [15][17] 的靜態連接數——一個拓撲代理量,不是萃取出的物理電容。這不削弱證明:定義本引擎 (及其祖先)儲存語意的浮接 tie-break 消費的是*同一個量*,所以支配不等式在模型之內是 精確的——證明談的是模擬器的語意,而那正是位元精確契約所針對的對象。至於連接數是否忠實地 排序晶粒上的物理電容,那是關於整個模型家族的保真度問題,與本研究正交,我們對此不作任何 主張。

5.4.3 儲存自我排除

> **自我排除原理**。支配測試最深刻的性質,在於是誰沒通過它。儲存細胞正是*靠*贏得 浮接 tie-break 才能運作:以電荷保存一個位元,意味著在群組浮接時自己是群中最大電容的成員。 因此局部支配性的電容不是儲存的偶然屬性——它就是儲存的物理本質,在這個模型裡如此,在晶粒 上亦然。最需要保護的節點(OAM 細胞、調色盤 RAM、動態閂鎖、5.4.1 節的那兩個 ALU 節點)恰恰 是無法滿足 cap(x) < cap(每一個鄰居) 的節點,所以它們*自動*保有 P-1 的汙染。分類器不需要暫存器 清單、不需要名稱比對、不需要逐晶片調校:當被要求找出可安全剪枝的節點時,它對安全的定義 使得儲存自己取消了自己的資格。解除汙染共接納 **986** 個節點(實測)——輕量的 pass 電晶體附屬節點,由同一條不等式可知,它們儲存的 電荷永遠不會被諮詢。

我們把性質 5.3 及其運用視為本研究最強的原創主張,並附上誠實所要求的精確界定。每一項 原料都是經典:電荷分享格(lattice)與按尺寸解析是 Bryant 的 [1];自動的電晶體層→閘級邏輯 抽取與以子圖同構為基礎的子電路圖樣識別是成熟的 EDA 實務 [5][6]——一個程式庫/結構圖樣 比對導向的家族,與我們基於物理性質(上拉旗標、電容比較、通道元件)的辨識形成對照;等電位 跳過則是選擇性追蹤之上的民間常識 [4][3]。我們 找不到先行研究的,是那一步具體的棋:在模型的電容度量上自動推導出的逐節點靜態支配不等式, 用來證明 tie-break 免疫,並在位元精確引擎中花在*事件壓制*上。我們的標題層級文獻掃描 (第 3 章)未發現撞題;對抗式外部審查的判定是「可推定為原創 (plausibly original)」,我們 也就以這個標籤呈現它,而不是更強的標籤。

5.4.4 與 IRSIM 節點尺寸的關係

最接近的先行研究值得單獨一段。IRSIM [3]——學術 VLSI 流程裡的主力開關級模擬器——體現了 一種實務:節點被指派小整數的*尺寸*(電容等級):預充匯流排可能被指派比一般節點更大的 尺寸,使電荷分享朝它的方向解析;儲存節點與暫態節點被區分開來;而尺寸格刻意保持粗糙以利 解析效率。這確實相鄰:它同樣是 Bryant 譜系中、針對電荷分享結果的電容等級推理。而差異正是 我們主張的重點。第一,推導方向:IRSIM 的尺寸是設計者提供、供解析器消費的註記;我們的 不等式是分類器從網表*推導*出來的,沒有任何註記。第二,事實花在哪裡:IRSIM 的尺寸是 讓一次仍然會發生的解析算得更簡單;我們的支配證明則壓制事件,讓解析根本不發生。第三,保證: 粗糙的尺寸格(刻意地)改變了模型的語意;我們的不等式是在*未變動*語意之內的保守證明, 以與未剪枝引擎的位元精確等價為關卡。我們把 IRSIM 列為最近的鄰居,正是因為把這條界線畫清楚, 是一個原創性主張欠讀者的交代。

5.5 零組態分類遍

四個剪枝全部由同一個載入期分類遍啟用,而這個遍體現了一個值得明說的方法論立場:**每一項安全事實都從組合系統的物理——連通性、旗標、電容與 handler 介面——推導而來,絕不來自名字、清單或 profile**。分類器裡沒有任何部分知道什麼是 OAM 細胞、什麼是狀態暫存器;同一份程式碼不需逐晶片組態就能分類 2A03、2C02 與主機板 TTL,而且一字不改即可分類這個家族格式 [15] [16] 中的任何其他網表。這件事的意義不止於優雅。基於名字的遮罩在網表已知的命名不規則面前是脆弱 的;基於 profile 的遮罩從構造上就不健全(一個從未操練到某節點的工作負載,對該節點什麼也 證明不了——引擎後來在「profile 事實 vs 推導事實」上的親身經歷,第 7 章,印證了這一點); 而由物理推導的遮罩,其可信度恰好等於背後的結構論證——那是位元精確契約唯一接受的貨幣。

這個遍在每次 `Reset()` 的尾端、任何 settle 之前執行,對第 4 章的平坦化陣列走 三個步驟(冷程式碼;在以秒計的載入裡,其成本以微秒計):

```
// 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
```

三點工程備註。第一,這兩項事實被合併成每節點單一個位元打包的位元組(bit 0 為 turn-on-unsafe,bit 1 為 turn-off-skip),讓熱路徑只讀一個陣列而非兩個,未來的剪枝是加一個 位元而不是加一個陣列;光是這次合併就實測 +0.64%(實測)——在這顆引擎上,連熱陣列的 *數量*都是可量測的量。第二,最後一行把供電護欄摺進同一機制:把 V_{CC} 與接地 標為 turn-off-skip,涵蓋掉入列走訪中歷史上明寫的供電測試,使其兩條展開的腿得以統一(其 commit 當時 +1.4–2.1%,實測)。第三,遮罩作為陣列最終徹底從熱路徑消失:第 7 章描述類別為 主鍵重編號如何讓每個剪枝類別成為連續的 id 區塊,把每次遮罩讀取變成對三個邊界 (A=460, S=1275, B=7532)的暫存器比較;每次 reset 都重新計算遮罩並留作基準真相,用以驗證 這些區間——若驗證失敗,則有一個安全退化回退(剪枝關閉、正確性保留),其精神即去優化防護 (deoptimization guard)[13]。區塊母體直接給出普查:457 個節點為 skip-and-unsafe,815 個為 skip-and-safe——合計即 P-2 的 1,272——另有 6,257 個為 no-skip-and-safe;總共 2,258 個節點 帶有剪枝位元,而活節點母體約有一半是可導通剪枝的。

5.6 實測效果

有兩個量測框架適用,我們兩者都報告,因為它們回答的是不同的問題。上線當時,每個剪枝都 個別以交錯成對(interleaved-paired)A/B 對其直接前身把關(協定見第 8 章):**P-1 +11.85%; P-2 +1.4%;P-3 +5.96%;P-4 +1.71%;遮罩合併 +0.64%**(皆為實測, 皆在各自 commit 時位元精確)。回溯來看,第 8 章的九輪同日消融階梯重建了歷史 commit 並在 單一協定下量測;在那裡,含 P-1 的階段較其前一階段跳升 **+26.4%**,含 P-2/P-3/P-4 加合併的階段跳升 **+7.2%**(實測)。這兩個框架在數字上不必 一致——階梯各階段是累積的 commit,因此把每個剪枝與歷史上同期落地、彼此無關的微小工作捆在 一起,而且兩組量測之間隔著數日的引擎演進與一個 .NET 版本——我們也誠實標示這種捆綁:+26.4% 的階差是 P-1 在階梯條件下孤立效果的上界,而 +11.85% 才是當時的孤立量測值。無論採哪個框架, 剪枝家族都是引擎累計 +94.6%(第 8 章)中最大的單一貢獻者。

表 5.2. 剪枝家族:條件、母體與上線當時增益(交錯成對中位數, 位元精確;實測)。

剪枝	壓制對象	靜態條件(逐節點)	母體	增益
P-1	端點狀態相等時的導通入列	若且唯若具上拉、且通道元件不含 ForceCompute 才視為安全	約一半活節點為安全	+11.85%
P-2	整個關斷入列	單一通道、無供電通道、無上拉/FC/callback、非 handler 驅動	1,272 個節點	+1.4%
P-3	(把 P-1 延伸到無上拉節點)	無旗標、無供電,cap 嚴格小於唯一鄰居的	986 個節點解除汙染	+5.96%
P-4	(多通道推廣)	同 P-3,但 cap 嚴格小於*每一個*鄰居的		+1.71%

在事件層級,效果就是直接刪除。比較 P-2 之前與 P-4 之後緊鄰時點的剖析器普查 (300,000 半週期, `full_palette`):總彈出從 159.6M 降到 126.1M——**刪除了 33.6M 次重算,約佔全部彈出的 21%**——無變化占比從 84.4% 降到 80.3%(實測)。組成的變化 一如理論預測:剪枝收割了浮接類別與入列端冗餘,殘餘逐步被 PullUp 與 Supply 類別主宰—— 5.7 節將顯示那是結構性的。注意百分比遞減的算術:只刪無變化彈出,即使移除了五分之一的全部 工作,無變化*占比*也下降得很慢,因為分母同時在縮小——誠實的讀表方式是看絕對彈出數, 而不是浪費百分比。

表 5.3. 剪枝各階段在事件與指令層級的效果(實測:剖析器於 300k hc;ETW PMC 於 1M hc;階梯階段 S1 = P-1 之前, S2 = + P-1,S3 = + P-2/3/4 + 合併)。

指標	P-1 之前(S1)	+ P-1(S2)	+ P-2/3/4(S3)
RecalcNode 彈出次數 / 300k hc	159.6M(P-2 前普查)		126.1M
彈出之無變化占比	84.4%		80.3%
已執行指令(B / 1M hc)	128.7	116.0	105.7
IPC	2.08	2.38	2.35
分支誤判(MPKI)	3.37	2.74	2.52
L1d miss(MPKI)	30.6	24.5	25.8

硬體計數器(完整階梯見第 8 章)印證了機制主張。剪枝刪的是*指令*,不只是抽象的事件: 單單兩個剪枝階段,已執行指令就下降 18%,朝整個階梯的 −33%(每百萬半週期 155.6B 到 104.4B,S0 到 S7)前進,而 IPC 維持在 2.1–2.4 區間——引擎是按比例少做工作,而不是把同樣的 工作做得更快。分支誤判在剪枝階段改善了約四分之一(每個被壓制的事件,同時也是一串難以預測 的派發分支被壓制),而 S2 的每指令 L1d miss 率在總 miss 數下降的同時也*改善了*—— 證據顯示被刪掉的彈出在記憶體行為上並不比倖存者便宜。72 趟階梯跑分中的每一趟、以及上線當時 的每一輪 A/B,都通過了全狀態 checksum 關卡。

5.7 剪枝所不能及之處

在 P-1 到 P-4 之後,倖存彈出中仍有 80.1% 重算出未變的值(表 5.1),殘餘由 PullUp(42.0%) 與 Supply(38.1%)類別主宰。我們以劃定這個家族的邊界為本章作結:就證據而言,這份殘餘無法 以靜態手段在結構上再削減——不是「我們還沒找到那個剪枝」,而是「那裡沒有可找的靜態事實」。 這個主張立足於三項獨立佐證,以不同方法、在不同時間取得。

- **經對抗式審查的機制論證**。PullUp 類的彈出,是在某條下拉閘極翻轉後重算一個上拉節點; 值會不會變,取決於該節點*其餘*下拉閘極狀態的 OR——那是即時狀態,逐事件不同,沒有 逐節點不變量。任何預測器都必須讀那些閘極狀態,而那正是 O(1) 快速路徑已在執行、且即為其 全部成本的掃描。因此壓制這次彈出,就得先做這次彈出的工作——一筆自我挫敗的交易。我們把 這份分析連同專案累積的死路帳本,提交給刻意對抗式的外部審查,並指示對方找出漏洞;審查同意: 這個類別不存在便宜的預測器。
- **窮盡的靜態普查**。若殘餘中存在任何可靜態分類的子類,它會以分類器語彙下的某個母體 現身。我們建了診斷工具(P-2b 普查,把 P-2 的條件向單通道*上拉*節點延伸),量得候選 母體僅佔彈出的 **0.04%**(實測)——比最小的已上線剪枝回報低三個數量級,與「不存在 可利用結構」一致。這個剪枝光憑母體數字就被關閉,未曾實作。
- **動態事實實驗**。殘餘*可以*動態預測——做法是為每個節點維護其當前導通路徑的 事實——第 9 章詳述那兩次完整的嘗試。維護形態(P-5)實測毛利 +13.76%、淨值 −6.84%:約佔執行時間 7% 的維護地板超過了戰利品,正是 5.2.2 節殺死 prune-merge 的 同一套經濟學。零維護復活版(P-5z)把殘餘劈成兩條腿:一條腿的事實可在壓制現場免費導出—— 健全、位元精確、實測效能中性(+0.16%/−0.51%)——而值錢的上拉腿,其所需事實可證明 綁定於上一次解析時的值:改

讀即時狀態就會發散(實測),所以健全性要求一份被維護的快照, 又回到維護地板。邊界從兩側同時關上。

> **本章發現**。在重算出未變值的約 84% 節點重算中,能以載入期證明移除的部分已被移除: 四個剪枝、2,258 個被遮罩節點、刪除約 21% 的全部重算,上線當時增益 +11.85%、+1.4%、+5.96% 與 +1.71%,全部位元精確。剩餘的部分,受到讓這個模型值得模擬的同一套 物理保護: 殘餘的值取決於即時導通路徑狀態(組合邏輯的再確認)或解析時的電荷快照(儲存), 兩者都不容許靜態憑證。因此,要在這顆引擎上進一步降低事件數,要嘛付一筆事件攤提不起的維護稅(第 9 章),要嘛整個離開位元精確契約——那就不再是這顆引擎了。剩下的上行空間必須來自別處: 來自更快地解析倖存事件(第 6 章),以及來自記憶體系統本身(第 7 章)。

第 6 章 基數特化派發

本書前文所述的剪枝家族,攻擊的是事件的*數量*:在「重新評估的結果可被證明不變」的事件 進入佇列之前,就把它們刪掉。本章攻擊的是互補的量——每一個倖存事件的成本。在引擎承襲而來的 形式中——經由 MetalNES [17] 而來的 Visual 6502 `chipsim.js` [15]——每一次彈出(pop) 都要支付完整的一般性代價:清除上一次走訪的暫存(scratch)狀態、以目前 ON 的 pass 電晶體為通道 種下一次廣度優先搜尋、對發現的成員累積旗標 OR、經優先序表解析、再把結果寫回每一個成員。這份 一般性是語意所要求的——語意必須處理任意大小與形狀的導通群組;但它不是資料所要求的。在自晶粒 擷取的 NMOS 網表上,導通群組壓倒性地微小,而且其大小往往*可證明*——有時是靜態的、在載入期; 有時是動態的、在事件觸發當下只需讀幾個位元組。基數特化派發就是對這些證明的系統性利用:載入期 把每個節點分到一個派發類別;執行期先證明 active 的通道相連元件(CCC)大小恰為一或恰為二、再 就地解析;證明一失敗,立即退回未經修改的一般走訪。

我們先把學術定位講明,因為本章是全書新穎性最低的材料,而誠實要求在此具有約束力。認識到「沒有任何 pass 通道的節點構成永久的單例 CCC」一事,隱含於 Bryant 的開關級劃分 [1],並在 COSMOS 中被明確化——COSMOS 乾脆把這類元件直接編譯成布林程式碼 [2]。動態判定*目前 active* 的子網路,則是 IRSIM 的創始想法 [3]。我們新增的是一個實作架構——把基數測試提升(hoist)進 派發決策,讓一般機制在常見情形下根本不被架設——連同對位元精確義務的仔細列舉,使大小 2 的 就地路徑成為保語意的改寫而非近似。實測收益足以支撐本章的篇幅:動態單例路徑是第 8 章九階段消融階梯中最大的單一階(+18.7%),成對路徑是最後一階(+7.0%)。

6.1 觀察:群組大小分布與派發類別

特化的理由是經驗性的,所以我們從實測的工作形狀開始。本節所有數字皆為 DEBUG 組建剖析器的 事件計數;DEBUG 組建執行與 Release 完全相同的演算法,因此產生完全相同的事件母體;工作負載為 `full_palette.nes`,跑 100,000 個主時脈半週期(hc)。引擎在該視窗內執行約 42.85M 次彈出——約每半週期 418 次——其中一次*彈出*(pop)指從目前的安定波(settle wave)清單 取出一個節點並重新評估。這些彈出之中,60.4% 由 6.2 與 6.3 節描述的快速路徑解析、完全不觸及 群走訪機制;39.5%(16.94M 次)進入真正的群走訪。走訪本身極小:其中 77.1% 建出恰好兩個節點的 群組,16% 達到三個節點,只有 5.7% 達到四個以上。BFS 前緣幾乎完全不前進——平均 BFS 深度 (從種子經 ON 電晶體到最遠成員的跳數)為 1.13,第 99 百分位是 3,觀測到的最大值是 14。若以 整體事件流而非走訪為分母,光是大小 2 的群組就佔了*所有*彈出的 30.5%。

這個分布是矽的直接印記。1980 年代的 NMOS 晶片絕大多數是空乏型負載(depletion-load)的閘輸出——節點由負載元件上拉、經增強型電晶體下拉到地——這類節點沒有任何通往其他訊號節點的 pass 電晶體通道,因此永遠不可能與其他節點共享一個 CCC。確實存在的 pass 電晶體結構(進入閂鎖的 傳輸閘、暫存器檔的存取電晶體、匯流排多工器)大多只有一個電晶體深:一個儲存節點藏在單一存取 元件之後、一條匯流排腿藏在單一驅動器之後。長導通鏈確實存在——深度 14 的走訪是真實的,而寬 匯流排群組正是引擎必須做對的結構——但它們是罕見事件,不是常態。其工程後果——這個結論在本書 的負結果中一再以判決的姿態出現——是:任何把每次走訪的額外負擔攤提在大走訪上的方案, 都是在跟資料對賭:位元平行 BFS 實驗(慢 156×,第 9 章)正是因此失敗。有利可圖的方向是 相反的——把大小一與二做到語意所允許的範圍內盡可能接近免費。

派發架構是每節點一個位元組的分類 `cls`,在每次 `Reset()` 時與剪枝 遮罩一併計算一次,並在每次彈出的開頭被讀取。表 6.1 給出類別定義與其實測母體。

表 6.1. 派發類別、其載入期定義與實測母體(full_palette.nes; 節點占比以存活節點為分母,彈出占比以每 100k hc 的全部彈出為分母)。

類別	定義(載入期)	節點數	執行期處理
cls1 —— 靜態單例	沒有通往任何訊號節點的 pass 通道;不帶 {callback, ForceCompute, supply} 任一旗標	3,929(26.7%)	O(1) LUT 解析 (6.2 節),無條件執行
cls2 —— 動態單例候選	具有 ≥1 條 pass 通道;不帶上述任一排除旗標	10,784(73.2%)	執行期基數測試 (6.3–6.4 節);失敗則退回 BFS
cls0 —— 一般	帶 callback 或 ForceCompute 旗標,或屬供電 解析	~16(≈0.1%;≈0.4% 的彈出)	一律走一般群走訪

表 6.2. 每 100k hc 的實測事件層級工作形狀 (DEBUG 剖析器;事件計數與 Release 相同)。

量	實測值
總彈出數	≈42.85M(≈每 hc 418 次)
不經群走訪即解析的彈出	60.4%
進入群走訪的彈出	16.94M(39.5%)
走訪大小分布	大小 2:77.1%;大小 3:16%;大小 4+:5.7%
大小 2 走訪佔全部彈出之比	30.5%
BFS 深度(自種子起算跳數)	平均 1.13;p99 = 3;最大 = 14
無變化彈出(所有成因)	80.1%(結構性 PullUp/Supply 殘餘 42.0% + 38.1%)

關於表 6.1 有兩點說明。第一,類別母體解釋了這個家族為何重要:全晶片將近四分之三的節點是 cls2 候選,而且——因為任一瞬間絕大多數 pass 電晶體的閘極都是 OFF,這個事實已可從 80.1% 的無變化率看出——候選節點的執行期測試成功遠多於失敗。第二,cls0 小到幾乎不存在。十六個一般路徑 節點是行為式記憶體的 callback 錨節點與八個 ForceCompute 匯流排節點;它們合計 0.4% 的彈出占比, 正是每一次想特化*它們*的嘗試都死於母體匱乏(population starvation)的原因 (6.6 節)。

6.2 cls1:靜態單例

靜態單例類別是已知技術,我們也如實歸功。一個節點的電晶體鄰接關係若不含任何通往其他訊號節點的通道——觸及它的每一顆電晶體要嘛是閘極連接(閘極連接永遠不會合併 CCC)、要嘛是直通供電 軌的通道——則僅憑網表結構本身,它就永久構成恰好一個節點的 CCC。這是 Bryant 開關級

劃分中最 平凡的一層 [1],而 COSMOS 走得比我們更遠,把這類元件整個編譯出模擬器、變成層級化的布林程式碼 [2]。我們刻意不編譯(本專案的編譯後端一律更慢,其指令快取原因在第 9 章量化);我們讓單例 保持直譯,但把它的評估壓到資訊理論下限。

對單例群組 $\{n\}$,一般解析——對成員做旗標 OR、再查 256 項優先序 LUT——塌縮為如下: 把節點自身的旗標位元組(上拉,加上任何執行期外部驅動 SetHigh/SetLow 旗標)與一個 Gnd 位元 (若它任一顆通往 GND 的電晶體為 ON)及一個 Pwr 位元(若任一顆通往 VCC 的電晶體為 ON)做 OR; 查 LUT;存回。閘極狀態以一個對節點行內 payload 的短 OR-all 迴圈讀取——由於狀態是 0/1 位元組, 構造上即無分支——整個評估只觸及一條 `NodeInfo` 快取行、寥寥幾個閘極狀態位元組與 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.
```

實作的兩個細節承載了精確性論證。第一,旗標位元組每次呼叫都重新讀取,而不是摺入預先計算的預設值:cls1 節點可以在執行期獲得與卸下外部驅動旗標(時脈腳與 handler 驅動的腳就會如此), 而讀取即時旗標讓這些旗標恰如其經一般路徑那樣搭上 LUT 的優先序。第二,類別資格*不*要求 上拉。最初的分類有此要求,直覺是上拉保證有驅動的解析;但純浮接的單例依一般路徑自己的浮接規則,解析為其最大電容成員的前一狀態——對單節點群組而言,那個成員就是節點本身。單例上的「保持前值」 在結構上是 no-op,快速路徑用「乾脆不存回」重現它。拿掉上拉要求把靜態涵蓋率從存活節點的 23.1% 擴大到 26.7%,並驗證 checksum 完全一致。我們特別記下這個模式,因為它一再出現:精確性論證永遠 是對照一般路徑*在特化母體上的行為*,而不是對照某個理想化模型;而好幾個百分點的涵蓋率, 一再藏在「充分但非必要」的條件後面。

6.3 R-1:動態單例

cls1 用盡了僅憑結構可證明的部分。其餘 73.2% 的節點至少有一條 pass 通道,所以它們的 CCC *有可能*長大——但*此刻*會不會長大,只取決於控制那些通道的閘極目前的狀態,而那只是 幾個位元組讀取之遙。R-1 的觀察是:BFS 的第一步,若在 BFS 的任何架設成本付出之前先做,對最常見 的情形就是一個完整的基數證明。

性質 6.1(動態單例)。 設 n 為 cls2 節點,且在評估當下,每一顆把 n 的通道連到其他訊號節點的電晶體,其閘極皆為 OFF。則 n 的導通群組恰為 $\{n\}$,且 `ResolveSingleton(n)` 的就地解析對引擎全部 狀態的效果,與一般路徑 `ComputeNodeGroup(n)` 加上成員寫回,逐位元組相同。

證明很短,因為走訪自己的定義就完成了工作。BFS 只會從已被納入的成員出發、經由「閘極目前為 ON 的通道」納入新成員。種子的所有通道皆 OFF,前緣就離不開種子;群組即 $\{n\}$。(網表的 通道端點 c_1/c_2 在 NMOS 中可互換,且每節點的通道清單涵蓋 兩個方向,所以不存在不對稱的逃逸。)至於位元相同的主張,我們列舉一般路徑原本會做的事。暫存 狀態清除只觸及上一個群組的成員,不留任何可觀察痕跡。納入種子會清掉它的待彈出去重旗標——而 彈出迴圈本來就會為正被彈出的節點清掉它。對單成員群組的旗標 OR,就是種子自己的旗標位元組加上 對種子供電閘極的 Gnd/Pwr 掃描——恰好就是 `ResolveSingleton`。浮接情形保持前值, 而 6.2 節已說明這對單例是 no-op。成員寫回是單獨一次 `SetNodeState(n, v)`,兩種形式 完全相同;候選類別又在載入期排除了 callback 與 ForceCompute 節點,所以一般路徑的 callback 入列收尾在這個母體上必然不觸發。所有可觀察後果完全重合。

執行期測試失敗——某條通道的閘極為 ON——時,以「未改動任何狀態」之姿跳往一般走訪,所以失敗的代價只有掃描本身。掃描從節點的行內 payload 讀取閘極 id 與狀態,常見情形下只有一條快取行; 電晶體鄰接關係溢出行內 payload 的節點,則退回對共享電晶體陣列的帶終止符清單掃描。一個小但有 量測的工程注記:把這個派發用直接 `goto` 攤平進回退路徑、而不是累積一個布林值再合併 控制流,值約 +0.5%(四個交錯成對批次,位元精確)——在這種每事件成本量級下,JIT 出口程式碼的 形狀在牆鐘時間裡看得見。

```
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 是消融階梯中最大的單一階:中位數吞吐 +18.7%(67,955 到 80,657 hc/s,階段 S0→S1, 九輪輪替,所有跑分皆過 checksum 閘門)。效能計數器毫不含糊地把機制指認為*工作刪除*而非 同樣工作上的效率提升:每 1M 半週期的已執行指令從 155.6 B 降到 128.7 B(−17.3%),IPC 基本持平 (2.11 到 2.08),而每千指令的 L1d miss 率反而從 27.3 *升*到 30.6——被刪掉的 BFS 簿記是 管線化良好、無 miss 的工作,所以殘餘指令流的 miss 密度略增。牆鐘收益幾乎恰好跟著指令刪除走, 這正是當被消除的工作本身並未停滯時所應預期的結果。

我們誠實地界定歸功。IRSIM 的核心想法是 active 子網路在事件當下動態決定、而非由靜態劃分固定 [3];在這個譜系裡,R-1 沒有貢獻新模型——它是一個*早退執行過濾器*(early-out execution filter):把走訪的第一次前緣擴張提升進派發決策,使得在「答案已經確定」的 60% 彈出 情形下,走訪的架設(暫存清除、群組緩衝、旗標累加器)根本不被建立。主張的內容是一個實作架構 及其實測後果,僅此而已。

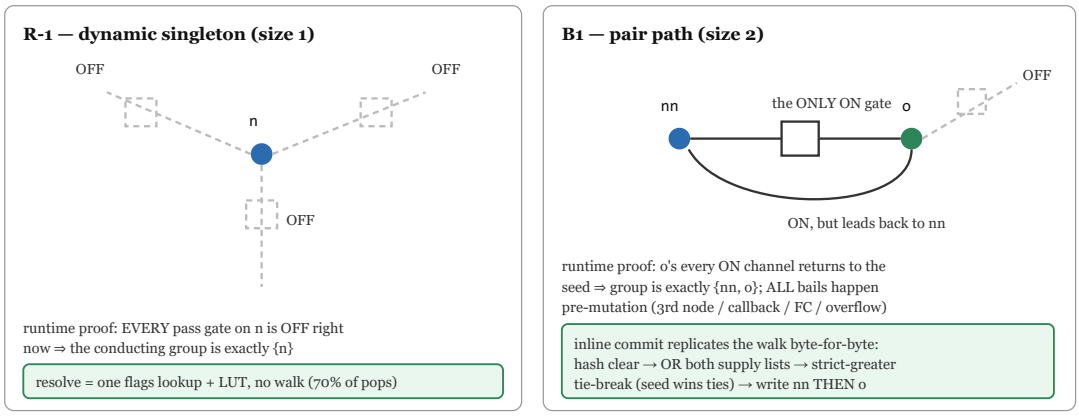

圖 6.1. 兩條基數特化路徑的執行期證明。左:R-1 動態單例(6.3 節)——所有 pass 閘極此刻皆 OFF,元件必為 {n};右:B1 成對路徑(6.4 節)——唯一導通閘極通向 o,且 o 的所有導通通道都折返種子,元件必為 {nn, o};所有放棄條件都在任何變異之前檢查,就地解析逐位元組複製走訪的寫入。

6.4 B1:成對路徑

R-1 之後,單例母體已被處理,而表 6.2 說只剩恰好一個類別大到值得處理:大小 2 的群組,佔 倖存走訪的 77.1%、全部彈出的 30.5%。這些就是晶片的存取電晶體在工作——一條 ON 通道把被彈出的 節點連到一個鄰居。一般走訪用全套裝置處理這樣的群組:暫存清除、納入種子、前緣擴張(發現一個 鄰居然後終止)、旗標累積、值解析、雙成員寫回、callback 掃尾。B1 在 cls2 掃描中發現第一個 ON 閘極的那一點就地解析它——但必須先*證明*群組恰為 $\{n, o\}$ 這一對,且每一條 證明義務都在改動任何引擎狀態之前履行完畢。

> **性質 6.2(成對)**。 設 n 為 cls2 節點, 其通道掃描發現一個通往鄰居 o 的 ON 閘極,並假設:(i) $o \neq n$; (ii) n 沒有其他通道閘極為 ON;(iii) o 為行內駐留(inline-resident),且不帶 callback 與 ForceCompute 任一旗標;(iv) o 的每一條 ON 通道都通回 n。則 n 的導通群組恰為 $\{n, o\}$,且下方的就地解析對引擎全部狀態的效果——包括待彈出去重 hash、值、寫入順序與下一波的入列順序——與 `ComputeNodeGroup(n)` 加上成員寫回, 逐位元組相同。

證明中基數的部分與性質 6.1 同構:從 n 出發的 BFS 前緣只能跨越 n 唯一的 ON 通道、抵達 o;從 o 出發只能跨越 o 的 ON 通道,而依 (iv) 它們全部回到已被 納入的 n。群組在 $\{n, o\}$ 封閉。然而 B1 的實質內容並不是這個觀察——而是 一般走訪在計算值之*外*還做的那一串事,每一件都可觀察、每一件就地路徑都必須一模一樣地 複製。我們逐條列舉,因為它們所代表的工程紀律——把舊路徑的副作用當成規格,逐位元組對待—— 正是本章主要的方法論內容。

1. **改動前回退**。 性質 6.2 的全部四個條件都在寫入任何東西之前測試。自通道 ($o = n$)回退;種子上的第二個 ON 閘極回退——即使那個閘極的通道通往*同一個* 鄰居 o:在這種情形下群組仍會是這一對,但證明就得處理並聯通道,我們選擇了較便宜、保守的 測試;溢出或帶 callback/ForceCompute 的鄰居回退;o 通往任何第三節點的 ON 通道回退。 由於最後一個測試之前沒有任何改動,回退後的一般走訪在未受擾動的狀態上重新執行,構造上即精確。 這種「先防護、再提交」的形狀,與去優化防護(deoptimization guard)[13] 是同一種紀律:特化路徑 必須證明自身的適用性,並無損地讓位。

2. **成員去重 hash 清除**。 BFS 納入一個成員時,會清除該成員的待彈出旗標,取消*同*一波中稍後對它排定的任何重新評估——它的值剛剛已作為這個群組的一部分被解析。就地路徑必須執行 完全相同的 `dedupHash[o] ← 0`。這個清除攸關正確性(load-bearing):漏掉它會讓 o 在同一波內再次彈出、在 n 的寫入之後重新解析,而這在測試中造成了發散。這種 副作用在以值為中心的演算法解讀裡看不見,在位元精確的解讀裡卻是致命的。
3. **雙節點供電掃描**。 群組的旗標字組是兩個成員旗標位元組的 OR,再加上一個 Gnd 位元 (若*任*一成員的任一顆通往 GND 的電晶體為 ON),Pwr 亦然。OR 累積與順序無關,所以就地 路徑對兩份供電清單的直線式 OR-all,即使造訪順序不同,也產生與走訪那種提早跳出式累積相同的 字組。
4. **一模一樣的浮接 tie-break**。 若旗標字組為零,這一對是純浮接,而一般規則——最大電容 成員的前一狀態,以*嚴格*比較、平手由先見者勝——必須以其精確形式重現。對這一對而言即 `cap[o] > cap[n] ? state[o] : state[n]`:嚴格大於、平手種子勝,恰好就是 `GetNodeValue` 對群組緩衝 [n, o] 迭代的結果。我們如本書他處一樣強調: 這個 tie-break 不是數值細節——它是晶片的儲存機制,把 > 換成 ≥ 就是另一台機器。
5. **寫入順序**。 走訪按群組緩衝順序寫成員:先種子、再鄰居。每次 `SetNodeState` 會把以該節點為閘極的相依節點附加到*下*一波的清單,而波內彈出順序是 Gauss-Seidel、屬可觀察語意(第 2 章)。因此就地路徑先寫 `SetNodeState(n, v)` 再寫 `SetNodeState(o, v)` ——同樣的順序,因而同樣的下一波入列順序,因而同樣的下游彈出 順序。
6. **callback 與 ForceCompute 排除**。 種子的 cls2 分類已在 n 上排除這兩個旗標; 回退 (iii) 在 o 上排除它們。於是一般路徑的 callback 入列收尾在被接納的母體上必然不 觸發,而 ForceCompute 的 Gnd+Pwr 互消——LUT 確實有編碼,但它與群組成員資格的交互方式,我們 不願在就地路徑重新驗證——永遠不會出現。保守排除比證明便宜。

```
// 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)
```

驗證走過本專案維護的每一道閘門:DEBUG 事件計數檢查——總彈出數維持在每 100k hc 42,848,807 不變,確認 B1 是「事件如何被解析」的純改寫、而非「哪些事件發生」的改變——三個 Release 全狀態 checksum(300k/400k/1M hc)、自我測試網表,以及精靈密集的《Super Mario Bros.》 1000 萬半週期 checksum 閘門。保守回退依設計留下了未收的涵蓋率:在走訪為大

小 2 的 30.5% 彈出中, 成對路徑每 100k hc 捕捉 8.34M 次彈出(全部彈出的 19.5%);其餘在條件 (ii)–(iv) 上回退,照舊 走一般走訪。我們沒有追擊殘餘:每一項放寬(並聯通道、溢出鄰居、ForceCompute 成對)都為一小片 母體增加證明面,而此時家族的否決準則(6.6 節)已然定型。

6.5 實測效果

表 6.3 彙整派發家族的實測效果。R-1 的 +18.7% 已於 6.3 節討論。B1 以 S7 階段進入消融階梯: 中位數吞吐較 S6 +7.0%(123,545 到 132,243 hc/s,同日交錯;最佳一跑 135,828);與緊鄰的 整理性 commit 分離歸因後約 +8.9%。量測一律不鎖頻,以同日多輪 round-robin 交錯、單一 benchmark 進程、中位數回報控制變異。

表 6.3. 派發家族的實測效果(中位數;消融階梯 9 輪 × 8 階段,400k hc,checksum 閘門;計數器來自 1M-hc 的 PMC 量測遍)。

效果	量測
R-1(階梯 S0→S1)	+18.7%(67,955 → 80,657 hc/s)
R-1 指令刪除	155.6 → 128.7 B 指令/1M hc(−17.3%);IPC 2.11 → 2.08
B1(階梯 S6→S7)	+7.0%(123,545 → 132,243 hc/s);分離歸因後約 +8.9%
B1 指令刪除	111.3 → 104.4 B 指令/1M hc(−6.2%);IPC 2.41 → 2.34
B1 事件涵蓋	每 100k hc 8.34M 次彈出(19.5% 的彈出)就地解析;總彈出數不變

S7 的計數器剖面以較小的尺度重複 R-1 的簽名:已執行指令下降 6.2%,IPC 微降,L1d miss 率 持平(12.5 到 13.0 MPKI)——再一次是「管線化良好機制的刪除」,不是快取效應。但 B1 刪掉的 指令並非均勻分布在管線的餘裕上:被移除工作中相當大的一部分——群組緩衝維護、暫存清除、間接成員迭代——坐落在第 8 章指認為引擎真正限制資源的*相依載入鏈*上。

有一個框架上的區分值得講清楚。前一章的剪枝降低事件數:彈出乾脆不發生。派發不動事件流 —— 彈出數不變性檢查正是 B1 驗證的一部分——而是降低每事件的常數。因此兩個家族乘法式地疊加、不 爭奪同一份獎金,但有一個微妙的交互:剪枝刪掉的是最便宜、最可預測的彈出,因而*抬高*了 倖存者的平均成本——這正是派發家族較晚的步驟,在一個已從剪枝吸收 +60% 的引擎上仍然有利可圖 的部分原因。

6.6 否決清單(Kill List)

派發家族在 B1 之後封閉,而我們以與「採納」相同的證據標準記錄這次封閉,因為這個家族大部分的探索其實發生在負空間。下列每一個候選,要嘛在真引擎上實測,要嘛被一個本身立足於實測母體的 分析論證否決。表 6.4 摘要;其後對兩個帶有一般性教訓的候選展開敘述。

表 6.4. 以實測母體否決的派發家族候選(full_palette.nes, 100k hc;「—」= 在值得動工之前即以分析否決)。

候選	實測母體	判決
cls3:callback 單例快速路徑	6 個節點;67,968 次彈出 = 0.16% 的彈出	結構上健全(已驗證 callback 假電晶體確實落在供電桶中),但母體已死;這些彈出 100% 是 無變化——值恆為 0,唯一的效果是 callback 入列。
把 ForceCompute 納入 cls2	8 個節點;101,120 次彈出 = 0.24% 的彈出	原則上位元精確(LUT 已編碼 FC 的 Gnd+Pwr 互消),但八個全是高扇出的溢出匯流排節點,其 通道通常為 ON——反正都會落入 BFS。
在去重 hash 位元組中夾帶入列時派發提示	—	在有利可圖的方向上不健全:同一波中較晚的彈出可以在提示凍結之後把某個閘極切成 ON,於是 過期的「快速」提示算出錯值。安全的方向只會增加 BFS 落入;維護提示的真確性正是計數器式快速 路徑那條死路(實測為負)。
BFS 落入時重用部分掃描	第一個 ON 閘極索引平均 < 2	「省下」的工作是重讀 L1 常駐、餵給一個預測良好分支的快取行——不在攜帶相依鏈上;沒有 收益可拿。
以 epoch 清除去重 hash	—	8 位元 epoch 每約 21 次 settle 呼叫即繞回,產生假的「已入列」判定——發散;它想省下的 儲存,一個可能的解釋是本就被 store buffer 隱藏(未單獨驗證)。
按類別分佇列 / 波內重排 / 批次彈出	—	波內順序是 Gauss-Seidel、屬可觀察語意;任何重排依規格而言就是另一台模擬器,不是 優化。
波內局部群組記憶化 (memoization)	BFS 深度 p50 = 1	群組成員資格是即時閘極狀態的函數;不存在可供攤提的「重複大走訪」母體,而有效性追蹤正是 「維護執行期事實」的地板(P-5 的教訓,第 9 章)。
無分支、容忍重複項的入列	入列分支約 97.7% 單邊	一個以 97.7% 被預測的分支,移除它沒有任何成本可省;重複項還會破壞波清單的容量 保證。

其中兩項值得展開。*入列時提示*的想法——當某次閘極變化把一個節點入列時,順手記下這次 變化是導通還是關斷,讓之後的彈出可以省掉自己的重新掃描——失敗的原因可以一般化:在 Gauss-Seidel 波中,入列時記下的資訊是一張快照,而快照與消費它的彈出之間執行的任何彈出都可能 使其失效。提示的有利可圖方向(信任它、跳過工作)因此恰好就是不健全的方向。這與否決 P-5 dominant-bypass 家族(第 9 章)的「即時狀態 vs 快照」區分是同一件事的縮影;引擎在彈出當下唯一可信的事實,是它在彈出當下重新推導出來的事實,而唯一付得起的重新推導,就是 6.3–6.4 節 那些 O(1) 證明。

*波內重排*這一條的直白值得保留。按類別排序彈出、批次處理單例、延後走訪,在紙上都會 改善區域性與分支一致性——但其中每一種都改變了同一波中較晚的彈出觀察到哪些值,因為在這個模型裡,波內順序就是語意。這裡不存在「大致等價」:浮接 tie-break 與 Gauss-Seidel 可見性規則使波 的順序成為機器定義的一部分。我們在專案早期用唯一可用的方式驗證了這一點——發散——此後這條 規則一直是絕對的。

來自鄰近家族的最後一個候選,強化了本章的篩選準則:把 `cls` 位元組本身編碼為 id 區間比較(就是讓剪枝遮罩變免費的那個變換)量得中性(−0.28%),因為類別載入餵的是一個預測 良好的分支、而非坐在攜帶相依鏈上。連同表 6.4,這固定了此後這顆引擎上所有提案都必須通過的 三條件檢核表:有利可圖的目標必須是*靜態的*(無需執行期真值維護即可證明)、*熱的*(母體以彈出的整數百分比計,而非零頭),且*在攜帶鏈上*(被刪的工作必須承載延遲,而非 管線填充)。R-1 與 B1 三者皆滿足;家族裡其餘一切至少敗在其中一條,以上量測即為證據。

B1 出貨之後,這個家族的終局如下:全部彈出約 80% 不經一般群組機制即解析——60.4% 作為單例、 落在一條約五次相依載入的鏈上,另 19.5% 走就地成對路徑——cls0 全體佔 0.4% 的彈出,殘餘的一般 走訪則是真正不規則的結構(匯流排、深 pass 鏈),BFS 對它們本來就是正確的工具。引擎剩餘的時間, 屬於結構性的無變化殘餘(80.1% 的彈出,且不存在靜態子集——已以三種獨立方式確認)以及這個已 被最小化的解析本身的相依載入延遲;後者正是重排各章、以及為本書收尾的天花板分析的主題 [21]。

第 7 章 自我捕捉資料重排

前幾章的優化——動態單例快速路徑、P-1 至 P-4 入列剪枝、條件子句重排——全都作用在事件排程上:它們刪除事件驅動引擎原本要執行的工作。本章描述一種不同性質的介入:它完全不動事件排程,改而改寫網表的*識別碼空間*(identifier space)。在載入期,引擎計算一個節點 id 的置換(permutation),並在置換後的編號下重建所有資料結構。這個置換服務兩個不同的目標。第一個是*區間剪枝*(range-prune):把節點排序,使每個剪枝類別佔據一段連續的 id 區塊,將引擎中最熱的查表轉換為一對暫存器比較——消融階梯上 4.2% 的一階。第二個是*自我捕捉初次觸碰鍵*(self-captured first-touch key):在每個區塊之內,依生產安定級聯(settle cascade)實際初次觸碰節點的順序排列節點——這份軌跡是引擎在載入期間從自己身上捕捉到的——再帶來 5.0% 的一階。

本章同時也是一個關於「效能死路如何被索引」的故事。在這些技術上線的一個月前,節點重編號曾在本專案中被量測、被判定毫無價值,並記入死路目錄。那次量測本身沒有任何錯誤;錯的是從中得出的推論——被判死的是那個*工具*,而不是它當時被要求服務的*目標*。7.1 節重建那段經過,因為我們相信,重啟此案的那條診斷鏈,比這些技術本身更具可遷移性。7.2 與 7.3 節描述兩個機制;7.4 節呈現分離出 locality 鍵真正買到什麼(順序,而非密度)的實驗;7.5 節則把這個構造放到 inspector-executor 與軌跡導向佈局文獻 [7, 8, 9, 10, 11, 12] 之中定位。

7.1 捲土重來的死路

7.1.1 經典 RCM,以及它為何誠實地失敗

反向 Cuthill-McKee(Reverse Cuthill-McKee,RCM)重編號降低稀疏圖的頻寬:在圖中相鄰的頂點獲得相近的識別碼,使鄰居的資料共享 cache line。當初促成在此嘗試它的假說很自然。引擎的群走訪——每事件一次、沿 ON pass 電晶體進行、用以界定通道相連元件(CCC)[1] 的 BFS——從一個節點跳到它的通道鄰居;而 `SetNodeState` 中的入列走訪,則從一個閘極節點跳到它所控制之電晶體的通道端點。若這些鄰居住在相同的 cache line 上,走訪的 miss 應當變少。2026 年 5 月的量測結果:開機暫態約 1.04×,穩態約 0.98×——在雜訊之內,等於什麼都沒有。這個判定以「重編號已死」之名進入了專案的死路目錄。

附在那個判定上的診斷當時就已正確,值得精確重述,因為它預言了其後的一切。引擎的熱工作集——1-byte 狀態陣列、目前活躍節點的 16-byte `NodeInfo` 紀錄、波佇列——量級約 15 KB,在穩態模擬期間常駐於 L1 資料快取。RCM 解決的是*容量*問題:它減少一次走訪所觸碰的相異 cache line 數,使其裝得進、或順暢流過一個有限的快取。當走訪的工作集本來就裝得下,就沒有容量問題可解,降低頻寬是在回答一個機器並沒有在問的問題。

7.1.2 三個獨立的確認

一個月後,三個實驗接連進行,從不同方向攻擊引擎的記憶體行為,全數受位元精確協定(golden checksum)把關,並以交錯成對(interleaved-paired)方法量測。其聯合結果摘要於表 7.1。

表 7.1. 2026-06-10 的三個延遲隱藏實驗(C# 引擎,full_palette,交錯成對)。每一個都攻擊記憶體行為;每一個都一無所獲或更糟。全數實測 [21]。

實驗	測試什麼	結果
把波佇列從 `int` 縮小為 `ushort`(串流資料 −60 KB)	串流的佇列流量是否有任何成本	−0.07%(中性)
對接下來幾次彈出(pop)的 `NodeInfo` 紀錄做軟體預取	是否還有可隱藏的記憶體延遲	−0.3% 至 −1.1%
純共活性(co-activity)locality 重編號:每半週期熱集的 cache-line 密度改善 45%,接近裝填理論的理想值	即便近乎完美的 locality 改善是否能撼動牆鐘時間	−0.36%(中性)

第三列值得強調,因為它是這個負結果最強的可能形式。該實驗不只是沒能找到好佈局;它*找到了*一個——一份儀器化的共活性 profile 把每半週期觸碰的相異 `NodeInfo` cache line 數降低了 45%,接近實測活動在裝填理論上的理想值——而這個近乎完美的 locality 改善仍然沒有帶來任何牆鐘收益。locality 作為一個目標,並非優化不足;它被充分優化了,而且一文不值。

> **發現。** 三個獨立實驗確認了同一件事:引擎的每事件成本地板(量測當時每次彈出 ≈20 ns)不是 cache-miss 綁定,而是*相依載入鏈*(dependent-load chain)綁定。每個事件是一條短的串行鏈——載入節點的紀錄、載入它所指向的東西、依結果分支——成本是這條鏈以相依載入計的*長度*,而其中每一個載入個別都是 L1 命中。把位元組搬得更近,改變的是資料住在哪裡;它縮短不了鏈。

7.1.3 目標的翻轉

如果綁定在鏈長,唯一的槓桿就是刪除鏈節。這個重新框定把重編號機具從一個失敗的優化轉化為一個賦能機制,因為有一類鏈節是置換可以直接刪除的:那些唯一用途是回答節點某個*靜態*問題的載入。一個靜態的逐節點事實——載入期定案、終網表一生不變——不需要從一張以節點 id 索引的表中提取;它可以被編碼進節點 id 本身:把節點排序,使該事實為真的所有節點佔據一段連續的 id 區間,查表就變成 id(已在暫存器中,因為迴圈正在迭代 id)與一個常數(同樣在暫存器中)之間的比較。那個載入從鏈上消失;沒有任何東西取而代之。

因此,五月的判定與六月的成功兩者皆正確,而這個表面矛盾的化解,正是本章的核心方法論教訓:**死路以目標索引,不以工具索引**。「重編號已死」對重編號當時被要求服務的目標(locality,在一個沒有 locality 問題的引擎上)是真命題,對一個還沒有人拿來問它的問題(把靜態事實編碼為 id 區間,在一個瓶頸正是這種編碼所刪除之載入的引擎上)則是假命題。對任何受相依鏈綁定、而非受 miss 綁定的引擎,其一般形式是:目前由查表回答的每一個靜態逐元素事實,都是改由元素的*位置*來回答的候選——載入時排序一次,執行期永遠只比較。

7.2 區間剪枝:類別為主鍵重編號

7.2.1 引擎中最熱的表

前一章描述的剪枝家族,實作上是一張逐節點遮罩 `PruneMask`:一個 15 KB 的位元組陣列,帶兩個語意不同的位元——bit 0 標記節點為*導通不安全*(turn-on-unsafe;P-1 的結構性安全汙染(taint)——無上拉節點與 ForceCompute 通道元件,減去被 P-3/P-4 解除汙染(un-taint)的「電容小於所有鄰居」子集);bit 1 標記節點為*關斷跳過*(turn-off-skip;P-2 的度 1 無驅動葉節點類別,加上供電跳過摺疊之後的供電軌)。兩個位元都是靜態的:在 `Reset` 時從網表結構算出一次,模擬期間永不改變。

這張遮罩的消費者是引擎中最熱的單一迴圈:`SetNodeState` 內的入列走訪。每當某節點的狀態翻轉,引擎就迭代該節點所閘控元件的緊湊電晶體列表——一次把四個 `ushort` 條目當成一個未對齊的 `ulong` 讀入——並為每個元件決定是否把其通道端點入列到下一個安定波(settle wave)。在每 100k hc 約 42.85 百萬次彈出(pop)、每次彈出多次端點檢查的規模下,這個決定每模擬秒執行數億次。重編號之前,每次決定都讀一次 `PruneMask[c]`,而端點 id c 從快取的角度看,是對這個 15 KB 陣列的隨機索引。陣列常駐 L1,載入必定命中——但它仍是一個 4–5 週期、餵給條件分支的相依載入,坐在最熱迴圈的關鍵鏈上。按 7.1 節的分析,它正是值得刪除的那種鏈節;事實上,它是這個迴圈所含*唯一*可刪除的鏈節。

7.2.2 四個區塊,兩個比較

兩個剪枝位元定義出四個類別,而類別為主鍵重編號(class-major renumbering)給每個類別分配一段連續的 id 區塊。id 0(保留)、1 與 2(電源與接地軌)被釘住;其餘所有節點以剪枝類別為主鍵排序。在 lowering 後的 NES 網表上,量得的邊界為 $A = 460$、$S = 1275$、$B = 7532$:

```
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
```

區塊順序有兩個承重的細節。第一,兩個 skip 區塊都放在兩個 no-skip 區塊之下,於是關斷跳過測試塌縮為單一比較 $c < S$。由於供電軌佔據 id 1 與 2,位於第一個區塊起點 3 之下,同一個比較也順帶回答了「這個端點是不是供電軌?」——歷史上的顯式守衛 `c2 != ngnd && c2 != npwr` 與其後繼者供電跳過遮罩摺疊,都被這一個比較免費吸收。第二,no-skip ∩ unsafe 區塊放在*最後*,因為重編號之後創建的假 handler 節點(行為式記憶體的 callback 目標)永遠恰屬此類——無上拉,故 unsafe;callback,故 no-skip——因此它們延伸的是開放端的最終區塊,不擾動任何邊界。以量得的邊界計,三個區塊分別容納 457、815 與 6,257 個節點,≈14.7K 節點 id 空間的其餘部分(加上後附的 handler 節點)落在最終區塊。

在每個區塊之內,節點仍需要一個次要排序鍵;既然 locality 大約一文不值(7.1.2 節),類別為主鍵的排序覆蓋掉鄰接關係並無任何犧牲。最初上線的次要鍵是對訊號流的一個盲目靜態近似:從主時脈節點出發、沿入列走訪實際遍歷的邊(從一個節點到它所閘控之電晶體的通道端點)、忽略閘極狀態的 BFS。未被到達的節點按原始相對順序沉到所屬區塊的尾端。7.3 節將以一個量測得到的鍵取代它。

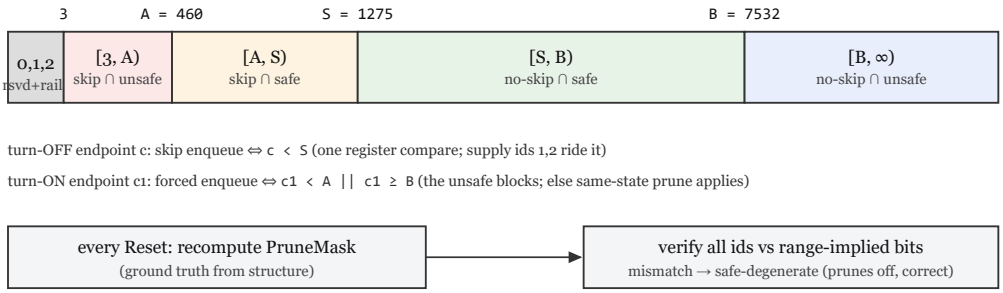

圖 7.1. 類別為主鍵的 id 空間。每個剪枝類別佔據一段連續區塊([3, *A*) skip∩unsafe、[*A*, *S*) skip∩safe、[*S*, *B*) no-skip∩safe、[*B*, ∞) no-skip∩unsafe;量得邊界 *A*=460、*S*=1275、*B*=7532),於是熱入列走訪所消費的靜態剪枝事實,變成對區塊邊界的暫存器比較:關斷跳過 ⇔ $c < S$;導通不安全 ⇔ $c < A \,||\, c \geq B$。供電軌(id 1、2)搭關斷跳過比較的便車;重編號後的 handler 節點附加於最終區塊。

7.2.3 永不信任佈局:驗證與安全退化回退

錯誤的區塊邊界不會讓程式崩潰;它會無聲地錯誤剪枝。失效模式從隱微(壓掉一次有影響的重新評估,以發散的方式汙染狀態)到怪誕(把電源軌當成可入列節點、對 VCC 做群走訪)都有。因此本設計拒絕信任佈局。遮罩計算被原封不動地保留,並在每次 `Reset` 時作為*基準真相*(ground truth)執行:先前用來產生執行期查表的同一套結構分類,如今產生一份參考,在第一個安定波獲准執行之前,逐節點驗證所宣稱的區間:

```
// 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
```

不匹配時,引擎回退到*安全退化*(safe-degenerate)邊界:$S = 3$ 使關斷跳過比較只對供電軌為真(強制性的守衛),而 $A = B = \infty$ 把每個節點都視為導通不安全。這會完全停用 P-1/P-2/P-3/P-4 剪枝——引擎執行更多 no-op 重新評估、跑得更慢——但它在*任何*編號之下都維持位元精確的正確性,包括恆等編號、手工搭建的自測網表,以及任何結構違反排序內建假設的未來網表。這是 Hölzle、Chambers 與 Ungar [13] 的去優化防護(deoptimization guard)模式,從編譯後程式碼移植到資料佈局:只有在一個便宜的執行期檢查認證了優化表示所依賴的不變式時,才使用優化後的表示;檢查一失敗,系統立即退化為較慢、但平凡正確的表示。熱路徑只編譯比較形式——沒有模式分支、沒有旗標;退化情形純粹經由邊界值達成。驗證成功之後,Release 版會釋放此時已死的 15 KB 遮罩陣列;由於未觸碰的記憶體在快取上毫無成本,這是衛生而非速度。

7.2.4 實測效果

在第 8 章的消融階梯中,區間剪枝這一步(S4→S5,commit `51e046d`)把中位數從 112,887 推到 117,671 hc/s,+4.2% 的一階。必須坦白聲明一項保留:階梯的 S5 階段同時把執行環境從 .NET 10 換到 .NET 11,所以這一階把佈局變更與執行環境升級混在一起。上線當時的同執行環境、交錯成對

的隔離量測為 +3.56%(兩次量測活動中 19/20 與 11/12 成對勝)——在當時是 P-1 剪枝本身以來最大的熱路徑勝利。

第 8 章的硬體計數器佐證了這個機制。S4→S5 這一步把 L1d miss 從 26.3 砍到 13.2 MPKI——整條階梯中最大的單一 L1d 變動——而 AMD 原生計數器那一輪顯示,每 1M hc 的 L1d *存取*次數從 S0 的 75.6 billion 降到 S5 的 44.1 billion:這些載入不是 miss 得更少,而是*消失了*,這正是「每次端點檢查刪除一個載入」所預測的。(S5 的已執行指令略升,每 1M hc 104.3→107.8 billion,IPC 從 2.31 降到 2.25——收益在鏈的結構,不在指令數,7.4 節會展開這一點。)同一步的 L1i MPKI 從 0.209 升到 0.284;我們把它歸因於執行環境遷移與程式碼佈局的擾動,而非資料重排,但階梯無法把兩者分開,我們把它標記為一個誠實的混淆因素。

7.2.5 適用性的邊界

7.1.3 節結尾的推廣——每一個靜態逐元素查表都是位置編碼的候選——在清單上的下一個候選身上受到了檢驗,而結果誠實地為這個技術劃界。快速路徑派發在彈出迴圈中讀一個逐節點類別位元組(`IsPureLogic` / cls1 對 cls2 的判別子);既然該類別是靜態的,同一招理應適用,於是次日建造並量測了把類別摺進額外 id 區塊的 cls-range 變體。結果是中性:−0.28%,7/20 成對勝,隨即被撤銷。這個診斷補完了模型,而非反駁它。類別位元組的載入餵的是一個*被預測的分支*——亂序核心會投機越過它,所以該載入的延遲不在關鍵路徑上,刪掉它買不到任何東西。相形之下,`PruneMask` 的載入坐在入列走訪的*攜帶*相依鏈(carried dependence chain)上,那裡每一個週期的延遲都是一個週期的牆鐘時間。由此得出此技術的三重檢核表:事實*必須是靜態的*(否則佈局會失效)、位點*必須是熱的*(否則收益量不出來)、被刪除的載入*必須位於攜帶鏈上*(否則投機早已把它藏起來)。三者同時成立,區間編碼才划算;在這個引擎上,剪枝遮罩是唯一同時滿足三者的表。

7.3 自我捕捉初次觸碰鍵

7.3.1 從靜態猜測到實測順序

7.2.2 節的盲目 clk-BFS 次要鍵,是對安定級聯訪問節點順序的一個靜態猜測。在區間剪枝實驗期間,一個以*實測*共活性 profile 為次要鍵的儀器化變體在早期測試中達到 +5.12%——明顯優於盲目鍵,但在操作上不討喜:它需要從一次執行傾印 profile 檔、再載入到後續執行,帶著離線 profile 所有的陳舊、ROM 不匹配與組態漂移風險 [12, 20]。催生最終形式的問題是:如果引擎能量測自己,為什麼還要有檔案?上線的設計讓引擎成為自己的 profiler。載入時,它把自己建構兩次,並在兩次建構之間捕捉自身生產級聯的真實初次觸碰順序。結構是三遍載入(圖 7.2):

1. **第 0 遍——分類。** 在恆等 id 下建構網表,執行 `Reset`,讓尋常的結構分類算出 `PruneMask`。捕捉每個節點的兩個剪枝類別位元;它們是後續兩遍的主排序鍵。
2. **第 1 遍——重建、暖機、捕捉。** 以暫時性的盲目 clk-BFS 次要鍵做類別為主鍵重建。由於重建後佈局的區間通過 7.2.3 節的 Reset 期驗證,事件剪枝是*開啟*的:被觀察的級聯是真實的、已剪枝的生產排程,不是未剪枝的近似。先讓晶片暖機 1,024 hc 越過重置暫態,再讓 32,768 hc 跑過安定迴圈的一份*冷儀器化副本*,為每個節點記錄其初次彈出的序號。把捕捉到的順序經由第 1 遍置換轉譯回恆等 id。
3. **第 2 遍——最終建構。** 以第 0 遍的類別位元為主鍵、捕捉到的初次觸碰順序為次要鍵重建(從未被觸碰的節點按原始相對順序沉到所屬區塊尾端)。生產執行就在這個佈局下進行。

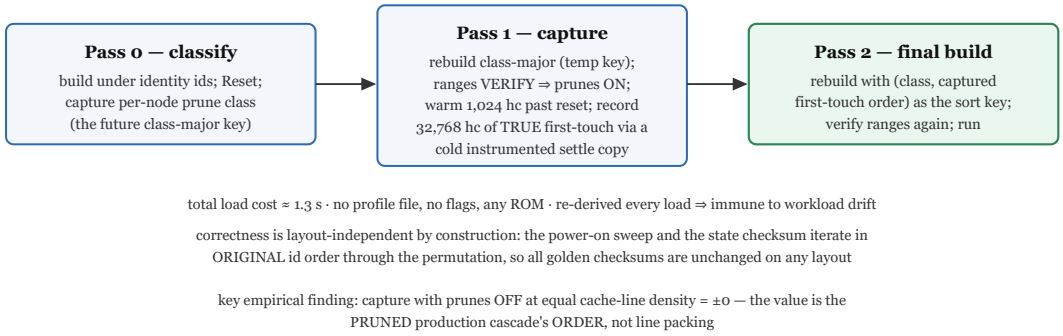

圖 7.2. 三遍自我捕捉載入。第 0 遍在恆等 id 下分類剪枝類別;第 1 遍以暫時性靜態鍵做類別為主鍵重建,暖機 1,024 hc,並經由安定迴圈的冷儀器化副本捕捉 32,768 hc 的真實初次彈出順序(剪枝已驗證且開啟,故觀察到的級聯就是生產排程);第 2 遍以捕捉到的順序作為區塊內鍵重建。總載入成本 ≈1.3 s;最終建構的熱迴圈毫髮無傷。

7.3.2 冷儀器化安定迴圈副本

這個捕捉儀器值得描述,重點在它刻意*不是*什麼。它不是加在 `ProcessQueue` 上的旗標、callback 掛鉤或條件分支——那些都會向最終建構的熱迴圈課稅,來支付一個只在載入期發生的量測。取而代之,捕捉遍驅動晶片跑過安定迴圈的一份獨立、冷的副本,它精確鏡映生產迴圈的雙緩衝波排空與時脈切換,外加一行:

```
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]]
```

這份副本只在第 1 遍期間執行,並隨之拆除;生產版 `ProcessQueue` 逐位元組不受影響。這個重複的失效模式也在構造上有界:即使儀器化副本在語意上偏離了生產迴圈,損害也只限於一個*較差的 locality 鍵*——效能退步,絕不是正確性缺陷——因為類別區塊、區間驗證與全部正確性機具都獨立於次要鍵。

7.3.3 為何正確性在構造上與佈局無關

一個每次佈局變更都得對行為測試 ROM 重新驗證位元精確性的重編號方案,在操作上是站不住的。引擎改為維護一個結構性論證:*任何*置換都產生相同的物理事件序列。

性質 7.1. 模擬的可觀察行為在節點重編號下不變。事件驅動核心恰有一個依賴 id 順序的位點:上電時的 `RecomputeAllNodes` 掃描,它把每個節點入列一次,其入列順序播種第一個安定波的波內(Gauss-Seidel,可觀察)順序。該掃描——如同全狀態 checksum——透過儲存的置換以*原始* id 順序迭代。其後每一個波的順序都由級聯本身決定,而級聯是網表連通性的函數,與節點 id 無關。因此,重編號後的執行重放完全相同的物理事件序列,並重現與恆等編號執行相同的 golden checksum。

這個性質不是被假設的;它是被稽核出來的(對 id 順序相依位點的搜尋,正是讓上電掃描浮現為唯一一個的過程),並且持續被再確認:本章實驗中的每一個佈局變體——盲目鍵、profile 鍵、捕捉鍵、剪枝關閉下的捕捉——都重現了全部三個 golden checksum(300k/400k/1M hc)與精靈密集的 10M-hc SMB1 門。7.2.3 節的驗證加回退,涵蓋了佈局確實承載語意的那一個地方(區間編碼的剪枝事實),把環閉合。

7.3.4 成本、漂移免疫與實測效果

整個三遍載入花費約 1.3 s,只在啟動時一次——兩次網表建構加 33,792 hc 的儀器化暖機,相對於動輒數分鐘到數小時的模擬場次。它所取代的那個反覆發生的替代方案(profile 檔)帶著一種更隱微的成本:陳舊。離線 profile 把佈局綁定到產生它的工作負載、引擎版本與組態;自我捕捉*在構造上*免疫於這種漂移,因為鍵在每次載入時都從實際的晶片、實際的 ROM、實際的引擎二進位重新導出。沒有檔案格式、沒有旗標,也沒有任何方法餵給引擎一份錯誤的 profile,因為 profile 已不再以工件形式存在。

關於捕捉視窗代表性的一句話。在每視訊框 714,732 hc 之下,32,768-hc 的軌跡觀察到的不足單一框的百分之五——它不能、也不企圖取樣工作負載完整的行為多樣性。它捕捉的是*初次觸碰*順序,由早期級聯結構支配:時脈扇出、相位產生器、任何工作負載最初幾千個半週期內就觸發的暫存器與匯流排路徑。視窗從未到達的節點沉到所屬區塊尾端,正反兩面皆無貢獻。經驗問題在於:在這個早期視窗學到的順序是否可泛化;答案是,在 400k hc 量得的 +6.17% 在 1M-hc checksum 跑與 10M-hc 精靈密集 SMB1 門中原封不動地延續——早期級聯順序顯然是網表活動結構的穩定性質,而非所觀察的特定開機序列的性質。

實測:階梯步 S5→S6(commit `3e4a571`)為中位數 +5.0%(117,671→123,545 hc/s)。隔離的成對量測為相對於盲目靜態鍵 **+6.17%(20/20 成對勝)**——而且,對設計具決定性地,自我捕捉鍵在同一執行檔的正面對決中以 +4.94%(16/16)擊敗*載入的離線 profile*。線上自我剖析勝過陳舊的離線剖析,是經典的動態對靜態 PGO 結局,它正當化了把檔案模式整個刪除(commit `a553d38`)。

7.4 順序,而非密度

對 +6.17% 的自然解讀會是:捕捉鍵把共活躍節點裝進更少的 cache line——locality 的故事終獲平反。數據反駁了這個解讀,而這個反駁是本章最具啟發性的量測。儀器是一個 cache-line 密度計數器:每半週期觸碰的相異 `NodeInfo` cache line 數,對整次執行取平均。在它之下比較了四個次要鍵(表 7.2)。

表 7.2. 四鍵實驗: `NodeInfo` cache-line 密度對牆鐘收益(C#,full_palette,交錯成對,全數位元精確;實測[21])。密度與收益不相關;剪枝關閉的捕捉是受控對照。

次要鍵	行/hc	牆鐘相對盲目鍵
盲目 clk-BFS(靜態)	118.4	基線(+0%)
離線共活性 profile	109.5	+1.55%
自我捕捉,剪枝**關閉**	110	±0.0%
自我捕捉,剪枝**開啟**(上線版)	105.9	+6.17%

受控對照在最後兩列之間。在事件剪枝*停用*下捕捉的*初次*觸碰順序,達到與 profile 鍵基本相同的 cache-line 密度(110 對 109.5 行/hc),而收益是*零*。同一個捕捉機制,在剪枝啟用下執行——觀察生產引擎實際將要執行的排程——以僅略好的密度(105.9)賺到 +6.17%。密度是必要的,但遠遠談不上充分;有效成分是順序的*血統*(lineage)。這把鍵當且僅當它是已剪枝生產級聯初次彈出序列的忠實影像時,才值 +6.17%。

硬體計數器從另一側說了同一件事。區間剪枝步 S5 把 L1d MPKI 砍半(26.3→13.2);自我捕捉步 S6 接著在*完全沒有進一步 miss 下降*之下加上它的 +5.0%(13.2→12.6 MPKI,邊際)。無論 S6 買到什麼,它不是靠 miss 更少買到的。它也不是靠做得更少買到的:每 1M hc 的已執行指令在這一步其實*上升*(107.8→111.6 billion——事件數在構造上完全相同,所以這是程式碼產生的擾動),而 IPC 從 2.25 爬到 2.39。一個執行更多指令、miss 相同、卻提早 5% 完成的變更,把它的收益完全花在延遲上——同樣的相依鏈,以機器更能容忍的順序走過。結合 7.1 節「引擎受相依鏈綁定」的發現,圖像是一致的:收益騎在一個 L1 常駐結構被走訪的*順序*上,不在走訪了多少,也不在走訪執行了多少指令。

> **微架構假說——明確標示為假說。** 收益*不*來自哪裡(cache miss、行密度),我們有量測背書;它*究竟*來自哪裡,我們只能假設,因為機器沒有暴露任何能解析它的計數器(本機的 LLC 計數器失能,而載入到使用(load-to-use)的延遲分布無法從軟體觀察)。三個不互斥的候選:(i) *預取器串流訓練*(prefetcher stream training)——當 id 跟隨初次觸碰順序時,級聯的存取近似遞增位址順序,next-line 與 stride 預取器認得這種模式,把部分相依載入延遲轉換為預取命中(hits-under-miss);(ii) *記憶體層級平行度*(memory-level parallelism)——連續排程節點的相鄰 id,讓亂序視窗內重疊的載入落在已在途的 cache line 上;(iii) *微視窗空間密度*(micro-window spatial density)——短時間視窗內的相關存取模式,即空間記憶體串流(spatial memory streaming)[14] 所利用的範疇,獎勵與排程吻合的佈局,即使總體密度不變。三者都符合「牆鐘動、miss 計數器不動」的特徵;沒有一個被證實。我們把它們標記為假說,而非發現。

7.5 與 Inspector-Executor 及軌跡導向佈局的關係

本章工具箱中沒有任何東西是沒有祖先的,我們把譜系說清楚。整體形狀——先用一個便宜的 *inspector* 跑過不規則計算的實際存取模式,再於 *executor* 執行之前重組資料(及/或迭代)——是為不規則科學計算發展出來的 inspector-executor 範式:CHAOS 執行期程式庫 [7]、Ding 與

Kennedy 的執行期資料與計算重組 [8],以及 Strout、Carter 與 Ferrante 的執行期重排組合框架 [9]。按追蹤到的存取序列為資料排序同樣早有先例:快取意識(cache-conscious)結構佈局與 Chilimbi 的串流式抽象 [10, 11]、profile 導向的資料放置 [12],以及已部署工具鏈中的 profile 引導資料佈局 [20]。我們的驗證加回退紀律,其模式借自動態去優化 [13]。在這個背景之下,此處的構造是一個*變體*,其具體差異有五:

- **同進程且自足**。 模擬器是它自己的 inspector:不需編譯器支援、不需儀器化工具鏈、沒有 profile 工件、沒有獨立的訓練跑。inspector 是 executor 自身內迴圈的一份冷副本,整個循環——分類、捕捉、重建——在一次 1.3 s 的載入內完成。
- **事件驅動的軌跡,而非陣列存取串流**。 被捕捉的對象是開關級網表 [1] 上一個已剪枝離散事件安定排程 [4] 的初次彈出順序,而不是陣列走訪的索引串流;7.4 節剪枝開/剪枝關的對照顯示,軌跡的*血統*——它對生產排程的忠實度——就是全部的有效載荷,這是 [8, 10, 11, 12] 的 locality 目標框架不會浮現的區別。
- **服務於非 locality 目標的全 id 空間重建**。 置換改寫網表的整個識別碼空間,其首要目的是語意編碼——位置即述詞(position-as-predicate)、刪除查表(7.2 節)——locality 被降級為次要鍵。文獻中的重組是為 locality 優化資料放置;在這裡,量得的 locality 價值約為零,重排照樣划算。
- **經驗證的回退**。 佈局的語意負載(區間編碼的剪枝事實)在每次 Reset 時對獨立重算的基準真相認證,不匹配時回退到安全退化、仍位元精確的形式——一個施加在資料佈局、而非編譯後程式碼上的去優化防護(deoptimization guard)[13]。
- **位元精確作為硬性門檻**。 本章的每一個佈局都重現了全狀態 checksum;正確性在經稽核的構造上與佈局無關(性質 7.1),而不只是靠測試。

每個成分單獨拿出來都是已知的。這個組合——一個事件驅動模擬器,在每次載入時從自身的生產軌跡重新導出自己的記憶體佈局、用置換把自己最熱的靜態述詞編碼為暫存器比較、對重算的基準真相認證該編碼、並在認證失敗時安全降級——就我們所知未見於先行文獻,我們把它作為本章的貢獻提出,並以 7.1 節的死路索引教訓作為其方法論上的伴侶。

第 8 章 評估

本章把前面各設計章節的所有主張化為量化數據。其核心是一座八階段的消融階梯(ablation ladder):在同一天、同一台機器上,把引擎歷史上的每個 commit 從原始碼重建,並在完全相同的協定下進行 benchmark;接著以硬體效能計數器重新檢視這座階梯,並與同一網表家族的其他公開模擬器並列比較 [15, 16, 17, 18, 19]。我們給方法論的篇幅異常地大——超出系統類評估的慣例——因為本專案最具啟發性的發現之一本身就是「方法論的」:一次把資料快取 miss 削減 67% 的變更,結果只換到個位數的牆鐘時間改善。在較不偏執的協定下,這個結果會被悄悄誤報;而它重塑了第 5–7 章工程結果應有的解讀方式。通篇之中,每個數字都是本專案實際取得的量測;凡是結果為負面、混雜、或弱於初次發表時的說法,我們都直說。

8.1 環境與方法論

8.1.1 機器、工作負載與驗證閘門

所有量測皆在同一台機器上進行:AMD Ryzen 7 3700X(Zen 2 微架構,8 核 / 16 執行緒,每核 32 KB L1 資料快取與 32 KB L1 指令快取、每核 512 KB 私有 L2、32 MB 共享 L3),作業系統為 Windows 11。C# 引擎依受測階段執行於 .NET 10 或 .NET 11——每個歷史 commit 都以它當時出貨的 runtime 建置,這個選擇我們會在 8.2 節回頭討論。除非另行註明,本章所有資料皆於 2026-06-12 蒐集。

工作負載為 `full_palette.nes`(社群開源的 NES 測試 ROM,顯示完整 NES 調色盤的靜態畫面,取自公開的 nes-test-roms 測試集),一個 NROM 測試 ROM,在 2A03 持續執行的同時操練 2C02 的完整繪圖管線。吞吐量跑模擬 400,000 個主時脈半週期(hc);硬體計數器跑模擬 1,000,000 hc,以攤平取樣的啟動成本。同一時間恰好只有一個 benchmark 行程在跑,機器上沒有其他前景負載。每個階段我們回報各輪的中位數,並附上該階段最佳值作為輔助數字;我們從不以單一跑分作為報告依據。

每一跑——無一例外——都以涵蓋全部節點狀態的全狀態 FNV-1a checksum 作閘門:吞吐量跑在 400,000 hc 為 0x9174E19D961CB6E5,計數器跑在 1,000,000 hc 為 0x6D4CCBCE2E9CD599(300,000 hc 的 golden 0x794A43ABDF169ADA 與一個 10,000,000 hc 精靈密集的 *Super Mario Bros.* 閘門,在專案歷史上一路把關各項採用決策)。這道閘門有兩個目的。其一,它在評估時強制執行第 2 章的位元精確契約:任何產出「快但錯」模擬的階段都會被自動排除。其二——更平凡卻同樣重要——當歷史 commit 被重建時,它防範悄悄建錯的二進位:過期的建置產物或用錯 runtime 的建置,要嘛重現 golden 狀態軌跡,要嘛不會。8.2 節的全部 72 次階梯跑、8.3 節的全部歸因跑、8.4 節的全部計數器跑,皆通過各自的閘門。

8.1.2 重建歷史:worktree 與 commit 身分

消融階梯並不重新實作引擎的早期版本;它重建它們。每個階段都是版本庫歷史中的真實 commit [21],checkout 到隔離的 git worktree 並從原始碼編譯。這消除了一整類消融假象——「停用功能 X 的基準線」對於功能 X 出現之前實際存在的程式碼,是出了名地不具代表性——代價是一處混雜(conflation),我們在 8.2.2 節誠實討論:一個 commit 可能在具名技術之外夾帶附帶變更。各階段 commit 依階梯順序為:a80dab4~1(S0,基線分叉)、a80dab4(S1,R-1 動態單例路徑)、

ed8c457(S2,P-1 同態 (same-state) 剪枝)、6bdc25b(S3,P-2/P-3/P-4 剪枝加遮罩合併)、00ab4fa(S4,條件重排與供電跳過摺疊)、51e046d(S5,區間剪枝重編號,同時把專案移到 .NET 11)、3e4a571(S6,自我捕捉初次觸碰鍵)、2749105(S7,B1 成對路徑)。

8.1.3 變異控制協定

現代桌上型處理器並不是跑在單一頻率上:3700X 會機會性地把時脈拉到遠高於 3.6 GHz 基礎時脈之上,實際達到的頻率取決於溫度、近期負載歷史與電源管理狀態。本章量測一律不鎖頻,變異由協定控制:同一時間恰好一個 benchmark 行程;一個實驗內的所有階段於同日「每輪輪替(round-robin)交錯」,讓每一輪依序各執行每個階段一次,使緩慢的熱漂移平均地記在每個階段帳上;各輪取中位數回報;每一跑都通過全狀態 checksum 閘門;歷史 commit 一律於隔離的 worktree 原樣重建(8.1.2 節)。對低於約 1% 的效應,採用 8.6 節所述的配對交錯。階梯中執行緒不釘核,以對歷史各階段保持一致(可選的親和性工具晚於其中數個階段才問世);其對變異的實測效果在 8.6 節回報。

8.2 消融階梯

8.2.1 結果

表 8.1 與圖 8.1 呈現這座階梯:八個階段九輪——72 跑,全部通過 checksum 閘門——每跑 400,000 hc,於 2026-06-12 單一場次量得。每一列在前一列之上新增一個技術家族;「階差」欄是相對前一階段的中位數對中位數改善。

表 8.1. 消融階梯:9 輪 × 8 階段,輪替交錯,每跑 400,000 hc,全部 72 跑皆通過全狀態 checksum 閘門(2026-06-12)。「TFM」為各 commit 出貨時的目標框架。吞吐量單位為每秒半週期數。

階段	Commit	新增	TFM	中位數 hc/s	最佳	階差
S0	a80dab4~1	基線分叉(靜態 fast path、SoA 佈局)	net10	67,955	69,677	—
S1	a80dab4	+ R-1 動態單例	net10	80,657	82,806	+18.7%
S2	ed8c457	+ P-1 同態剪枝	net10	101,964	103,818	+26.4%
S3	6bdc25b	+ P-2/P-3/P-4 + 遮罩合併	net10	109,841	113,571	+7.2%
S4	00ab4fa	+ 條件重排 + 供電跳過摺疊	net10	112,887	114,833	+2.8%
S5	51e046d	+ 區間剪枝(類別為主鍵重編號;+ .NET 11)	net11	117,671	119,464	+4.2%
S6	3e4a571	+ 自我捕捉初次觸碰鍵	net11	123,545	126,781	+5.0%
S7	2749105	+ B1 成對路徑	net11	132,243	135,828	+7.0%

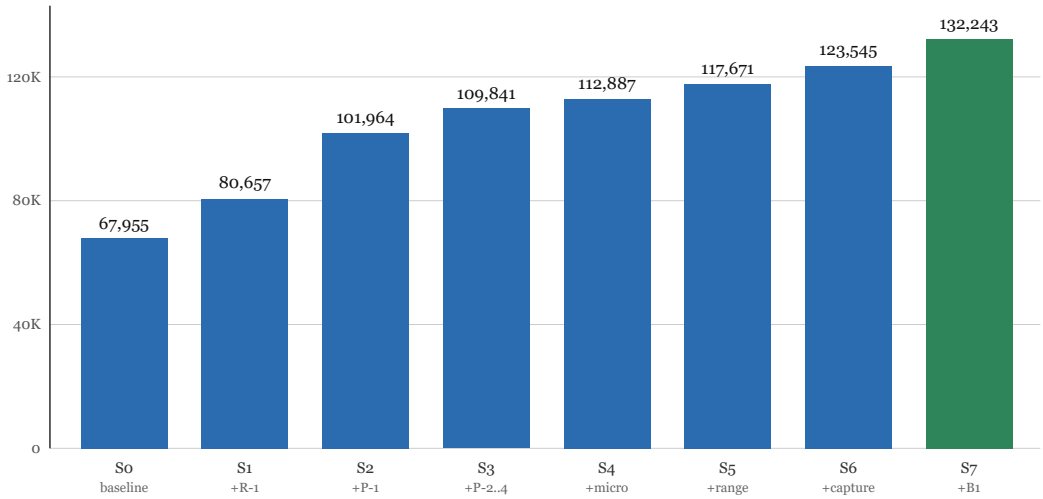

圖 8.1. 消融階梯:各階段中位數吞吐量(hc/s,400,000 hc 跑,9 輪輪替交錯,2026-06-12)。每一根長條都位元精確:全部 72 跑皆重現 golden 全狀態 checksum。

> **累計結果。** S0 → S7 中位數吞吐量 +94.6%——引擎幾乎翻倍——同一台機器、同一天、同一協定,且每一跑都重現 golden checksum。沒有任何階段以正確性換取速度;位元精確契約在整座階梯上全程成立。

8.2.2 逐階段評註

S0——基線並不天真。 67,955 hc/s 的起點已經納入非託管的 SoA(structure-of-arrays)佈局、帶去重 hash 的雙緩衝波清單,以及 cls1 節點的靜態 fast path——即完全沒有 pass 通道的那 3,929 個節點(網表的 26.7%),其解析永遠不需要群走訪。以家族標準而言,S0 已經比我們量測過的每一個外部模擬器都快(8.5 節)。因此這座階梯量的是本書各項技術相對於一個稱職的事件驅動引擎,而不是相對於稻草人。

S1——R-1 動態單例,+18.7%。 R-1 把單例 fast path 從靜態隔離的 cls1 節點延伸到 10,784 個 cls2 節點(網表的 73.2%)——這些節點的 pass 電晶體只是「恰好」在評估當下全部 OFF:一次檢查即可證明導通群為 $\{n\}$,解析便坍縮成一次 O(1) 查表。其幅度由事件剖面解釋:全部節點重算約有 70% 解析為單例,所以 R-1 把最熱迴圈最常見的情形,從一次 BFS 加 LUT 的走訪變成寥寥數條指令。計數器資料(8.4 節)補上一個細節:S1 反而讓 IPC 微降(2.11 → 2.08)並推高每指令的 miss 率,同時把已執行指令砍掉 17%——留下的指令更少,而倖存者更受延遲束縛。這筆交換是決定性的正收益,但它預示了階梯其餘部分所處的運作情境(regime)。

S2——P-1 同態剪枝,+26.4%,最大的單一階差。 P-1 在入列之前壓掉「兩個通道端點狀態已經一致」的導通(turn-on)事件,並受第 5 章的結構性安全汙染標記(taint)約束(排除無上拉節點與 ForceCompute 通道元件)。對照浪費剖面,其幅度是合理的:全部佇列彈出有 80.1% 解析為無狀態變化,所以任何健全、能刪掉一片可證明為空之事件的機制,刪掉的不只是該次解析,還連帶刪掉彈出、去重 hash 流量,以及每個死事件拖在後面的下游入列嘗試。為求誠實我們註明:P-1 的天真形式——不帶 taint——曾可見地發散(數個 frame 之內黑畫面):這個剪枝家族的價值與其安全分析不可分割,因為 taint 所保護的浮接 tie-break 正是晶片的儲存機制 [1]。

S3——P-2/P-3/P-4 與遮罩合併,+7.2%。 本階段加入度 1 無驅動葉節點的關斷隔離剪枝(P-2)、為 P-1 找回可證明 tie-break 免疫節點的電容支配解除汙染(un-taint)(P-3)、其多通道一般化(P-4),以及把各安全陣列合併為單一位元打包遮罩。在各自的採用 commit 上,它們分別量得 +1.4%、+5.96%、+1.71% 與 +0.64%——乘法合成約 +10%,對上階梯量得的 +7.2%。我們兩個數字都回報,並相信階梯:採用時的量測是在不同日子、對著移動中的基準線取得,而同日輪替交錯的合成量測才是權威協定。整個 P 家族合計刪掉約 21% 的節點重算,熱路徑成本為零,所有判定皆於載入期預先計算。

S4——微優化,+2.8%。 兩個變更共享這一階差:依實測選擇性重排熱點 `SetNodeState` 條件的各合取項(profiler 顯示首項約 97% 為真,作為閘門幾乎無用),以及供電跳過摺疊——標記電源與接地軌,使關斷走訪的兩個通道端點變成一個均勻的展開迴圈。各自在採用時量得約 +1–2%。這是階梯給的提醒:傳統的微調仍然有回報——但只及結構性技術所給的零頭。

S5——區間剪枝,+4.2%。 第 7 章的類別為主鍵重編號重排節點 id 空間,使各剪枝類別成為連續的 id 區塊(本網表上邊界為 A = 460、S = 1275、B = 7532),把每事件的安全查表從相依的記憶體載入變成對常數的暫存器比較。本階段的計數器特徵極為戲劇化——每指令 L1d miss 砍半(8.4 節),因為類別為主鍵佈局同時也讓熱集變得緻密——但牆鐘時間只動了 +4.2%,這是下文發展的「受延遲綁定」解讀的第一個有力證據。

> **混雜警示。** S5 是唯一一個其 commit 帶有第二項變更的階段:從 .NET 10 移到 .NET 11。階梯尊重各 commit 的出貨 runtime,所以這個 +4.2% 的階差把技術與 runtime 綁在一起。該技術自身在採用時的量測——對其直接父代配對且交錯——為 +3.56%,這把 runtime 升級的貢獻界定為一小幅殘餘。我們沒有進一步重審這次拆分;讀者應把 S4 → S5 的階差理解為「區間剪枝加 runtime」,其中技術佔了大部分。

S6——自我捕捉初次觸碰鍵,+5.0%。 三遍載入(分類;重建並暖機;透過 settle 迴圈的一份冷儀器化副本捕捉 32,768 hc 的真實初次彈出順序;再次重建)以生產級聯自身的存取順序重新導出節點 id 空間,載入成本約 1.3 秒。其計數器特徵與 S5 恰好相反:幾乎沒有進一步的 miss 下降,牆鐘增益卻大於 S5。有效成分是順序、不是密度——這是重排工作的核心發現,也是它與以 miss 數為目標函數的快取感知佈局文獻 [10, 11, 12] 的分野。

S7——B1 成對路徑,+7.0%。 B1 把基數特化從大小 1 的群延伸到大小恰為 2 的群:當節點恰有一個 ON 閘極、且鄰居的 ON 通道全部通回種子時,{種子, 鄰居} 這一對就地解析,逐位元組複製一般走訪的成員 hash 清除、供電掃描、tie-break 形式與寫入順序。母體規模支撐這個增益:全部群走訪有 77.1% 是大小 2,B1 適用子集涵蓋全部彈出的 19.5%。與緊鄰的整理性 commit 分離歸因後,效應約為 +8.9%(8.3 節)。

8.3 歸因方法論

由歷史 commit 建成的階梯繼承了歷史的不整潔:兩個具名階段之間可能夾著與技術無關的整理性 commit;若它本身有效能效應,階梯會悄悄把它記到具名技術頭上。我們採用的通則:一個階梯階差,只有在每個夾在中間的 commit 都已被直接的交錯量測證明中性、或被併入該階差的描述之後,才歸因給具名技術。這很費工——在有爭議的階差附近大約使量測預算翻倍——但它是對消融研究最

常見失敗模式的唯一防線;那個失敗模式不是壞統計,而是壞的「出處(provenance)」。依此程序檢驗 S6 與 S7 之間唯一的中介 commit:直接的交錯量測證明其為中性;分離歸因後,B1 的效應約為 +8.9%。

8.4 硬體計數器研究

8.4.1 計數器方法論

我們在 Windows ETW 效能監控計數器取樣下,對每個階梯階段以每跑 1,000,000 hc 進行剖析;取樣間隔為:指令與週期 1,048,576 事件、cache miss 與分支誤判 65,536 事件;事件總數以樣本數 × 間隔重建。取樣帶來約 5% 的負擔,所以本節交付的是「比率」——MPKI、IPC、每模擬半週期的指令數——而非絕對吞吐量;後者表 8.1 已無汙染地提供。表 8.2 與圖 8.2 給出各階段剖面;表 8.3 對三個階段以 AMD 原生快取事件(資料快取存取、指令快取擷取與 miss)做第二輪,得到的是存取次數與 miss 比率而非每指令率,作為獨立的交叉檢核。

表 8.2. 各階梯階段的硬體計數器剖面(ETW PMC 取樣,每跑 1,000,000 hc,2026-06-12)。MPKI = 每千條已執行指令的 miss 數;末欄為每 1M 模擬半週期的已執行指令總數,單位為十億(B)。

階段	IPC	L1i MPKI	L1d MPKI	分支誤判 MPKI	已執行指令(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

表 8.3. 三個階段的 AMD 原生快取事件輪(每跑 1,000,000 hc):絕對存取次數與 miss 比率,獨立於表 8.2 的每指令正規化。

階段	L1d 存取(B)	L1d miss 率	L1i fetch(B)	L1i miss 率
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%

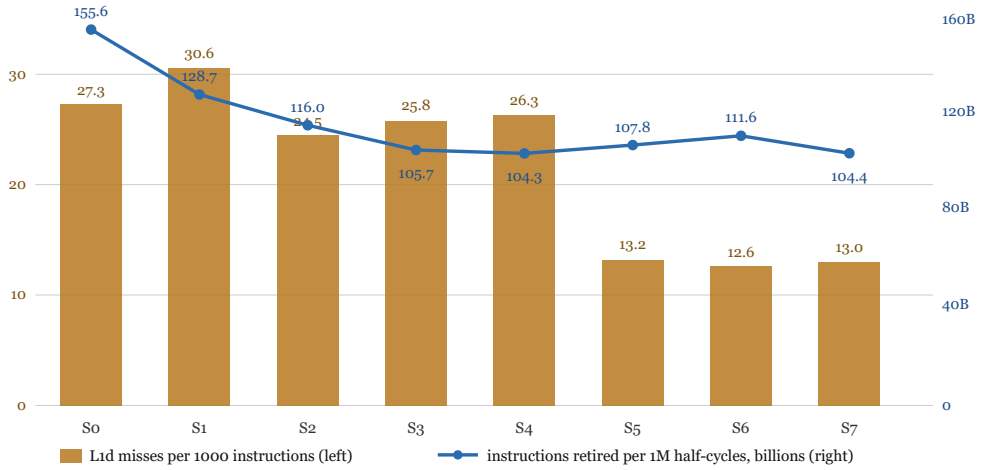

圖 8.2. 計數器在階梯上的軌跡:每千指令 L1d miss(長條,左軸)對每 1M hc 已執行指令總數(折線,右軸)。miss 率在 S5 砍半而牆鐘只動 +4.2%;指令數沿著階梯單調下降三分之一。

8.4.2 發現 1:指令擷取免費——為什麼直譯在此勝過編譯

在每個階段,L1i miss 都落在 0.169 至 0.300 MPKI 之間——原生輪並以絕對量證實了這幅圖像:每百萬半週期 3.2–4.6 B 次的指令擷取中,只有 0.59–0.74% 未命中 32 KB 指令快取。整個 settle 迴圈——派發、各 fast path、群走訪、解析 LUT、兩種剪枝走訪,以及自 S7 起的行內成對路徑——都舒適地裝進 L1i。程式碼足跡確實隨著 fast path 增加而成長(i-fetch MPKI 從 S0 到 S7 幾乎翻倍),但起點低到它從未接近「有影響」的程度。

> **發現 1。** 直譯器在指令遞送上實質上不付任何代價。這就是一個當初在系統層面令我們意外的結果在計數器層面的解釋:這個模擬器的每一個編譯或 AOT(ahead-of-time)變體都輸了,而且輸得很慘——產生碼後端慢 3–6 倍、編譯式錐評估慢 45–84 倍(第 9 章)。COSMOS 傳統的編譯式開關級模擬 [2] 拿資料驅動派發去換直線程式碼體積;在一張平坦的 26.8K 電晶體網表上,這筆交換恰好在本引擎分文不付的地方引爆程式碼工作集,而資料側的成本——計數器顯示那才是真正的成本——絲毫未動。

8.4.3 發現 2:砍掉 67% 的 miss 只買到個位數——引擎受延遲綁定,而非受 miss 數綁定

S4 → S5 的轉換把每指令 L1d miss 率砍半,26.3 → 13.2 MPKI。與整座階梯的指令數下降複合之後,絕對 miss 約降三分之二:S0 的 27.3 MPKI × 155.6 B 指令約為每百萬半週期 4.25 B 次 miss,對上 S5 的約 1.42 B 次——絕對量 −67% 的削減,並由原生輪獨立佐證(75.6 B 次存取 × 5.6% 對 44.1 B 次 × 3.2%,即 4.2 B 對 1.4 B 次 miss)。牆鐘時間對這三分之二 cache miss 削減的回應:+4.2%。

反過來,S5 → S6——自我捕捉初次觸碰鍵——幾乎完全不動 miss 計數器(13.2 → 12.6 MPKI),卻交出 +5.0%——比那個把 miss 率砍半的階段「更大」的增益。第 7 章的順序對密度實驗在原始碼層面顯示了同樣的解離:在剪枝停用下套用的捕捉鍵達到相同的每半週期快取行足跡(110 對 105.9 NodeInfo 行/hc)卻得到 ±0.0%,而同一種捕捉在剪枝啟用下取得時得到 +6.17%。等密度、不同順序、相反結果。

發現 2。 本引擎受「相依載入鏈的延遲」綁定,而非受 cache miss 的「數量」綁定。能與其他工作重疊的 miss 是便宜的;昂貴的是那條序列鏈——節點 id → NodeInfo → 電晶體表 → 鄰居狀態——其環節在前驅解析完成之前無法發出,而硬體預取無法預判它,因為位址序列依資料而定(streaming-predictor 文獻明確針對的是互補的、空間相關的情形 [14])。有回報的優化,是那些「從鏈上刪除環節」的(區間剪枝以暫存器比較取代遮罩載入;B1 以行內端點讀取取代走訪),或是把存活的鏈按已剪枝級聯實際走訪的順序排程的(自我捕捉鍵)。這顛倒了古典快取感知佈局文獻 [10, 11, 12] 與 profile 導引資料擺放 [20] 的目標函數——它們優化的是 miss 數;我們的重排機構在結構上是一個 inspector–executor [7, 8, 9],但 inspector 必須捕捉的是執行的「順序」,不是親和圖。

8.4.4 發現 3:剪枝刪的是指令,不只是事件

已執行指令總數沿著階梯從每百萬半週期 155.6 B 單調降到 104.4 B——減少 33%——同時 IPC 維持在 2.08–2.41 的帶域內。絕對分支誤判降得更多:從 S0 約每百萬半週期 509M 次(3.27 MPKI × 155.6 B)降到 S7 約 280M 次,−45% 的削減。事件壓制機構於是在體系結構層面以真實的工作刪除現形:P-1 至 P-4 在入列之前移除約 21% 的重算,而每個被刪除的事件把它的彈出、它的解析、它的 hash 探測、它的誤判分支一併帶走。這是把 Ulrich 的 activity-exclusive 原則 [4] 再往前推一步:古典事件驅動模擬拒絕評估「沒有改變的」;剪枝家族拒絕「入列」可證明不可能改變的。

8.4.5 階梯上的 IPC,與 LLC 限制

IPC 欄值得逐階段細讀。S1 讓 IPC 微降(2.11 → 2.08)同時砍掉 17% 的指令——R-1 移除的是容易、預測良好的走訪指令,留下更稠密的鏈綁定殘餘;每指令 miss 率的「上升」(27.3 → 30.6 MPKI)伴隨絕對 miss 的「下降」,提醒我們:當分母移動時,每指令正規化會誤導。S2 憑誤判削減(3.37 → 2.74 MPKI)把 IPC 抬到 2.38:被刪除的無變化彈出正是預測差的那些。S5「儘管把 miss 砍半」IPC 卻降到 2.25——與受延遲綁定的解讀一致:被移除的 miss 並不是卡住指令退役(retirement)的那些停頓。整個帶域的水準——一個指標追逐式的直譯器在 Zen 2 核心上跑出 2.1–2.4 IPC——顯示各 fast path 讓管線維持相當程度的餵飽;這不是一個閒在 0.5 IPC 等 DRAM 的引擎。

限制。 在我們的 ETW 組態下,這台機器沒有可用的最後一層快取(last-level cache)計數器來源,因此我們無法直接回報 L3 或 DRAM 流量。兩個間接論證界定了這個缺口:引擎的熱資料集約 15 KB,輕鬆常駐 L1;且原生事件輪顯示 L1d miss 比率為 3.2–5.6%,流入的是每核 512 KB 的私有 L2——L2 之外不太可能還有多少流量。我們回報這個缺口,而不是粉飾它。

8.5 家族對照

AprVisual 屬於建立在同一批晶粒導出網表上的一個小模擬器家族 [15, 16]。2026-06-08 我們在這台機器上以無頭(headless)方式編譯並量測各公開成員,同一 ROM、同一單位(整機主時脈每秒半週期數)——這些原始專案皆未內建 hc/s 吞吐量量測,我們以實驗性修改其原始碼加上同單位輸出(詳見 2.4 節)——為的是把引擎放在它的親屬之間定位,而不是對照不可通約的已發表數字。表 8.4 給出結果。

表 8.4. 同一台機器、同一 ROM、同一單位上的網表家族(外部成員於 2026-06-08 量測;AprVisual 為表 8.1 的 S7 引擎)。perfect6502 為求完整而列入,但在此軸上不可排名:它模擬的是一顆孤立的 6502,沒有 PPU、沒有電荷模型,是不同的工作單位。

模擬器	語言 / 譜系	吞吐量	相對 AprVisual
VisualNes [19]	C++,chipsim.js 逐字移植	約 24K hc/s	慢約 5.6 倍
perfect6502 [18]	C,僅 6502 的優化重寫	約 29K(僅 6502 單位)	不可比較
MetalNES [17]	C++,直系祖先	約 54K hc/s	慢約 2.5 倍
AprVisual [21]	C#	約 136K hc/s(最高紀錄)	—

這張表有兩種重要讀法。其一,對 MetalNES 的 2.5 倍是本書貢獻的誠實度量,因為 MetalNES 正是我們所移植的那個引擎:這個差距恰好等於表 8.1 的階梯加上基線分叉的工作,沒有任何演算法語意差異可資混淆。一份受控語言實作以 2.5 倍勝過其優化過的 C++ 祖先——同時加上祖先所沒有的位元精確體制——以實測反駁了一個廣泛存在於效能工程實務中的一般印象(我們未找到其正式文獻出處):受控語言不適合這種指標追逐的核心。其二,絕對尺度:136K hc/s 等效於約 68 kHz 的矽主時脈(主時脈每兩個半週期跳動一次)、約 5.7 kHz 的 6502 核心時脈(hc/24),以及約 17 kHz 的 PPU 像素時脈(hc/8)。對上實機的 42,954,552 hc/s,差距約 316 倍。我們直白地說:NES 的即時開關級模擬在單核上仍然遙不可及,而第 9 章將以實測的負結果論證:我們嘗試過的任何軟體路線——編譯、GPU 卸載、位元平行、多執行緒——都無法補上這個差距。

8.6 量測衛生

我們以本專案最終收斂出的量測實務作結,因為本章有數個結果小於隨性 benchmark 設定的雜訊地板,沒有這些實務便無法重現。

變異控制工具。 可選的 `--pin` 模式(執行緒親和性釘到單一實體核心、行程高優先權、停用 EcoQoS——刻意不碰頻率)把跑間變異減半,變異係數 1.5% → 0.94%,輸出位元精確。它預設關閉,讓發表的排行榜數字 [21] 反映一個無特權行程;在階梯中為求跨階段一致也保持關閉;它是採用時 A/B 的正確工具——其變異縮減直接換得統計檢定力。

熱誠實。 同一個二進位在同一台機器上,涼爽的早晨與熱透的下午之間散布約 10%。因此在本章處理的效應量級下,跨日的原始吞吐量比較毫無意義;此處每一個比較性主張皆為同日量測,而階梯的各階段又另以輪替交錯,使場內漂移平均地記在所有階段帳上,而非碰巧最後跑的那一個。

sub-1% 效應的配對交錯。 批次式 A/B——先跑完 A 的全部、再跑 B 的全部——把與時間相關的漂移直接嵌進對比之中,曾屢次誤導我們。採用的協定是:預先建好兩個二進位,每輪交替基準/實驗順序,回報中位數、截尾平均與配對勝場數(本書通篇以「k/n」回報)。它具體地證明了自己的價值:一個早期的 fast-path 變體在批次下讀來模稜兩可、在配對下決定性地分曉。

總結紀律。 不鎖頻;同日輪內輪替交錯;單一 benchmark 行程;中位數回報;sub-1% 效應採配對交錯;每一跑都通過 checksum 閘門。這些步驟沒有一個是新穎的;每一次都全部套用的紀律,才讓表 8.1 的階梯有資格被相信。

第 9 章 負結果與單核邊界地圖

前幾章報告了奏效的技術:派發快速路徑、事件剪枝、識別碼空間轉換,以及它們累計產生的 +94.6%。本章報告所有沒有奏效的東西,並論證這些失敗不是殘渣,而是本專案最主要的科學產出。一份只記錄勝利的優化帳本描述的是一條軌跡;一份連同失敗——以及其母體數(population)、機制、與各自量測條件——一併記錄的帳本,描述的則是一道*邊界*——亦即:在哪個區域之內,這份網表的位元精確、事件驅動、單核開關級模擬器還能被加速,而越過它就不能。我們分七個部分呈現這張邊界地圖:為什麼我們把負結果視為一級工件(§9.1);抽象化路線與可化約性悖論(§9.2);平行化嘗試及其四道結構之牆(§9.3);維護態地板——以專案中最具啟發性的單一失敗來講述(§9.4);較小死路的普查,連同關閉它們的實測母體數(§9.5);其背後反覆出現的反模式型錄(§9.6);以及由此得出的邊界陳述(§9.7)。

9.1 為什麼負結果在這裡是一級成果

本專案文件化的成果,依其自身的定位,就是一個可證偽的負結果:在所述契約之下——完整的 Bryant 開關級語意 [1]、對三條 golden checksum 加上一個精靈密集的 1000 萬半週期閘門的逐節點位元精確、單顆 CPU 核心——這份網表的即時模擬不可達,而人們可能寄望抵達它的各條路線,已逐一被量測關閉。這種形式的主張,其價值完全取決於背後的紀律。我們方法論的兩項性質,使本章的負結果可信而非軼事。

第一,每一個負結果都是*量測過的*負結果。每一項失敗的技術,要麼被實作出來、並以與勝利相同的交錯成對(interleaved-paired)協定量測(每輪內輪替兩個組建、跨多輪取中位數、每輪都以 checksum 閘門把關),要麼在實作之前就被一次母體量測殺掉——剖析器計數顯示該技術所能省下的事件太少、太便宜、或太分散,不足以支付其成本。全章我們區分三個等級的負結果:在所述條件下的實測損失(例如 IR 直譯器的 −2.5%);有量測支持的結構性論證(例如錐壓縮普查顯示不存在更粗的粒度);以及健全性失效——該技術不只是輸,而是與 golden 模型發散(例如 §9.4 中上拉 pin 的即時狀態推導,或 §9.5 的提早停止 settle(under-settling))。

第二,每一個負結果都附帶其適用範圍,因為本專案曾因過度推廣負結果而吃過虧。引擎的「天花板」本身就被宣告過四次——大約在每秒 80K、100K、120K 與 127K 半週期處——又被推翻四次,其中最具決定性的一次,是 R-1 動態單例路徑在一個我們已宣告榨乾的組態上又加上 +18.7%。數個在批次式 A/B 協定下量得的 3% 以下負結果,後來在其中一個(無分支快速路徑掃描)於交錯成對下翻成小幅勝利之後,被判定可疑;而一個重編號「死路」,在其目標從快取局部性改成刪除相依載入之後,翻成 +3.6%(第 7 章)。因此,負結果的誠實形式不是「X 不可能」,而是「X,以這個形式、在這個引擎上、輸這麼多、出於這個機制」——而機制才是可重用的部分。

9.2 抽象化路線

面對緩慢的直譯器,一個自然的想法是拉高抽象層次:建立中間表示、做層級化(levelize)、編譯它。這是 COSMOS [2] 的譜系——它把 Bryant 的開關級模型編譯成布林求值程式,在其年代的同步 MOS 電路上對直譯式模擬取得決定性勝利。我們在這份網表上嘗試了那把梯子的每一階。每一階都比事件驅動直譯器慢,而其原因可以合成為單一一句結構性陳述。

9.2.1 IR 直譯器:正確、覆蓋活躍節點的 23%,以及 −2.5%

第一次嘗試刻意保守:把最乾淨的純邏輯節點抽取成逐節點真值表,經由一個 IR 求值器派發,其餘一切留在開關級路徑上。抽取是位元精確的——整機 NES checksum 不變——並覆蓋了 3,390 個節點,即活躍網表的 23%。以交錯成對量測,它輸了:中位數 −2.54%,九輪中零輪獲勝。

機制才是重點。乾淨到能抽取成真值表的節點,恰恰就是開關級引擎已經用其 O(1) 單例快速路徑解析的節點:寥寥數條指令加上 256 項優先序 LUT 的一次查表。IR 求值——讀閘極狀態、打包索引、追逐閘極清單、查一張表——比它所取代的東西*更重*。與此同時,引擎解析得慢的節點(多節點導通群、電荷分享匯流排),正是那些無法表示為固定輸入之布林函數、因而根本無法抬升進 IR 的節點。於是抽象層把本已最小的工作包進額外負擔,卻對昂貴的工作毫無作為。一個乾淨、正確的 IR,在這個工作負載上,是純粹的成本。

9.2.2 編譯式求值:慢 3–6 倍到 84 倍,以及 i-cache 反轉

下一階是編譯。兩個獨立的努力都到達了這一階。較早的(S4 時期)AOT 批次後端——C# 程式碼發射、一條 LLVM 路徑、位元切片與稠密變體——實測全部比事件驅動引擎慢 3–6 倍,原因在演算法層面、任何程式碼品質都修不了:批次後端每半週期重新求值約 14.7K 個節點,而真正改變的只有幾百個。這等於要求編譯器在執行上勝過 30–150 倍的工作冗餘,它辦不到。

較晚的(Escape-1)努力則在抽取出的邏輯模型上端到端量測了盲掃式(oblivious)路線。直譯式盲掃求值——把全部約 6,000 個抽取節點掃到定點,每半週期約 6.5 次鬆弛迭代,因為雙向 pass-transistor 網路無法層級化——每半週期跑 539 µs,對上 golden 引擎的 12.0 µs:慢 45 倍,每半週期執行約 39,000 次節點求值,而事件驅動引擎只執行幾百次。把同一趟掃描用 Roslyn 編譯成直線程式碼反而*更糟*:每半週期 1,004 µs,慢 84 倍。編譯後的掃描把約 6,000 次節點求值展開成約 700 KB 的機器碼,每半週期串流 6.5 次——超出 32 KB 的 L1 指令快取一個數量級以上——於是處理器前端在重新擷取程式碼時停滯,執行單元卻閒置。盲掃式掃描的編譯之所以自我挫敗,正是因為「盲掃」意味著程式碼足跡隨整個電路的規模成長。

這個「i-cache 反轉」——編譯比直譯慢——是抽象化階梯上最具診斷力的單一數據點,而 2026-06-12 的硬體計數器消融(第 8 章)提供了它的反面。在事件驅動引擎自身歷史的全部九個量測階段中,L1 指令快取 miss 落在每千指令 0.169 至 0.300 之間:全內聯的 settle 迴圈塞進指令快取還綽綽有餘兩個數量級。事件驅動直譯器是一支跑在資料上的小程式;編譯後的掃描則是一支跑在同樣資料上的巨大程式。在一個資料足跡(約 15 KB 熱集)本已常駐快取的工作負載上,拿常駐的程式碼迴圈去換串流式的程式碼足跡,是純粹的損失。

那麼,*選擇性*的編譯器能不能贏——只編譯熱的多節點群,就像追蹤式 JIT 編譯熱路徑那樣?我們在動手寫後端之前正是為此做了剖析,而剖析在實作之前就把它殺掉了。在 300K 個半週期上(當日引擎的 181.4M 次重新求值),69.5% 的事件走的是編譯無從改進的 O(1) 快速路徑;其餘 30.5%——群走訪——分散在 4,752 個不同節點上,分布近乎平坦:最熱的單一節點只占群走訪工作的 0.14%,前 10 名占 1.2%,前 50 名占 6.0%,前 200 名也只占 21.6%。就算只想覆蓋一半的群走訪工作,也得編譯數千個節點專屬的本體,等於零碎地重建那 700 KB 的足跡;而導通群平均 2.25 個節點(在剪枝後的引擎上如今是 1.13–1.4),群的內部幾乎沒有東西可以讓直線程式碼省。追蹤式編譯獲勝的兩個前提——集中度與深度——都不存在。

9.2.3 錐壓縮:1.1 倍的普查

如果稠密求值會輸、逐節點編譯又無從著力,剩下的抽象化就是保留事件驅動的稀疏性、但把事件單位變粗:以組合邏輯錐(combinational cone)層級的事件取代節點層級的事件,以閂鎖與匯流排邊界為界,期望換來少一個數量級的、原子化的事件。我們直接量測可得的壓縮率,而不是先蓋排程器。把組合節點依通道連通性凝聚,得到 2,737 個錐,平均大小 2.0 個節點;重播一段 trace,90 個節點層級事件凝聚成 81 個錐層級事件——結構壓縮 1.1 倍。把錐的定義放寬到包含閘極連通性,並不會產生中間粒度:網表塌縮成一個 5,349 節點的巨錐,而觸發它就又回到盲掃式求值。兩者之間沒有膝點。

> **粒度發現**。這份網表中的組合邏輯錐平均 2.0 個節點;事件驅動引擎的平均導通群是 1.13–1.4 個節點。直譯器已經運作在網表的天然最小粒度上。不存在更粗的可重用結構單位可供抽取,因此也不存在任何抽象層能把直譯器零碎進行的工作批次化——這些工作的零碎,不是實作上的偶然,而是電路結構使然。

9.2.4 可化約性悖論及其消解

陳述抽象化之牆最尖銳的方式,是把它說成一個悖論。Escape-1 管線以全自動、可證偽的方式證明:晶片動態活動的約 98.9% 可化約為布林邏輯加暫存器,只有約 1.1% 是真正的類比行為(電荷分享匯流排、動態儲存);其上還坐著一個 88 倍的理想化 Amdahl 上限。自動的電晶體層→閘級邏輯抽取,以及以子圖同構為基礎的子電路識別/圖樣比對,是既有的成熟做法 [5][6];我們以物理性質為據的識別(上拉旗標、電容比較、通道元件)與「先驗證再啟用(verify-then-enable)」變體的差異,在於用 golden 模擬器作為神諭(oracle),而非元件庫/結構圖樣比對。然而,建立在抽取邏輯之上的每一種求值策略——直譯、編譯、錐批次——都比它所抽象的開關級引擎慢,倍數從 1.0 到 84。可化約性沒有轉換成速度。

消解有三條腿,全部經過量測。第一,可化約的部分是*逐節點*可化約,卻*不可層級化*:NMOS pass 電晶體是雙向的(哪一側是「輸入」由執行期哪一側在驅動決定),而兩相透明閂鎖使透明路徑成環,於是依賴圖的 94% 是一個雙向 SCC,抽取出的邏輯在結構上注定要跑約 6.5 次鬆弛迭代。即使是理想化的稠密掃描、每節點 2.5 個週期,每半週期也要花約 97,500 個 CPU 週期——約為事件驅動引擎預算的兩倍——這還在任何實作低效之前。第二,事件驅動引擎的每事件成本已經位於記憶體延遲地板(第 8 章):不存在可供編譯器移除的直譯負擔——就 Ulrich 原始論證「只模擬活動本身才是最主要的節省」[4] 的意義而言——本引擎正是經由 chipsim.js [15] 與 MetalNES [17] 一脈承自那個傳統,而活動率約為每半週期 3% 的節點。第三,上述 i-cache 計數器顯示直譯器作為一個迴圈不付任何代價,而每一個編譯變體都為「程式大小等同電路」付出沉重代價。

為完整起見,補上 GPU 數據點:單一模擬實例映射到 GPU 上作為一個 workgroup(安定波(settle wave)是一條串行依賴鏈,單一實例用不了更多),比 CPU 引擎慢 10.7 倍,同時只占用裝置的約 1/76。許多獨立實例才是 GPU 的天然形狀——但多實例吞吐量從來不是本專案的目標,而且它對單一位元精確實例的延遲毫無幫助。

表 9.1. 抽象化路線,實測。「Golden」指各次比較當日的事件驅動開關級引擎;所有實際跑過的變體在計時前都驗證為位元精確(Escape-1 模型則是在覆蓋節點上行為精確)。

路線	實測結果	失敗機制
混合 IR 直譯器(真值表節點)	−2.5%(0/9 輪)	可抽取節點恰為 O(1) 快速路徑節點;IR 把最小的工作包進額外負擔
AOT 批次後端(C# emit、LLVM、位元切片)	慢 3–6 倍	每 hc 重新求值約 14.7K 個節點而僅數百個改變;codegen 勝不過 30–150 倍的工作冗餘
選擇性/巨集區塊 codegen	實作前即被剖析否決	無集中度:前 200 個節點 = 群走訪工作的 21.6%;群平均約 2 節點
盲掃式掃描,直譯	慢 45 倍(539 vs 12.0 µs/hc)	每 hc 約 39,000 次節點求值(6,000 節點 × 6.5 次迭代)vs 數百個事件
盲掃式掃描,編譯(Roslyn)	慢 84 倍(1,004 µs/hc)	約 700 KB 直線程式碼 × 6.5 次掃描 ≫ 32 KB L1i;前端飢餓
巨集事件/錐粒度	1.1 倍結構壓縮	錐平均 2.0 節點 ≈ 導通群;不存在更粗的單位
GPU,單一實例	慢 10.7 倍	串行 settle 鏈 ⇒ 單一 workgroup ⇒ 僅用到裝置約 1/76

9.3 平行化

兩種平行分解被實作並量測;兩者都輸掉一個數量級以上,而事後剖析收斂到四道結構之牆——任何想平行化這個引擎的嘗試都必須翻越它們。

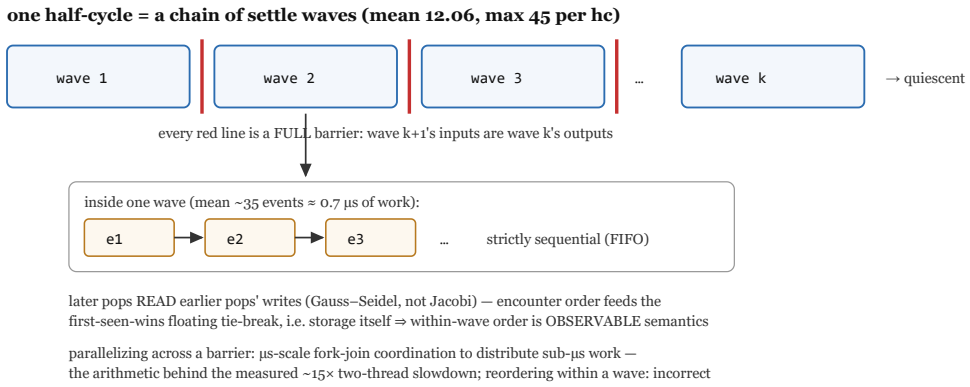

圖 9.1. 安定波是屏障、波內順序是語意。每個半週期平均排空 12.06 個波(最大 45),波界是完全同步點;波內事件嚴格依 FIFO 順序執行,較晚的彈出讀到較早彈出的寫入(Gauss–Seidel),而遭遇順序餵入先見者勝的浮接 tie-break——即儲存本身。跨屏障平行化要以微秒級協調分發次微秒級工作(實測雙執行緒慢約 15 倍);波內重排則直接改變模擬結果。

第一次嘗試是最自然的那種:NES 內有兩顆大晶片,那就把 2A03 與 2C02 放到兩條執行緒上步進(每個安定波 fork-join 一次)。實測:比單執行緒*慢* 15 倍。第二次嘗試徹底換了軸線:維持單執行緒,但讓群走訪資料平行化,把逐事件的 BFS 換成 Ligra 式的位元向量稠密前沿掃描,在一個 8K 節點 /

16K 電晶體的 PPU 子集上評估。演算法是正確的——checksum 逐位元相同——而且慢 156 倍。兩種不同的分解、兩種不同的失敗量級、一個共同根因:兩者都假設圖的大小蘊涵可利用的體量,但真正起作用的量是*每步工作量分布*,而那個分布小得殘酷。

四道牆,連同其實測尺寸:

1. **事件粒度 vs 通訊粒度**。處理一個事件約花 70–82 個 CPU 週期——在 4 GHz 下約 20 ns。跨核心搬移一條 cache line 的延遲,依公開微架構資料的常見量級估計約為 40–80 ns(未在本機單獨量測)。以此估計,任何跨核共享節點狀態,*每搬一條 line* 就要付出約二至四個事件的時間,這還沒把任何同步原語算進去。工作單位比通訊單位還小,這把分散式的經濟學整個反轉:把一個事件運到另一顆核心,比直接執行它還貴。

2. **安定波是屏障**。雙緩衝的 settle 迴圈每半週期平均排空 12.06 個波(最多 45),而每個波邊界都是一個完整同步點:第 k+1 波的輸入是第 k 波的輸出。在每半週期約 418 個事件下,平均每個波持有約 35 個事件——約 0.7 µs 的工作——夾在屏障之間。跨這種粒度做 fork-join,等於花費微秒級的協調去分發次微秒級的工作,這就是 15 倍損失背後的算術。本章 §9.5 的提早停止 settle 實驗封死了顯而易見的漏洞:深波不能被截斷或近似,所以屏障數量沒有商量餘地。

3. **波內順序是可觀察語意**。在一個波之內,引擎是 Gauss–Seidel 式的:同一波中較晚的事件會讀到較早事件的寫入,而浮接電容 tie-break——晶片真正的儲存機制——對那個順序敏感。把一個波切到多顆核心上會改變交織順序,從而改變被模擬的狀態;這不是效能轉換,而是語意改變。(這件事用了最便宜的方式確認:單執行緒上的波內重排實驗就已讓 checksum 發散。)

4. **不存在薄切口**。依賴圖的 94% 是單一雙向 SCC,且 CPU 與 PPU 透過共享的主機板節點持續互動,因此無論是靜態空間分割還是晶片邊界,都無法在同步點之間產生獨立的工作。第一次嘗試裡的「兩顆晶片」,在波粒度下電氣上是一台機器。

因此,要讓平行化變得可行需要改變什麼,是定義良好的——而每一個選項都打破契約。要把同步攤提到許多波上,需要波彼此獨立,而它們依構造就不獨立。要把兩顆晶片解耦,需要為其互動延遲設界,這就拿位元精確去換一個近似視窗。樂觀式平行離散事件模擬——先行投機、衝突時回滾——是經典的逃生口,但在約 35 事件的波與 94% 的 SCC 之下,可以預期衝突率將近乎全面(未實作驗證),而檢查點狀態是整個節點陣列。至於位元平行路線,敗因不在同步而在佔用率:當 77% 的群走訪只造訪兩個節點、平均 BFS 深度 1.13 時,一個 256 位元的稠密前沿每一步都做了必要工作的 100–200 倍。這些都不是工程上的不足;四道牆全是「網表加語意契約」的性質。

9.4 維護態地板

在已出貨的剪枝之後,引擎中最大的未解成本,是當某個驅動者斷開時重新求值一個節點,結果卻發現它沒變——因為另一個驅動者仍然撐著它。結構殘餘的數字(§9.5)顯示,全部事件彈出(pop)中有 80.1% 是這種「無變化」。本專案中最具啟發性的失敗——在五天裡分三次追擊,每次都逼出更銳利的結論——就是嘗試用一個逐節點的執行期事實去跳過它們。我們完整講述它,因為它終結於一條量化定律,而非一則軼事。

9.4.1 P-5:一個正確、確實觸發、卻輸給自身簿記的優化

想法(P-5,「支配驅動者旁路」)是:為每個節點記錄當前決定其值的那一顆電晶體;當閘極 g 關斷時,若端點 c 記錄的支配驅動者存在且不是 g,就可以跳過 c——另一個導體撐著它,這次斷開不可能改變它。第一個實作對其觸發的情形可證明健全,在 300K 與 1M 半週期上驗證位元精確,而且真的剪到了東西:保留簿記但停用跳過的分解量測顯示,跳過本身的收益是 +3.4%。完整功能實測 −12.05%(16 輪 0 勝),因為維護——在每次解析時捕捉支配驅動者、把刻意無分支的供電掃描變成有分支的計數加捕捉、以及以每幀約 117M 次寫入的頻率寫一個 29 KB 的陣列——單獨就要 −15.41%。

9.4.2 重建:同一道過路費背後更大的獎賞

一次從零重建(分支 `dominant-bypass` [21])把 16 位元的支配閘極 id 換成更便宜、更廣的 1 位元述詞——「此節點目前的值由它*自己的*接地/電源/上拉撐住,與 pass 連接無關」——並重新量測分解。獎賞遠比第一個形式所暗示的大:單看旁路本身值 +13.76%,壓制了約 40% 的節點重新求值。過路費卻沒有縮小:維護 −15.6%,外加一筆沒有免費選項的儲存稅。把 pinned 位元打包進 1-byte 狀態陣列,寫入免費,卻迫使每一次熱導通讀取都要加一個遮罩(−4.98% 地板,淨 −8.36%);給它獨立陣列可移除遮罩,卻新增一條約 50M 次寫入的隨機串流(淨 −6.84%,所得最佳)。把捕捉融合進群走訪——它本來就掃描成員供電——量測為中性:走訪造訪的成員遠多於受益的候選集合。

表 9.2. P-5 成本分解(重建形式,相對 P-4 基線,交錯成對,所有變體皆位元精確)。收益真實且巨大;其下的地板更大。

項目	實測	內容說明
跳過收益(僅旁路)	+13.76%	刪掉的重新求值;約 40% 的彈出被壓制
維護(每次解析時捕捉)	−15.6%	有分支的捕捉 + 在延遲綁定迴圈上新增一條隨機寫入串流
儲存稅,選項 (i):位元打包進 NodeStates	−4.98%	每次熱導通讀取都要遮罩;淨 −8.36%
儲存稅,選項 (ii):獨立 1-byte 陣列	淨 −6.84%(最佳)	無讀取遮罩,但旗標自成一條約 50M 次寫入的串流
維護融合進群走訪	中性	為所有被走訪成員做捕捉 ≈ 它所取代的針對性重掃

9.4.3 P-5z:把維護整個刪掉——以及可證明無法解放的那條腿

第三輪(分支 `dominant-bypass-2`)直攻地板本身,觀察到 pinned 述詞的一部分可分解為「靜態結構 × 即時狀態」——而即時狀態 `NodeStates` 本來就由引擎維護。對於恰有一條自身接地通道、且位於乾淨元件中的 6,338 個節點,跳過可以在跳過點零簿記地推導:若且唯若 `NodeStates[ProbeGate[c]] != 0` 就跳過關斷端點 c——那條接地通道的閘極是 ON,接地在優先序 LUT 中壓過所有其他貢獻,這次斷開不可能改變該節點,連暫態也包含在內。一張唯讀表、零寫入、零新增維護狀態。它通過了我們手上的每一道閘門,包括 10M 半週期的 SMB1 跑,並在每 100K 半週期壓制 5.3M 次重新求值——占全部彈出的 13.6%。

它實測為**中性**:中位數 +0.16%,成對 10/20(一個區間編碼變體為 −0.51%)。這筆算術精確而無情。關斷走訪每觸發一次跳過要做 9.4 次端點測試(50.1M 次測試買到 5.3M 次跳過),而這一類避開的彈出是引擎*最便宜*的——約 30 個週期的單接地葉節點解析——所以損益平衡要求測試成本低於約 3 個週期,而這恰恰就是它那兩個餵分支載入的價格。一個健全、維護免費、能跳過 13.6% 事件的跳過,什麼也沒買到,因為壓制「次數」不是目標函數;被壓制的*週期數*才是。

值錢的那條腿結局不同。重建形式 +13.76% 的質量中有 64% 來自上拉情形——「PullUp 把節點釘在高位」。把*那個*述詞從即時狀態推導,**讓 checksum 發散**:上拉 pin 的健全性證明綁定的是節點*在其上一次解析時*的值,而 settle 中途的暫態可能在兩次解析之間短暫違反即時狀態讀取。§9.4.2 那個被維護的 pinned 位元從來不是實作上的便利——它*就是*證明所需要的解析時快照。有利可圖的那條腿,在原理上就無法從維護中解放。

> **維護態地板**。三次獨立嘗試——活躍驅動者計數器(−6.0%)、支配驅動者 id(維護 −15.4%)、以及兩種儲存編碼下的 pinned 位元(合計約 −7% 地板)——收斂到一條定律:在這個延遲綁定的引擎上,任何需要逐解析執行期事實的跳過述詞,其維護成本都高於它所刪除的重新求值,在每一種已嘗試的編碼下皆然;而述詞可分解為「靜態結構 × 即時狀態」的那唯一一條腿,只搆得到引擎最便宜的事件,淨值為零。普遍的教訓,以精確的代價換來:*即時狀態不是上一次解析時的狀態*——關於節點保值的證明,綁定的是它上一次被解析時取得的快照,而保存那個快照就是不可化約的過路費。

> 對重建分支的一次對抗性審查,還發現了兩個其乾淨 checksum 從未行使過的*潛伏*健全性漏洞:ForceCompute 元件的同組節點會破壞釘高(pinned-high)證明(那個集合是 2C02 的精靈/OAM 匯流排,在 `full_palette` 中靜止),而成員迴圈內存在一個過期(staleness)視窗。該分支通過的 checksum 是倖存者偏差,不是證明——這正是精靈密集的 SMB1 跑被升格進永久閘門集合的原因。負結果紀錄必須能就這種失效模式接受稽核:一個「位元精確」主張的強度,僅止於其背後工作負載的多樣性。

9.5 較小的死路,連同其母體數

在幾道大牆的周圍,是一圈較小的已關閉問題。每一個都不是靠直覺、而是靠一次實測母體數或一次實測損失關閉的;我們兩者都記錄,因為母體數正是防止這些問題被重新打開的東西。

表 9.3. 較小的死路。「母體數」是該技術所能觸及的節點、電晶體或事件的實測占比;母體冷或趨近於零的技術,無論其單事件價值如何都已關閉。

死路	實測母體數/結果	判定與機制
已出貨 lowering 之外的網表常數摺疊	定點下 76 個可摺疊類 / 111 顆電晶體 (0.41%);可串聯融合 0;重複/並聯元件 0;剩餘合計 <0.8%——全部是冷的(無事件)	由普查關閉:無事件 ⇒ 無收益,與健全性無關
弱/空乏(weak/depletion)電晶體旗標	母體為零——兩份原始 2A03/2C02 網表的每一列該旗標皆為 false [16];上拉改以線段屬性表示	模型表面,不是被錯過的優化
死端(未被觀察葉節點)跳過	診斷把 38% 的 BFS 工作標為「無閘極 + 未被觀察」;三個跳過變體全都弄壞 CPU;真正封閉的子集 <1%	偽陽性:葉節點狀態仍經由群走訪向外流動;「無閘極扇出」≠「無觀察者」
計數器快速路徑(維護活躍驅動者計數)	−6.0%	寫入路徑(每次電晶體翻轉)的執行頻率約為其節省之讀取的 10 倍;§9.4 定律的一個實例
DFS 群走訪(取代 BFS)	−2.28%(17/20),位元精確	順序無關(群是集合);BFS 之所以贏,只因結果緩衝區兼作佇列、使其免費——DFS 得為堆疊付費
彈出迴圈中的軟體預取;16 位元佇列;純局部性重編號	負/中性/中性(密度 −45%,牆鐘時間 ±0)	熱集常駐 L1;瓶頸是相依載入*延遲*而非 miss 數——位址揭曉得太晚,預取 [14] 無從遮掩任何東西
settle 深度封頂(「近似深尾」)	封頂 33(放棄約 0.58% 的 settle):1,000 hc 內發散;封頂 8:CPU 在第 1 幀崩潰(PC → $0040/BRK)	健全性失效,不是優雅的交換——settle 深度 7–45 是網表真正的關鍵路徑
剪除結構殘餘(無變化的 PullUp/Supply 彈出)	80.1% 的彈出是無變化(PullUp 42.0% + Supply 38.1%);靜態單通道子集 0.04%	以三條獨立途徑關閉:剖析器普查、P-5 家族(§9.4)、以及一次外部諮詢稽核——述詞本質上是執行期狀態
估算器導向的化約(記憶體細胞、匯流排網絡(bus fabric)、時脈相位浪費)	實測 17.3–25.0% / 3.8% / 0.3%,對上諮詢預測的 30–40% / 7% / 20–35%;平均 BFS 深度 1.13 ⇒「不存在巨型 BFS」	每一根被預測的槓桿在量測後縮水 2–100 倍;記憶體占比是真的,但將其行為化會違反逐節點位元精確

其中三項值得多說一句。*死端跳過*一項是本專案的典型偽陽性:38% 的母體看起來像帳上最大的一塊未動獎賞,而三個實作全都破壞了 CPU 執行,因為一個沒有閘極扇出的節點,仍會經由雙向通道走訪把狀態向外傳播——在 pass-transistor 網表中,觀察關係不是閘極驅動的扇出關係。*settle 封頂*一項以罕見的終局性關上了「近似計算」的門:深的 settle 尾巴不是可以在小誤差下截斷的數值精煉,而是電路的邏輯關鍵路徑;提早停止 settle 不會退化,而是出軌——不到一千個半週期就發散,激進封頂則直接硬崩潰。而*結構殘餘*一項是這道邊界最大的一面:全部事件工作的五分之四是確認無

變化的重新求值,它是事件驅動局部無知的直接代價(引擎無法便宜地知道某顆兄弟下拉仍撐著該節點),而每一種購買這份知識的機制——計數器、id、pinned 位元、靜態子集——都已被標價並且輸了。浪費與速度是同一個現象從兩個方向看。

9.6 反模式型錄

從個別結果退一步看,有五種失敗形狀在本專案的十三個月裡反覆出現,頻繁到值得命名。每一種都附上專案中最清晰的單一實例;多數有兩個以上。

1. **狀態快取謬誤(state-caching fallacy)**。「維護一個小的執行期事實,讓熱路徑可以跳過工作。」這個事實的寫入路徑幾乎總是比它加速的讀取路徑跑得更頻繁,而在延遲綁定的迴圈上,新增一條隨機存取寫入串流是所能加上最昂貴的東西之一。實例:活躍驅動者計數器(−6.0%)、整個 P-5 家族(§9.4)。本可及早殺掉兩者的診斷問句:*這個事實多常被寫入,相對於跳過多常被觸發?*

2. **微分支陷阱(micro-branch trap)**。「這個測試只是一個比較——它是免費的。」在熱走訪上執行的防護,是按走訪的頻率付費,不是按跳過的頻率;P-5z 每觸發一次跳過執行 9.4 次端點測試,而每次測試兩個餵分支的載入,恰好吃光了被省下的約 30 週期彈出。已出貨的區間剪枝(第 7 章)是建設性的反例:它之所以贏,正是因為把這類測試轉換成對常數的暫存器比較,刪掉載入而不是增加載入。

3. **小 N SIMD 妄想(small-N SIMD delusion)**。「圖有 14.7K 個節點、26.8K 顆電晶體——工作肯定寬到可以向量化或位元切片。」起作用的寬度是每步工作量分布,而這裡它是 1.13–2.25 個節點。位元平行 BFS(慢 156 倍、位元相同)是純粹的案例:一個 256 車道的前沿掃描,其中 77% 的時間只有兩條車道承載資訊。專案帳本裡有四條死路共享這個根因;檢查清單上的那一項,是在假設體量之前先量測*走訪大小分布*。

4. **編譯器微管理(compiler micromanagement)**。「把控制權從 JIT 手上拿走,它就會更快。」Native AOT 編譯在我們的實作中實測 −5.5%;一個可能的解釋是它放棄了執行期的動態 profile 導引優化(未單獨驗證)。強制激進內聯把一個方法養大到超過 JIT 自身的內聯門檻,實測 −6%,與下游內聯級聯被摧毀的解讀一致。這些效應活在編譯器/微架構的交互作用裡、不在演算法裡——只能逐組態實測,不能從原則推出。

5. **細粒度平行幻覺(fine-grained parallelism illusion)**。「兩顆晶片,兩條執行緒。」每次同步的工作量子(一個約 35 事件的波,約 0.7 µs)比分發它的成本還小,而語意禁止放大這個量子(§9.3)。最乾淨的雙執行緒切分慢 15 倍,是這個型的典型實例;教訓可推廣到任何「同步週期由模型決定、而非由實作者選擇」的分解。

五者共同的祖先是「以貌取因」的誤診:從容易看見的結構(圖大小、晶片數、「浪費」工作占比)推理,而不是從真正主宰的分布(每波事件數、每走訪節點數、每跳過寫入數)推理。本專案每一個出貨的勝利之前都有一次剖析器普查;本章每一個未經普查就先實作的條目都輸了。

9.7 邊界陳述

現在我們可以陳述引擎當前數字的意義,以及它周圍的東西。實測峰值是每秒約 136K 半週期(消融階段 S7,最佳 135,828;中位數 132,243),在一顆 Zen 2 核心上——每模擬一幀約 5.4 秒牆鐘時間,與真晶片的 42,954,552 hc/s 相差約 316 倍。在同一台機器、同一工作負載、同一單位下量測的公開電晶體級 NES/6502 模擬器家族中,這是最快的,以 2.5 倍領先其直系祖先(MetalNES [17] 約 54K、VisualNes [19] 約 24K;perfect6502 [18] 只模擬 6502,單位不可比;這些原始專案皆無內建的 hc/s 量測,數字來自我們對其原始碼的實驗性插樁,詳見 2.4 節)——而這個家族內比較才是該主張的公允範圍,不是一個泛用的最高級。

> **性質 9.1(單核邊界)**。在三項約束的合取之下——(i) 含浮接電容 tie-break 的完整開關級語意 [1],對 golden checksum 逐節點位元精確;(ii) 事件驅動求值;(iii) 單顆 CPU 核心——實測引擎位於一道 Pareto 前沿:對求值模型的每一種抽象化實測慢 1.0–84 倍(§9.2);每一種平行分解實測慢 15–156 倍(§9.3);每一種維護態事件剪枝實測在損益平衡點或其下,且有利可圖的那條腿可證明需要被維護的快照(§9.4);而剩餘的無變化工作(80.1% 的彈出)不存在高於 0.04% 的靜態、可位元精確剪除的子集(§9.5)。放鬆任何一項約束,改變的是工件本身,而不是這個工件的速度。

因此,什麼*會*移動這個數字,是具體的。第一,記憶體延遲與時脈頻率:瓶頸是 L1 常駐工作集上的相依載入延遲,因此更高 IPC、更低載入延遲的核心,可望把硬體世代的進步直接轉換成吞吐。第二,第 7 章那一類殘餘的識別碼空間與派發精煉——以個位數百分比計、可累積,且其上界遠不及一個數量級。而依本章的證據,什麼*不會*移動它:在這套語意下加更多核心;在任何已嘗試或已剖析的粒度上做編譯;以及在逐節點位元精確契約下做抽象化——98.9% 的可化約性結果是真實的,而且是*另一個*工件(供分析用的已驗證閘級模型)的正確地基,但它無法讓*這個*工件更快。

邊界地圖就是交付物。未來若有人要為這種規模的平坦、雙向、兩相 pass-transistor 網表實作開關級模擬器,他從本章繼承的,不只是「事件驅動直譯器是勝出的形狀」這個知識——Ulrich 的活動原理 [4],歷經四十年硬體變遷仍然成立——還包括每一條替代路線落敗的實測原因、關閉較小問題的母體數、五種需要自我檢測的自欺形狀,以及棋盤上剩下的、精確的合法走步。我們主張,這是一份不同於——而且比——引擎在任何排行榜上的名次更耐久的貢獻 [21]。

第 10 章 結論與未來工作

本專著要誠實回答一個狹窄的問題:對一台完整的商用遊戲主機——NES 的 2A03 CPU、2C02 PPU 與主機板膠合邏輯,lowering 之後約 14.7K 節點、26.8K 顆 NMOS 電晶體——做位元精確、事件驅動的開關級模擬,在一顆量產處理器的單一核心上、以受控語言——選用受控語言,部分正是為了實測檢驗一個廣泛存在於效能工程實務中的一般印象(我們未找到其正式文獻出處):受控語言不適合這種指標追逐的核心——且不犧牲模型可觀察語意的任何一個位元,究竟能跑多快?答案是量測出來而非論證出來的:在一顆 Zen 2 核心上約每秒 136,000 個主時脈半週期——約為矽實機速度的 0.3%,約為本引擎所承襲的優化 C++ 祖先的 2.5× [17],約為原始 JavaScript 模擬器逐字移植版的 5.6× [15, 19]。距即時(real time)所餘的約 316× 差距,依本專案自身負結果的證據,沿此路線無法弭平。本結尾章重述各項貢獻及其實測幅度,提煉我們認為能走出這份程式碼庫的教訓,平實陳述效度威脅,並以約束引擎本身的那些負結果來界定未來工作的邊界。

10.1 貢獻總結

第 1 章列出了五項貢獻。我們在此附上實測幅度重述之,順序依第 8 章消融階梯的歸因排列。下列所有數字均為同一台機器上同日、輪替交錯(round-robin interleaved)的中位數,每一跑都以全狀態 checksum 閘門把關,各階段皆自真實歷史 commit 重建。

1. 帶靜態安全證明的剪枝家族(P-1 至 P-4)。當最終的重新求值可被證明為 no-op 時,引擎僅憑載入期結構加上兩端點的即時狀態,就把入列本身壓掉。這個家族建立在一個結構性的安全汙染標記(safety taint)之上——無上拉的節點與 ForceCompute 通道元件被排除在外,因為它們的浮接 tie-break *就是*晶片的儲存機制——以及本專案最強的原創性主張:電容支配解除汙染(un-taint):電容嚴格小於其所有通道鄰居的節點,永遠不可能成為任何含有鄰居的群組中電容最大的成員,因此永遠贏不了 Bryant 的電荷分享 tie-break [1],因此經過它的同態(equal-state)合併可證明為無效。實測:這個家族刪掉約 21% 的全部節點重新求值;在階梯上貢獻 +26.4% 的階差(P-1)及再 +7.2%(P-2/P-3/P-4 加遮罩合併),且全階梯的已執行指令數下降 33%(每百萬半週期由 155.6B 降至 104.4B),IPC 維持在 2.1 至 2.4 之間。我們將其定位於始自 Ulrich [4] 的活動壓制(activity suppression)譜系之中;佇列前(pre-queue)、靜態證明的形式看來是其中的新元素。

2. 基數特化派發(R-1 與 B1)。兩個執行期證明:active 通道相連元件(CCC)的大小為一(所有 pass 閘目前皆關閉)或恰為二(僅一個 ON 閘,且其遠端只會繞回種子),各自就地(inline)解析,同時逐位元組複製一般走訪的順序敏感語意。實測:R-1 是階梯上單一最大的階差,+18.7%;B1 貢獻 +7.0%(S6→S7,同日交錯;與緊鄰的整理性 commit 分離歸因後約 +8.9%)。我們稱之為變體而非發明:IRSIM 早已動態量測 active 子網路的大小 [3],我們的貢獻在於早退(early-out)執行過濾器,以及對其位元精確義務的示範。

3. 自我捕捉初次觸碰資料重排。三遍式載入:先將每個節點分類到剪枝類別;再以類別為主鍵重建 id 空間,使熱迴圈的安全檢查變成對三個邊界的暫存器區間比較(每次 reset 都對重新計算的基準真相驗證,並依去優化防護(deoptimization guard)模式 [13] 配備安全退化回退);然後透過 settle 迴圈的一份冷儀器化副本,捕捉生產級聯 32,768 個半週期的真實初次彈出(first-pop)順序,以該順序作為區域性(locality)鍵再重建一次。實測:類別為主鍵的區間形式在階梯上貢獻 +4.2%,捕捉鍵再貢獻 +5.0%;與自檔案載入的實測 profile 正面對決,捕捉鍵以 +6.17% 勝出。這個機制完全落在

inspector-executor 與軌跡導向佈局(trace-driven layout)的譜系之中 [7, 8, 9, 10, 11, 12, 20];新元素在於模擬器在每次載入時於同一進程內捕捉*自身的*級聯——不需 profile 檔、適用任何 ROM、在構造上即免疫於工作負載漂移——而實證上令人意外之處在於它*為何*有效,即下文的第二個教訓。

4. 以物理為依據的零組態安全分類。儲存、匯流排與記憶體節點從不以名字辨識。上拉旗標、通道元件 union-find、受驅動接腳排除與電容比較,依其物理本質辨識它們;儲存會自動把自己排除在不安全的剪枝之外,因為在這種製程裡,浮接節點上的大電容*就是*儲存的本體。我們將此呈現為實作紀律而非研究主張——自動的電晶體層邏輯抽取與以結構圖樣比對為導向的子電路識別,在文獻中已有成熟家族 [5, 6];我們的辨識依據是物理性質(上拉旗標、電容比較、通道元件)而非元件庫圖樣——但這正是剪枝家族從 2A03 延伸到 2C02 與主機板時,完全不需逐晶片組態的原因。

5. 一張可證偽的效能邊界地圖。含硬體計數器的九階段消融(第 8 章)、每一條抽象化與平行化路線的實測失敗(第 9 章),以及「維護執行期事實」的地板(P-5 家族:毛利 +13.76%,最佳淨值 −6.84%;其中健全的零維護腿實測為中性,有價值的腿則可證明需要解析時(resolution-time)快照而非即時狀態)。這張地圖的標題級結構事實:導通群平均 1.13–1.4 節點;80.1% 的彈出(pop)為無變化,且佔主導的 PullUp/Supply 殘餘沒有靜態子集(三個獨立方向確認);依賴圖的 94% 是單一個雙向強連通元件(SCC);以及波內順序屬於可觀察語意。

> **標題級結果**。S0→S7 累計:單核中位數吞吐 +94.6%,每一階段皆位元精確;計數器顯示引擎受限於相依載入*延遲*而非 miss *數量*:重排把 L1d miss 從 27.3 砍到 13.2 MPKI 而牆鐘時間只動個位數,且 L1i miss 從未超過 0.30 MPKI——這正是「在此區間直譯勝過編譯」的計數器層級解釋。

表 10.1. 以實測幅度重述各項貢獻(同日消融,400k hc,中位數)。「變體」之判定依第 3 章的先行研究稽核。

貢獻	實測幅度	相對先行研究之地位
P-1 同態(same-state)入列剪枝(+安全汙染標記)	+26.4% 階差	可信為原創(佇列前靜態壓制)
P-2 隔離剪枝;P-3/P-4 電容支配解除汙染(un-taint)	+7.2% 階差;家族刪掉約 21% 的重新求值	P-3/P-4 為最強主張;最近鄰為 IRSIM 的尺寸量測實務 [3]
R-1 動態單例	+18.7% 階差	變體(在動態 CCC 尺寸量測 [3] 之上的基數早退)
B1 成對路徑	+7.0% 階差(分離歸因後約 +8.9%)	變體(執行期模板匹配)
類別為主鍵重編號 + 區間剪枝	+4.2% 階差	建立於 [10, 12] 之上的工程;去優化防護依 [13]
自我捕捉初次觸碰鍵	+5.0% 階差;對載入 profile +6.17%	同進程自我捕捉看來為新;譜系 [7, 8, 9, 11, 20]
邊界地圖 + 負結果	峰值約 136K hc/s;距即時差距 ≈316×	方法論貢獻

10.2 可推廣的教訓

10.2.1 事件壓制需要證明,而非啟發式

每一個最終出貨的壓制機制,都帶著一個基於解析語意的結構性論證;每一個倚賴「看似合理的啟發式」的機制都失敗了,而且敗得很大聲。P-1 最初的天真版本產生了黑畫面:它剪掉了經過浮接儲存節點的同態合併,而那些節點的 tie-break 結果恰恰取決於被壓掉的那次合併。支配旁路(dominant-bypass)家族的零維護變體,有一條腿的證明是健全的(且實測利潤為零),另一條腿的直覺寫法——讀取支配節點的即時狀態——在 100k 半週期內就發散了,因為底層的證明綁定的是*解析時*的值,而非當下的值。至於所有近似中最誘人的不足安定(under-settling),即使只截斷最深的 0.58% 安定鏈,也會在一千個半週期內讓被模擬的 CPU 出軌。對於狀態機制建立在 tie-break 上的事件驅動模擬器,普遍的教訓是:不存在「大致安全」。作為補償的仁慈是,這類系統的失效是災難性且立即的,而非隱微的,這使得 checksum 閘門成為有效且廉價的證偽器——本專案的每一個候選壓制,都在幾分鐘內被它接受或摧毀。

10.2.2 佈局的價值可能在順序,而非密度

不規則程式碼的資料佈局文獻是圍繞 miss 數組織起來的 [8, 10, 11, 12, 14],而我們自己的計數器顯示了這種框架為何可能誤導。類別為主鍵的重排把 L1d miss 率砍半(27.3 降至 13.2 MPKI;AMD 原生 miss 比率由 5.6% 降至 3.2%),牆鐘收益卻只有 +4.2%;接著自我捕捉鍵再增加 +5.0%,而 miss 計數器幾乎紋風不動。對照實驗更為尖銳:在剪枝關閉下取得的捕捉鍵,達到與獲勝鍵相同的 cache line 足跡(每半週期 110 對 105.9 條 NodeInfo line,對照盲鍵的 118.4),卻*毫無*收益;而在剪枝開啟下取得的捕捉則收得 +6.17%。等密度、不同血統、相反結果。佈局的價值在於它吻合已剪枝生產級聯的*順序*——縮短並重疊亂序核心必須串行走過的相依載入鏈——而不在於它把 cache line 塞得多滿。對於熱集常駐 L1、受延遲鏈綁定的指標程式碼,去優化佈局文獻所點名的那個指標是一種範疇錯誤;目標函數必須是那條串行鏈。

10.2.3 量測紀律是第一級的交付物

三項實務讓本專著中的數字值得信任。第一,量測一律不鎖頻,以同日多輪 round-robin 交錯、單一 benchmark 進程、中位數回報控制變異:這台機器的吞吐在涼機日與熱透日之間變動約 10%,因此未在同一場次內交錯的跨階段比較,量到的是天氣。第二,對每一次 benchmark 跑都做 checksum 閘門(階梯全部 72 跑通過),這把每一個效能實驗都轉化為回歸測試。第三,以歷史重建做歸因:當 S6→S7 的階差看起來過大時,與緊鄰的整理性 commit 分離歸因,把 B1 隔離在約 +8.9%,且分布完全分離。親和性釘選並關閉 EcoQoS 把跑間變異減半(cv 由 1.5% 降至 0.94%),是我們背書的衛生工具。

10.2.4 對變體誠實,成本低而有回報

先行研究稽核的結論是:我們的解析語意在功能上等同去掉 X 態的 MOSSIM II [1];群走訪就是文獻所稱的 CCC 求值;而 R-1、B1 與區間剪枝是已知實務的變體 [2, 3, 13]。把這件事說白不花任何成本,卻換來專注:力氣集中在稽核找不到對應先例的兩個主張(佇列前剪枝與支配解除汙染)以及邊界地圖上,而不是重新推導並過度宣稱四十年的開關級工程。我們向任何同類專案推薦這項練習——在論文動筆之前,先寫一張對照文獻的判定表。

10.3 效度威脅

單一機器。本專著的每一個數字都量自同一顆 AMD Ryzen 7 3700X(Zen 2)。階梯對各技術的*排序*很可能是穩健的;階差大小則是這顆 Zen 2 上的事實,而延遲綁定的診斷應在 load-to-use 延遲與亂序視窗大小不同的核心上重新驗證。

單一工作負載家族。吞吐與事件統計來自 `full_palette.nes`,並以一段 10M 半週期、精靈密集的《超級瑪利歐兄弟》執行作為對抗性正確性門。兩者僅涵蓋一台主機、兩張源自晶粒的 NMOS 網表、以及 NROM 卡匣。結構性發現——群組大小分布、80.1% 的無變化占比、單一巨大 SCC——是這些網表的性質;我們預期在其他 Visual6502 家族的晶粒上會有類似形狀 [15, 16],但尚未量測。

計數器方法論。逐階段硬體剖面使用 ETW PMC 取樣,負擔約 5%;因此對那些跑我們只回報比率(MPKI、miss 比率、IPC),不回報絕對吞吐。本機沒有可運作的最後層級快取(last-level cache)計數器來源,故 L3/DRAM 行為是由約 15 KB 的熱集與 3.2% 的 L1d miss 比率推論而得,並非實際觀測。這個推論令人安心,但它終究是推論。

位元精確是 checksum 見證的,不是證明的。300k/400k/1M 半週期的全狀態 FNV-1a checksum 加上 10M 的 SMB1 門,對所測試過的輸入而言是強證據;但它不是對所有程式的等價性證明。不足安定實驗顯示此系統中的發散又快又響,這提高了我們對「這些門能抓到真實錯誤」的信心,但限制仍然成立。

10.4 未來工作,誠實設界

轉向分析價值。最站得住腳的下一筆投資不是速度。量得晶片約 98.9% 可化約為邏輯的同一套抽取機制,可支撐對運行中矽晶片的閘級*視圖*:具名暫存器監看點(網表本來就命名了 `a0..a7`、`pcl0..pcl7` 等及其同類 [16])、Φ1/Φ2 時脈相位圖、匯流排競爭診斷,以及抽取邏輯的 Verilog/Graphviz 匯出。這些都不與位元精確引擎競爭;它們全是只有引擎的保真度才能獨力成就的價值,也是本專案的邊際工時如今最划算之處。

行為式 OAM 分叉,在契約之外。實測結構剖析把全部事件彈出的約 25% 歸因於記憶體單元流量(OAM 與調色盤預充,時脈驅動)。把那些單元換成行為式陣列會刪除內部節點,因而違反定義本引擎的「位元精確、全狀態 checksum」契約——這正是我們明文禁止的抽象化風格。作為一個獨立標示的分叉,它是正當的未來工作,其硬上限由該事件占比決定:約 1.3×,不會更多。我們現在就把上限說出來,讓這個分叉日後無法被過度推銷。

硬體會移動地板。計數器說,引擎把時間花在 L1 常駐工作集上的相依載入延遲,而指令擷取實質免費。這正是未來硬體不需任何軟體修改就能幫上忙的唯一區間:更短的 load-to-use 延遲、更深的亂序視窗、更高的持續 IPC 都直接換算成速度。反過來,實測的負結果——多執行緒慢 15×、GPU 慢 10.7×、編譯慢 3–6×、位元平行慢 156×——說明更多核心、更寬的向量與加速器幫不上忙,因為平均 1.13 節點的群組沒有可批次化的規則性,且每事件預算(約 20 ns)比一次跨核 cache line 搬運還小。

重新量測我們自己的天花板。本專案發表過的每一個吞吐天花板——80K、100K、120K、127K 半週期每秒——後來都倒了,而且每一個都是倒在量測之下,不是倒在論證之下。仍然成立的結構性論證(沒有靜態子集的無變化殘餘;巨大 SCC;順序即語意)屬於另一種類:它們是「路線已關閉」的理由,從三個獨立方向驗證,而不是外插的趨勢線。我們的最終立場因此一分為二:我們有信心斷言約

316× 的即時差距*沿此路線*不可達;但對「任何特定的 hc/s 數字就是終點」,我們毫無信心斷言。現行政策——累積小型、checksum 閘門把關、配對量測的勝利;不預先以「太小」否決任何事——持續有效。

10.5 可得性

本專著量測的一切皆已公開 [21]。位於 `github.com/erspicu/AprVisual` 的儲存庫包含 C# 引擎(`src/AprVisual.S1/`)、產出第 9 章負結果的已棄用實驗分叉、自足的 benchmark 套件,以及量測腳本。配套網站 `erspicu.github.io/AprVisual` 收錄文章系列與公開排行榜,其中含有本文所引用的上傳跑分。第 8 章背後的原始數據——逐跑的消融階梯與逐階段的計數器摘要——隨手稿原始檔一併提供(`MD/paper/data/`),連同在其他硬體上重建並重新閘門化階梯每一級所需的逐階段 commit 識別碼與 golden checksum。源自晶粒的網表本身屬於其抽取者所有 [15, 16],依其上游授權以引用而非隨附(vendored)的方式使用。我們希望這張邊界地圖被以它被建造的方式使用:作為一組可證偽的主張,每一條都附著一行可以證明它錯誤的命令列。

致謝

在本工作——引擎、量測與本稿件——的準備過程中,作者使用了生成式 AI 工具(主要為 Anthropic 的 Claude,經由 Claude Code 環境),協助實作與重構 C# 引擎程式碼、 編排 benchmark 與硬體計數器量測、分析結果、調查相關文獻,以及起草與潤飾本專論英文與 繁體中文兩個版本的文字。使用這些工具之後,作者已徹底審查、測試並依需要編輯了內容—— 每一個被採納的引擎修改都受附錄 A.3 的位元精確 checksum 閘門把關,每一個效能數字 都是原始數據收錄於附錄 B、C 的實際量測——並對本出版品的內容、科學準確性與原創性 負完全責任。

本聲明即 arXiv 生成式 AI 語言工具政策所要求的揭露(文字生成工具的重大使用必須在 作品中報告),亦為 IEEE 投稿政策(PSPB 作業手冊)對後續期刊或會議投稿所要求的致謝揭露。 就 IEEE 政策要求的「章節級指明」:上述程度的 AI 輔助——由作者主導的素材與量測數據出發 的全文起草,再經作者審查與編輯——適用於本專論的*全部*章節與附錄;SVG 圖同樣由 AI 依量測數據起草、經作者審查;本稿件沒有任何部分例外。依循兩者的共同規範,任何 AI 系統 皆非本工作的作者或共同作者;著作者身分與問責完全屬於人類作者。

本工作站在開放矽考古社群的肩膀上:Visual 6502 專案 [15]、Quietust 的 Visual 2A03/2C02 網表 [16],以及 MetalNES [17]、perfect6502 [18]、VisualNes [19] 模擬器。

參考文獻

文獻條目保留英文原文(學術慣例);全部條目均已逐一查證存在(2026-06-13,IEEE Xplore / ACM DL / Google Patents / 各專案公開頁面)。

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 一引擎、benchmark 套件與公開排行榜:github.com/erspicu/AprVisual; 系列文章:erspicu.github.io/AprVisual.

附錄 A. 可重現性

A.1 機器與執行方式

本專論所有量測均在同一台機器上完成:AMD Ryzen 7 3700X(Zen 2,8C/16T,每核 32 KB L1d + 32 KB L1i),Windows 11, 各階段使用的 .NET 10/11 如表所列。工作負載:`full_palette.nes`(NROM;社群開源的 NES 測試 ROM,顯示完整 NES 調色盤的靜態畫面,取自公開的 nes-test-roms 測試集),整機模擬。benchmark 進入點:

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

時脈不鎖頻;以同日多輪 round-robin 交錯與中位數統計控制變異(方法見 §8.1), 任一時刻只有一個 benchmark 進程在跑。選用的 `--pin` 參數(核心親和性 + High 優先權 + 關閉 EcoQoS;不動頻率)可將跑次間變異減半且位元精確, 但為了與排行榜可比,正文數字均不帶它量測。

A.2 階段對應 commit

表 A.1. 第 8 章消融階段對應的版本庫 commit、runtime 與各階段新增內容。歷史 commit 在隔離的 git worktree 中原樣重建。

階段	Commit	Runtime	新增
S0	`a80dab4~1`	.NET 10	基線分支(靜態 fast path)
S1	`a80dab4`	.NET 10	R-1 動態單例
S2	`ed8c457`	.NET 10	P-1 同態剪枝
S3	`6bdc25b`	.NET 10	P-2/P-3/P-4 + PruneMask 整併
S4	`00ab4fa`	.NET 10	條件重排 + supply-skip 摺疊
S5	`51e046d`	.NET 11	區間剪枝(類別為主鍵重編號)
S6	`3e4a571`	.NET 11	自我捕捉初次觸碰鍵
S6b	`a553d38`	.NET 11	檔案模式移除(歸因對照;中性)
S7	`2749105`	.NET 11	B1 成對路徑

A.3 黃金 checksum(位元精確閘)

表 A.2. 每個被採納的修改都必須重現的全狀態 FNV-1a checksum。checksum 透過排列映射以「原始節點 ID 順序」迭代,因此構造上與記憶體佈局無關。

閘	工作負載	長度	Checksum
主要	full_palette.nes	300,000 hc	`0x794A43ABDF169ADA`
階梯閘	full_palette.nes	400,000 hc	`0x9174E19D961CB6E5`
長程	full_palette.nes	1,000,000 hc	`0x6D4CCBCE2E9CD599`
精靈密集	Super Mario Bros.	10,000,000 hc	`0x7DE79A1BD3D8536F`

全部 72 次消融跑次(9 輪 × 8 階段)均在 400k hc 通過 checksum 閘;沒過閘的跑次視為「量到了另一個程式」而捨棄。

附錄 B. 消融原始數據

表 8.1 與圖 8.1 背後的完整 9 輪 × 8 階段矩陣(hc/s,每跑 400,000 hc, 不鎖頻,每輪內 round-robin 順序)。各欄中位數即正文引用的階段結果。數據檔保存於版本庫 `MD/paper/data/ablation_runs.csv` [21]。

表 B.1. 消融階梯原始數據:9 輪 × 8 階段(hc/s,每跑 400,000 hc,皆通過 checksum 閘)。

輪	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
中位數	**67,955**	**80,657**	**101,964**	**109,841**	**112,887**	**117,671**	**123,545**	**132,243**

附錄 C. 硬體計數器原始數據

ETW PMC 取樣(PerfView 收集、xperf 萃取),每階段 1,000,000 hc,樣本過濾至模擬器 進程。取樣間隔:指令與週期為 1,048,576 事件/樣本;cache miss 與分支預測錯誤為 65,536 事件/樣本。估計事件數 = 樣本數 × 間隔;§8.4 的每百萬 hc 正規化與 MPKI 比率均由此 導出。數據檔保存於 `MD/paper/data/pmc_summary.csv` [21]。

表 C.1. PMC 取樣原始數據(每階段 1,000,000 hc;事件估計值 = 樣本數 × 取樣間隔)。

階段	計數器	樣本數	估計事件數
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

限制:取樣式歸因;Windows 通用 `DcacheAccesses` / `CacheMisses` 計數源在本 CPU 上無輸出 —— 第二輪改用 AMD 原生源(`DCAccess` / `ICFetch` / `ICMiss`)補上 §8.4 引用的存取次數分母;本機沒有可用的最後階快取(LLC)計數器。