Label-Efficient and Bottleneck-Exposing CPU Performance Modeling with Architectural Invariants

Weihang Li **Amir Yazdanbakhsh**† **Krishbin Paudel**‡ **Santosh Pandey**‡* **Hang Liu***
Rutgers University †Google DeepMind ‡University of South Florida
wl730@scarletmail.rutgers.edu

***Abstract*—Machine learning (ML) CPU performance models offer orders-of-magnitude speedups over traditional cycle-level simulators, yet face two fundamental barriers to practical adoption. First, generating labeled training data requires thousands of hours of computationally expensive cycle-level simulation, severely limiting early-stage design exploration. Second, existing models predominantly predict a single, scalar CPI metric. This black-box output obscures pipeline bottleneck dynamics, leaving architects without the actionable guidance needed to optimize microarchitectural designs. We argue these two barriers are intrinsically linked: reducing complex pipeline physics to a single scalar forces models to rely on prohibitively large, expensive labeled datasets to implicitly infer hidden microarchitectural interactions. To address this, we propose PRISM, an ML-based CPU performance modeling framework. PRISM replaces scalar CPI prediction with a structured Multi-Stage CPI Stack (MSCS), a multi-dimensional representation capturing bottleneck proportions across key pipeline stages. This output not only natively exposes actionable bottleneck attributions, but mathematically encodes fundamental architectural invariants: instruction throughput conservation and monotonic stall-pressure propagation. We explicitly embed these invariants into the training process as physics-constrained losses. Because these rules dictate pipeline structure independent of simulation labels, they provide a powerful supervision signal on raw, unlabeled instruction traces. This enables semi-supervised learning, allowing the model to leverage abundant unlabeled trace data alongside a small set of expensive cycle-level simulation labels during training. Our evaluation across SPEC benchmarks shows PRISM achieves a 3.1% average CPI error, matching state-of-the-art scalar predictors while uniquely unlocking detailed bottleneck decomposition. By leveraging semi-supervised learning, PRISM matches full-label accuracy using 40% fewer expensive cycle-level simulation labels.**

I. Introduction

Architecture performance modeling remains a cornerstone of computer architecture research and microprocessor design. Historically, the architecture community has relied on three primary paradigms for performance evaluation. Each paradigm offers distinct trade-offs among simulation speed, accuracy, and explainability. First, high-abstraction analytical models offer rapid evaluation [3], [6], [7], [9], [11], [13], [17], [24]. However, these models often struggle to accurately capture the complex microarchitectural interactions of modern out-of-order (OoO) pipelines. Second, cycle-level simulators like gem5 provide high-fidelity modeling [2]. Unfortunately, they suffer from slow execution speeds that hinder large-scale design space exploration. Third, ML-based performance prediction has emerged as an essential tool for early-stage design exploration [12], [15], [16], [18]. Rather than replacing cycle-level simulators, ML models deliver ultra-fast evaluation speeds to help architects navigate massive design spaces.

Traditional analytical models and cycle-level simulators have largely reached their fundamental simulation speed and/or accuracy limits. However, despite their extraordinary evaluation speeds, ML models have yet to achieve widespread practical adoption because microarchitecture engineers struggle to trust predictions they cannot physically verify. To transition ML-based performance modeling from an academic proof-of-concept into a trusted, industrial-grade engineering tool, it must provide actionable feedback that architects know how to treat and analyze. Therefore, our work focuses on differentiating ML-based performance modeling by directly addressing the fundamental limitations that obscure microarchitectural visibility. The state-of-the-art ML-based performance model, Concorde [16], represents a meaningful advance in pure predictive capability. Concorde achieves 2-3% in-distribution prediction accuracy. It can also adapt to unseen workloads with only 32 to 2048 samples. Furthermore, by compressing a workload's instruction trace into a fixed-length feature vector, it evaluates performance in constant time regardless of the program's execution length. Despite these strengths, Concorde and prior ML-based models [16], [18], [25] share two limitations that constrain their real-world utility.

[Challenge 1] Limited bottleneck attribution with CPI. Existing ML-based performance models predominantly predict only a scalar CPI performance metric. This black-box output tells architects how fast a design executes, but it provides limited insight into pipeline bottleneck dynamics [10], [21]. In practice, architects optimize microarchitectures by first identifying the primary performance bottleneck and then selectively tuning the responsible resources. A single, scalar CPI prediction fundamentally cannot provide this necessary isolation. Therefore, when a learned model outputs only an opaque CPI metric, architects have no way to determine whether the root cause of a performance shift is frontend starvation, backend contention, or branch mispredictions. As a result, while the ML-based prediction itself may be nearly instantaneous, it fails to solve the broader engineering challenge; architects are still forced to fall back on multiple iterations of manual profiling and cycle-level telemetry analysis just to interpret the model's output. Because of this diagnostic blind spot, architecture optimization guided solely by scalar CPI degenerates into an

*Equal corresponding authors: spandey1@usf.edu, hl1097@soe.rutgers.edu

uninformed parameter search. To truly accelerate design space exploration, an effective learned model must expose both the overall CPI and its underlying bottleneck structure, ensuring its predictions provide actionable guidance.

[Challenge 2] The high computational cost of label acquisition. The efficiency of ML-based models is fundamentally tied to the availability of massive labeled datasets. Generating these labels, which are the ground-truth performance metrics (e.g., CPI) extracted from a detailed simulator, requires thousands of hours of computing time [16]. This high computational cost makes early-stage microarchitectural exploration prohibitively expensive [1], [20]. Moreover, this massive labeling cost recurs every time architects retarget the learned models to a new microarchitecture. An intuitive remedy would be to somehow leverage cheaply obtainable unlabeled data, such as raw instruction traces (e.g. DynamoRIO [4]) processed through fast analytical models. However, a naïve application of this unlabeled data is ineffective. To extract actual value from these traces, the learning process requires meaningful architectural signals that are decoupled from the exhaustive long tail of cycle-level simulation. Unfortunately, scalar CPI prediction lacks a structural invariant to serve as this necessary supervision signal on unlabeled samples.

We argue that these two fundamental barriers share a common cause. *Predicting a single scalar CPI discards the structured, physically grounded behavior of the CPU pipeline. Furthermore, expecting a learned model to implicitly infer the entirety of these complex microarchitectural interactions through the narrow lens of a single metric creates an intractably difficult learning objective.* As such, the guiding design principle of our work is to preserve the high predictive accuracy and architectural simplicity of state-of-the-art models like Concorde [16], restructuring the prediction target to be architecturally grounded. To mitigate these limitations, we propose PRISM, an ML-based performance modeling framework that deliberately anchors the prediction task in a structured Multi-Stage CPI Stack (MSCS) instead of relying on a scalar CPI. The MSCS representation serves simultaneously as a practical prediction target for architects, a bottleneck-exposing diagnostic interface, and a physics-constrained loss [19] carrier. This multifaceted formulation yields a synergistic benefit, uniquely unlocking a unified solution to both bottleneck attribution and label efficiency barrier in ML-based CPU performance modeling. To realize this solution, PRISM introduces the following contributions:

1 First, we structure the prediction output as a streamlined MSCS vector. Building upon prior work [8], we simplified the MSCS representation to represent performance as a 4×4 matrix of core bottleneck attributions. Specifically, the columns of this matrix correspond to four critical pipeline stages: Decode, Dispatch, Issue, and Commit. The rows map to four distinct bottleneck states: Base, Frontend Pressure, Backend Pressure, and Squash Pressure. By explicitly normalizing the rows of this matrix, the ML model outputs directly bottleneck-exposing microarchitectural attributes rather than abstract latent variables.

2 Second, we extract architectural invariants from the MSCS to create physics-constrained losses. The MSCS mathematically encodes two universal physical laws of the pipeline. The first is CPI consistency, which ensures instruction throughput conservation across all stages. The second is bottleneck visibility monotonicity, which captures monotonic stall-pressure propagation trends. We refer to these as "physics-based" constraints because they enforce universal conservation laws of instruction flow (e.g., instructions entering the pipeline must ultimately commit or flush), independent of any specific architectural design parameter. We embed these invariants directly into the model training process as physics-constrained losses. As such, this process generates the meaningful supervision signals necessary to *learn from unlabeled data*, entirely decoupled from cycle-level simulation.

3 Third, we use these physics-constrained losses to unlock semi-supervised learning with unlabeled data. Because these physical constraints are derived from the pipeline structure itself rather than simulation labels, they provide a strong supervision signal on *unlabeled* instruction traces. This formulation natively enables semi-supervised training, where both labeled and unlabeled data are jointly used in each training batch. For labeled samples, the model is supervised by both ground-truth simulation labels and physical constraints losses. For unlabeled samples, the model is supervised solely by the physical constraints, which enforce physically consistent predictions without requiring any ground-truth simulation labels. This allows the model to learn from abundant, inexpensive trace data while relying on only a small set of expensive cycle-level simulation labels.

Summary of results. Our evaluation across a range of SPEC benchmarks shows that PRISM achieves a 3.1% average CPI error. This matches the accuracy of state-of-the-art scalar predictors while uniquely unlocking detailed bottleneck decomposition. Crucially, by leveraging this semi-supervised learning on cheaply generated unlabeled traces, PRISM matches full-label accuracy using merely 40% fewer cycle-accurate simulation labels. Furthermore, we use this fast and accurate predicted MSCS to guide CPU design space exploration. The detailed bottleneck feedback of the predicted MSCS guides design space exploration to a superior Pareto front. In addition, we use this MSCS representation to prune redundant parameter interactions in Shapley value calculations. Our MSCS-guided pruning achieves up to a $20\times$ reduction in evaluation time, while remaining highly correlated (Spearman $\rho = 0.953$) with the full Monte Carlo baseline.

II. PRISM DESIGN

Figure 1 presents the PRISM workflow, structured around three design points that build on one another. First we replaces the scalar CPI output with a 16-dimensional MSCS vector spanning four pipeline stages and four bottleneck states, establishing the structured prediction target that the remaining designs rely on. Then we extracts two architectural invariants from this MSCS structure, CPI consistency ($\mathcal{L}_{\text{const}}$ and bottleneck monotonicity $\mathcal{L}_{\text{mono}}$), and embeds them as

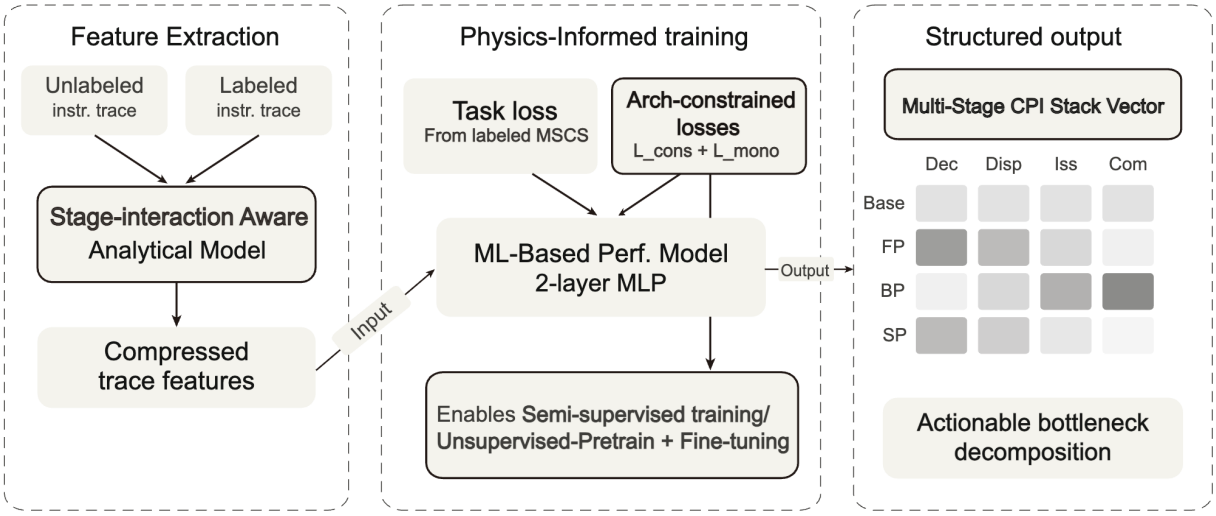

Fig. 1: Overview of PRISM. An instruction trace is pre-processed and passed through a stage-interaction-aware analytical model to produce a fixed-length feature vector. An MLP predicts the 16-dim MSCS vector, trained with architecture-constrained losses. PRISM supports three training paths: (1) supervised learning with labeled data only, (2) semi-supervised learning using both labeled and unlabeled data jointly(PRISM default), and (3) unsupervised pretraining on unlabeled data followed by supervised finetuning. At inference, the predicted MSCS provides both CPI and per-stage bottleneck attribution.

physics-constrained losses that provide a supervision signal on unlabeled data, enabling both semi-supervised learning and unsupervised pretraining. Finally we introduces a stage-interaction-aware analytical model that transforms a raw instruction trace into a fixed-length feature vector encoding per-stage pressure distributions, capturing inter-resource pipeline dynamics that Concorde's per-component feature extraction inherently discards.

A. MSCS Vector

We build upon the original MSCS [8] with one light extension and a simplification to build an MSCS vector. First, we add a Decode stage to the original three-stage stack (Dispatch, Issue, Commit), capturing frontend stalls at the earliest observable point in the pipeline before they propagate downstream. Second, we consolidate the original fine-grained per-component CPI breakdown into three unified pressure buckets: Frontend Pressure (I-cache/ITLB misses), Backend Pressure (ALU contention, D-cache misses, and data dependencies), and Squash Pressure (branch mispredictions). This consolidation reduces the complexity of the prediction target without sacrificing the bottleneck resolution architects need, since the dominant design decisions map cleanly to these three categories.

1) MSCS Vector Formulation: The MSCS vector ($\mathbf{V}_{\text{MSCS}}$) comprises 16 elements organized as four stage-level proportion vectors:

$$\begin{aligned} \mathbf{V}_{\text{MSCS}} &= [\mathbf{v}_{\text{Decode}},\, \mathbf{v}_{\text{Dispatch}},\, \mathbf{v}_{\text{Issue}},\, \mathbf{v}_{\text{Commit}}]^T \\ \mathbf{v}_s &= [Base_s,\, FP_s,\, BP_s,\, SP_s], \\ \text{s.t.} \quad & Base_s + FP_s + BP_s + SP_s = 1, \\ & s \in \{\text{Decode, Dispatch, Issue, Commit}\} \end{aligned} \tag{1}$$

Each stage s is decomposed into four normalized slot proportions. $Base_s$ captures productive utilization: the fraction of the pipeline capacity used to advance instructions. FP_s (Frontend Pressure) captures stalls caused by I-cache/ITLB misses, reflecting instruction supply bottlenecks upstream of the decode stage. BP_s (Backend Pressure) aggregates all execution-side stalls, including D-cache misses, execution-unit contention, and data dependencies. SP_s (Squash Pressure) captures pipeline flushes due to the branch mispredictions. Because every slot in every cycle is classified into exactly one category, the four proportions at each stage sum to unity, making each element is directly interpretable as a fractional bottleneck attribution. This constraint is satisfied by construction in Eq. 3: dividing raw slot counts by the total available slots ensures that the proportions sum to unity before any model sees the data. Each element of ($\mathbf{V}_{\text{MSCS}}$) is directly interpretable as the fraction of pipeline capacity consumed by a specific bottleneck type at a specific stage, the same quantities an architect would read from a hardware performance counter. At inference, PRISM predicts ($\mathbf{V}_{\text{MSCS}}$) in a single forward pass, providing both scalar CPI and per-stage bottleneck breakdown without any post-hoc analysis.

2) MSCS Vector Accounting Algorithm: Following Eyerman et al. [8], performance is quantified by tracking slot utilization every cycle. For a given stage, the total available capacity per cycle is $W_{\min}$ slots, where $W_{\min}$ is the minimum width across all pipeline stages:

$$\begin{aligned} W_{\min} = \min(&W_{\text{fetch}},\, W_{\text{decode}},\, W_{\text{rename}}, \\ &W_{\text{dispatch}},\, W_{\text{issue}},\, W_{\text{commit}}) \end{aligned} \tag{2}$$

Using $W_{\min}$ uniformly ensures that the base slot count is identical across all stages, which is a necessary precondition for

the CPI consistency invariant formalized in Section II-B. Algorithm 1 presents the per-cycle slot accounting logic applied at each pipeline stage. We derive the per-cycle accounting logic from Eyerman et al. [8] with two modifications that tailor the representation for ML-based prediction.

Algorithm 1 presents the per-cycle accounting logic at each stage. Table I defines the per-stage conditions for detecting frontend and backend stalls. The algorithm accumulates raw slot counts $Base_s^{raw}$, FP_s^{raw}, BP_s^{raw}, SP_s^{raw} over the execution window, which are the absolute number of slots sampled in each category. Following the original MSCS paper, residual slots ("other" parts in [8]) are attributed to FP when they occur within the front-end pressure condition (e.g., fetch bubbles), and to BP otherwise (e.g., structural stalls).

Algorithm 1 Generalized MSCS Accounting Algorithm for Stage s

Require: Pipeline stage $s \in$ {Decode, Dispatch, Issue, Commit}, minimum width $W_{\min}$, correct-path instructions processed this cycle U, I-cache/ITLB miss flag I_{miss}, branch misprediction flag B_{miss}

```
full_slots ← W_min
Base_s^raw ← Base_s^raw + U
if U < full_slots then
    lost_slots ← full_slots − U
    if ObserveFEPressure(s) then
        if ICache_miss then
            FP_s^raw ← FP_s^raw + lost_slots
        else if B_miss then
            SP_s^raw ← SP_s^raw + lost_slots
        end if
    else if ObserveBEPressure(s) then
        BP_s^raw ← BP_s^raw + lost_slots
    end if
end if
```

TABLE I: Per-Stage FE/BE conditions (DDQ: Decode-to-Dispatch Queue)

Stage (s)	ObserveFEPressure(s)	ObserveBEPressure(s)
Decode	Fetch buffer empty	DDQ full
Dispatch	DDQ empty	ROB or RS full
Issue	RS empty	RS not empty and have non-ready inst
Commit	ROB empty	ROB head inst not done

Over an execution window of N_{window} cycles, let $S_{total} = N_{window} \times W_{min}$ be the total available slots. We convert the raw counts to the MSCS proportion vector $\mathbf{v}_s = [Base_s, FP_s, BP_s, SP_s]$ (Eq. 1) by normalization:

$$\mathbf{v}_s = \left[\frac{Base_s^{raw}}{S_{total}}, \frac{FP_s^{raw}}{S_{total}}, \frac{BP_s^{raw}}{S_{total}}, \frac{SP_s^{raw}}{S_{total}}\right] \tag{3}$$

Since $U \leq W_{\min}$ every cycle and all lost slots are attributed to exactly one pressure category, the four proportions sum to unity by construction.

The scalar CPI follows directly from the base proportion at any stage s:

$$CPI = \frac{1}{Base_s \cdot W_{min}} \tag{4}$$

Because $W_{\min}$ is shared and every slot is classified exactly once, Eq. 4 evaluates to the same value at every pipeline stage. This cross-stage CPI consistency is a hard constraint on instruction throughput, and we exploit it as a label-free supervision signal in Section II-B.

B. Architecture-Constrained Loss Function

The MSCS vector introduced in Sec. II-A is not an arbitrary structured output: it encodes two physical conservation laws of the OoO pipeline that hold for any correct prediction, independent of workload or microarchitectural configuration. We formalize these laws as differentiable loss terms and embed them directly into the training objective. Because both constraints are derived from the pipeline structure rather than simulation labels, they provide a supervision signal on unlabeled instruction traces, which are the mechanism that enables label-efficient training.

1) CPI Consistency: As established in Section II-A, the scalar CPI computed from $Base_s$ via Eq. 4 is identical at every pipeline stage by construction. This follows from instruction throughput conservation: every correct-path instruction must traverse all stages, so the effective IPC observed at Decode, Dispatch, Issue, and Commit must be the same. A well-trained model must respect this invariant in its predictions.

Labeled data. When ground-truth labels are available, we anchor each stage's predicted CPI to the simulation ground truth CPI_{golden}:

$$\mathcal{L}_{const} = \frac{1}{|S|}\sum_{s\in S}(CPI_s - CPI_{golden})^2 \tag{5}$$

where $S = \{\text{Decode, Dispatch, Issue, Commit}\}$ and $CPI_s = 1/(Base_s \cdot W_{\min})$ is the CPI predicted by the model at stage s.

Unlabeled data. Without a ground-truth anchor, the same conservation law is enforced by minimizing the variance of the predicted CPI across stages:

$$\mathcal{L}_{const}^{self} = \frac{1}{|S|}\sum_{s\in S}\left(CPI_s - \overline{CPI}\right)^2, \quad \overline{CPI} = \frac{1}{|S|}\sum_{s\in S} CPI_s \tag{6}$$

This self-supervised form requires no labels: any instruction trace processed through the analytical model of D3 can receive a consistency gradient signal. In our semi-supervised setting, labeled samples use $\mathcal{L}_{const}$ and unlabeled samples use $\mathcal{L}_{const}^{self}$.

2) Bottleneck Visibility Monotonicity: The second invariant governs how bottleneck pressures propagate across pipeline stages. An instruction that reaches a later stage has already survived frontend stalls and branch misprediction flushes upstream. Consequently, the observable Frontend Pressure (FP) and Squash Pressure (SP) must monotonically decrease from Decode to Commit. Conversely, later stages are physically closer to the execution units and memory hierarchy, where long-latency stalls are directly exposed. Backend Pressure (BP) must therefore monotonically increase from Decode to Commit. These trends are structural properties of the

MSCS: they hold in every ground-truth measurement, for any workload and any microarchitectural configuration.

We enforce this invariant with a hinge loss that penalizes any predicted violation. Let the stages be ordered as $s_1 =$ Decode, $s_2 =$ Dispatch, $s_3 =$ Issue, $s_4 =$ Commit:

$$\begin{aligned}\mathcal{L}_{mono} = &\sum_{i=1}^{3} \max\big(0,\ FP_{s_{i+1}} - FP_{s_i}\big) \\ &+ \sum_{i=1}^{3} \max\big(0,\ SP_{s_{i+1}} - SP_{s_i}\big) \\ &+ \sum_{i=1}^{3} \max\big(0,\ BP_{s_i} - BP_{s_{i+1}}\big)\end{aligned} \tag{7}$$

Like $\mathcal{L}_{const}^{self}$, this term requires no simulation labels and applies equally to labeled and unlabeled samples.

Total Loss: The complete training objective combines the primary task loss with both architectural constraints:

$$\mathcal{L}_{total} = \mathcal{L}_{task} + \lambda_1 \mathcal{L}_{const} + \lambda_2 \mathcal{L}_{mono} \tag{8}$$

where $\mathcal{L}_{task}$ is the mean squared error against the ground-truth MSCS vector, and λ_1, λ_2 are empirically tuned hyperparameters. For unsupervised pretraining on unlabeled data, $\mathcal{L}_{task}$ is dropped and the model is trained on $\mathcal{L}_{const}^{self}$ and $\mathcal{L}_{mono}$ alone. This allows the model to leverage abundant unlabeled traces jointly with a small set of cycle-level simulation labels during training, extracting structural supervision from the pipeline physics without requiring additional simulation cost.

C. Stage-Interaction Aware Analytical Model

PRISM's output is a per-stage MSCS vector, yet per-resource features compute each structure's throughput bound in isolation, carrying no information about how much of a stall is absorbed by intervening buffers before it reaches a later stage. Without per-stage input features, the model must infer this attenuation solely from raw microarchitectural parameters, increasing the learning burden and reducing generalization to unseen configurations. We address this by propagating FE and BE pressure through the pipeline using inter-stage buffer capacities as filters, producing per-stage features that mirror the MSCS structure and already satisfy the monotonicity invariants of the architecture-constrained losses by construction. Details are presented in Appendix A.

1) First-Order Pressure Model: If a resource requires $T_{resource}$ cycles to process N instructions at sustained throughput $W_{\min}$, the stall slots it induces are:

$$P_{resource} = T_{resource} \times W_{\min} - N \tag{9}$$

We source FE pressure from I-cache miss latency and BE pressure from Concorde's ROB model, as these two resources dominate frontend starvation and backend contention across the target workload range.

TABLE II: Inter-stage queue structures and capacities.

Structure	Symbol	Value
Fetch-to-Decode Queue	C_{FDQ}	$2 \times W_{decode}$
Decode-to-Dispatch Queue	C_{DDQ}	$2 \times W_{dispatch}$
Dispatch-to-Issue Queue	C_{DIQ}	$2 \times W_{issue}$
Issue Queue (IQ)	C_{IQ}	`numIQEntries`
Load Queue (LQ)	C_{LQ}	`numLQEntries`
Reorder Buffer (ROB)	C_{ROB}	`numROBEntries`

2) FE and BE Pressure Propagation: We model propagation by having inter-stage buffers absorb pressure: a buffer of capacity C absorbs up to C stall slots, passing only the excess $\max(0, P - C)$ to the next stage. Table II defines the buffer capacities.

FE Pressure Propagation. FE pressure originates at fetch and propagates downstream, with each stage draining its upstream buffer before observing the stall. Because the RS is a subset of the ROB, Issue starves before Commit. The sequential absorption structure guarantees $P_{FE,Decode} \geq P_{FE,Dispatch} \geq P_{FE,Issue} \geq P_{FE,Commit}$.

BE Pressure Propagation. BE pressure originates at commit and propagates upstream, partially absorbed by instruction-level parallelism between commit and issue. At Dispatch, the MSCS stalls when either the ROB or the RS is full; the binding path is whichever yields higher pressure. The upstream absorption structure guarantees $P_{BE,Decode} \leq P_{BE,Dispatch} \leq P_{BE,Issue} \leq P_{BE,Commit}$.

3) Output Feature Representation: The propagation produces 8 pressure sequences per trace, one FE and one BE stream at each of the four pipeline stages. Each sequence spans K analysis windows (e.g., $K = 250$ for a 100K-instruction trace with 400-instruction windows). Following Concorde, each sequence is compressed into a 101-dimensional empirical distribution, yielding $4 \times 2 \times 101 = 808$ stage-aware features concatenated with Concorde's original per-resource features and the microarchitectural configuration parameters.

III. EVALUATION

A. Experiment Settings

1) Dataset Construction: We evaluate 17 SPEC CPU2017 benchmarks [5], detailed in Table III. For each benchmark, SimPoint [23] identifies the three highest-weight execution intervals of 100M instructions each, which are then simulated in gem5 to extract instruction traces, scalar CPI, and MSCS labels. Each interval is partitioned into 100K-instruction data points with a 400-instruction context window, yielding 9,600 data points in total. The input feature vector concatenates Concorde's original microarchitectural embeddings with the per-stage pressure distributions from D3. The ground-truth label for each data point is the 4×4 MSCS matrix.

2) Micro-architectural Parameter Space: We adopt the same microarchitectural parameter space as Concorde [16], summarized in Table IV. Each data point is simulated under a distinct randomly sampled configuration, requiring the model

to learn generalized architectural interactions rather than interpolate within a narrow neighborhood.

TABLE III: Evaluated benchmarks and trace counts. All the benchmarks have the same number of data points

Benchmark Name	Traces
508.namd_r, 526.blender_r, 623.xalancbmk_s, 631.deepsjeng_s, 638.imagick_s, 641.leela_s	900
602.gcc_s, 607.cactuBSSN_s, 619.lbm_s	600
600.perlbench_s, 605.mcf_s, 620.omnetpp_s, 621.wrf_s, 644.nab_s, 648.exchange2_s, 654.roms_s, 657.xz_s	300
Total (17 benchmarks)	**9,600**

TABLE IV: Design space parameters and value ranges.

Parameter	Range
Fetch / Decode / Rename width	4–12
Dispatch / Issue width	4–12 (tied)
Writeback / Commit width	4–12
Fetch buffer size	8, 16, 32, 64
Branch predictor	BiMode, TAGE
ROB size	64–1024
Issue queue size	32–128
Load queue size	16–256
Store queue size	16–256
Int ALU / Mult-Div count	1–8
FP ALU / Mult-Div count	1–8
Read/Write ports	1–8
L1 cache size (kB)	16, 32, 64, 128, 256
L2 cache size (kB)	512, 1024, 2048, 4096
L1d / L1i / L2 associativity	2, 4, 8, 16, 32

B. Results

1) In-Distribution Accuracy and Ablation: We compare PRISM against Concorde and ablated variants across two model capacities ([256,128] and [512,256]) on the full 408K dataset with a 15:1 train/test split. Figure 2 reveals four findings.

Structured prediction matches scalar accuracy. PRISM ([512,256]) achieves 3.1% MAPE, matching Concorde's 3.0%, while additionally providing per-stage bottleneck attribution that scalar predictors cannot offer by design.

Physics losses are essential, not optional. Predicting a 16-dimensional MSCS vector without architectural constraints degrade to 4.7% MAPE: predicting a richer target increases the learning difficulty unless the physical structure guides the model. Enforcing $\mathcal{L}_{const}$ and $\mathcal{L}_{mono}$ closes this gap entirely (4.7% to 3.1%), confirming that architecture-constrained losses are the mechanism that makes structured prediction viable.

D3 stage-aware features contribute independently. Replacing Concorde's per-resource features with D3's per-stage pressure distributions reduces MAPE from 3.6% to 3.1%, capturing inter-stage pipeline dynamics that per-resource features computed in isolation cannot represent.

Structured prediction scales with model capacity; scalar prediction does not. Concorde's accuracy improves marginally from 3.2% to 3.0% as model capacity doubles, indicating a methodological ceiling. Under the same capacity increase, PRISM reduces error by 24% (4.1% to 3.1%), confirming that the MSCS prediction task has headroom to benefit from larger models in a way scalar prediction does not.

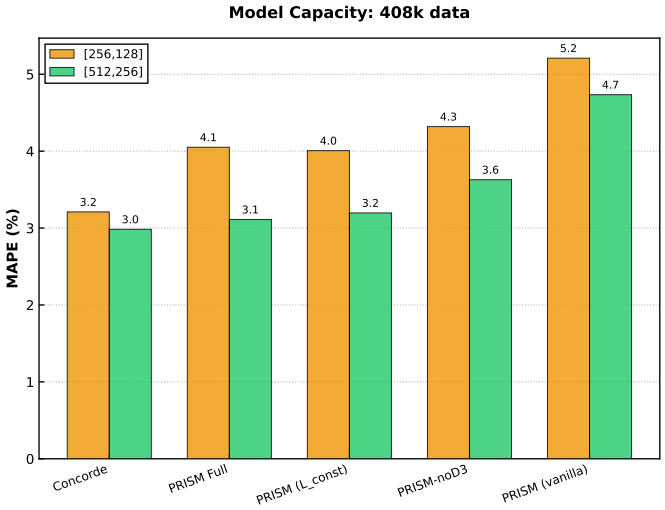

Fig. 2: Model accuracy and ablation study across different network capacities.

2) Label Efficiency: Label acquisition is the primary bottleneck in performance modeling. Figure 3 defines four variants that progressively add MSCS output (V1), physics losses (V2), and semi-supervised learning (PRISM) on top of Concorde. All variants use the 183K dataset with a 15:1 train/test split. For PRISM, the unlabeled portion consists of traces processed through the D3 analytical model without gem5 simulation.

V1 (+MSCS only) is consistently the worst performer, confirming that the 16-dimensional prediction task is inherently harder than scalar CPI, physical constraints are essential to compensate.

PRISM matches Concorde's full-label accuracy (4.0% MAPE) using only 60% of labeled data, reducing cycle-accurate simulation cost by 40% while additionally providing per-stage bottleneck attribution. The advantage is most pronounced in the low-label regime: at 5% labels, PRISM achieves 9.1% MAPE versus V2's 10.1% and Concorde's 9.8%; at 20% labels, PRISM reaches 6.1% versus V2's 6.9% and Concorde's 6.8%. The consistent gap between PRISM and V2 across all labeling ratios demonstrates that semi-supervised training on unlabeled traces amplifies the effect of physics constraints beyond mere regularization within the labeled set.

IV. DSE via Differentiable Multi-Stage Bottleneck Analysis

A. MSCS-Guided Design Space Exploration

Scalar CPI gradient ($\partial\text{CPI}/\partial\theta$) compresses all pipeline overlap effects into a single number. When opposing bot-

TABLE V: Label efficiency experiment variants.

Name	Output	Description	Data
PRISM	16-dim MSCS	Concorde + MSCS + Physical Loss + Semi-Supervised Learning	Labeled + Unlabeled
Concorde	Scalar CPI	Concorde	Labeled only
V1	16-dim MSCS	Concorde + MSCS	Labeled only
V2	16-dim MSCS	Concorde + MSCS + Physical Loss	Labeled only
V3	16-dim MSCS	Concorde + MSCS + Physical Loss + Pre-Train then Fine-tuning	Labeled + Unlabeled

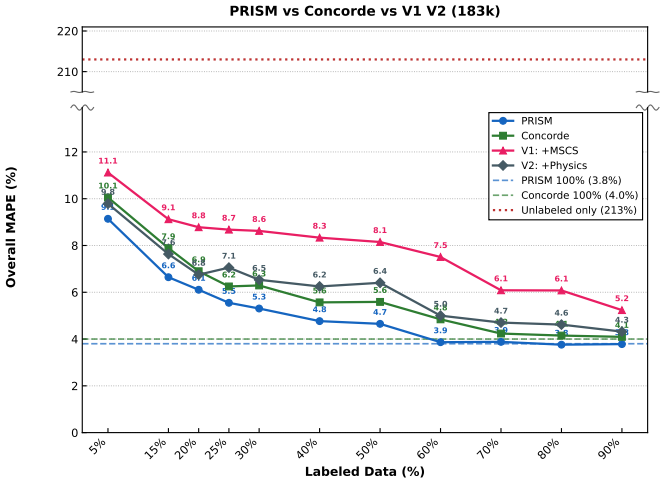

Fig. 3: Label efficiency on the 183k dataset (train/test = 15:1). For supervised methods (Concorde, V1, V2), only $X\%$ of the training set is used as labeled data. For PRISM, the remaining $(1-X)\%$ serves as unlabeled input processed through analytical models without cycle-accurate simulation. Dashed lines indicate 100%-label accuracy floors for PRISM (3.8%) and Concorde (4.0%). Variant definitions are in Table V.

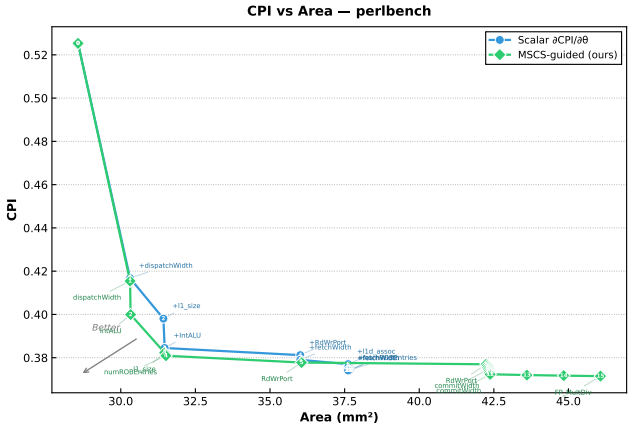

Fig. 4: CPI vs. Area trajectory on 600.perlbench_s. MSCS-guided DSE reaches a lower CPI at a comparable area through bottleneck-informed decisions, while scalar CPI gradient plateaus due to masked overlap effects.

tleneck changes cancel each other, or when a parameter's contribution is masked by a concurrent dominant bottleneck, the scalar gradient stalls: all search directions appear equally unproductive. The MSCS vector exposes both failure modes directly and enables a corrective action in each case, as formalized in Table VI. A mathematical proof of the existence of these two optimization opportunities is in Appendix B.

We compare MSCS-guided search against scalar CPI gradient using PRISM as the surrogate model for both, under an area-constrained DSE setting on 600.perlbench_s. Figure 4 shows that MSCS-guided DSE achieves lower CPI at every comparable area budget.

At $31\,\text{mm}^2$, MSCS-guided search reaches CPI ≈ 0.383 while the scalar gradient remains at 0.393. This gap reflects the *decouple* action: the scalar signal sees $\Delta\text{CPI} \approx 0$ because opposing BP and FP shifts cancel, while MSCS identifies the cancellation and selects the parameter (numROBEntries) that resolves the dominant component.

Beyond $38\,\text{mm}^2$, the scalar gradient plateaus at CPI ≈ 0.386 with no further improvement. MSCS-guided search continues to find productive adjustments out to $42\,\text{mm}^2$, steadily reducing CPI to 0.37. This continued progress reflects the *protect* action: rather than reclaiming a parameter whose contribution is masked by a concurrent bottleneck, MSCS preserves it and returns to it once the masking bottleneck is resolved, avoiding the backtracking that causes the scalar gradient to stall.

V. MSCS-Guided Shapley Value Pruning

Shapley values [22] answer a specific question that scalar CPI gradients cannot: across all possible combinations of microarchitectural parameters, how much does each individual parameter contribute to performance on average? This order-independent attribution accounts for interactions between parameters, which single-parameter perturbation misses entirely. The cost, however, is prohibitive: computing exact Shapley values for 18 parameters requires evaluating every possible ordering of those parameters, which amounts to over 21,000 surrogate model calls even with Monte Carlo approximation. The MSCS vector provides a way to identify which parameters actually interact, before running any Shapley computation, and skip the expensive joint evaluation for parameter pairs that do not.

A. Pipeline Influence Vectors.

For each parameter θ_i, we compute a 12-dimensional *pipeline influence vector* $\mathbf{v}_i \in \mathbb{R}^{12}$ (4 stages $\times$ 3 pressure

TABLE VI: MSCS-guided DSE: overlap detection enables two corrective actions that scalar CPI gradient cannot perform. A proof of this claim is presented in the Appendix B.

	Overlap Manifestation	Scalar CPI Error	MSCS Correction
Increase (decouple)	ΔCPI ≈ 0 (ΔBP > 0, ΔFP < 0 cancel)	Search stalls; all directions appear flat	Detects cancellation; selects param to decouple
Decrease (protect)	ΔCPI ≈ 0 (true effect pipeline-masked)	Misjudges as useless; incorrectly reclaims	Detects hidden contribution; preserves for future use

types) via finite differencing against the baseline configuration $\boldsymbol{\theta}^*$:

$$\mathbf{v}_i = \mathrm{MSCS}(\theta_i^{\min}, \boldsymbol{\theta}^*_{-i}) - \mathrm{MSCS}(\boldsymbol{\theta}^*) \tag{10}$$

where $\boldsymbol{\theta}^*_{-i}$ denotes all parameters held at default except θ_i, which is set to its minimum. Each vector captures how removing a parameter redistributes stall pressure across pipeline stages and bottleneck types.

B. Interaction Detection via Cosine Similarity.

Two parameters interact strongly in the Shapley sense when they compete for the same bottleneck: relieving one reduces the other's apparent value, so their joint contribution is less than the sum of their individual contributions.

We quantify pipeline influence similarity using cosine similarity, which measures the angle between two influence vectors: a value near one means the parameters affect the pipeline in the same way, a value near zero means their effects are orthogonal. This comparison requires no additional surrogate evaluations; it operates entirely on the fingerprints already computed above.

Formally, we measure the mechanistic overlap between parameters θ_i and θ_j as:

$$s_{ij} = \frac{|\mathbf{v}_i \cdot \mathbf{v}_j|}{\|\mathbf{v}_i\| \cdot \|\mathbf{v}_j\|} \tag{11}$$

A high s_{ij} indicates that both parameters affect the same pipeline stages and pressure types, they compete for the same pressure reduction budget, implying strong Shapley interaction. A low s_{ij} indicates orthogonal effects, implying negligible interaction. This requires $\binom{d}{2}$ dot products on already-computed vectors, zero additional surrogate evaluations. Interestingly, RpStacks [14] uses the same metric to merge redundant execution paths by their event-penalty vectors; our application is dual, comparing *parameters* by their MSCS vectors rather than *paths* by their penalty vectors.

C. Parameter Grouping and Pruned Computation.

We construct an interaction graph where (i, j) is an edge iff $s_{ij} > \tau$, and partition parameters into connected components $\{G_1, \ldots, G_K\}$. Exact Shapley values are computed independently within each group, with out-of-group parameters held at the baseline configuration. The total cost reduces from $O(d!)$ to $\sum_k n_k!$, where $n_k = |G_k|$.

We connect any two parameters whose influence vector similarity exceeds a threshold τ and identify the resulting groups of mutually interacting parameters. Shapley values are then computed exactly within each group, with parameters outside the group held at their baseline values. Because groups are small relative to the full parameter set, the number of required evaluations drops dramatically:

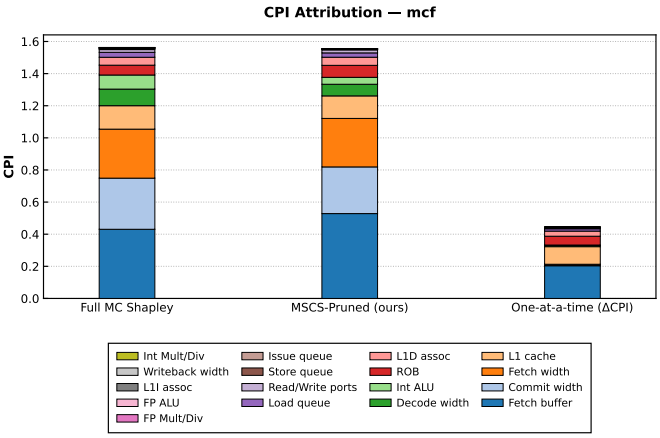

Fig. 5: Shapley value pruning result of 605.mcf_s.

D. Results.

Figure 5 compares three attribution methods on `605.mcf_s`. Full Monte Carlo Shapley (2000 permutations) serves as ground truth. MSCS-pruned Shapley closely reproduces the full attribution structure: the relative ordering and magnitudes of all major contributors are preserved, including `FetchBuf`, `CommitW`, `FetchW`, `L1$`, `IntALU`, `ROB`. In contrast, One-at-a-time ΔCPI drastically underestimates parameters whose value arises from interactions: `FetchW` drops from $+0.305$ to near zero, and `CommitW` drops from $+0.318$ to -0.017 (direction reversed), because their contributions only manifest when other width parameters are simultaneously present.

Table VII quantifies the trade-off. MSCS-pruned Shapley achieves Spearman $\rho = 0.953$ with $20\times$ fewer evaluations and $19\times$ faster wall-clock time than full Monte Carlo, while One-at-a-time attribution shows poor rank correlation ($\rho = 0.538$), confirming that single-parameter perturbation cannot capture the interaction structure of CPU design spaces.

VI. Conclusion

We presented PRISM, an ML-based CPU performance modeling framework that replaces scalar CPI prediction with the Multi-Stage CPI Stack (MSCS), a 4×4 matrix of bottleneck proportions across four pipeline stages that simultaneously

TABLE VII: Shapley attribution accuracy and cost on `605.mcf_s`

Method	ρ	MSE	Evals	Time
Full MC Shapley	1.000	—	21,857	5,678s
Pruned Shapley (ours)	0.953	0.0011	1,092	296s
One-at-a-time	0.538	0.0527	18	5s

serves as a prediction target, an interpretability interface, and a physics-constrained loss carrier. By extracting two architectural invariants from the MSCS structure,CPI conservation and pressure monotonicity, and embedding them as label-free losses, PRISM enables training with unlabeled instruction traces for the first time in CPU performance modeling, matching Concorde's accuracy (3.1% vs. 3.0% MAPE) with 40% fewer labels while natively providing bottleneck decomposition. The predicted MSCS further enables bottleneck-aware design space exploration, producing a better Pareto front on perlbench than scalar CPI based search, and accelerates Shapley value-based attribution by 20× while preserving near-identical parameter correlation (Spearman $\rho = 0.953$ on mcf). PRISM demonstrates that richer prediction targets, grounded in architectural physics, can break the accuracy–explainability–label-efficiency trade-off that has constrained ML-based performance modeling to date.

REFERENCES

[1] C. Bai, Q. Sun, J. Zhai, Y. Ma, B. Yu, and M. D. Wong, "BOOM-Explorer: RISC-V BOOM microarchitecture design space exploration framework," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2021, pp. 1–9.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[3] H. Bong, N. Kang, Y. Kim, J. Kim, and H. Jang, "Intervalsim++: Enhanced interval simulation for unbalanced processor designs," *IEEE Computer Architecture Letters*, vol. 24, no. 1, pp. 1–4, 2025.

[4] D. L. Bruening, "Efficient, Transparent, and Comprehensive Runtime Code Manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, 2004.

[5] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 41–42. [Online]. Available: https://doi.org/10.1145/3185768.3185771

[6] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[7] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Transactions on Computer Systems (TOCS)*, vol. 27, no. 2, pp. 1–37, 2009.

[8] S. Eyerman, W. Heirman, K. Du Bois, and I. Hur, "Extending the performance analysis tool box: Multi-stage cpi stacks and flops stacks," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2018, pp. 179–188.

[9] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[10] K. Hegde, P.-A. Tsai, S. Huang, V. Chandra, A. Parashar, and C. W. Fletcher, "Mind mappings: enabling efficient algorithm-accelerator mapping space search," ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 943–958. [Online]. Available: https://doi.org/10.1145/3445814.3446762

[11] A. S. Karkhanis and J. E. Smith, "A first-order superscalar microprocessor performance model," in *Proceedings of the 31st annual international symposium on Computer architecture (ISCA)*, 2004, pp. 338–349.

[12] S. J. Kaufman, P. M. Phothilimthana, Y. Zhou, and M. Burrows, "A learned performance model for the tensor processing unit," *CoRR*, vol. abs/2008.01040, 2020. [Online]. Available: https://arxiv.org/abs/2008.01040

[13] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: Association for Computing Machinery, 2006, p. 185–194. [Online]. Available: https://doi.org/10.1145/1168857.1168881

[14] J. Lee, H. Jang, and J. Kim, "Rpstacks: Fast and accurate processor design space exploration using representative stall-event stacks," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 255–267.

[15] L. Li, S. Pandey, T. Flynn, H. Liu, N. Wheeler, and A. Hoisie, "Simnet: Accurate and high-performance computer architecture simulation using deep learning," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 2, pp. 1–24, 2022.

[16] A. Nasr-Esfahany, M. Alizadeh, V. Lee, H. Alam, B. W. Coon, D. Culler, V. Dadu, M. Dixon, H. M. Levy, S. Pandey *et al.*, "Concorde: Fast and accurate cpu performance modeling with compositional analytical-ml fusion," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, 2025, pp. 1480–1494.

[17] D. B. Noonburg and J. P. Shen, "A framework for statistical modeling of superscalar processor performance," in *Proceedings Third International Symposium on High-Performance Computer Architecture*. IEEE, 1997, pp. 298–309.

[18] S. Pandey, A. Yazdanbakhsh, and H. Liu, "Tao: Re-thinking dl-based microarchitecture simulation," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 8, no. 2, May 2024. [Online]. Available: https://doi.org/10.1145/3656012

[19] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational physics*, vol. 378, pp. 686–707, 2019.

[20] B. Reagen, J. M. Hernández-Lobato, R. Adolf, M. Gelbart, P. Whatmough, G.-Y. Wei, and D. Brooks, "A case for efficient accelerator design space exploration via bayesian optimization," in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2017, pp. 1–6.

[21] A. Renda, Y. Chen, C. Mendis, and M. Carbin, "Difftune: Optimizing cpu simulator parameters with learned differentiable surrogates," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 442–455.

[22] L. S. Shapley *et al.*, "A value for n-person games," 1953.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. New York, NY, USA: Association for Computing Machinery, 2002, p. 45–57. [Online]. Available: https://doi.org/10.1145/605397.605403

[24] S. Van den Steen, S. De Pestel, M. Mechri, S. Eyerman, T. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout, "Micro-architecture independent analytical processor performance and power modeling," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 32–41.

[25] S. Wadle, Y. Zhang, V. Singh, and K. Sankaralingam, "Neuroscalar: A deep learning framework for fast, accurate, and in-the-wild cycle-level performance prediction," *arXiv preprint arXiv:2509.22410*, 2025.

APPENDIX A
D3 ANALYTICAL MODEL DETAILS

A. Frontend and Backend Pressure Sources

We classify pressure into frontend (FE) and backend (BE) based on their distinct effects on the pipeline. FE pressure manifests as pipeline bubbles: when the frontend stalls, empty slots propagate forward and starve subsequent stages. BE pressure induces structural backpressure: when a backend resource is saturated, it blocks earlier stages from making progress.

Frontend pressure models only I-cache misses. T_{Icache} represents the extra stall cycles caused by instruction cache misses (not total execution cycles), so no $-N$ term is needed:

$$P_{FE,raw} = T_{Icache} \times W_{min} \tag{12}$$

Backend pressure uses Concorde's ROB dynamical system, which jointly captures instruction dependencies, execution latencies, data cache miss penalties, and finite ROB size. T_{ROB} is the total cycle count for processing N instructions:

$$P_{BE,raw} = T_{ROB} \times W_{min} - N \tag{13}$$

Why only I-cache and ROB. For the frontend, the dominant source of sustained stalls—long enough to drain inter-stage buffers—is instruction cache misses. Branch mispredictions are captured separately: the misprediction rate from the external oracle predictor is already part of the ML model's input, and the MSCS constraint $Base + FP + BP + SP = 1$ ensures the squash component is fully determined once FP and BP are predicted. For the backend, the ROB model is the most comprehensive of Concorde's per-resource models. The effects captured by the ALU throughput and load queue capacity models are largely subsumed by the ROB dynamical system, as execution port contention and memory access delays ultimately manifest as longer instruction finish times in the dependency graph.

Why not model squash pressure. We do not propagate squash pressure through the analytical model for three reasons. First, branch misprediction rates from the oracle predictor are already encoded in the input features. Second, since $Base + FP + BP + SP = 1$ at each stage, the squash component is fully determined once the other three are predicted. Third, adding squash propagation would introduce 808 additional input features without providing information not already captured, increasing the risk of noise.

B. ILP-Dependent Absorption (C_{ILP})

The MSCS accounting algorithm detects backend stalls at the issue stage when no RS instruction is ready (`prod(first non-ready instr)`). The absorption between commit and issue depends on workload ILP: independent instructions can continue issuing while the ROB head is stalled.

We quantify this using Concorde's dependency analysis. The full ROB dynamical system computes T_{ROB} with ROB size and in-order commit constraints. We additionally compute T_{crit}, the critical path length, by retaining only the dependency constraints:

$$\begin{aligned} s_i^{crit} &= \max\{f_d^{crit} \mid d \in Dep(i)\} \\ f_i^{crit} &= s_i^{crit} + Lat(instr_i) \\ T_{crit} &= \max_i \ f_i^{crit} \end{aligned} \tag{14}$$

T_{crit} is the longest path through the dependency DAG—the minimum cycles needed if all resources were infinite. The difference $T_{ROB} - T_{crit}$ represents cycles where the pipeline has available independent work. We convert this to an absorption capacity:

$$C_{ILP} = (T_{ROB} - T_{crit}) \times W_{min} \tag{15}$$

When $T_{crit} \approx T_{ROB}$ (low ILP), $C_{ILP} \approx 0$ and issue sees nearly the same backend pressure as commit. When $T_{crit} \ll T_{ROB}$ (high ILP), C_{ILP} is large and issue is shielded from backend stalls.

C. FE Propagation Details

$P_{FE,Decode}$ and $P_{FE,Dispatch}$ each subtract the inter-stage queue between adjacent stages (C_{FDQ} and C_{DDQ}). From dispatch onward, the MSCS algorithm detects frontend starvation at issue when the RS (= IQ + LQ) is empty, and at commit when the ROB is empty. Since instructions are allocated entries in both the ROB and RS at dispatch, but release their RS entry after issue while retaining their ROB entry until commit, instructions in the RS are a subset of those in the ROB. The RS must therefore drain before the ROB can. $P_{FE,Issue}$ subtracts the dispatch-to-issue queue plus the full RS capacity ($C_{IQ} + C_{LQ}$). $P_{FE,Commit}$ further subtracts the remaining ROB capacity ($C_{ROB} - C_{DIQ} - C_{IQ} - C_{LQ}$), representing instructions that have left the RS but remain in the ROB awaiting commit. The sequential structure guarantees monotonicity: each step subtracts a non-negative quantity.

D. BE Propagation Details

$P_{BE,Commit}$ takes the raw ROB pressure directly. $P_{BE,Issue}$ subtracts C_{ILP}: independent instructions continue issuing while the ROB head is stalled. For dispatch, the MSCS algorithm checks whether the ROB or RS is full (OR condition), producing two paths to stalling dispatch: (1) after issue stalls, dispatch continues filling the RS until it is full, absorbing $C_{IQ} + C_{LQ}$; (2) the ROB fills up directly from commit, absorbing C_{ROB}. The binding constraint is whichever yields higher pressure (i.e., smaller absorption). $P_{BE,Decode}$ further subtracts the decode-dispatch queue. Monotonicity is structurally guaranteed: both dispatch paths yield values $\leq P_{BE,Issue}$.

APPENDIX B
PROOF OF STRUCTURAL SUPERIORITY OF MSCS-GUIDED DSE

We formalize why MSCS-guided DSE decisions are structurally superior to scalar CPI gradient for parameter selection and resource reclaim.

Setup. Let θ denote a microarchitectural configuration. The MSCS vector decomposes pipeline utilization at each stage s into proportional components: $s \in \{\text{decode}, \text{dispatch}, \text{issue}, \text{commit}\}$

$$\text{Base} + \sum_T T_s = 1, \quad T \in \{\text{FP}, \text{BP}, \text{SP}\}, \tag{16}$$

where Base is the fraction of useful throughput and T_s is the fraction lost to bottleneck type T. Base is stage-invariant (the same correct-path instructions traverse all stages). CPI follows directly:

$$\text{CPI} = \frac{1}{\text{Base} \cdot w_{\min}}, \tag{17}$$

where $w_{\min}$ is the minimum pipeline width.

For a parameter perturbation $p \mapsto p+\delta$, let θ' denote the perturbed configuration. Define $\Delta T_s(p) = T_s(\theta) - T_s(\theta')$, with positive values indicating improvement (pressure reduced). From Eq. (16):

$$\Delta\text{Base} + \sum_T \Delta T_s = 0, \quad \text{so} \quad \Delta\text{Base} = -\sum_T \Delta T_s. \tag{18}$$

The CPI improvement under perturbation is:

$$\Delta\text{CPI}(p) = \text{CPI}(\theta) - \text{CPI}(\theta'). \tag{19}$$

Since $\text{CPI} = 1/(\text{Base} \cdot w_{\min})$ is monotonically decreasing in Base, and $\Delta\text{Base} = -\sum_T \Delta T_s$ (Eq. 18), the sign of ΔCPI is determined by the sign of $\sum_T \Delta T_s$. Scalar CPI observes only this sum. MSCS observes each ΔT_s individually across types and stages.

Definition 1 (Overlap-free score)**.** *For a candidate parameter p, the overlap-free score is:*

$$S(p) = \sum_{T \in \{\text{FP},\text{BP},\text{SP}\}} \max_s \max\big(0,\ \Delta T_s(p)\big). \tag{20}$$

$S(p)$ measures the total improvement potential of increasing p by one stride, with each bottleneck type contributing its best-stage effect independently.

The score applies three operations: $\max(0, \cdot)$ isolates beneficial effects per type, preventing cross-type cancellation; $\max_s$ selects the stage where each type's improvement is largest, before pipeline overlap absorbs it; $\sum_T$ aggregates the per-type benefits independently.

Lemma 1. *For all configurations θ and parameters p:*

$$S(p) \ \geq \ \sum_T \Delta T_s(p). \tag{21}$$

Equality holds if (condition i) $\Delta T_s \geq 0$ *for all T (no cross-type cancellation), and* (condition ii) ΔT_s *is stage-invariant for each T (no pipeline overlap).*

Proof. For any fixed stage s and each type T:

$$\Delta T_s \ \leq \ \max(0,\ \Delta T_s) \ \leq \ \max_{s'} \max(0,\ \Delta T_{s'}). \tag{22}$$

Summing Eq. (22) over all types T:

$$\sum_T \Delta T_s \ \leq \ \sum_T \max_{s'} \max(0,\ \Delta T_{s'}) \ = \ S(p). \tag{23}$$

Equality requires both steps of Eq. (22) to be tight: the first holds with equality if $\Delta T_s \geq 0$ for all T (condition i); the second iff ΔT_s is the same at all stages for each T (condition ii). □

Theorem 1. *Scalar CPI gradient has two provable failure modes in greedy DSE that the MSCS decomposition resolves:*

1) ***Increase (cancellation):*** *When different bottleneck types respond to parameter p in opposite directions (e.g., $\Delta FP > 0$ but $\Delta BP < 0$), their effects cancel in the sum $\sum_T \Delta T_s \approx 0$, causing $\Delta CPI(p) \approx 0$. Scalar CPI sees a flat landscape and the search stalls. By Lemma 1, $S(p) \gg 0$ in this case, so MSCS correctly identifies p as beneficial and continues the search.*
2) ***Reclaim (overlap):*** *When reducing a parameter yields $|\Delta CPI(p_{dec})| < \varepsilon$, scalar CPI judges the resource as reclaimable. However, individual $|\Delta T_s|$ may be large with opposite signs, indicating that the parameter has significant but currently cancelling effects on different bottleneck types. Reclaiming it eliminates these effects; if later configuration changes break the cancellation, the lost resource must be re-acquired at additional area cost. MSCS detects the large per-type effects and blocks the reclaim.*

In both cases, the ΔCPI *safety valve (reject any step where* $\Delta\text{CPI} < 0$*) ensures that MSCS decisions never worsen CPI relative to the current configuration.*

Proof. Part 1. By Lemma 1, $S(p) \geq \sum_T \Delta T_s$ always holds. When cross-type cancellation is present, $\sum_T \Delta T_s \approx 0$ while $S(p) \gg 0$, because $S(p)$ clips negative contributions before summing and thus cannot cancel. Since ΔCPI has the same sign as $\sum_T \Delta T_s$, scalar CPI inherits the near-zero signal and stalls; $S(p)$ does not. The ΔCPI safety valve (reject any step where $\Delta\text{CPI} < 0$) ensures that acting on $S(p)$ never worsens CPI.

Part 2. Scalar CPI judges reclaimability by $|\Delta\text{CPI}|$, which depends on $|\sum_T \Delta T_s|$. When individual $|\Delta T_s|$ are large but sum to near zero, the scalar test passes despite significant per-type effects. Reclaiming such a parameter costs no CPI now, but forces re-acquisition at additional area cost when later configuration changes break the cancellation. MSCS inspects each ΔT_s separately, detects the hidden effects, and blocks the reclaim, avoiding this area overhead. □