Concorde: Fast and Accurate CPU Performance Modeling with Compositional Analytical-ML Fusion

Arash Nasr-Esfahany*
Google & MIT
Cambridge, MA, USA
arashne@mit.edu

Mohammad Alizadeh*
Google & MIT
Cambridge, MA, USA
malizadeh@google.com

Victor Lee
Google
Sunnyvale, CA, USA
victor.w.lee@gmail.com

Hanna Alam
Google
Haifa, Israel
hannaalam@google.com

Brett W. Coon
Google
San Jose, CA, USA
bwc@google.com

David Culler
Google
Sunnyvale, CA, USA
dculler@google.com

Vidushi Dadu
Google
Sunnyvale, CA, USA
vidushid@google.com

Martin Dixon
Google
Portland, OR, USA
mgdixon@google.com

Henry M. Levy
Google & University of Washington
Seattle, WA, USA
hanklevy@google.com

Santosh Pandey*
Google & Rutgers University
New Brunswick, NJ, USA
santosh.pandey@rutgers.edu

Parthasarathy Ranganathan
Google
Sunnyvale, CA, USA
parthas@google.com

Amir Yazdanbakhsh
Google DeepMind
Sunnyvale, CA, USA
ayazdan@google.com

Abstract

Cycle-level simulators such as gem5 are widely used in microarchitecture design, but they are prohibitively slow for large-scale design space explorations. We present `Concorde`, a new methodology for learning fast and accurate performance models of microarchitectures. Unlike existing simulators and learning approaches that emulate each instruction, `Concorde` predicts the behavior of a program based on compact performance distributions that capture the impact of different microarchitectural components. It derives these performance distributions using simple analytical models that estimate bounds on performance induced by each microarchitectural component, providing a simple yet rich representation of a program's performance characteristics across a large space of microarchitectural parameters. Experiments show that `Concorde` is more than five orders of magnitude faster than a reference cycle-level simulator, with about 2% average Cycles-Per-Instruction (CPI) prediction error across a range of SPEC, open-source, and proprietary benchmarks. This enables rapid design-space exploration and performance sensitivity analyses that are currently infeasible, e.g., in about an hour, we conducted a first-of-its-kind fine-grained performance attribution to different microarchitectural components across a diverse set of programs, requiring nearly 150 million CPI evaluations.

---

*Work done at Google.

CCS Concepts

• **Computing methodologies → Model development and analysis**; *Machine learning*; • **Computer systems organization → Processors and memory architectures**; • **General and reference → Performance**.

Keywords

CPU performance modeling, microarchitectural bottleneck analysis, analytical–ML fusion, design space exploration, performance attribution, Shapley value analysis, machine learning for microarchitecture

1 Introduction

Microarchitecture simulators are a key tool in the computer architect's arsenal [4, 6, 11, 14, 19, 28, 72, 75, 82]. From SimpleScalar [11] to gem5 [14, 59], simulators have enabled architects to explore new designs and optimize existing ones without the prohibitive costs of fabrication. CPU simulation, in particular, has become increasingly important as hyperscale companies like Google (Axion) [85], Amazon (Graviton) [7], Microsoft (Cobalt) [62] increasingly invest in developing custom CPU architectures tailored to their specific workloads.

The landscape of CPU performance modeling is characterized by a critical tension between model accuracy and speed. This trade-off manifests in the various levels of abstraction employed by different

performance models [14, 32, 87]. At one end of the spectrum lie analytical models [2, 87], which provide simplified mathematical representations of microarchitectural components and their interactions. Although they are fast, analytical models often lack the detailed modeling necessary to capture the dynamics of modern processors accurately. At the other end of the spectrum reside cycle-level simulators like gem5 [14], which can provide high-fidelity results by meticulously modeling every cycle of execution. However, this level of detail comes at a steep computational cost, becoming prohibitively slow for large-scale design space exploration [32, 47, 55], programs with billions of instructions, or detailed sensitivity studies.

Recognizing the limitations of conventional methods, there has been growing interest in using machine learning (ML) to expedite CPU simulation [54, 55, 61, 71]. Rather than explicitly model every cycle and microarchitectural interaction, these methods learn an approximate model of the architecture's performance from a large corpus of data. A typical approach is to pose the problem as learning a function mapping a sequence of instructions to the target performance metrics. For example, recent work [50, 54, 55, 71] train sequence models (e.g., LSTMs [35] and Transformers [88]) on ground-truth data from a cycle-level simulator to predict metrics such as the program's Cycles Per Instruction (CPI).

These methods show promise in providing fast performance estimates with reasonable accuracy. However, relying on black-box ML models operating on instruction sequences has several limitations. First, the computational cost of these methods scales proportionally with the length of the instruction sequence, i.e. $O(L)$ where L is the instruction sequence length. The $O(L)$ complexity limits the potential speedup of these methods, e.g., to less than 10× faster than cycle-level simulation with a single GPU [55, 71]. This speedup is mainly due to replacing the irregular computations of cycle-level simulation with accelerator-friendly neural network calculations [71]. By contrast, analytical models can be several orders of magnitude faster than cycle-level simulation (and current ML approaches) because they fundamentally operate at a higher level of abstraction, i.e., mathematical expressions relating key statistics (e.g., instruction mix, cache behavior, branch misprediction rate, etc.) to performance.

Second, existing ML approaches must learn all the dynamics impacting performance from raw instruction-level training data. In many cases, this learning task is unnecessarily complex since it does not exploit the CPU performance modeling problem structure. For example, TAO's Transformer model [71] must learn the performance impact of register dependencies from per-instruction register information, even though there exist higher-level abstractions (e.g., instruction dependency graphs [61]) that concisely represent dependency behavior (§3.2). By ignoring the problem structure, blackbox methods require a significant amount of training data to learn. For example, TAO trains on a dataset of 180 million instructions across four benchmarks and two microarchitectures [71], with further training required for each new microarchitecture.

To address these challenges, we propose a novel approach to performance modeling—compositional analytical-ML fusion—where we decompose the task into multiple lightweight models that work together to progressively achieve high fidelity with low computational complexity. We demonstrate this approach in Concorde, a CPU performance model that uses simple analytical models capturing the first-order effects of individual microarchitectural components, coupled with an ML model that captures complex higher-order effects (e.g., interactions of multiple microarchitectural components).

Concorde achieves constant-time $O(1)$ inference complexity, independent of the length of the instruction stream, while maintaining high accuracy across diverse workloads and microarchitectures. Unlike existing ML methods that operate on instruction sequences, Concorde predicts performance based on a compact set of *performance distributions*. It trains a lightweight ML model—a shallow multi-layer perceptron (MLP)—to map these performance distributions to the target performance metric. We focus on modeling CPI in this paper as it directly reflects program performance, though in principle our techniques could be extended to other metrics. Concorde's ML model generalizes across a large space of designs, specified via a set of parameters associated with different microarchitectural components (§3). Given the performance distributions for a program region (e.g., 1M instructions), predicting its CPI on any target microarchitecture is extremely fast; it requires only a single neural network evaluation, taking less than a millisecond.

Concorde derives a program region's performance distributions through a two-step process: trace analysis and analytical modeling. Trace analysis uses simple in-order cache and branch predictor simulators to extract information such as instruction dependencies, approximate execution latencies, and branch misprediction rate. Next, analytical models estimate the bottleneck throughput imposed by each CPU component (e.g., fetch buffer, load queue, etc.) in isolation, assuming other CPU components have infinite capacity. For each CPU component, Concorde uses the distribution of its throughput bound over windows of a few hundred instructions as its performance feature. For each memory configuration, Concorde executes the per-component analytical models independently to precompute the set of performance distributions for all parameter values. The analytical models are lightweight, completing in 10s of milliseconds for a million instructions. Precomputing the performance distributions is a one-time cost, enabling nearly instantaneous performance predictions across the entire parameter space.

Concorde's unique division of labor between analytical and ML modeling simplifies both the analytical and ML models. Since the analytical models are not directly used to predict performance, they are relatively easy to construct. Their main goal is to provide a first-cut estimate of the performance bounds associated with each microarchitectural component (akin to roofline analysis [18]), without the burden of quantifying the combined effect of multiple interacting components. The ML model, on the other hand, starts with features that correlate strongly with a program's performance, rather than raw instruction sequences. Its task is to capture the higher-order effects ignored by the analytical models, such as the impact of multiple interacting bottlenecks. The net result is a method that is as fast as analytical models while achieving high accuracy.

Concorde enables large-scale analyses that are deemed impractical with conventional methods. As one use case, we consider the problem of fine-grained performance attribution to different microarchitectural components: *What is the relative contribution of different microarchitectural components to the predicted performance of a target architecture?* We present a novel technique for answering this question using only a performance model relating microarchitectural parameters to performance. Our technique applies the concept

of Shapley value [78] from cooperative game theory to provide a fair and rigorous attribution of performance to individual components. The method improves upon standard parameter ablation studies and may be of interest in other use cases beyond Concorde.

We present a concrete realization of Concorde, designed to approximate the behavior of a proprietary gem-5 based cycle-level trace-driven CPU simulator. We train Concorde on a dataset of 1 million random program regions and architectures, to predict the impact of 20 parameters spanning frontend, backend, and memory (totaling 2.2×10^{23} parameter combinations). The program regions are sampled from a diverse set of SPEC2017 [40], open-source, and proprietary benchmarks. The key findings of our evaluation are:

- Concorde's average CPI prediction error is within 2% of the ground-truth cycle-level simulator for unseen (random) program regions and architectures, with only 2.5% of samples exceeding a 10% prediction error. Ignoring the one-time cost of analytical modeling, Concorde is five orders of magnitude faster for predicting the performance of 1M-instruction regions. For long 1B instruction programs, Concorde accurately estimates performance (average error ~3.2%) based on randomly-sampled program regions, seven-orders of magnitude faster than cycle-level simulation.
- In predicting CPI for a realistic core model (based on ARM N1 [73]), Concorde is more accurate than TAO [71], the state-of-the-art sequence-based ML performance model, trained specifically for the same core configuration. It achieves an average prediction error of 3.5%, compared to 7.8% for TAO.
- For a 1M-instruction region, precomputing all the performance distributions takes the CPU time equivalent of 7 to 107 cycle-level simulations, depending on the granularity of parameter sweeps. These performance features enable rapid performance predictions for 1.8×10^{18} to 2.2×10^{23} parameter combinations.
- Concorde enables a first of its kind, large-scale, fine-grained performance attribution to components of a core based on ARM N1, across a diverse set of programs using our Shapley value technique. This large-scale analyses requires more than 143M CPI evaluations, but takes only about one hour with Concorde.

2 Motivation and Insights

Consider a cycle-level simulator like gem5 as implementing a function that maps an input program and microarchitecture configuration to a performance metric such as CPI. Formally, $y = f(\vec{x}, \vec{p})$, where $\vec{x} \triangleq (x_1, ..., x_L)$ denotes the input program comprising L instructions, $\vec{p} \triangleq (p_1, ..., p_d)$ the parameters specifying the microarchitecture, and y the CPI achieved by program $\vec{x}$ on microarchitecture $\vec{p}$. Our goal is to learn a fast and accurate approximation of the function f from training examples derived from cycle-level simulations.

Supervised learning provides the de facto framework for learning a function from input-output examples. However, a critical design decision involves how to best represent the learning problem, including the selection of representative features and an appropriate model architecture. Several recent efforts [54, 55, 71] represent the function f using sequence-based models, such as LSTMs [35] and Transformers [88], operating on raw or minimally processed instruction sequences. As discussed in §1, these blackbox sequence models inherently limit scalability and increase the complexity of the learning task. Our key insight is a novel decomposition of the
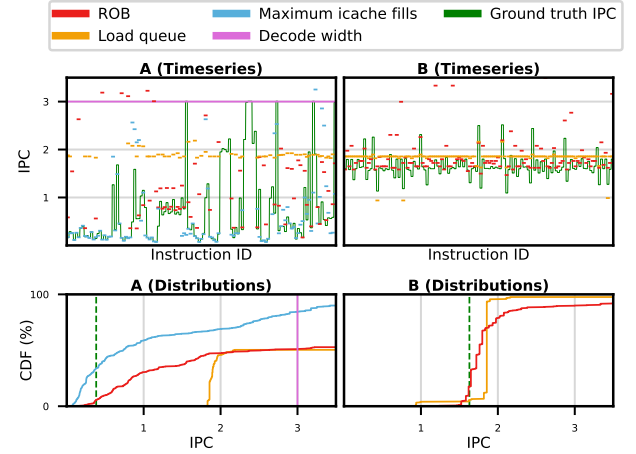
Figure 1: Per-resource analytical modeling produces a rich performance characterization of a program.

function f, comprising of two key stages. First, an analytical stage uses simple per-component models to extract compact performance features capturing the overall performance characteristics of the program. Second, a lightweight ML model predicts the target CPI metric efficiently based on these performance features.

Deriving compact performance features. The foundation of our analytical stage is to characterize the bottleneck throughput imposed by each CPU resource[1] individually, under the simplifying assumption that all other CPU components operate with unlimited capacity. For instance, to analyze the impact of the reorder buffer (ROB) size, we evaluate the program's throughput in a hypothetical system constrained only by the ROB size and instruction dependencies (i.e., a perfect frontend with no backend resource bottlenecks other than the limited ROB size). Focusing on one resource at a time enables relatively straightforward analyses (see §3.2 for examples). Formally, given a program $\vec{x}$, we compute the bottleneck throughput $z_i = A_i(\vec{x}, p_i)$ for each CPU component, where $A_i(\cdot, \cdot)$ is the analytical model for the ith component, parameterized by p_i. To capture program phase changes, we calculate this throughput over small windows of consecutive instructions (e.g., a few hundred instructions).

Figure 1 shows an illustrative example of these throughput calculations for four microarchitectural parameters (ROB size, Load queue size, maximum I-cache fills, and decode width) on two programs. The top plots display the timeseries of the throughput bounds derived by our analytical model for each parameter (details in §3.2.1) across 400-instruction windows, and the ground truth Instructions per Cycle (IPC) for the same windows. For both programs, the throughput bound timeseries explain the IPC trends well. For example, for program A, initially the IPC (green line) aligns with the maximum I-cache fills bound (cyan segments); subsequently, the IPC is around the smaller of the ROB, decode width, and maximum I-cache fills bounds in most instruction windows. Similarly, for program B, the bounds for ROB and Load queue overlap with the IPC. The maximum I-cache fills and decode width throughput bounds are much higher for program B (not shown in the figure).

---

[1]For simplicity, this section focuses on CPU parameters. Concorde handles a few parameters such as cache sizes differently (§3).
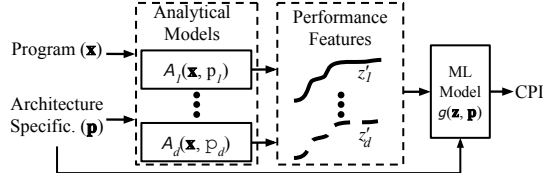
Figure 2: Concorde's compositional analytical-ML structure

Although the minimum of the per resource throughput bounds provides an estimate of IPC, it is not accurate. As shown in Figure 1, despite their overall correlation, the IPC frequently deviates from the exact minimum bound. This is not surprising. The analytical models make simplifying approximations, including ignoring interactions between multiple resource bottlenecks. In reality, resource bottlenecks can overlap, resulting in a net IPC lower than any individual bound. Nonetheless, the per-resource analysis provides informative features for predicting performance, capturing key first-order effects while leaving it to the ML model to capture higher-order effects.

The last step of deriving compact performance features (independent of program length L) is converting throughput timeseries into *distributions*, as depicted in the bottom plots in Figure 1. We encode these distributions using a fixed set of percentiles from their Cumulative Distribution Functions (CDF). Converting timeseries to CDFs is inherently lossy (e.g., joint behaviors across timeseries are not retained). However, as Figure 1 shows, the CDFs are still informative for predicting IPC. In particular, the IPC (vertical dashed line) aligns well with the lower percentiles of the smaller throughput bounds (e.g., Maximum I-cache fills and ROB for program A) — an implication of the IPC's proximity to the minimum throughput bound in most instruction windows. As our experimental results will show, a simple ML model can learn to accurately map these CDFs to IPC.

Concorde's compositional analytical-ML structure. Figure 2 illustrates the two-stage structure of Concorde. To predict the performance of program $\vec{x}$ on a given microarchitecture $\vec{p}$, Concorde first uses per-component analytical models to derive performance features $\vec{z} \triangleq (z_1', \ldots, z_d')$, where z_i' represents the distribution of the throughput bound for parameter p_i. These features, along with the list of parameters, are then passed to a lightweight ML model $\hat{y} = g(\vec{z}, \vec{p})$ to predict the CPI.

An important consequence of modeling each component (parameter) separately in the analytical stage is the ability to *precompute* the performance features for a program ($\vec{x}$) across the entire microarchitectural design space. In particular, our approach eliminates the need to evaluate the Cartesian product of all parameters, which would require exponential time and space. Instead, Concorde sweeps the range of each CPU parameter (once or per memory configuration depending on the parameter), precomputing the feature set $\{A_i(\vec{x}, p_i) | \forall p_i, \forall i\}$. To predict the performance of $\vec{x}$ on a specific microarchitecture $\vec{p}$, Concorde retrieves the pertinent precomputed features corresponding to $p_1, \ldots, p_d$ and evaluates the ML model $g(\vec{z}, \vec{p})$.

3 Concorde Design

We present a concrete realization of Concorde designed to approximate a proprietary gem5-based cycle-level trace-driven simulator. Inevitably, some aspects of Concorde (esp., analytical models) depend on the specifics of the reference architecture. We detail the
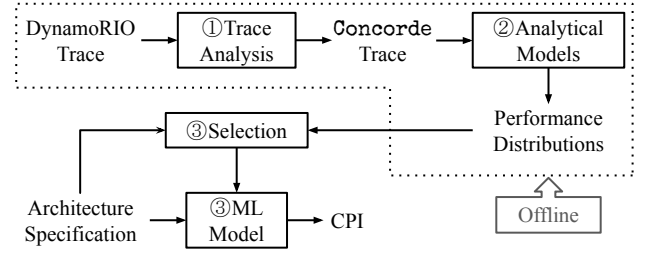

Figure 3: Design overview

Table 1: Large space of design parameters

Parameter	Value Range	ARM N1 value
ROB size	1,2,3,…,1024	128
Commit width	1,2,3,…,12	8
Load queue size	1,2,3,…,256	12
Store queue size	1,2,3,…,256	18
ALU issue width	1,2,3,…,8	3
Floating-point issue width	1,2,3,…,8	2
Load-store issue width	1,2,3,…,8	2
Number of load-store pipes	1,2,3,…,8	2
Number of load pipes	0,1,2,…,8	0
Fetch width	1,2,3,…,12	4
Decode width	1,2,3,…,12	4
Rename width	1,2,3,…,12	4
Number of fetch buffers	1,2,3,…,8	1
Maximum I-cache fills	1,2,3,…,32	8
Branch predictor	Simple, TAGE	TAGE
Percent misprediction for *Simple* BP	0,1,2,…,100	–
L1d cache size (kB)	16,32,64,128,256	64
L1i cache size (kB)	16,32,64,128,256	64
L2 cache size (kB)	512,1024,2048,4096	1024
L1d stride prefetcher degree	0 (OFF),4 (ON)	0(OFF)

design for our in-house cycle-level simulator while emphasizing concepts that we believe apply broadly to CPU modeling.

Our cycle-level simulator processes program traces captured by DynamoRIO [17] and features a generic parameterized Out-of-Order (OoO) core model similar to gem5's O3 CPU model [57]. The architecture consists of fetch, decode, and rename stages in the frontend; issue, execute, and commit stages in the backend; and uses Ruby [58] for modeling the memory system.[2] We focus on modeling the impact of 20 key design parameters on CPI, as summarized in Table 1, though our approach can be extended to other design parameters.

Figure 3 outlines Concorde's key design elements: ① **Trace analysis** which augments the input DynamoRIO [17] trace with information needed for Concorde (§3.1); ② **Per-resource analytical models** which transform the processed trace into performance distributions (§3.2); and ③ **Lightweight ML model** which predicts the CPI based on performance distributions (§3.3). The first two stages perform a one-time, offline computation for a given DynamoRIO trace. At simulation time, Concorde supplies the precomputed performance distributions and target microarchitecture's design parameters to the ML model, enabling nearly instantaneous CPI predictions.

---

[2]We use a fixed LLC size of 4MB, and a cache replacement policy similar to gem5's TreePLURP, a pseudo-Least Recently Used (PLRU) replacement policy. Our cache allocation policy is the same as gem5's standard allocation policy, always allocating lines on reads and writebacks. A cache line is not allocated on sequential access (for L2 and LLC) or a unique read for LLC. We use writeback for all L1i, L1d, L2, and LLC. We use memory BW of 37GB/s with latency of 90ns, and do not model memory channels.

3.1 Trace Analysis

The raw input trace to Concorde is captured using DynamoRIO's drmemtrace client [16], which provides detailed instruction and data access information for the target program. This trace is then processed into a Concorde Trace, which includes per-instruction information needed for our analytical models. We categorize this information into microarchitecture independent and microarchitecture dependent features, as detailed below.

Microarchitecture independent. This category includes data derived directly from the DynamoRIO trace: *(i) Instruction dependencies*, including both register and memory dependencies, *(ii) Program counters (PC)* for all instructions, *(iii) Data cache lines* for Load instructions, *(iv) Instruction cache lines* for all instructions, *(v) Instruction Synchronization Barriers (ISB)*, and *(vi) Branch types* (Direct unconditional, Direct conditional, and Indirect branches) for branch instructions.

Microarchitecture dependent. *(i) Execution latency:* Our analytical models require an estimate of the execution latency for each instruction. For non-memory instructions, we estimate the latency based on the opcode and corresponding execution unit (e.g., 3 cycles for integer ALU operations). Store instructions also incur a fixed, known latency, as the architecture uses write-back (with store forwarding). Load instructions, however, have variable latency depending on the cache level. To estimate their latency, we perform a simple in-order cache simulation (per memory configuration) to determine the cache level for each Load. We then map each cache level to a constant latency (e.g., L1 → 4 cycles, L2 → 10 cycles, LLC → 30 cycles, RAM → 200 cycles). *(ii) I-cache latency:* To model the fetch stage, our analytical models need an estimate of I-cache access times, which we obtain by performing a simple in-order I-cache simulation (per memory configuration). *(iii) Branch misprediction rate*, which we obtain by simulating the target branch prediction algorithm on the DynamoRIO trace. Our implementation supports two branch predictors: *Simple*, a branch predictor that mispredicts randomly with a pre-specified misprediction rate, and *TAGE* [8, 77].

Improving memory modeling. The execution times assigned to Load instructions by the above procedure can be highly inaccurate in some cases, leading Concorde's analytical models astray. As we will see, Concorde's ML model can overcome many errors in the analytical model. However, in extreme cases at the tail, Concorde's accuracy is affected by discrepancies between the results of trace analysis and the program's actual behavior (§5.2.1).

The key challenge with analyzing Load instructions is that their execution times can change depending on the time and order in which they are issued. Of course, a simple in-order cache simulation cannot capture timing-dependent effects. However, we now discuss a refinement atop the basic cache simulation that addresses two large sources of errors in estimating Load execution times. Our approach is built on two principles for accounting for the effects of conflicting cache lines and instruction order *without* running detailed timing simulations.

Consider two Load instructions accessing the same cache line, with the data not present in cache. In the in-order cache simulation, the first Load is labeled as a main memory access (200 cycles), while the second Load is labeled as an L1 hit (4 cycles). Now suppose these Loads are issued at around the same time in the actual OoO core, e.g., first Load at cycle 0 and second Load at cycle 1. Naïvely using the

---

Algorithm 1 A trace-driven state machine for memory

```
for all cache_line do                    ▷ State variable initialization
    exec_times[cache_line] ← Execution times of load instructions accessing
                              cache_line from in-order cache simulation
    access_counters[cache_line] ← 0            ▷ Number of accesses
    last_req_cycles[cache_line] ← 0            ▷ Cycle of last request
    last_resp_cycles[cache_line] ← 0           ▷ Cycle of last response
end for

function RespCycle(req_cycle, instr)
    cache_line ← instr.cache_line
 ▷ req_cycle must be non-decreasing for requests to the same cache line
    Assert req_cycle ≥ last_req_cycle[cache_line]
    if is_load(instr) then         ▷ Adjustment for load instructions only
        prev_resp_cycle ← last_resp_cycles[cache_line]
        access_number ← access_counters[cache_line]
        exec_time ← exec_times[cache_line][access_number]
        resp_cycle ← max(req_cycle+exec_time, prev_resp_cycle)
        last_resp_cycles[cache_line] ← resp_cycle
        access_counters[cache_line]++
    else                    ▷ Nothing special for non-load instructions
        resp_cycle ← req_cycle + estimated execution time of instr
    end if
    return resp_cycle
end function
```

---

cache simulation results, we might conclude that the first Load completes at cycle 200 and the second Load at cycle 5. But, in reality, both Loads will complete after 200 cycles because the second Load must wait for the first Load to fetch the data from main memory into L1 cache. This example motivates our first principle: *the response cycle for consecutive Loads accessing the same cache line is non-decreasing*.

Next, consider the same scenario, but with the two Loads issued in reverse order in the OoO core (e.g., due to a register dependency). With this reversed order, the second Load (issued first) becomes a main memory access, and the first Load (issued second) becomes an L1 hit. Thus, our second principle: *the access levels of Loads with the same cache line is determined by their issue order, not the instruction order (used in cache simulation)*. We incorporate these principles into a trace-driven state machine for memory (Algorithm 1). The function RespCycle returns the response cycle (execution completion cycle) for an instruction issued at cycle *req_cycle*. For non-Load instructions, it simply uses the execution time estimated by the standard procedure described earlier. For Loads, however, it adjusts the execution time to account for their cache line and issue times. We use this memory model in the analytical models of ROB and Load queue, which are sensitive to Load execution latencies (§3.2). The memory model is fast and does not materially increase the cost of analytical modeling.

3.2 Analytical Models

As discussed in §2, Concorde's primary features are a set of throughput distributions associated with each potential microarchitectural resource bottleneck. We describe the derivation of these distributions for various resources in §3.2.1. We then discuss a few auxiliary features in §3.2.2 that capture nuances not covered by the primary features, further improving the ML model's accuracy.

The bulk of Concorde's design effort has gone into analytical modeling. Before delving into details, we highlight a few lessons from our experience. Our guiding principle has been to capture the *performance trends* imposed by a microarchitectural bottleneck, without being overly concerned with precision. As our results will show (§5.2),

the ML model serves as a powerful backstop that can mask significant errors in the analytical model. Thus, we have generally avoided undue complexity (admittedly a subjective metric!) to improve the analytical model's accuracy. Our decision to simply analyze each resource in isolation (§2) is the clearest example of this philosophy.

Isolated per-resource throughput analysis is similar to traditional roofline analysis [18], but we perform it at an unusually fine granularity to analyze the impact of low-level resources (e.g., an issue queue, fetch buffers, etc.) on small windows (e.g., few 100s) of instructions. The details of such analyses depend on the design, but we have found three types of models to be useful: (i) closed-form mathematical expressions, (ii) dynamical system equations, (iii) simple discrete-event simulations of a single component. We provide examples of these methods below.

3.2.1 Per-Resource Throughput Analysis. We calculate the throughput of each CPU resource over fixed windows of k consecutive instructions, assuming no other CPU component is bottlenecked. The parameter k should be small enough to observe phase changes in the program's behavior, but not so small that throughput fluctuates wildly due to bursty instruction processing (e.g., a few instructions). We have found that any value of k in the order of the ROB size, typically a few hundred instructions, works well.

Given a program region (e.g., 100K-1M instructions), Concorde divides it into consecutive k-instruction windows and calculates the throughput bound for each window, per CPU resource and parameter value (Table 1). Concorde converts all throughput bound timeseries into distributions (CDFs) to arrive at the set of performance distributions for the entire microarchitectural design space.

Memory parameters (L1i/d, L2, L1d prefetcher degree) do not have separate throughput features; they affect the instruction execution latency and I-cache latency estimates (§3.1) used in CPU resource analyses. Specifically, the throughput computations for ROB, and Load/Store queues rely on instruction execution latencies. Concorde performs throughput calculations for these resources per L1d/L2/prefetch configuration using the corresponding execution latency values in the Concorde trace. Similarly, the I-cache fills throughput calculations are performed per L1i/L2 cache size.

ROB. The ROB is the most complex component to model, encapsulating out-of-order execution constrained by instruction dependencies and in-order commit behavior. For an instruction i, we define $\text{Dep}(i)$ as its immediate (register and memory) dependencies obtained via trace analysis (§3.1), a_i as its arrival cycle to the ROB, s_i as its execution start cycle, f_i as its execution finish cycle, and c_i as its commit cycle. We calculate the throughput induced by a ROB of size ROB using the following instruction-level dynamical system:

$$a_i = c_{i-\text{ROB}},\tag{1}$$

$$s_i = \max\Big(a_i, \max\big\{f_d \,|\, d \in \text{Dep}(i)\big\}\Big),\tag{2}$$

$$f_i = \text{RespCycle}\Big(s_i, \text{instr}_i\Big),\tag{3}$$

$$c_i = \max(f_i, c_{i-1}),\tag{4}$$

for $i \geq 1$, where $c_i = 0$ for $i \leq 0$ by convention. Equation (1) enforces the size constraint of the ROB. Equation (2) accounts for the instruction dependency constraints. Equation (3) uses the function shown

in Algorithm 1 (§3.1) to determine the finish time of each instruction.[3] Equation (4) models the in-order commit constraint. Finally, the throughput for the jth window of k instructions is calculated as:

$$thr_{ROB}^j = \frac{k}{c_{kj} - c_{k(j-1)}}.\tag{5}$$

Load/Store queue. The Load and Store queues bound the number of issued memory instructions that have yet to be committed (in order). We briefly discuss the Load queue model (Store queue is analogous). It is identical to the ROB model, with two differences: (i) the calculations are performed exclusively for Load instructions, (ii) there are no dependency constraints: a Load is eligible to start as soon as it obtains a slot in the queue. After computing the commit cycle for each Load, we derive the throughput for each k-instruction window similarly to Equation (5). In these calculations, non-Load operations are assumed to be free and incur no additional latency.

Static bandwidth resources. These resources impose limits on the number of instructions (of a certain type) that can be serviced in a single clock cycle. For example, Commit, Fetch, Decode, and Rename widths constrain the throughput of all instructions. The throughput bound imposed by these resources is trivially their respective width. In contrast, issue queues restrict the throughput for a specific group of instructions, e.g., ALU, Floating-point, and Load-Store issue widths in our reference architecture. To compute the throughput bound imposed by such resources, we compute the processing time of the instructions that are constrained by that resource and assume non-affected instructions incur no additional latency. For instance, the throughput bound induced by the ALU issue width in the j-th window of k consecutive instructions is given by:

$$thr_{ALU}^j = \frac{k}{n_{ALU}^j} \times \text{ALU issue width},\tag{6}$$

where n_{ALU}^j is the number of ALU instructions in window j.

Dynamic constraints. Some resources impose constraints on a dynamic set of instructions determined at runtime based on the microarchitectural state. Analyzing such resources is more challenging. Two strategies that we have found to be helpful are to use simplified performance bounds or basic discrete-event simulation. We briefly discuss these strategies using two examples.

Load/Load-Store Pipes. Finite Load and Load-Store pipes limit the number of memory instructions that can be issued per cycle. Store instructions exclusively use Load-Store pipes, while Load instructions can utilize both Load pipes and Load-Store pipes. The allocation of instructions to these pipes depends on dynamic microarchitectural state, e.g., the precise order that memory instructions become eligible for issue and the exact pipes available at the time of each issue. Rather than model such complex dynamics, we derive simple upper and lower bounds on the throughput. Let n_{Load} and n_{Store} denote the number of Load and Store instructions in a k-instruction window, LSP the number of Load-Store pipes, and LP the number of Load pipes. The worst-case allocation of pipes is to issue Loads first using all available pipes, and only then begin issuing Stores using the Load-Store pipes. This allocation leaves the Load pipes idle while Stores

---

[3]We execute Equation (3) in order of instruction start times s_i to satisfy Algorithm 1's requirement for non-decreasing request cycles.

are being issued. It results in the maximum total processing time: $T_{max} = n_{Load}/(LSP+LP) + n_{Store}/LSP$, and thus a lower-bound on the throughput of the pipes component: $thr_{lower} = k/T_{max}$.

The best-case allocation is to grant Stores exclusive access to Load-Store pipes while concurrently using Load pipes to issue Loads. Once all Stores are issued, the Load Store pipes are allocated to the remaining Loads. Analogous to the lower bound, we can derive an upper bound on the throughput thr_{upper} based on this allocation (details omitted for brevity). We summarize these bounds using the distribution of thr_{lower} and thr_{upper} over all instruction windows.

I-cache fills and fetch buffers. We model these resources using simple instruction-level simulations. Here, we focus on I-cache fills for brevity. The maximum I-cache fills restricts the number of in-flight I-cache requests at any given time. This is a dynamic constraint, because whether an instruction generates a new I-cache request depends on the set of in-flight I-cache requests when it reaches the fetch target queue. Specifically, new requests are issued only for cache lines that are not already in-flight. We estimate the throughput constraint imposed by the maximum I-cache fills using a basic simulation of I-cache requests. This simulation assumes a backlog of instructions waiting to be fetched, restricted only by the availability of I-cache fill slots. Instructions are considered in order, and if they need to send an I-cache request, they send it as soon as an I-cache fill slot becomes available. We record the I-cache response cycle for each instruction in the simulation, and use it to calculate the throughput for each window of k consecutive instructions similarly to Equation (5).

3.2.2 Auxiliary Features. In addition to the primary features described above, we describe a few auxiliary features that capture nuances not covered by per-resource throughput analysis. We evaluate the impact of these auxiliary features in §5.2.2.

Pipeline stalls. Unlike resource constraints, modeling the effects of pipeline stalls caused by branch mispredictions and ISB instructions as an isolated component is not meaningful. The impact of stalls on performance depends on factors beyond the fetch stage, for instance, the inherent instruction-level parallelism (ILP) of the program, how long it takes to drain the pipeline, and how quickly the stall is resolved [30]. Rather than try to model these complex dynamics analytically, we incorporate two simple groups of features to assist the ML model with predicting the impact of pipeline stalls. First, we provide basic information about the extent of stalls: (i) the distribution of the number of ISBs in our windows of k consecutive instructions; (ii) the distribution of the count of the three branch types (§3.1) per instruction window, (iii) the overall branch misprediction rate obtained from trace analysis. Additionally, we provide the overall throughput calculated by our analytical ROB model (§3.2.1) for varying ROB sizes, ROB $\in \{1,2,4,8,...,1024\}$. The intuition behind this feature is that pipeline stalls effectively reduce the average occupancy of the ROB, lowering the backend throughput of the CPU pipeline. Therefore, the ROB model's estimate of how throughput varies versus ROB size can provide valuable context for how sensitive a program's performance is to pipeline stalls.

Latency distributions. We augment our primary throughput based features from §3.2 with three instruction-level latency distributions collected from the ROB model. Specifically, we provide the distribution of the time that instructions spend in the issue ($s_i - a_i$), execution

($f_i - s_i$), and commit ($c_i - f_i$) stages of the ROB model (Equations (1) to (4)) for ROB $\in \{1,2,4,8,...,1024\}$.[4] These latency distributions provide additional context that can be useful for understanding certain nuances of the performance dynamics. For example, the execution latency distribution indicates whether a program is load-heavy, which can be useful for predicting memory congestion.

3.3 ML Model

The final component of Concorde's design is a lightweight ML model that predicts the CPI of a program on a specified architecture. The model is a shallow multi-layer perceptron (MLP) (details in §4) that takes as input a concatenation of (i) the performance distributions corresponding to the target microarchitecture (§3.2.1), (ii) the auxiliary features (§3.2.2), and (iii) a 20-dimensional vector of parameters ($\vec{p}$) representing the target microarchitecture (Table 1). We train the ML model on a dataset constructed by randomly sampling diverse program regions and microarchitectures. We simulate each sample program region and sample microarchitecture using the cycle-level simulator to collect the ground-truth target CPI. To train the ML model, we use a loss function that measures the relative magnitude of CPI prediction error, as follows:

$$Loss(\hat{y}, y) = \frac{|\hat{y} - y|}{y}, \tag{7}$$

where $\hat{y}$ denotes the predicted CPI and y denotes the CPI label.

4 Concorde's Implementation Details

Trace analyzer and analytical models. We implement the trace analyzer and analytical models in C++. Trace analysis performs in-order cache simulation (per memory configuration) and branch prediction simulation (for TAGE). To precompute the performance features for a program, we run the trace analyzer for each memory configuration to derive the Concorde trace, and then run the analytical models for all parameter values of each CPU resource independently. Our current implementation uses a single thread, but all analytical model invocations could run in parallel. To calculate performance distributions, Concorde uses a window size of $k = 400$.

Dataset. Unless specified otherwise, Concorde uses a dataset with 789,024 data points for training, with an additional 48,472 unseen (test) data points reserved for evaluation. Every data point is constructed by independently sampling a microarchitecture, and a 100k-instruction region. To sample a microarchitecture, we independently pick a random value from Table 1 for every parameter. Concorde's large microarchitecture space ($\sim 2 \times 10^{23}$) ensures that test microarchitectures are almost surely unseen during training, preventing memorization. To sample a program region, we sample a program from Table 2, sample a trace of the chosen program randomly with probability proportional to trace length, and sample a region randomly from this trace. Figure 4 shows the average overlap of test program regions with their closest training region (the training region with maximum instruction overlap) for every program. The overlap is 16.86% on average, and less than 10% for the majority of programs.

Lightweight ML model. Concorde's ML component uses a fully connected MLP with a 3873-dimensional input layer and two hidden

---

[4]The execution latency does not depend on ROB size; therefore, we only include one copy of the execution latency distribution feature.

Table 2: Workload space with 5486B instructions from 29 programs

| Type	Name	Traces	Instructions (M)
Proprietary	Compression **(P1)**	4	1845
	Search1 **(P2)**	168	17854
	Search4 **(P3)**	170	23188
	Disk **(P4)**	168	23441
	Video **(P5)**	268	26981
	NoSQL Database1 **(P6)**	168	30283
	Search2 **(P7)**	84	52989
	MapReduce1 **(P8)**	84	56677
	Search3 **(P9)**	1334	69277
	Logs **(P10)**	191	75845
	NoSQL Database2 **(P11)**	84	91274
	MapReduce2 **(P12)**	84	104750
	Query Engine&Database **(P13)**	790	1195128
Cloud Benchmark	Memcached **(C1)**	8	2791
	MySQL **(C2)**	84	9283
Open Benchmark	Dhrystone **(O1)**	1	174
	CoreMark **(O2)**	1	335
	MMU **(O3)**	132	18475
	CPUtest **(O4)**	138	95215
SPEC2017	505.mcf_r **(S1)**	19	197232
	520.omnetpp_r **(S2)**	20	214749
	523.xalancbmk_r **(S3)**	20	214749
	541.leela_r **(S4)**	20	214749
	548.exchange2_r **(S5)**	20	214749
	531.deepsjeng_r **(S6)**	20	214749
	557.xz_r **(S7)**	38	408022
	500.perlbench_r **(S8)**	41	440235
	525.x264_r **(S9)**	44	472447
	502.gcc_r **(S10)**	94	999282
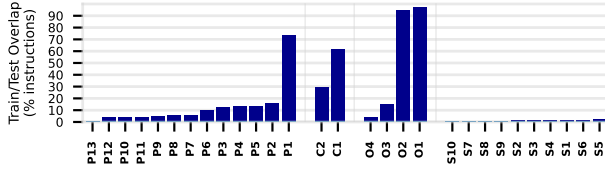
Figure 4: Average test/train overlap across benchmarks

layers with sizes 256 and 128 that outputs a scalar CPI prediction. For encoding every input distribution to the ML model, Concorde uses a 101-dimensional encoding which includes 50 fixed equally-spaced percentiles of the original distribution, 50 fixed equally-spaced percentiles of the size-weighted distribution,[5] and the average value. Table 3 shows the breakdown of the input dimensions to the features detailed in §3. Note that in the first column, we do not include throughput distributions for static bandwidth resources that remain constant throughout the entire program such as Commit width. In the last column, we use one-hot vectors for the branch predictor type and the state of prefetching. We use the AdamW [56] optimizer with weight decay of 0.3, learning rate of 0.001 that halves after {10,14,18,22}k steps, and batch size of 50k to train for 1521 epochs.

Table 3: ML model's 3873-dimensional input.

(§3.2.1) Per-resource throughput analysis	(§3.2.2) Pipeline stalls	(§3.2.2) Latency distributions	(Table 1) Target microarchitecture
$11 \times 101 = 1111$	$4 \times 101 + 1 + 11 \times 1 = 416$	$(1 + 2 \times 11) \times 101 = 2323$	$19 + 2 \times 2 = 23$

---

[5]The size weighted distribution is a transformation of the original distribution of a non-negative random variable in which we weight every sample by its value. This transform highlights the tail of the original distribution.
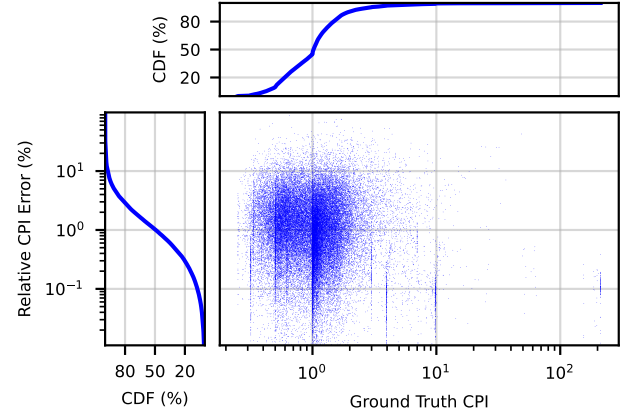
Figure 5: Scatterplot of Concorde's CPI prediction error vs. the CPI for unseen (test) pairs of 100k-instruction regions and microarchitectural parameters. The plots on the sides show the distributions of CPI and prediction error. The average error is 2%, with only 2.5% of samples having larger than 10% error.
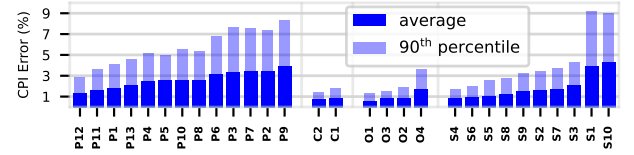
Figure 6: Error breakdown across benchmarks

5 Evaluation

We evaluate Concorde's CPI prediction accuracy and speed in §5.1. In §5.2, we dive deeper into its accuracy and our design choices.

5.1 Concorde's Accuracy and Speed

Accuracy on random microarchitectures. To highlight the generalization capability of Concorde across microarchitectures, we first evaluate its accuracy on the unseen test split of the dataset (§4), where microarchitectures are randomly sampled. Figure 5 illustrates Concorde's relative CPI prediction error (Equation (7)) vs. the ground-truth CPI from our gem5-based cycle-level simulator. The top and left plots besides the axes show the distributions of the CPI and Concorde's prediction error across all samples. Concorde achieves an average relative error of only 2.03%. Moreover, its error has a small tail; only 2.51% of test samples have errors larger than 10%. Recall from §4 that such accuracy cannot be achieved by memorization since the microarchitectures in our test dataset are not seen in the training samples. Figure 6 shows the error breakdown across programs. While some programs are more challenging than others, the average error and P90 is capped at 4.2% and 8.9%, respectively. Furthermore, the errors do not correlate well with the per program train/test overlaps in Figure 4. For instance, Concorde's average error is less than 1% for S4 and S6, and only slightly over 1% for P12, all of which have train/test overlaps less than 3.5%. This highlights Concorde's effectiveness in generalizing (in distribution) across program regions.
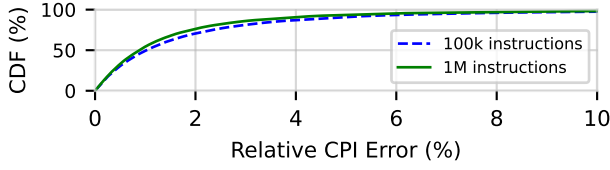
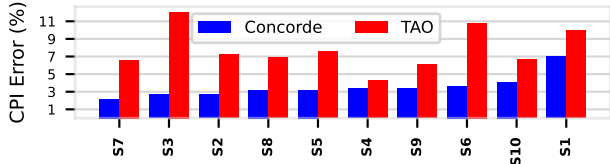Figure 7: Concorde is more accurate on longer program regions.

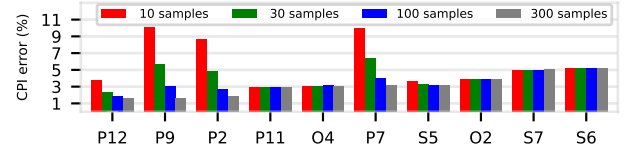Figure 8: Concorde is more accurate than TAO on all programs.

Figure 9: Accuracy for long programs vs. number of samples
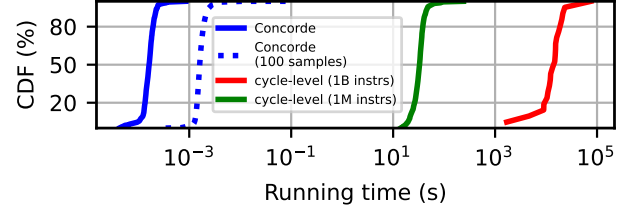

Figure 10: Concorde is five/seven orders of magnitude faster than a cycle-level simulator on 1M/1B-instruction program regions.

Longer program regions. Recall that one of the main design goals of Concorde is to avoid a run time cost that scales with the number of instructions ($O(L)$). Hence, unlike cycle-level simulators that operate on sequence of instructions, Concorde takes as input a fixed-size performance characterization of the program independent of the program length. To evaluate Concorde on longer program regions, we create a new dataset similar to the original one (§4) with longer program regions of 1M instructions and re-train Concorde on it. Figure 7 shows the distribution of Concorde's relative CPI prediction error over the unseen test split of this dataset (solid green line). The average error is 1.75% and only 1.82% of cases have larger than 10% error, which is slightly better than Concorde's accuracy on the original 100k-instruction region dataset (dashed blue line). We hypothesize that this is because the average CPI has less variability over longer regions due to the phase behaviors getting averaged out (which we confirmed by comparing the CPI variance in the two cases). This reduced variance makes the learning task easier for longer regions, boosting Concorde's accuracy.

Accuracy on ARM N1. To assess Concorde's accuracy on a realistic microarchitecture, we evaluate its CPI predictions for ARM N1 (Table 1), using the 100k-instruction regions in the test split of our dataset (§4). It has an average error of 3.25% with 4.39% of program regions having errors larger than 10%, which is a slight degradation in the accuracy compared to random microarchitectures. We believe that this is because randomly sampled microarchitectures are more likely to have a single dominant bottleneck while ARM N1 is designed to be balanced.

Comparison with TAO [71]. We compare Concorde with TAO, the previous SOTA in sequence-based approximate performance modeling. Unlike Concorde, TAO does not generalize without additional retraining beyond a single microarchitecture. Hence, we train it for ARM N1 on a dataset of 100M randomly sampled instructions from SPEC2017 programs (Table 2). Figure 8 compares TAO's CPI prediction accuracy on 100k-instruction regions from SPEC2017 programs with Concorde's; Concorde is more accurate for every single program. This is despite the fact that Concorde is trained on random microarchitectures whereas TAO is specialized to ARM N1.

Accuracy on long programs. Using Concorde's CPI predictions for finite program regions as the building block, we can estimate the CPI for arbitrarily long programs by randomly sampling program regions and averaging their predicted CPIs. As an example, we use the 1M-instruction region model to predict the CPI for programs with 1B instructions. Figure 9 shows Concorde's accuracy in predicting CPI for ARM N1, across ten such 1B-instruction programs, with four sampling levels. As shown, with as little as 100 samples, Concorde's error gets below 5% for every program, with an average error of 3.5%. Using 300 samples, the average error decreases to 3.16%.

Concorde's Speed. Figure 10 shows the running time distribution of Concorde and our gem5-based cycle-level simulator. We measure the running time of Concorde and the cycle-level simulator on a single CPU core. For these experiments, we simulate from the first instruction of each trace, to avoid extra warmup overheads for the cycle-level simulator. The average running time of Concorde (solid blue) is 168μsec. Compared to our cycle-level simulator, Concorde achieves an average speedup of more than 2×10^5 for 1M-instruction regions. Furthermore, Concorde's running time does not change with the length of the instruction region (e.g., 100k $\rightarrow$ 1M) since the size of its input distributions are fixed. In contrast, the cycle-level simulator's running time scales with the program region length, e.g., 487$\times$ by increasing the length from 1M (green) to 1B (red) instructions. Recall that to estimate the CPI of 1B-instruction programs in Figure 9, we used Concorde's predictions on randomly sampled 1M-instruction regions. The dashed dotted line in Figure 10 shows the running time distribution for processing 100 samples, measured on the same CPU. Even with 100 sequential samples, Concorde's average running time (1.7$m$sec) is about 10^7 times faster than the cycle-level simulator for programs with 1B instructions. Additionally, the running time of the cycle-level simulator exhibits a high variance due to its dependence on the number of cycle-level events, which varies with programs and microarchitectures. In contrast, Concorde's running time has minimal variance since its computation is deterministic irrespective of the program or the microarchitecture. Note that the reported speedups do not include the benefits of batching Concorde's calculations on accelerators such as GPUs, which would further amplify its advantage.
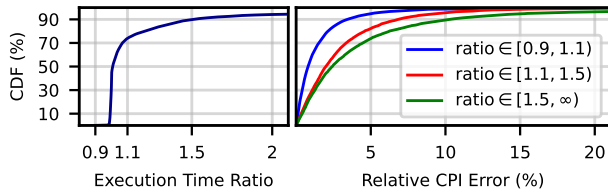
Figure 11: Although the ML component of `Concorde` corrects for a large portion of errors in estimates of instruction execution times from trace analysis, this error plays a significant role in the tail of `Concorde`'s error distribution.

5.2 Deep Dive

5.2.1 What constitutes Concorde's error tail? Recall from §5.1 that `Concorde` has a small error tail, where tail is defined as cases with larger than 10% error. Here, we detail our attempts to understand some of the factors responsible for the tail.

Discrepancy in raw execution times. Recall that `Concorde`'s analytical models use approximate instruction execution times derived in trace analysis (§3.1). As we discussed in §3.1, these estimated executions times can differ from the actual values observed during timing simulations. Figure 11 (left) shows the distribution of the ratio of the actual instruction execution times in timing simulations to their estimates from trace analysis, across 100k-instruction regions in our test dataset (§4). More than 10% of program regions have a ratio larger than 1.5. These discrepancies can occur for a variety of reasons, including memory congestion, partial store forwarding, etc. that we do not account for in trace analysis.

With high errors in their raw inputs, our analytical models will be inaccurate. We bucketize program regions based on the above ratio into three buckets, and plot `Concorde`'s prediction error distribution for samples in each bucket (Figure 11, right). The result shows that `Concorde`'s prediction error increases for the buckets with larger execution time discrepancy. But its accuracy remains quite high, even with significant discrepancies, e.g., achieving an average error of 4.53 in cases with ratio larger than 1.5. This shows that the ML component of `Concorde` can correct for significant errors in the analytical models. Nonetheless, errors in execution time estimates from trace analysis account for a large portion of the tail of `Concorde`'s error distribution. Among test program regions that have errors larger than 10%, 41.5% have execution time ratios larger than 1.5 (whereas only about 10% of all program regions have a ratio larger than 1.5).

Table 4: `Concorde` successfully learns the effect of branch prediction.

Number of branch mispredictions	$[0, 1000)$	$[1000, 5000)$	$[5000, \infty)$
`Concorde`'s average error (%)	2.16	2.12	**1.82**
%(`Concorde`'s error > 10%)	3.11	2.43	**1.95**

Branch prediction. Recall from §3.2 that unlike other CPU components, `Concorde` does not analytically model branch mispredictions. Instead, it relies on a set of auxiliary features that are helpful for learning the effect of pipeline stalls. We will show in §5.2.2 that these features indeed boost `Concorde`'s overall accuracy. Here, we study whether branch mispredictions are another source of `Concorde`'s error tail. Table 4 categorizes `Concorde`'s accuracy based on the number of branch mispredictions in 100k-instruction regions of the test
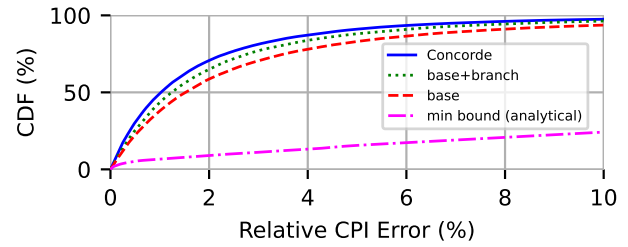
Figure 12: Ablation of `Concorde`'s design components

dataset (§4). Intriguingly, `Concorde`'s accuracy improves as the number of mispredictions increases, with an average error of 1.82% in regions with over 5,000 branch mispredictions. We hypothesize that this is because programs with large number of stalls have low parallelism and simpler dynamics, making them easier to predict. This result confirms that `Concorde`'s branch-related features are sufficient.

5.2.2 Ablation study. Recall from §3.2.2 that `Concorde` uses a few auxiliary features to augment the primary per-component throughput distributions. We train several variants of `Concorde` to understand the impact of these features. For reference, we begin with a simple minimum over the per-component throughput bounds (no ML). As shown by the pink line in Figure 12, this has poor accuracy, achieving an average error of 65% (and 11% in cases with no branch misprediction). `Concorde`'s base ML model, which takes as input the per-component throughput distributions along with the branch misprediction rate, significantly boosts accuracy (red line), achieving an average error of 3.32% with errors exceeding 10% in only 4.48% of cases. Further adding the auxiliary features (§3.2.2) related to pipeline stalls (green) and instruction latency distributions (blue) provides incremental accuracy improvements, reducing average error to 2.4% and 2.03% and the percentage of samples with errors larger than 10% to 3.7% and 2.51%, respectively.

In addition, we ablated the ML model size, and the choice of k, the length of instruction windows for throughput calculations (§3.2). Expanding the model to three hidden layers of sizes 512, 256, and 128 slightly lowers the average error on random microarchitectures from 2.03% to 1.85%, while reducing it to a single hidden layer of size 256 increases the error to 3.91%. Varying $k \in \{100, 200, 400\}$ did not have a significant effect on our results.

5.2.3 Preprocessing cost. Precomputing the performance features for a 1M-instruction region for all the 2.2×10^{23} parameter combinations in Table 1 takes 3959 seconds on a single CPU core — equivalent to the time required for 107 cycle-level simulations with similar warmup. This includes 195s for trace analysis (§3.1) and 3764s for analytical modeling (§3.2). Trace analysis comprises one TAGE, 40 D-cache, and 20 I-cache simulations. The dominant factors in analytical modeling are 40×1024 ROB model invocations (3327s) and 40×256 invocations of the Load/Store queue models (211s/211s). The precomputed performance features occupy 24MB in uncompressed NumPy [38] format.

Table 1 sweeps all parameters in increments of 1, but such a fine granularity is typically not necessary in practice. Quantizing the parameter space can significantly reduce the precomputation time. For example, considering powers of 2 for ROB, Load and Store queues,
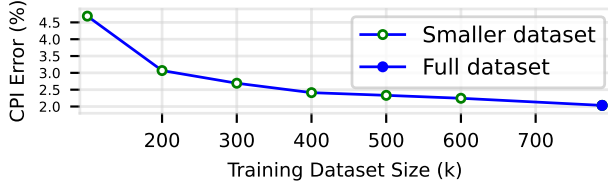
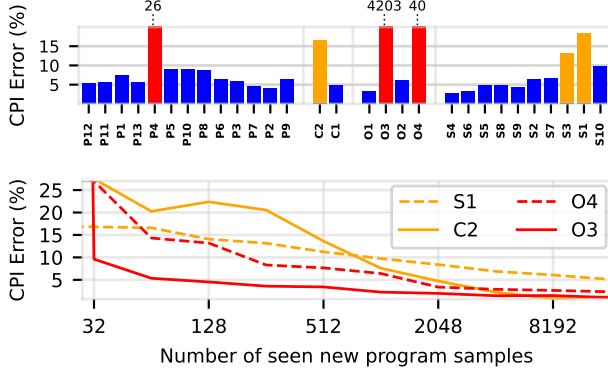Figure 13: Impact of training dataset size on Concorde's accuracy

Figure 14: Errors can be high on unseen programs (*top*). However, Concorde recovers quickly as it trains on their samples (*bottom*).

i.e, $\mathtt{ROB} \in \{1,2,4,...,1024\}$, $\mathtt{Load/Store}$ queue $\in \{1,2,4,...,256\}$, reduces the analytical modeling time to 63s , lowering the total preprocessing time for the resulting 1.8×10^{18} parameter combinations to 257s (7 cycle-level simulations). Techniques like QEMU [12] could further reduce trace analysis time [25].

5.2.4　Training cost. ML training takes 3 hours on a TPU-v3-8 [1] cloud server with 8 TensorCores (2.7 hours on an AMD EYPC Milan processor with 64 cores). To generate the training dataset (§4), we only run trace analysis and analytical modeling for one (randomly selected) microarchitecture for each program region.

Using 512 cores, it takes 19.4 hours to create the more expensive 1M-instruction region dataset with 837,496 data points. This includes 16.8 hours for cycle-level simulations (to generate the CPI labels), 2.2 hours for trace analysis, and 26 minutes for analytical modeling.

Although training is a one-time cost, it can be reduced at a slight degradation in model accuracy. Figure 13 shows that reducing the training dataset size to 200k samples gradually increases the relative CPI error from 2.01% to 3.07%. Further reduction to 100k samples increases the error to 4.67%.

5.2.5　Out-of-Distribution (OOD) Generalization. Like any ML model, we expect Concorde to be trained on a diverse dataset of programs representative of programs of interest. However, to stress test program generalization, for every program, we train Concorde on a dataset that excludes all its traces and evaluate the accuracy of the resulting model on that program. Figure 14 (top) shows the average OOD error for all programs. As expected, the error increases, with some programs being affected more than others. 23 programs (blue) have OOD error below 10%. The 3 programs with the highest error (red) are synthetic microbenchmarks testing specific microarchitectural capabilities. These programs are unlike any other in the dataset.

For instance, O3 (a memory test), has much higher CPIs compared to other programs in Table 2. The 3 remaining programs (orange), with OOD error of about 15%, are real workloads that stand out from the others. For example, as we will see in §6, S1 has the highest sensitivity to cache sizes among all workloads in Table 2.

Compared to generalization across microarchitectures, OOD generalization across programs is not a major concern. Programs and benchmarks used for CPU architecture exploration are relatively stable. For example, SPEC CPU benchmarks are only updated every few years, and we similarly see infrequent updates to our internal suite of benchmarks. Nevertheless, we quantify the cost of "onboarding" new programs into Concorde for O3, O4 (2 highest red bars), and S1, C2 (2 highest orange bars). For each of these programs, we train Concorde on all other programs together with a varying number of samples from the new program. As Figure 14 (bottom) shows, 2k (8k) samples from the new program are enough for Concorde to reach within 5% (2%) of the error floor achieved by the model trained on the full dataset with $\sim 30k$ samples per program (Figure 6). O3 and O4 have the steepest drop in error, which is likely due to the regularity of these synthetic benchmarks.

5.2.6　Can Concorde predict metrics other than CPI? Although we focused on CPI in designing our analytical models, Concorde's rich performance distributions are useful for predicting other metrics as well. To illustrate this point, we retrain Concorde's ML model (without changing hyperparameters) to predict the average *Rename queue occupancy* (%) and *average ROB occupancy* (%), on the same dataset used for CPI (§4). On unseen test samples, Concorde achieves an average prediction error of 2.50% and 2.23%, respectively, vs. the ground-truth metrics from our gem5-based simulator.

6　Fine-Grained Performance Attribution

Beyond predicting performance, architects often need to understand *why* a program performs as it does on a certain design. In this section, we present a methodology for fine-grained attribution of performance to different microarchitectural components. Our method can be used in conjunction with any performance model $y = f(\vec{\mathbf{x}}, \vec{\mathbf{p}})$ relating microarchitectural parameters to performance. But as we will see, it is computationally impractical for expensive models such as cycle-level simulators. Concorde's massive speedup over conventional methods makes such large-scale, fine-grained analyses possible.

Concretely, our goal is to quantify the relative impact of different microarchitectural parameters $\vec{\mathbf{p}}$ on the performance of a program $\vec{\mathbf{x}}$. This requires identifying the dominant performance bottlenecks. Many existing performance analysis techniques (e.g., Top-Down [92], CPI stacks [29]) rely on hardware performance counters to identify bottlenecks. We seek to obtain similar insights using only a performance model $y = f(\vec{\mathbf{x}}, \vec{\mathbf{p}})$ like Concorde that outputs the (predicted) performance of a program given the microarchitectural parameters. Performance of a single microarchitecture $\vec{\mathbf{p}}$ provides no information about which of the parameters p_i are important. Thus, we use parameter *ablations*, where we change some parameters and observe their impact on performance. Intuitively, parameters that have a large effect on performance when modified are more important. Parameter ablations are commonly used to understand the impact of design choices [23, 37, 67, 91]. A typical approach is to start with microarchitectural parameters $\mathbf{p}^{base}$ representing a *baseline* design, and modify
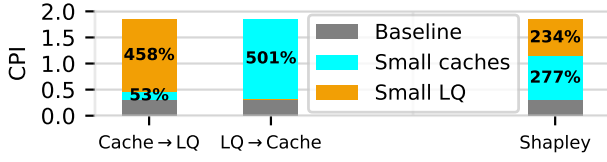
Figure 15: Changing the order of parameter ablations (Cache → Load Queue vs. Load Queue → Cache) leads to different conclusions about their relative importance. Shapley values provide a fair, order-independent performance attribution to design parameters.

one parameter (dimension) at a time to reach a *target* design with parameters $\mathbf{p}^{target}$. After each parameter change, the incremental change in performance is reported as the contribution of that parameter to the total performance difference between $\mathbf{p}^{base}$ and $\mathbf{p}^{target}$.

Although this methodology is standard, it can be difficult to draw sound conclusions from parameter ablations when there are multiple inter-related factors effecting performance. The issue is that the *order* of parameter ablations can change the perceived importance of different factors. To illustrate, suppose we are interested in quantifying the relative impact of (i) limited cache size and (ii) limited Load queue size on a memory-intensive workload. As a baseline, we consider a "big core" with all parameters set to their largest value in Table 1, particularly: L1d/L1i cache = 256kB, L2d cache = 4MB, and Load queue = 256. Figure 15 shows the CPI achieved by this baseline (Grey bars) on a sample trace from the `Search3` workload.

Next, we consider two parameter ablations atop the baseline, where we reduce the cache sizes and the Load queue size to our target values: L1d/L1i cache = 64kB, L2d cache = 1MB, Load queue = 12. In one ablation, we first reduce the cache sizes and then the Load queue size; in the other ablation, we reduce the Load queue size first, then reduce cache size. The two left bars in Figure 15 show the CPI trajectory for both routes, along with the CPI increase associated with reducing cache sizes and Load queue size in each case. The overlaid numbers show the percentage CPI increase relative to the baseline following each parameter change.

The two orders of parameter ablations lead to entirely different conclusions. The (Cache → Load queue) order suggests that reducing the Load queue size has about 9× larger impact than reducing the cache sizes. The (Load queue → Cache) order, on the other hand, says that reducing the Load queue has negligible effect and the performance degradation is almost entirely caused by reducing the cache sizes. Neither of these interpretations is correct. The reality is that the effects of cache and Load queue size are intertwined. A large Load queue can mitigate the performance hit of small caches for this workload (due to increased parallelism). Similarly, a large cache size can perform well despite a small Load queue (since Load instructions complete quickly). It is only when *both* the Load queue and cache sizes are small that we incur a large performance hit.

Shapley value: a fair, order-independent attribution. A natural way to remove the bias caused by a specific order of parameter ablations is to consider the average of *all* possible orders. Let Π denote the set of all permutations of the parameter indices $D \triangleq \{1, ..., d\}$. Each permutation $\pi \in \Pi$ corresponds to one order of ablating the parameters from $\mathbf{p}^{base}$ to $\mathbf{p}^{target}$, resulting in a different value for the incremental effect of modifying parameter i.

Specifically, define $\mathbf{p}_\pi(j) \triangleq (\mathbf{p}^{target}_{\pi_{1:j}}, \mathbf{p}^{base}_{\pi_{j+1:d}})$ to be the jth microarchitecture encountered in the ablation study based on order π, i.e., parameters $\pi_1, ..., \pi_j$ are set to their target values and the rest remain at the baseline. Let k denote the position of parameter i in the order π. Then, the incremental effect of parameter i in order π is: $\Delta^\pi_i \triangleq f(\mathbf{x}, \mathbf{p}_\pi(k)) - f(\mathbf{x}, \mathbf{p}_\pi(k-1))$. To assign an overall attribution to parameter i, we take the average over all permutations:

$$\varphi_i \triangleq \frac{1}{|\Pi|} \sum_{\pi \in \Pi} \Delta^\pi_i, \tag{8}$$

where $|\Pi| = d!$ is the total number of permutations.

The quantity defined in Equation (8) is referred to as the *Shapley value* [78] in economics. The concept arises in cooperative game theory, where a group of M players work together to generate value $v(M)$. Shapley's seminal work showed that the Shapley value is a "fair" distribution of $v(M)$ among the players, in that it is the only way to divide $v(M)$ that satisfies certain desirable properties (refer to [78] for details). In our context, the "players" are the different microarchitectural components, and the "value" to be divided is the performance difference between the baseline and target microarchitectures.[6] The Shapley value is used in many areas of science and engineering [5, 13, 33, 60, 64–66], but to our knowledge, we are the first to apply it to performance attribution in computer architecture.

The rightmost bar in Figure 15 shows the Shapley values corresponding to cache and Load queue sizes in the above example. The Shapley value correctly captures that small caches and small Load queue sizes are together the culprit for high CPI relative to the baseline, with a slightly larger attribution to small caches.

Case study. To illustrate Shapley value analysis, we use it for fine-grained performance attribution in a target design based on the ARM N1 core [73] (parameters in Table 1) across our entire pool of programs. As baseline, we use the "big core" configuration mentioned above (perfect branch prediction, other parameters set to their max).

Computing Shapley values is computationally expensive. For each program (region), using Eq. (8) directly requires $d! \times d$ performance evaluations, where d is the number of parameters. We can calculate an accurate Monte Carlo estimate of Eq. (8) using a few hundred randomly sampled permutations, but even that requires a massive number of performance evaluations for large-scale analyses. For example, estimating Shapley values for our corpus of workloads (Table 2) using 2000 sample regions per program and 200 permutations of parameter orders requires ~143M CPI evaluations in total. This is impractical with existing cycle-level simulators; we estimate it would take about a month on a 1024-core server! With `Concorde`, the computation takes about an hour on a TPU-v3 [1] cloud server with 8 TensorCores.

Figure 16 shows the result of our analysis. The grey bars show the reference CPI achieved by the "big core" baseline, while the entire bars show the CPI achieved by ARM N1. Within each workload group, i.e., proprietary, cloud, open-source, and SPEC2017, the programs are sorted based on the relative CPI increase of ARM N1 compared to the baseline. For instance, in SPEC2017 benchmarks, `S1(505.mcf_r)` has the largest relative jump in CPI for ARM N1, whereas `S7(557.xz_r)` has the smallest relative CPI increase.

The colored bars in Figure 16 show the Shapley value for each microarchitectural component, i.e., how much each component in

---

[6]It is not difficult to see that: $\sum_i \varphi_i = f(\mathbf{x}, \mathbf{p}^{target}) - f(\mathbf{x}, \mathbf{p}^{base})$.
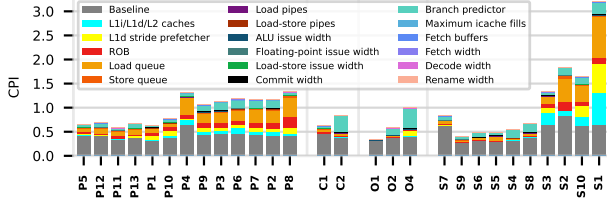
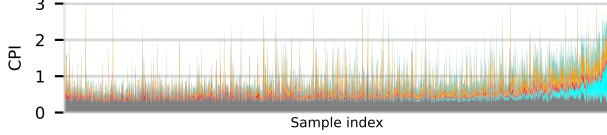**Figure 16: CPI attribution for ARM N1 across all workloads**

Figure 17: CPI attributions for all `Search3(P9)` sample regions

ARM N1 is responsible for the performance degradation relative to the baseline. This provides a bird's eye view of the dominant performance bottlenecks across the entire corpus of workloads. For instance, all the proprietary programs and half of the SPEC2017 programs are mainly backend bound on ARM N1, with prominent bottlenecks being the Load queue size and ROB size. A few programs such as S4(541.leela_r) (a chess engine using tree search) are frontend bound, with the TAGE branch predictor the most prominent frontend bottleneck. Cache sizes and L1 prefetching have a large effect on some SPEC2017 benchmarks (e.g., S10(502.gcc_r), S1) but a less pronounced impact on our proprietary workloads, perhaps in indication of our programs being cache-optimized.

For a deeper look, we can further zoom into the behavior of a single program. For example, Figure 17 shows the CPI attribution for all 2000 sample regions of P9, sorted on the x-axis based on their sensitivity to cache size. Although the P9 bar in Figure 16 shows limited sensitivity to cache sizes on average, the zoomed-in view in Figure 17 shows high sensitivity to cache size in about 10% of the sampled regions, highlighting the different phase behaviors in the program [39].

7 Related Work

Conventional CPU simulators. Conventional simulators [3, 4, 6, 11, 15, 19, 22, 28, 32, 34, 42, 63, 69, 72, 74, 75, 84, 94, 95] aim to balance speed and accuracy, leveraging higher abstraction levels [19, 32], or decoupled simulations of core and shared resources [28, 75]. While these methods achieve faster simulations, they often compromise flexibility or accuracy. Hardware-accelerated simulators [22, 47] improve speed but require extensive development effort for validation. Statistical modeling tools [27, 36, 46, 79–81, 90] reduce computation by sampling representative segments [80, 90] or generating synthetic traces [26, 68], but they trade off flexibility and detailed insights.

Analytical performance models. Analytical models [2, 21, 41, 48, 49, 86, 87] provide quick performance estimates using parameterized equations and microarchitecture-independent profiling. Such methods are ideal for crude design space exploration but often lack the granularity needed to capture intricate μ-architectural dynamics.

ML- and DL-based performance models. Conventional ML based models [24, 43–45, 51–53, 76, 89, 96] predict performance over constrained design spaces but often struggle with fine-grained program-hardware interactions. In contrast, Concorde demonstrates robust generalization across unseen programs and microarchitectures. Recent DL-based models [20, 54, 61, 70, 71, 83, 93] improve modeling at a higher abstraction levels at the cost of higher compute. Notably, PerfVec[54] (significant training overhead) and TAO[71] (additional finetuning for unseen configurations) emphasize per-instruction embeddings. Our work diverges by compactly capturing program-level performance characteristics using analytical models and fusing them with a lightweight ML model for capturing dynamic behaviors.

8 Final Remarks

The key lesson from Concorde is that decomposing performance models into simple analytical representations of individual microarchitectural components, fused together by an ML model capturing higher-order complexities, is very effective. It enables a method that is both extremely fast and accurate. Before concluding, we remark on some limitations of our work and directions for future research.

Concorde does not obviate the need for detailed simulation. It enables large-scale design-space explorations not possible with current methods (e.g., Shapley value analysis (§6)), but some analyses will inevitably require more detailed models. Moreover, Concorde needs training data to learn the impact of design changes (e.g., different parameters), which we currently obtain using a reference cycle-level simulator. In principle, Concorde could be trained on data from any reference platform, including emulators and real hardware.

As an ML approach, Concorde's accuracy is inherently statistical. Our results show high accuracy for a vast majority of predictions, but there is a small tail of cases with high errors. We have analyzed some of the causes of these errors (§5.2.1), and we believe that further improvements to the analytical models (e.g., explicitly modeling in-memory congestion) can further reduce the tail. But we do not expect that tail cases can be eliminated entirely. Alternatively, a large set of techniques exist for quantifying the uncertainty of such ML models [9, 10, 31]. Future work on providing confidence bounds would allow designers to detect predictions with high potential errors and crosscheck them with other tools.

Finally, Concorde was just one example of our compositional analytical-ML modeling approach. We believe that the methodology is broadly applicable and we hope that future work will extend it to other use cases, such as modeling multi-threaded systems, uncore components, and other architectures (e.g., accelerators).

Acknowledgments

References

[1] 2024. *TPU v3*. https://cloud.google.com/tpu/docs/v3
[2] Andreas Abel, Shrey Sharma, and Jan Reineke. 2023. Facile: Fast, Accurate, and Interpretable Basic-Block Throughput Prediction. In *IISWC*.
[3] Jung Ho Ahn, Sheng Li, O Seongil, and Norman P Jouppi. 2013. McSimA+: A Many-core Simulator with Application-level+Simulation and Detailed Microarchitecture Modeling. In *ISPASS*.
[4] Ayaz Akram and Lina Sawalha. 2019. A Survey of Computer Architecture Simulation Techniques and Tools. *IEEE Access* (2019).
[5] Johan Albrecht, Delphine François, and Koen Schoors. 2002. A Shapley Decomposition of Carbon Emissions without Residuals. *Energy policy* (2002).
[6] Marco Antonio Zanata Alves, Carlos Villavieja, Matthias Diener, Francis Birck Moreira, and Philippe Olivier Alexandre Navaux. 2015. SiNUCA: A Validated Micro-Architecture Simulator. In *HPCC*.
[7] Amazon. 2023. *AWS Unveils Next Generation AWS-Designed Chips*. https://press.aboutamazon.com/2023/11/aws-unveils-next-generation-aws-designed-chips
[8] SEZNEC Andre. 2006. A Case for (Partially)-TAgged GEometric History Length Predictors. *JILP* (2006).
[9] Anastasios N. Angelopoulos, Rina Foygel Barber, and Stephen Bates. 2024. Theoretical Foundations of Conformal Prediction. arXiv:2411.11824
[10] Anastasios N Angelopoulos, Stephen Bates, et al. 2023. Conformal Prediction: A Gentle Introduction. *Foundations and Trends® in Machine Learning* (2023).
[11] Todd Austin, Eric Larson, and Dan Ernst. 2002. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer* (2002).
[12] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *ATEC*.
[13] Leopoldo Bertossi, Benny Kimelfeld, Ester Livshits, and Mikaël Monet. 2023. The Shapley Value in Database Management. *ACM SIGMOD Record* (2023).
[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 Simulator. *SIGARCH Comput. Archit. News* (2011).
[15] Hadi Brais, Rajshekar Kalayappan, and Preeti Ranjan Panda. 2020. A Survey of Cache Simulators. *Comput. Surveys* (2020).
[16] Derek Bruening. 2024. *DynamoRIO: Tracing and Analysis Framework*. https://dynamorio.org/page_drcachesim.html
[17] Derek Lane Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph. D. Dissertation. Massachusetts Institute of Technology.
[18] Victoria Caparrós Cabezas and Markus Püschel. 2014. Extending the Roofline Model: Bottleneck Analysis with Microarchitectural Constraints. In *IISWC*.
[19] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation. In *SC*.
[20] Isha Chaudhary, Alex Renda, Charith Mendis, and Gagandeep Singh. 2024. COMET: Neural Cost Model Explanation Framework. *MLSys*.
[21] Xi E Chen and Tor M Aamodt. 2011. Hybrid Analytical Modeling of Pending Cache Hits, Data Prefetching, and MSHRs. *TACO* (2011).
[22] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. 2007. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In *MICRO*.
[23] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators. In *ISCA*.
[24] Christophe Dubach, Timothy Jones, and Michael O'Boyle. 2007. Microarchitectural Design Space Exploration Using an Architecture-Centric Approach. In *MICRO*.
[25] Tran Van Dung, Ittetsu Taniguchi, and Hiroyuki Tomiyama. 2014. Cache Simulation for Instruction Set Simulator QEMU. In *DASC*.
[26] Lieven Eeckhout, Sebastien Nussbaum, James E Smith, and Koen De Bosschere. 2003. Statistical Simulation: Adding Efficiency to the Computer Designer's Toolbox. *IEEE Micro* (2003).
[27] Lieven Eeckhout, John Sampson, and Brad Calder. 2005. Exploiting Program Microarchitecture Independent Characteristics and Phase Behavior for Reduced Benchmark Suite Simulation. In *IISWC*.
[28] Muhammad ES Elrabaa, Ayman Hroub, Muhamed F Mudawar, Amran Al-Aghbari, Mohammed Al-Asli, and Ahmad Khayyat. 2017. A Very Fast Trace-Driven Simulation Platform for Chip-Multiprocessors Architectural Explorations. *TPDS* (2017).
[29] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. 2006. A Performance Counter Architecture for Computing Accurate CPI Components. In *ASPLOS*.
[30] S. Eyerman, J.E. Smith, and L. Eeckhout. 2006. Characterizing the Branch Misprediction Penalty. In *ISPASS*.
[31] Jakob Gawlikowski, Cedrique Rovile Njieutcheu Tassi, Mohsin Ali, Jongseok Lee, Matthias Humt, Jianxiang Feng, Anna Kruspe, Rudolph Triebel, Peter Jung, Ribana Roscher, Muhammad Shahzad, Wen Yang, Richard Bamler, and Xiao Xiang Zhu. 2023. A Survey of Uncertainty in Deep Neural Networks. *Artificial Intelligence Review* (2023).
[32] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. 2010. Interval Simulation: Raising the Level of Abstraction in Architectural Simulation. In *HPCA*.
[33] Amirata Ghorbani and James Zou. 2019. Data Shapley: Equitable Valuation of Data for Machine Learning. In *ICML*.
[34] Nathan Gober, Gino Chacon, Lei Wang, Paul V Gratz, Daniel A Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The Championship Simulator: Architectural Simulation for Education and Competition. arXiv:2210.14324
[35] Alex Graves and Jürgen Schmidhuber. 2005. Framewise Phoneme Classification with Bidirectional LSTM and other Neural Network Architectures. *Neural Networks* (2005).
[36] Qi Guo, Tianshi Chen, Yunji Chen, and Franz Franchetti. 2015. Accelerating Architectural Simulation via Statistical Techniques: A Survey. *IEEE TCAD* (2015).
[37] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. In *MICRO*.
[38] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array Programming with NumPy. *Nature* (2020).
[39] Muhammad Hassan, Chang Hyun Park, and David Black-Schaffer. 2021. A Reusable Characterization of the Memory System Behavior of SPEC2017 and SPEC2006. *ACM TACO* (2021).
[40] Ranjan Hebbar SR and Aleksandar Milenković. 2019. SPEC CPU2017: Performance, Event, and Energy Characterization on the Core i7-8700K. In *ICPE*.
[41] Qijing Huang, Po-An Tsai, Joel S. Emer, and Angshuman Parashar. 2024. Mind the Gap: Attainable Data Movement and Operational Intensity Bounds for Tensor Algorithms. In *ISCA*.
[42] Christopher J Hughes, Vijay S Pai, Parthasarathy Ranganathan, and Sarita V Adve. 2002. RSIM: Simulating Shared-Memory Multiprocessors with ILP Processors. *IEEE Computer* (2002).
[43] Engin İpek, Sally A McKee, Rich Caruana, Bronis R de Supinski, and Martin Schulz. 2006. Efficiently Exploring Architectural Design Spaces via Predictive Modeling. *ACM SIGOPS* (2006).
[44] PJ Joseph, Kapil Vaswani, and Matthew J Thazhuthaveetil. 2006. A Predictive Performance Model for Superscalar Processors. In *MICRO*.
[45] PJ Joseph, Kapil Vaswani, and Matthew J Thazhuthaveetil. 2006. Construction and Use of Linear Regression Models for Processor Performance Analysis. In *HPCA*.
[46] Ajay Joshi, Aashish Phansalkar, Lieven Eeckhout, and Lizy Kurian John. 2006. Measuring Benchmark Similarity using Inherent Program Characteristics. *IEEE TC* (2006).
[47] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *ISCA*.
[48] Tejas S. Karkhanis and James E. Smith. 2004. A First-Order Superscalar Processor Model. In *ISCA*.
[49] Tejas S. Karkhanis and James E. Smith. 2007. Automated Design of Application Specific Superscalar Processors: An Analytical Approach. In *ISCA*.
[50] Aviral Kumar, Amir Yazdanbakhsh, Milad Hashemi, Kevin Swersky, and Sergey Levine. 2022. Data-Driven Offline Optimization for Architecting Hardware Accelerators. In *ICLR*.
[51] Benjamin C Lee and David M Brooks. 2006. Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction. In *ASPLOS*.
[52] Benjamin C Lee and David M Brooks. 2007. Illustrative Design Space Studies with Microarchitectural Regression Models. In *HPCA*.
[53] Jiangtian Li, Xiaosong Ma, Karan Singh, Martin Schulz, Bronis R de Supinski, and Sally A McKee. 2009. Machine Learning Based Online Performance Prediction for Runtime Parallelization and Task Scheduling. In *ISPASS*.
[54] Lingda Li, Thomas Flynn, and Adolfy Hoisie. 2023. Learning Independent Program and Architecture Representations for Generalizable Performance Modeling. arXiv:2310.16792
[55] Lingda Li, Santosh Pandey, Thomas Flynn, Hang Liu, Noel Wheeler, and Adolfy Hoisie. 2022. SimNet: Accurate and High-Performance Computer Architecture Simulation using Deep Learning. In *ACM SIGMETRICS/IFIP PERFORMANCE*.
[56] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *ICLR*.
[57] Jason Lowe-Power. 2024. *O3CPU*. https://www.gem5.org/documentation/general_docs/cpu_models/O3CPU
[58] Jason Lowe-Power. 2024. *Ruby Memory System*. https://www.gem5.org/documentation/general_docs/ruby/
[59] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan

Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152

[60] Richard TB Ma, Dah Ming Chiu, John CS Lui, Vishal Misra, and Dan Rubenstein. 2007. Internet Economics: The Use of Shapley Value for ISP Settlement. In *ACM CoNEXT*.

[61] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *ICML*.

[62] Microsoft. 2024. *Announcing the preview of new Azure VMs based on the Azure Cobalt 100 processor*. https://techcommunity.microsoft.com/blog/azurecompute/announcing-the-preview-of-new-azure-vms-based-on-the-azure-cobalt-100-processor/4146353

[63] Jason E Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. 2010. Graphite: A Distributed Parallel Simulator for Multicores. In *HPCA*.

[64] Christoph Molnar. 2020. *Interpretable machine learning*. Lulu. com.

[65] Stefano Moretti, Fioravante Patrone, and Stefano Bonassi. 2007. The Class of Microarray Games and the Relevance Index for Genes. *Top* (2007).

[66] Ramasuri Narayanam and Yadati Narahari. 2010. A Shapley Value-Based Approach to Discover Influential Nodes in Social Networks. *IEEE T-ASE* (2010).

[67] Quan M. Nguyen and Daniel Sanchez. 2023. Phloem: Automatic Acceleration of Irregular Applications with Fine-Grain Pipeline Parallelism. In *HPCA*.

[68] Sébastien Nussbaum and James E Smith. 2001. Modeling Superscalar Processors via Statistical Simulation. In *PACT*.

[69] Pablo Montesinos Ortego and Paul Sack. 2004. SESC: SuperESCalar simulator. In *ECRTS*.

[70] Santosh Pandey, Lingda Li, Thomas Flynn, Adolfy Hoisie, and Hang Liu. 2022. Scalable Deep Learning-Based Microarchitecture Simulation on GPUs. In *SC*.

[71] Santosh Pandey, Amir Yazdanbakhsh, and Hang Liu. 2024. TAO: Re-Thinking DL-based Microarchitecture Simulation. In *ACM SIGMETRICS/IFIP PERFORMANCE*.

[72] Avadh Patel, Furat Afram, and Kanad Ghose. 2011. MARSS: A Full System Simulator for Multicore x86 CPUs. In *DAC*.

[73] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, et al. 2020. The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC. *IEEE Micro* (2020).

[74] Alejandro Rico, Alejandro Duran, Felipe Cabarcas, Yoav Etsion, Alex Ramirez, and Mateo Valero. 2011. Trace-driven Simulation of Multithreaded Applications. In *ISPASS*.

[75] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *ISCA*.

[76] Kiran Seshadri, Berkin Akin, James Laudon, Ravi Narayanaswami, and Amir Yazdanbakhsh. 2022. An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks. In *IISWC*.

[77] André Seznec. 2011. A New Case for the TAGE Branch Predictor. In *MICRO*.

[78] Lloyd S Shapley. 1953. A Value for n-Person Games. *Contribution to the Theory of Games* (1953).

[79] Timothy Sherwood, Erez Perelman, and Brad Calder. 2001. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *PACT*.

[80] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *ASPLOS*.

[81] Timothy Sherwood, Suleyman Sair, and Brad Calder. 2003. Phase Tracking and Prediction. In *ISCA*.

[82] Kevin Skadron, Margaret Martonosi, David I August, Mark D Hill, David J Lilja, and Vijay S Pai. 2003. Challenges in Computer Architecture Evaluation. *IEEE Computer* (2003).

[83] Ondřej Sỳkora, Phitchaya Mangpo Phothilimthana, Charith Mendis, and Amir Yazdanbakhsh. 2022. GRANITE: A Graph Neural Network Model for Basic Block Throughput Estimation. In *IISWC*.

[84] Rafael Ubal, Julio Sahuquillo, Salvador Petit, and Pedro Lopez. 2007. Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. In *SBAC-PAD*.

[85] Amin Vahdat. 2024. *Introducing Google Axion Processors, our new Arm-based CPUs*. https://cloud.google.com/blog/products/compute/introducing-googles-new-arm-based-cpu

[86] Sam Van den Steen, Sander De Pestel, Moncef Mechri, Stijn Eyerman, Trevor Carlson, David Black-Schaffer, Erik Hagersten, and Lieven Eeckhout. 2015. Micro-Architecture Independent Analytical Processor Performance and Power Modeling. In *ISPASS*.

[87] Sam Van den Steen, Stijn Eyerman, Sander De Pestel, Moncef Mechri, Trevor E. Carlson, David Black-Schaffer, Erik Hagersten, and Lieven Eeckhout. 2016. Analytical Processor Performance and Power Modeling Using Micro-Architecture Independent Characteristics. *IEEE TC* (2016).

[88] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *NeurIPS*.

[89] Nan Wu and Yuan Xie. 2022. A Survey of Machine Learning for Computer Architecture and Systems. *Comput. Surveys* (2022).

[90] Roland E Wunderlich, Thomas F Wenisch, Babak Falsafi, and James C Hoe. 2003. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *ISCA*.

[91] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2019. Alleviating Irregularity in Graph Analytics Acceleration: a Hardware/Software Co-Design Approach. In *MICRO*.

[92] Ahmad Yasin. 2014. A Top-Down Method for Performance Analysis and Counters Architecture. In *ISPASS*.

[93] Amir Yazdanbakhsh, Christof Angermueller, Berkin Akin, Yanqi Zhou, Albin Jones, Milad Hashemi, Kevin Swersky, Satrajit Chatterjee, Ravi Narayanaswami, and James Laudon. 2021. Apollo: Transferable Architecture Exploration. arXiv:2102.01723

[94] Wu Ye, Narayanan Vijaykrishnan, Mahmut Kandemir, and Mary Jane Irwin. 2000. The Design and Use of SimplePower: A Cycle-Accurate Energy Estimation Tool. In *DAC*.

[95] Matt T Yourst. 2007. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *ISPASS*.

[96] Xinnian Zheng, Lizy K John, and Andreas Gerstlauer. 2016. Accurate Phase-Level Cross-Platform Power and Performance Estimation. In *DAC*.