Leveraging Heterogeneous Multicore for Single-Thread Performance

by

Shayne Matthew Wadle

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2026

Date of final oral examination: 12/01/2025

The dissertation is approved by the following members of the Final Oral Committee:

Karthikeyan Sankaralingam, Professor, Computer Science
Rahul Chatterjee, Associate Professor, Computer Science
Kirthevasan Kandasamy, Assistant Professor, Computer Science
Vikas Singh, Professor, Biostatistics and Medical Informatics

May you always seek improvement;

and those around you do the same.

ACKNOWLEDGMENTS

Thank you to my PI, Karu, for your patience and insights. My research grind will never stop.

Thank you to the professors I worked with: Jim, Deb, Matt, and Karu. I enjoyed curating a learning environment with each of you. The breadth of your pedagogies has informed my choices moving forward.

Thank you to my fellow students trudging through this choice (of graduate school) together. I am honored to call many of you friends.

Thank you to my friends for the steadfast conversations during turbulent times. Sometimes the best thing to do is vacation (twice).

Thank you to my family. The constant support network is invaluable and I appreciate the advice on how to play the game.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LISTINGS

ABSTRACT

As computer architecture enters an era where traditional performance scaling has stagnated, architects face the challenge of designing increasingly complex chips using feedback loops such as slow cycle-level simulations and non-representative benchmarks. These are increasingly out of step with real-world "in-the-field" workloads. This dissertation addresses the critical research gap in hardware introspection and field-testing by introducing three heterogeneous systems designed to leverage real-world data for single-thread performance gains. The first system, the Introspection Processing Unit (IPU), is a programmable RISC-V co-processor that enables at-speed hardware introspection and the first-of-its-kind in-the-field A/B testing for hardware designs. With less than 1% area overhead, the IPU allows developers to generate per-instruction cycle stacks and track fine-grained component utilization without specialized silicon, bridging the visibility gap between hardware and software. Second, NeuroScalar introduces a deep learning framework for cycle-level performance prediction of hypothetical hardware designs on production workloads. Utilizing a bidirectional LSTM model, NeuroScalar achieves over 95% accuracy in predicting instruction latencies and 90% accuracy in A/B testing rankings, all while maintaining a mere 1% overhead on commodity GPUs or utilizing the newly designed high-efficiency Neutrino accelerator. Finally, the State-Aware Heterogeneous Multicore (SAHM) architecture directly improves performance by specializing individual cores for specific application states (e.g., branch-heavy or cache-intensive phases). By migrating programs to cores that match their current runtime needs, SAHM achieves an average speedup of 15% in realistic settings and up to 20% in fully loaded systems. Together, these contributions establish a new paradigm for architectural design where in-the-field analytics and microarchitectural specialization provide a sustainable path for continued single-thread performance evolution.

1 INTRODUCTION

Architecture design is built upon methods that existed when performance improvement was "free": increasing transistor counts and power scaling meant new and more complex designs could be incorporated with little to no friction. Furthermore, when physical constraints limited single-core designs, architects adjusted by including multiple cores. Architects are designing the most complicated chips ever while leveraging little to no knowledge about what happens with the chips in the field. Benchmarks attempt to represent the real world workloads; however, computer science is a rapidly evolving field: machine learning trains on larger and larger datasets, more devices connect to the internet everyday, and new applications are being developed each minute.

The end of Moore's Law [152, 18, 117, 150, 89] and Dennard Scaling [45] has ushered in a new age for architects. The design constraints now include area, power, energy, and thermal limits in 3 dimensions from the combination of hundreds of interconnected components. Currently, the allocation of the budgets created by these constraints are informed by slow cycle-level simulations of benchmarks, RTL emulation, and the intuition of the architects built over the decades. It is not working anymore: chip over chip generational performance has stagnated [36].

To diagnose the reasons for this stagnation, data must be gathered on the individual components and their interactions, then analyzed to reveal bottlenecks and locations for optimizations. Architects have data from simulation and emulation; unfortunately, the data is limited by the fidelity of the simulation or by the time required to gather it (high accuracy runs on-the-order of weeks for one experiment). This accuracy-vs-time trade-off is costly because the hardware under test might never see the benchmark it was tested with run on it once it is fabricated. In this case, any hardware/software co-design that occurred has diminished returns. Thus, we must be able to gather data from real workloads, so called in the field, and make sense of it through analysis. We can leverage multicore

heterogeneous architectures to do just that. For this work, heterogeneity can be across forms of the cores, CPU/GPU or CPU/Accelerator, across ISAs, RISC-V/x86, or across microarchitecture, specialized branch prediction for example. Each combination gives rise to different opportunities due to the trade-offs between computational power and control granularity. Thus, we form three systems; each takes one combination and strives to tighten the feedback loop from having data to knowing what it tells us and potentially acting on it.

The first system, IPUs, introduce a RISC-V co-processor specifically for introspection, see Figure 2.1. It reads 32 signals from a host core and executes an analysis program on them. The output is writ through PCIe to memory either for later post-processing or off-loading. The IPU enables hereto unseen analysis and flexibility from a single performance monitoring tool. For software developers, it can be used to generate per-instruction cycle stacks without specialized hardware granting a new depth of analysis to debugging tools. For hardware producers, the IPU can track component utilization rates enabling a new plane of optimization through intra-core sharing. For hardware designers, the IPU enables in-the-field hardware A/B testing through an embedded FPGA. Now a design can be pushed to the masses and tested against real world applications in a secure test environment. Design choices no longer have to rely on slow cycle-accurate simulation; the statistics gathered from an IPU will make the choice clear. Furthermore, the mature eco-system around RISC-V means that new analysis programs are not limited to those presented - new analysis can be developed by anyone. Thus, the IPU improves single-threaded performance indirectly: it enables better hardware designs for the next commodity CPU as well as providing software developers new tools for performance analysis.

A second system, Neuroscalar, provides a new methodology in CPU hardware design: in-the-field testing of a proposed system; see Figure 3.1. A lightweight buffer is fed by the ROB collecting an instruction trace with a few key features. This buffer is then processed in epochs of 100,000 instructions through an LSTM machine learning model. The output is the delta between instruction retire cycles. We further post-process these outputs as an

indicator of A/B testing. Overall, the model is over 95% accurate for being within 1 cycle of the ground truth delta. This provides for 90% accuracy when predicting when one system is better than another per instruction. We develop a methodology to use a GPU to run the LSTM model that utilizes 1% overhead to prevent slow-downs for consumer machines when performing testing. Furthermore, we develop an accelerator, Neutrino, to process the LSTM; it is an 85× improvement in energy efficiency. Thus, Neuroscalar also takes an indirect approach to improving single-threaded performance: improve the next generation of CPUs.

The final system, SAHM, optimizes system throughput via specializing underlying microarchitecture to handle individual application states; see Figure 4.1. SAHM stands for state-aware heterogeneous multicore. First, we apriori establish a 16-state state space by binning four key performance metrics into high and low: branch misprediction ratio, L1 instruction cache miss rate, L1 data cache miss ratio, and L2 cache miss ratio. Then, while an application is executing, every 100ms we gather these metrics and determine which state the application is currently in. The application is migrated to the core whose specialization most closely matches the needs of that state. For example, an application in a phase where branches are hard to predict would have a high branch misprediction ratio. This would migrate the application to a core with more area and power devoted to branch prediction. The limit study found that if we model a 30% speed up when the application is placed on a beneficial core, one that matches the state, then on average, the program is sped up 27%. When we test a fully loaded system with migration cost and contention, the program is 20% sped up. Our migration cost sensitivity analysis showed a little decline to this end-to-end speed up. Therefore, leveraging program state in this unique way can provide ample head room in single threaded performance.

These systems enable new insights for architects and software developers. They provide a new basis on which to make design choices. In the end, each system is a step toward consistent single-thread performance gains.

Thesis contributions

This works contributions are summarized as follows:

- System design and implementation of a co-processor, the IPU, that is able to perform analysis duties without introducing slow down for the user, disturbing normal program behavior, or altering source code.
- Creation of a system for in the field A/B testing of hypothetical full system designs via deep learning on either a commodity GPU or dedicated accelerator, NeuroScalar with the Neutrino accelerator.
- Expansion on application phase behavior analysis and the system design to track and exploit it. This results in a limit study of the performance improvements available from a diverse microarchitecture multicore chip: SAHM.

Thesis Organization

The rest of this work is organized by system. Each system has a chapter that reflects and builds from the research papers that describe them, all currently in the process to be published [91, 143, 142]. The IPU is Chapter 2. NeuroScalar is Chapter 3. SAHM is Chapter 4. Extra material that does not add to the prose of some chapters resides in the appendices.

2 IPU: FLEXIBLE HARDWARE INTROSPECTION UNITS

Modern chip designs are increasingly complex, making it difficult for developers to glean meaningful insights about hardware behavior while real workloads are running. Hardware introspection aims to solve this by enabling the hardware itself to observe and report on its internal operation — especially **in the field**, where the chip is executing real-world workloads. Although prior work (performance counters, debug monitors etc. - see related work) has explored forms of introspection, success has been limited when confronted with three urgent challenges: 1. Lack of A/B Testing. In modern software development, new features are tested side by side to collect performance data on live workloads [47, 48]. No equivalent mechanism exists for hardware. Architects cannot "deploy" or "test" a new feature in the field, observe real performance impacts, then decide whether to commit that feature to the next silicon revision. 2. Obfuscated Hardware. Software developers do not see essential microarchitectural details, since current tools (e.g., performance counters) provide only coarse insights and hide the nuances of actual on-chip events [57]. 3. Obfuscated Software. Hardware designers rarely know how their designs behave under real-world software usage. Information from in-field use does not percolate back to chip designers. These issues have become especially acute because both hardware and software are expanding in complexity. Microarchitects must handle larger design spaces, while software developers have difficulty optimizing code for intricate, opaque hardware. The recently proposed Time-Proportional Event Analysis (TEA) module, which tracks per-instruction cycle stacks (PICS) to aid software optimization using a specialized hardware design [56, 57], argues performance counters are insufficient.

In this chapter, we revisit hardware introspection with a new approach designed to bridge these gaps comprehensively. We present a framework that not only offers fine-grained observability to software but also allows hardware designers to gather actionable insights **in the field**, ultimately informing the chip design cycle.

A central question for hardware introspection is: *How can we design future chips so that developers can capture, analyze, and derive insights from cycle-level data in real deployments?* Addressing this involves four key challenges: i) Capturing fine-grained hardware signals **in the field** without incurring prohibitive area or power costs; ii) Enabling programmability so that what introspection is done can be changed at runtime, rather than being fixed in silicon; iii) Deciding what to introspect on; iv) Controlling the data volume so continuous introspection does not overwhelm the system.

Our solution, the Introspection Processing Unit (IPU), outlined in Figure 2.1, tackles these challenges with a philosophy that balances efficiency and flexibility. Each IPU is built around a tiny RISC-V core placed in close physical proximity to the hardware block (a "Hardware-module Introspection Target" or HIT) whose signals it introspects. IPUs are integrated into HITs' physical hierarchy (similar to performance monitors) removing the need for costly wiring and, at design time, allowing engineers to profligately choose which signals are worth exposing - up to 32 signals per IPU. Chip designers, at design time, choose what the HIT is and attach an IPU to it - there can be multiple HITs and corresponding IPUs on a single chip. We expect HITs to be small - around 3 to 4 mm^2. Meanwhile, the actual analysis that runs on these signals is fully programmable, can be conceived of and implemented post-manufacturing any time during the chip's lifetime. From a system-level perspective, the IPU appears as a small PCIe device, providing a logical FIFO to send introspection outputs back to the host. This ensures that data volume remains manageable and that the host software stack can enforce bandwidth limits if needed. Chip designers can use software signing mechanisms to control 3rd party introspection for HITs. In this way, the IPU design forms a practical solution for hardware introspection delivering (i) efficient at-speed data capture, (ii) flexible analysis that allows rich hardware introspection, and (iii) low-overhead data handling.

This chapter's contributions are the definitions, system design, and implementation of the IPU, including an RTL implementation, that enables such a type of introspective

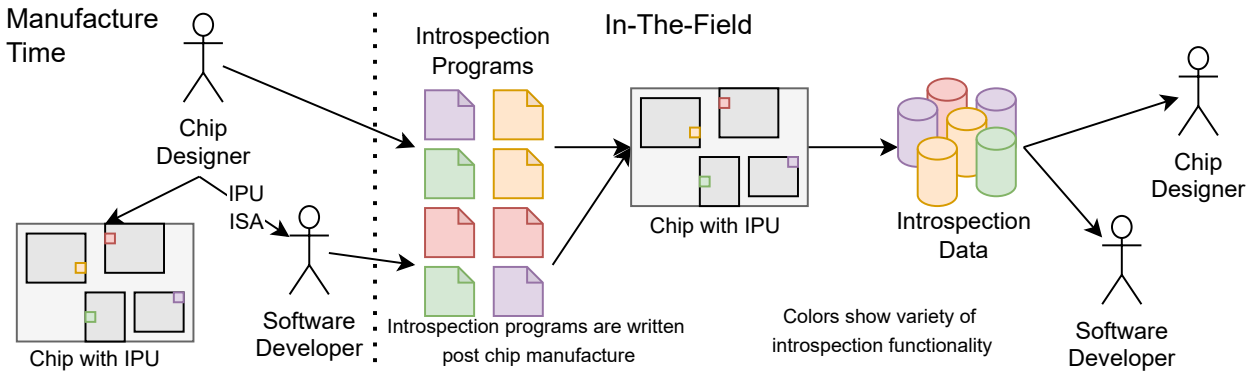


Figure 2.1: IPU overview

hardware. To evaluate the IPU, we show 4 case studies using the IPU along the 3 key problems.

• A/B testing. To address the lack of A/B testing for hardware, we demonstrate how the IPU can be used to evaluate a state-of-the-art entangled prefetcher [121]. Specifically, the IPU allows chip designers to emulate and evaluate a new prefetching algorithm on real workloads running in production systems—without modifying the hardware pipeline or incurring runtime slowdowns. This form of in-field evaluation enables comparative testing of candidate microarchitectural features. While fundamentally different from software A/B testing due to fabrication cost and rollback limitations, the IPU provides a practical mechanism for lightweight, post-silicon experimentation with hardware behaviors via introspection programs.

• Obfuscated hardware. Recent work has shown that per instruction cycle stacks (PICS) provide extraordinary insights into hardware behavior and opportunities for software optimization, way beyond traditional performance [56, 57]. However the implementations require specialized hardware to implement PICS. We show that PICS can readily be implemented as a program that simply runs on our IPU, with inputs being easy to access signals of the processor's microarchitecture. *In essence, we show that an IPU achieves the functionality of the specialized PICS hardware implementation while being programmable.*

• Obfuscated software. Conversely, hardware designers cannot observe enough about

the software. In the case of GPUs this becomes quite acute. In spite of their extensive performance counters and profiling libraries, GPU designers have little in the field data about the software being run on their chips. We show that by collecting fine-grain cycle-level hardware utilization rates of key streaming multiprocessor (SM) components through an IPU, hardware developers can observe opportunities for overlapped execution.

• Value histograms. In addition to these exemplary case studies, activation values from fully connected layers within machine learning can be gathered by an IPU via a histogram. This transparently and without slowdown enables software developers to study whether their models are suitable for other data formats.

In addition, the IPU is capable of providing the functionality of recent work like on-chip power estimation [151], historical works on data logging[24, 23, 15], monitoring engines[85, 50, 38, 28, 38], and specialized support for debugging and watchpoints [58] to enumerate a few.

Our results show the IPU executes each case study correctly, capturing the functionality of a specialized design. Second, it demonstrates IPUs achieve introspection capability infeasible with existing techniques. Third, it is efficient - the area overhead is $< 1\%$ and power overhead is $< 25mW$, corresponding to $< 1\%$ in the worst-case of the IPU being always active (with something as simple as 10% sampling, the overhead reduces by that factor). Our results span 15 SPECCPU benchmarks, 135 Champsim traces, 21 Gemm shapes, and 4,000 activations covering both CPU and GPU uses. The simulation testbed, IPU RTL, and introspection code will be released for others to build upon as an artifact.

It is important to clarify that the IPU architecture is not a collection of custom, per-use-case monitors, but a shared, programmable unit that runs different introspection binaries over the same general-purpose hardware. Across all four case studies, the same IPU variant can be used without modification. This programmability allows chip designers to explore hardware behavior in the field with significantly more flexibility than fixed-function PMUs. For example, our prefetcher emulation case study (Section 2.5) implements decision logic to

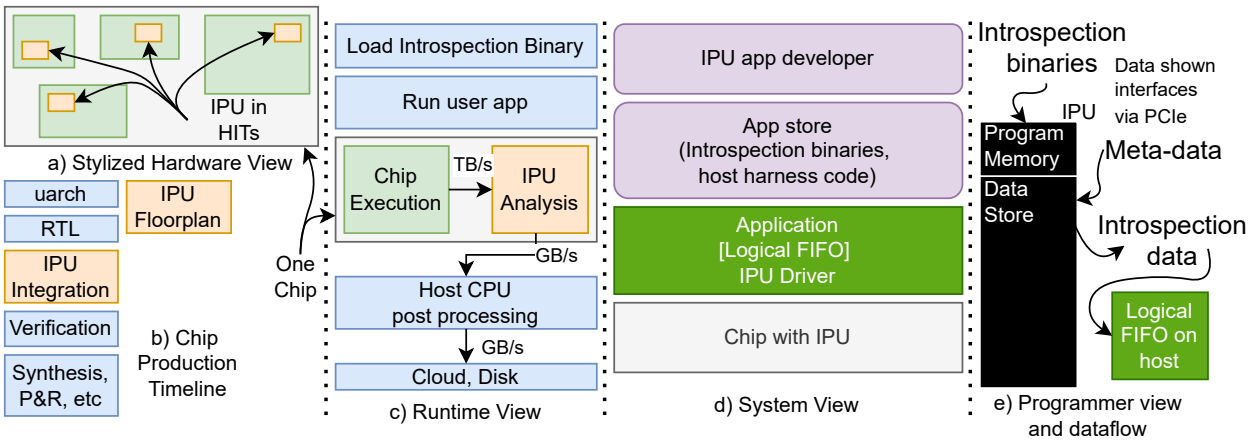


Figure 2.2: Cloud is purple, chip/HIT is green, blue is existing processes, orange is our contribution/new circuitry

test new algorithms on live workloads—functionality that is fundamentally infeasible with PMUs. While end users may not directly interact with IPUs, the design targets chip vendors, system integrators, and firmware developers who routinely require deep observability in controlled environments. The IPU provides them with a secure, low-overhead mechanism to introspect post-silicon behavior with new kinds of insights, ultimately informing the evolution of hardware design.

This chapter is organized as follows. An overview of the IPU and surrounding system is provided in Section 2.1, software and hardware architecture are detailed in Section 2.2 and Section 2.3 respectively. Evaluation methodology is in Section 2.4, and our results with the four case studies are in Section 2.5. Section 2.6 covers related work.

2.1 System Overview

We provide a full overview of the software and hardware components of an IPU as shown in Figure 2.2 before providing the details in the following sections. To provide some context, we outline two types of introspection needed. First, for simplicity one that can be done in other ways as well. Consider the outputs of a Tensorcore every cycle. One introspection program is to build histograms (say 1024 equal-sized bins) of distributions of values as

they are produced. The data producer would be Tensorcore, the signals are its output values, and the IPU would run instructions every cycle to determine the bins those values belong in. The histogram could be sent to the host's memory every 1 million cycles. Second, considering A/B testing - an introspection program could be instruction prefetch logic, with the data producer being a CPU's front-end block, with one signal - the `current PC` being fetched. The introspection output is the effective miss-rate and accuracy.

The programs run on the IPU that perform introspection we call "introspection binaries". The term "user program" refers to an application that runs on the chip - like web-browser, Photoshop, DL inference, etc. Introspection programs analyze user programs.

Hardware. The Introspection Processing Unit (IPU) is a modular design built around a simple in-order RISC-V core. Inclusion of common analytics functions beneficial to maintain speed creates the $\text{IPU}_{\texttt{lite}}$. The $\text{IPU}_{\texttt{pro}}$ has soft logic for finite state machine traversal introspection programs as we detail in Section 2.3. An IPU is responsible for monitoring an individual hardware component of the underlying chip: the Hardware-module Introspection Target (HIT). Our design associates one IPU per HIT (Figure 2.2(a)). Examples of possible HITs include individual sub-modules within a core, L2 controller, GPU SM sub-modules, a CPU core etc. The IPU will be flattened into the hierarchy of the HIT for placement and routing (P&R) purposes (Figure 2.2(b)). The IPUs are also connected to the underlying chip's on chip network (OCN), which is used to transfer produced introspection information out of the IPU. Introspection programs output small packets at long intervals (e.g. 1 million cycles) so the generated traffic does not impact the user program's performance.

Runtime View. The runtime view of a program is cascaded in Figure 2.2(c) with time flowing downward. The IPU driver exposes an API to configure and load the introspection code. The user application can specify what regions to analyze (e.g. specific tensor address regions). Once configuration is complete, the user application begins execution. On each data input to the IPU, it determines if the data is within the region to analyze. If so, the

introspection program executes on said data. While analysis is on-going, any further incoming data is dropped. Therefore, introspection programmers must be cognizant of the data rate specifications from the hardware designer. Once analysis execution completes, the IPU awaits the next data. In most introspection programs, outputs occur at intervals through a logical FIFO (Figure 2.2(e)). Once the user application ends, the IPU output can be post processed and, finally, exported onto disk or into the cloud.

System Architecture. The system architecture (Figure 2.2(d)) includes an API to expose IPUs and the IPUs themselves. At runtime, the IPU appears as a PCIe device programmable via a simple API, allowing hardware and software developers to download introspection programs. Each IPU is physically tied to its hardware block, and secure code execution is enforced via certificate-based code-signing using public/private key infrastructure, similar to Android/iOS app stores [7, 1]. This enables a secure, app-store-like model where trusted introspection programs—authored by chip designers, researchers, or developers—are deployed through configuration API calls.

Privacy. Figure 2.2(e) presents an abstraction to understand the IPU's privacy and security implications. An IPU, by design, cannot inject signals into the hardware HIT; it only accepts binaries and meta-data as inputs and emits introspection data. Integrity of binaries is addressed via code-signing. Richer program analysis techniques inherently risk leaking microarchitecture details—PICS [57], for instance, reveals bottlenecks that reflect design decisions—as earlier shown by Desikan et al. [41] using performance counters. The central privacy question is what unintended inferences introspection might allow; for example, end-users learning undisclosed HIT details. To mitigate this, we propose three policy modes: **closed**, where no third-party introspection is allowed; **restrictive**, requiring source-code review before code-signing; and **permissive**, permitting introspection from credentialed developers. These policies are enforced during code signing, with designers leveraging HIT semantics for decision-making. Techniques such as program verification [92], trace wringing [33], local differential privacy [44, 26, 29, 144, 53], and

secure multiparty computation [82] offer potential for future formal guarantees. A subtler issue involves using HIT_x introspection to infer data about unrelated HITs or chip-level behavior—akin to side-channel attacks or industrial espionage, and while possible, the IPU's risk is comparable to existing interfaces like performance counters. Our system-level API ensures strong user privacy by streaming only introspection data, without host state access (e.g., IP or MAC addresses). Lastly, trust between chip vendors and users remains a broader concern not unique to IPUs; telemetry and diagnostics (e.g., performance counters, JTAG) are often disabled in untrusted settings. CPUs and GPUs support confidential modes that disable monitoring, as with NVIDIA Hopper's secure execution mode [111], and Intel's diagnostic firmware requires signing. Similarly, IPUs can support SKU variants or boot-time configurations that disable hardware or introspection paths entirely. In untrusted deployments, IPUs would be disabled, aligning with industry norms and not detracting from their value in trusted or OEM-controlled contexts.

2.2 Software Architecture

This section describes the software implementation on top of IPUs with a deep dive on introspection programs. These programs, called "introspection binaries", include code that will be run on an IPU. The collection of IPUs on chip appear as a single PCIe device with distinct memory mapped areas for each IPU. The IPUs also expose a host API for configuring what introspection binary to run and the trigger logic. "User program" refers to an application that runs on the chip (CPU or GPU) - like web-browser, DL inference, etc.

Programmer's Model

An IPU is a programmable RISC-V core with a unique interface: 32 named inputs per execution. API calls configure which regions of the user program are analyzed. A small on-IPU memory is available, and the introspection program runs once per input set, repeating

if new data arrives or idling otherwise. Post-processing can be performed at program end before offloading results.

Data & Transfer Semantics. A HIT's signal interface to these 32 named values would be exposed to users through a formal documentation like an ABI Spec specifying semantics of signals and data arrival rate. Introspection developers must ensure that the HIT signal production rate matches data processing speed in the IPUs, or use sampling, or be cognizant that some data will be dropped if there is a mismatch. Introspection binaries are able to transmit data via host-memory mapped into the IPU's address space. The IPU can then issue simple memory instructions to the unit's memory hierarchy that are then transparently routed to a region in host memory. Using this, we can create a store for introspection results.

One might ask why we don't add input buffers to avoid dropping data when the IPU is busy. While small buffers delay overflow, they cannot prevent it if the introspection rate is slower than the data arrival rate—by Little's Law, loss is inevitable without stalling the HIT, which our architecture prohibits. Thus, the IPU drops inputs when active, and introspection programs must be designed with this in mind, using sampling, aggregation, or exploiting event sparsity. Though the IPU runs at 2 GHz in 7 nm, HIT modules may run faster (e.g., 3–4 GHz). We do not require clock synchronization; instead, we support three modes: (1) Accept reduced fidelity via sub-sampling (e.g., 1-in-2 cycles); (2) Use a fast buffer for asynchronous, windowed processing; or (3) Restrict introspection to low-frequency phases or optimize the IPU for speed.

Software API and Execution Management

For configuration, the IPU exposes a minimal host-side API which are facilitated via memory-mapped I/O to the IPU. To configure a binary for execution, `IPU_CONFIG_IMAGE(image)` specifies that an IPU should load a given introspection binary `image`. To configure when analysis begins, `IPU_CONFIG_START(addr)` sets the IPU to begin processing new data when the program to be analyzed reaches `addr`. `IPU_CONFIG_STOP(addr)` similarly sets when

Signal	#Bits	Reg	Semantics	Rate
itlb-miss	1	x0	Instruction TLB miss flag	1
icache-miss	1	x1	L1 Icache miss flag	1
recycle	1	x9	Recycle ROB unique IDs flag	1
fetch-pc-head	64	x11	Next PC to be fetched	1
...				

Table 2.1: Partial ABI Spec. Rate measured in cycles between data points.

to stop processing new data. To enable fine-grained analysis we expose `IPU_PAUSE()` and `IPU_RESUME()` to pause and resume introspection execution. Finally, the command `IPU_FINALIZE()` instructs the IPU to execute any clean-up code necessary for the introspection program. The API resembles CUPTI [110] or Linux's `perf_event` [49], allowing either a harness process to transparently configure and trigger IPU execution, or user binaries to use `PAUSE/RESUME` for fine-grained control.

IPU Program Lifetime

This section is an end-to-end example. The HIT is a CPU core and analytics code development is under a closed policy - the CPU core designer will also write the analytics code. The analysis is PICS generation; the Obfuscated Hardware case study (Section 2.5) - we encourage the reader to skim that first.

Pre-Fabrication. As part of the design of the CPU core, up to 32 signals are chosen to connect to the IPU - based on important signals in the microarch pipeline. No information needs to be released to the public because of the closed policy; yet, an internal ABI Spec would be created to facilitate analytics code development. A partial ABI Spec is seen in Table 2.1. Prior to verification, the IPU is flattened into the core layout and the HIT-IPU connections are made as outlined in Figure 2.2. A subtle issue is that the HIT designer needs to determine what the important signals are - by providing up to 32 we give them freedom to be profligate to allow rich analytics post-manufacture.

Development. With the ABI Spec defined, the analytics code can be developed, which is the PICS generation in our example here. To this end, the CPU designer references the

ABI Spec for each of the 17 signals necessary and identifies which input registers they are connected to. A portion of the code handling the instruction TLB miss event is shown below:

```
_main: regtimer 50000, psv_loop
psv_loop: beq x0, 1, itlbm_m
beq x1, 1, icache_miss; x1 is a HW inp sig
...
itlb_m: hash r1, x12; x12 is an HW inp sig
ld r2, r1, 0
addi r2, r2, 0x40
st r2, r1, 0
ret
...
```

In the development a 400,000 cycles sample rate is chosen to limit the output load; this does create approximation error which the CPU designer tests and finds it within acceptable limits - the overall PC ordering by most cycles used is correct. The CPU designer releases the analytics binary, output location, approximation error, and if a region of interest can be chosen onto the app-store.

Analysis. Now, a SW developer has encountered unexpected slowdowns in their application and wants to profile it. They can download the PICS generation binary on the host. The listing also indicates that output will be put in a file in disk and that the user can optionally specify a region of instructions to analyze. They include a few API calls at the top of their program source:

```
IPU_CONFIG_IMAGE("PICS-generation")
IPU_CONFIG_START(ROI_BEGIN)
IPU_CONFIG_STOP(ROI_END)
```

PC	Event Combination	Number of Cycles
0x7912d0	DTLB miss, DCache Miss	50000000
0x80dda0	Branch Mispredict	200000
...		

Table 2.2: A few rows of the output file from PICS generation analysis.

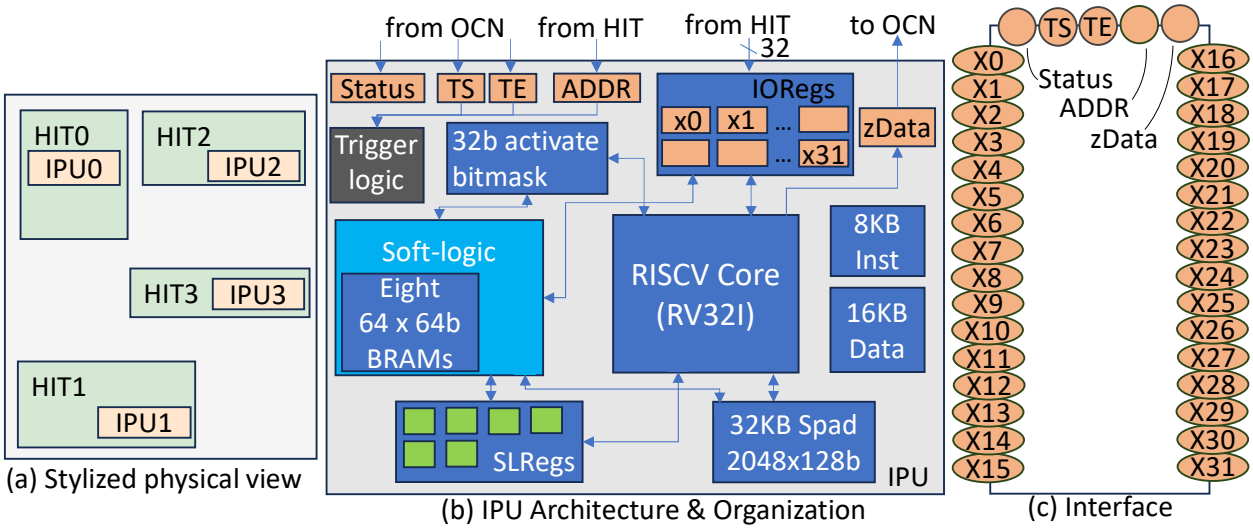


Figure 2.3: IPU hardware architecture.

```
<program code>
```

`ROI_BEGIN` and `ROI_END` are the beginning and end of the region of interest where the developer believes the slowdown to be. Essentially it sets the respective Program Counters as the address to monitor for activating the IPU.

Post-Analysis. The output file has a list formatted as in Table 2.2. Produced by host code, monitoring the IPU's introspection program. The developer uses the results for application performance tuning.

2.3 Hardware Architecture

A chip can have multiple IPUs. Each IPU observes hardware signals from its corresponding HIT and runs introspection binaries on those signals as stylized in Figure 2.3(a). The IPUs use the chip's OCN, which enables them to transmit introspection outputs (which are

infrequent) to and configuration from the host. The collection of IPUs is visible as a PCIe device and uses PCIe to interface with the host; each IPU is distinguished through distinct memory mapped regions. If the chip lacks an OCN, some form of network which connects the IPUs to the PCIe interface must be added as well.

Configuration of an IPU occurs through an API call: `IPU_CONFIG_IMAGE(image)`. This bundles the introspection binary (and trigger logic metadata - details in Section 2.3) and sends it to the appropriate IPU based on a given hardware device ID.

Interface overheads. HIT-IPU connections are short because the IPU is flattened into the HIT during P&R. Therefore, wiring overheads are negligible. Depending on the signals in a HIT and what the HIT itself is - there could be timing issues that can be addressed with standard buffering techniques used for performance counters. Consider a square HIT that is $2mm^2$. At a simple level, signals might need to traverse $2.8mm$ (half the perimeter of the HIT) to reach the IPU logic. To avoid timing issues for a high frequency design, one flip-flop might be necessary. Since this "far away" signal is buffered by one cycle, it implies all signals of this $\mathtt{HIT} \rightarrow \mathtt{IPU}$ must be buffered. HIT designers can use P&R feedback to judiciously select signals to avoid/minimize this.

There is no type of cross-chip wires. IPU configuration and output transfers occur over the chip's OCN and PCIe; both transmit small packets at long intervals meaning the traffic is negligible. We acknowledge, that even this meagre PCIe introspection traffic can introduce QoS and interference leading to non-linear slowdowns. Optimizations to this traffic management are future work.

Hardware organization. The IPU architecture comprises of four baseline components: a programmable core, a scratchpad SRAM, 32 input registers (IORegs), and 3 trigger registers: Trigger Start (TS), Trigger End (TE), and ADDR. The Trigger logic looks at the ADDR signal from the HIT along with TS and TE (programmed using API calls) to control when the IPU becomes active. This is shown in Figure 2.3(b) while the interface

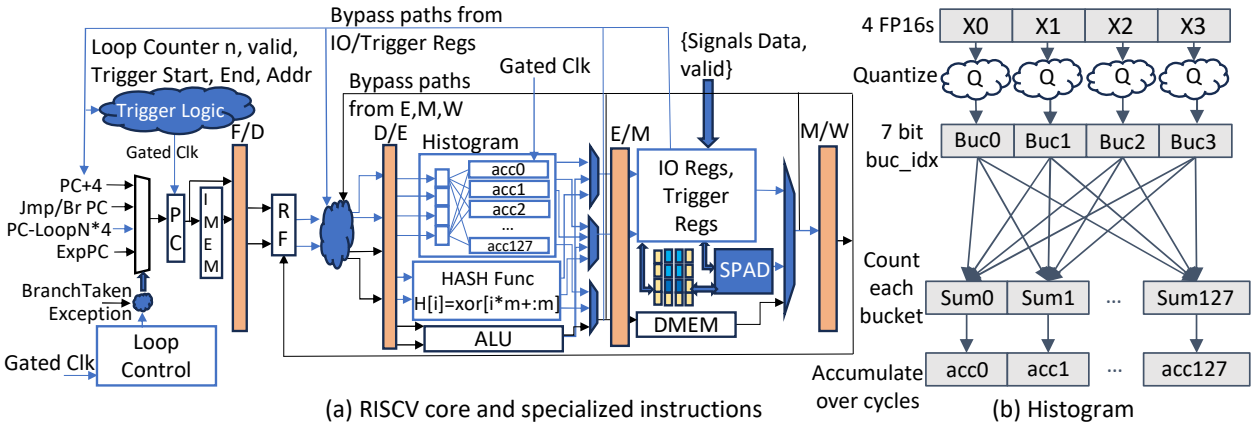


Figure 2.4: IPU Microarchitecture showing datapath and control-path changes

is laid out in Figure 2.3(c). To concretize the architecture, we select some sizes for these components: the core is 32-bit RISCV core (RV32I instruction set) using a data-SRAM and an instruction-SRAM; the scratchpad is 32KB.

Execution Model. An IPU has a 4-bit STATUS register, putting it in 6 states: PAUSED (P), ACTIVE-PAUSED (AP), ACTIVE-RUNNING (AR), FINALIZE (F), ERROR (E), and UNDEFINED (U). Its program structure includes 3 predefined program regions: `init`, `_main`, and `end`. Borrowing from the simplicity of micro-controllers, `init` is hardcoded to instruction memory address 0x0. The entire instruction memory is 8KB which amounts to 2048 instructions long. On power-on, PC is set to 0 and starts executing the code in `init`. By convention, `finish` is hard-coded to be 16 instructions from the bottom of the instruction memory at 0x7F0. When the IPU is set to the finalize state, it executes code in the `finish` function and transitions to the PAUSED state.

The execution model of code on an IPU is data-driven, i.e. when new inputs arrive the `_main` function is called if the IPU is in the ACTIVE-PAUSED state. If it is running code triggered by previous input, it will be in the ACTIVE-RUNNING state - data received when in this state is dropped. Whenever we show a datapath of X bits for the IO registers, there is an implied additional valid bit associated. This bit is used by to determine whether new input has arrived.

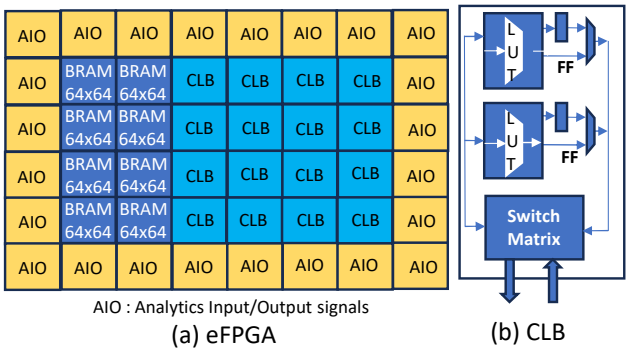


Figure 2.5: IPU_{pro} soft-logic design

Microarchitecture. We design two IPU variants for different regions of the analysis-vs-data-rate design space. **IPU**$_{\text{lite}}$ is a compact, cacheless RISC-V core with built-in primitives like histograms, loop counters, and hash functions for efficient, low-complexity introspection. **IPU**$_{\text{pro}}$ augments the RISC-V core with soft-logic—a lightweight embedded FPGA with configurable logic blocks and small BRAMs—enabling complex, high-throughput analysis tailored at runtime. It interfaces via memory-mapped registers and supports introspection programs that bundle RTL logic alongside control code. Each IPU is embedded with its associated HIT and communicates via the OCN, avoiding long wires and limiting system traffic due to low data rates. Multiple IPUs scale well: 5 IPU_{pro} and 10 IPU_{lite} consume just 0.65% of a 200 mm^2 die, and even full coverage across GPU SMs stays under 1% chip area overhead.

Limitations

While the proposed IPU framework demonstrates significant potential for addressing key challenges in hardware observability, we acknowledge certain limitations inherent in our study and design.

Workload Coverage. The case studies utilize specific benchmark suites and traces (SPEC-CPU, CVP2 traces, GEMM kernels). While diverse, these workloads may not fully represent the breadth and complexity of all potential in the field applications where IPUs could be deployed.

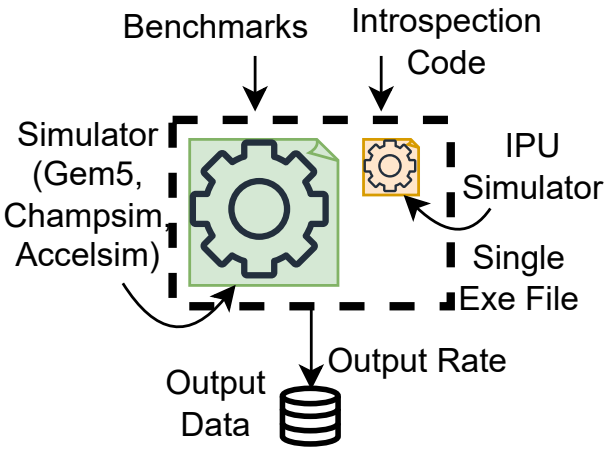


Figure 2.6: IPU System Testbed Flow for emulation

HIT Signal Selection. The utility of an IPU is fundamentally dependent on the foresight of chip designers to expose relevant and sufficient signals from the Hardware-module Introspection Target (HIT) during the initial design phase. The process of selecting these signals remain a practical consideration, which is substantially ameliorated by encouraging designers to be profilate in the number of signals to expose for HIT.

Security Guarantees. We propose policy-based access control and code-signing mechanisms to manage security. However, any introspection mechanism inherently increases the potential attack surface for information leakage as noted earlier. The development and formal verification of robust security and privacy protocols represent an important direction for future research.

2.4 Evaluation Methodology

To empirically evaluate the IPU, we conduct four case studies that we briefly describe in the introduction. Each demonstrates the IPU's ability to resolve 1 of the 3 problems: **in the field A/B testing**, **obfuscated hardware**, and **obfuscated software**. Table 2.3 describes the emulation and simulation testbeds we built.

Emulation and Simulation testbed. Our four case studies span prefetch engine (Champ-

Case Study	A/B Testing	Obfuscated HW	Obfuscated SW	Value Histograms
Benchmarks	135 Traces	15 Spec17	21 Gemms	20 GB/4k Activations
Simulator Config Matching	Champsim Entangled Prefetcher	Gem5 SE TEA	AccelSim QV100 Model	PyTorch arch values only
#lines of IPU code	300 Verilog	75	2	2
#bits into IPU	64	215	4	64
#signals from HIT	2	17	3	1
Rate (GB/s)	26.88	8	0.5	8
Output size	3B	6B	4B	384B
Output timing (cycles)	Program	400k	256	32k
Rate per HIT (1/s)	approx 0	15KB	15.6MB	8MB

Table 2.3: Methodology configurations

sim [54] simulator), core-microarchitecture (GEM5 [13] cycle level simulation), GPU cycle-level simulation (AccelSim [71]), and GPU values (at architecture level and hence simulated at PyTorch level). We built an IPU emulator for code development and to determine correctness of our introspection. For performance (time), we developed a co-simulation environment that adds an IPU simulator into Champsim, Gem5, and Accelsim (left Figure in Table 2.3). For area and power, we implemented RTL (and then synthesized) which was verified with an emulator for correctness. Table 2.3 also shows the number of lines of code for the `_main` function of each introspection program. Since the values histogram is essentially processing values which are part of architecture state, its simulation is done entirely in PyTorch. Overall, we have more than 171 applications simulated.

RTL Implementation. We implemented IPU_{pro} and IPU_{lite} in Verilog. Our implementation was verified for many input values against the introspection reference implementation. We use the AsAP7 7nm educational PDK [27]. For SRAMs we use CACTI scaled from 32nm to 7nm per [137]. We implemented our soft-logic using the FABulous design flow [73] to estimate area and power, and their synthesis flow for utilization. To determine soft-logic power, we used data from the reference introspection execution to create input traces. We used ASIC process flow of synthesis (DC Compiler/Primetime), APR(Innovus), and VCD based power estimation obtained from Netlist simulation of all case studies. The max clock frequency for the soft-logic and IPU_{pro} is 1.3 GHz and 2 GHz for the RISC-V core i.e.

IPU_{lite}. The HIT signals we need are described in the case studies and we show that the signals are readily available for any reasonable implementation of a GPU or core. The first two case studies are on a CPU and the third case study is on a GPU. To do comparison, our references are: a CPU Zen2 4-Core Complex [135] which needs 31 mm^2 area and consumes 4 watts of power [31] and a GPU SM that uses 3.475 mm^2 [86] of area and consumes 1 watt of power [161, 69].

2.5 Case studies

We now describe 4 case studies resolving the 3 key problems while spanning CPU and GPU and covering different HITs and signal types. **We emphasize that an IPU accurately captures each case study: enabling A/B testing that is impossible currently, constructing disruptive PICS stacks without dedicated single-function hardware, and gathering in the field characteristics of GPU utilization**. Figure 2.7 shows which hardware signals are connected to the IPU. Figure 2.8 depicts a utility result for each case study. For each case study, we cover the key problem, how our use is exemplary of the problem, what utility the introspection holds, and where applicable present a quantitative comparison, the design of the introspection code, the area and power, and whether any data is dropped and its effects on accuracy if so. An overview of the interface width, code length, and output characteristics is shown in Table 2.3. For context, PCIe bandwidth for A100 is 32 GB/sec [108]; our introspection output never exceeds 2.0 GB/sec.

This reflects worst-case raw output bandwidth under sustained introspection. In practice, IPU outputs are small and sparse—our case studies show useful analytics from just a few bytes every hundreds of thousands of cycles. Results are meant for in-system use: the GPU case study emits short utilization histograms, not full traces; the PICS study sends only aggregated delay signatures. Further aggregation or selective export would be done locally, avoiding raw trace streaming to the cloud.

PCs	Fetch, Dispatch, LSQ, Head, Flush, Commit
Control Sigs	bpred mis-speculate, exception, recycle, ROB-empty, SQ-full, memory violation, iTLB miss, iCache miss, dTLB miss, dCache miss, LLC miss

Table 2.4: PCs and Control Signals for Obfuscated Hardware case study

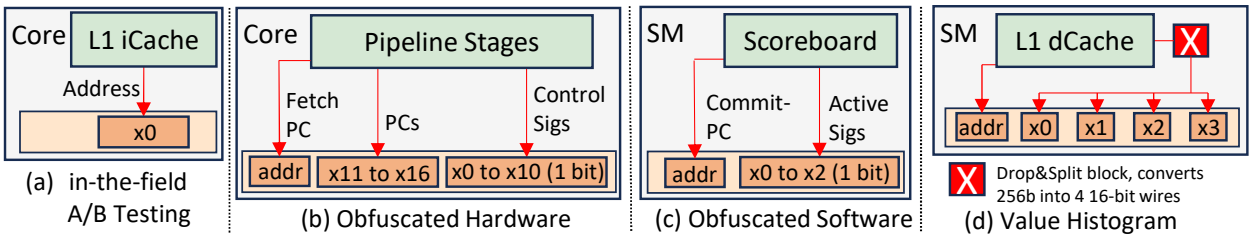
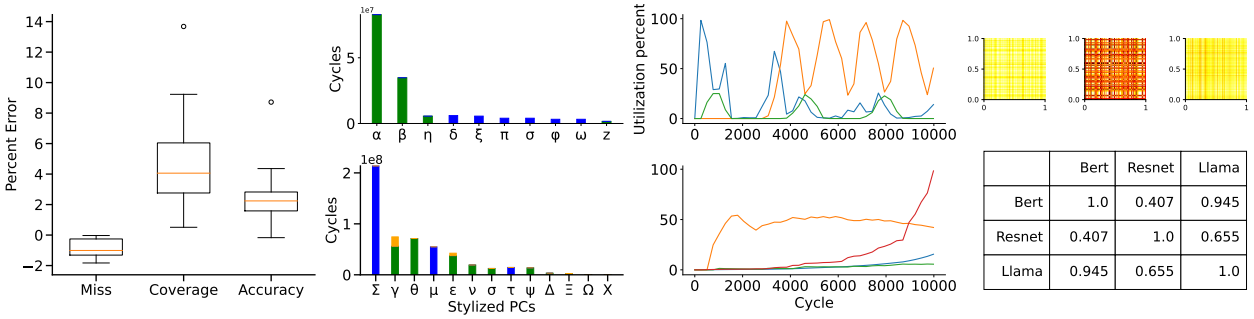


Figure 2.7: Case studies interfaces. (b)PCs and Control Sigs listed in Table 2.4. (c)Active Sigs are for the tensor core, SIMT, and memory subsystem.



(a) In the field A/B Testing (b) Obfuscated Hardware (c) Obfuscated Software (d) Value Histogram

In the interest of space, the labels for (b) are symbols. In reality they are PC values in the application binary.

Figure 2.8: Utility results of the four case studies. (a) Relative error for each metric in prefetch emulation in-silicon across 135 workload traces. (b) TOP-10 PICS for NAB and Libquantum benchmarks, showing instruction contributions to exposed cycles. ■ dcache miss, ■ Drain- SQ full, ■ Icache miss and Dcache miss (c) Cycle-level GPU utilization. ■ SIMT, ■ Tensor Core, ■ higher level memory, ■ SIMT sorted by utilization. (d) Distribution of weights for 3 benchmarks. Yellow is more similar; brown is less similar.

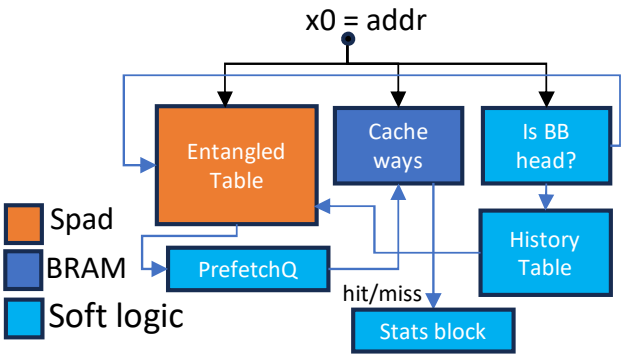


Figure 2.9: Organization of the soft-logic block for prefetcher

In the field A/B Testing

A chip designer cannot directly compare the effectiveness of hardware designs through deployment tests; this is unlike some software that deploys multiple designs and analyzes each for their effectiveness in real-world situations. It would be a hazard to have applications run on designs under test, so a tool to emulate the design's effectiveness while execution continues using the hardware made at manufacture time is ideal: this is complex analysis. To perform an effective test, all inputs should be captured meaning a high data arrival rate. Thus, the IPU_{pro} is utilized.

This case study tests the recently proposed entangled prefetcher [121] via emulating a form of the prefetcher in the IPU_{pro}'s soft logic. In one core complex, one such IPU_{pro} can be integrated to introspect the front-end of a core. The analysis produces the coverage, accuracy, and miss-rate for in the field applications. We achieve results within 1.8%, average of 1%, for miss rate of tests using simulations. **This demonstrates that the IPU can test hardware designs at speed in the field before fabrication enabling unprecedented analysis of new designs.**

HIT & Interface. The HIT is the CPU front end block, and we need essentially one data signal - fetch PC being issued by the processor core (depending on the decoupled front-end design, the signal could be different; a virtual or physical address depending on cache design). The analysis is performed over the entire program so no signal is connected to

ADDR nor are TS and TE configured.

Introspection Code. Implementing the prefetcher using RISCV code is too slow resulting in many dropped data (a state-machine traversal is heavily branchy code). Instead, we implement it on the soft logic and are able to run at one address per cycle with nearly full eFPGA utilization. The RTL design of the prefetch emulator is shown in Figure 2.9. Since we cannot inject anything into the HIT, our entangling design assumes that all L1 misses are L2 hits to determine entangling pairs (we measure the error this introduces). The execution of memory requests is still correct under this assumption - it does not change the contents of L1.

Performance analysis. The performance analysis here is simple - the design accumulates coverage, misses, and accuracy counters in hardware and emits them periodically (every 2^{31} cycles to the host) to avoid overflow. Thus, the traffic to host is minimal.

Simulation methodology and results. In our Champsim testbed we ran 135 CVP2 traces and compared IPU based prefetch to the original entangled prefetch implementation from the authors. Our results are nearly identical to the results from the original paper, as the only difference is the always-hit-in-L2 assumption, discussed below. To measure error, we compare our statistics to the reference simulation's statistics.

Analysis of approximations. Figure 2.8(a) shows coverage, accuracy, and miss-rate error across the traces. For each metric, our always-hit-in-L2 assumption leads to better prefetching than actual, outperforming on each statistic by less than 5% on average. In cases where the prefetch stats are high, the initial miss rate was very low (less than 0.25%) so the other prefetch stats are less meaningful. The Figure shows the distribution of errors in terms in min, max, and inner quartile. Note that we don't model cache pollution effects, which our results show has small impact on accuracy.

Area and Power. The area of an $\mathrm{IPU}_{\mathrm{pro}}$ is 0.22 mm^2. In comparison to the CPU reference, this is 0.7% area overhead; power is 20.8 mW, which is around 0.5% of the CPU reference (Section 2.4). On a chip with a single $\mathrm{IPU}_{\mathrm{pro}}$ instead of one per core complex, the area and

power overheads reduce to 0.175% and 0.125% respectively.

Takeaway 1. An IPU enables in-field A/B testing of policies on real workloads, allowing behavior inference before silicon redesign—previously infeasible for microprocessors.

Obfuscated Hardware

Modern hardware hides much of its internal complexity, leaving developers with limited visibility into how their programs actually execute. Performance counters have proven insufficient to bridge this gap [57]. Per-instruction cycle stacks (PICS) reveal which static instructions dominate execution time and what core events occur during each dynamic instance, enabling significant speedups[1]. This compute-intensive analysis demands high-rate signal access, making IPU_{lite}—placed per core complex—a natural fit. Unlike prior work that used dedicated RTL [57], IPU_{lite} constructs these stacks using its programmable core as illustrated in Figure 2.8b.

HIT & Interface. The HIT is the core pipeline of a CPU. Figure 2.7(b) shows the interface as listed in the left table including Control Sigs, which indicate long-latency events starting in the core. In addition, we have 6 virtual address PC values from 4 parts of the pipeline. To restrict program regions, users set the `TS` and `TE` registers with the fetch-PC connected to the `ADDR` register using the IPU API.

Introspection code. The introspection code has two phases: every cycle we update a Performance Signature Vector (PSV) which is a bit-mask that indicates which event has occurred for a particular dynamic instance of a PC. This follows a sequential if-else-if sequence across all supported hardware events, where if an event occurs in the core pipeline a load-modify-store sequence sets a specific bit of the appropriate PSV to a '1'. If an instruction is flushed, we store its associated PC value in the IPU's memory, so that we can

[1]Due to space limitations, we refer readers to the original TEA/PICS paper for design details [57]. Our aim here is to match the TEA behavior; the original paper already validates its utility, which we re-verified using our emulation testbed.

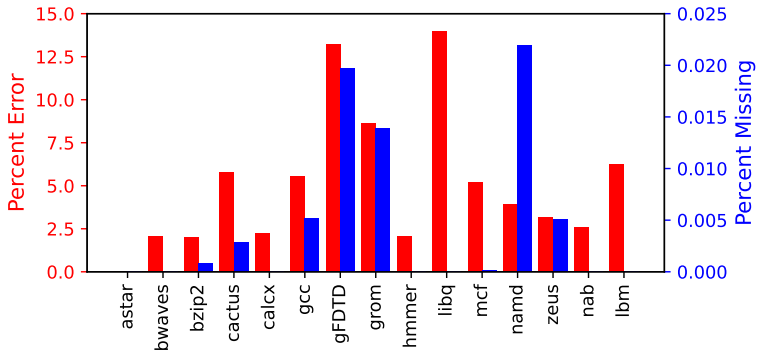


Figure 2.10: Average of the relative error for the PICS per benchmark in red and percent of total delay missed in blue

reference it after it commits. Every 400,000 cycles (TEA paper's design value) we update PICS which correspond to combining the delays of every dynamic instance of a PC into a single entry. The introspection code scans through the active list of PSVs to determine to which PSV we can attribute cycles and sends to the FIFO a payload comprising of PSV (PC + signature).

Performance analysis. 215 bits of data are used every cycle in this case study. In the common case (representing more than 75% of the cycles - our results and traces we obtained from TEA authors confirm this), no event is triggered when the ROB is sampled (as it isn't stalled/drained). In the 25% eventful cycles, typically a single long latency event occurs. Some times 2 or more events occur when the ROB is sampled (very rarely 3 events, and almost never more than that). Outside of the cycles where the ROB is sampled, there are typically 1-3 events in the processor pipeline which need to be processed, triggering 3-9 instructions of code. The introspection output data volume a few bytes of PSV data every 400,000 cycles.

Simulation methodology and results. As shown in Table 2.3, we use a gem5-based simulation. For the SPEC benchmarks, we ran 1 billion cycles of simulation after fast-forward 1 billion cycles (matching [57]'s methodology). Two example PICS stacks from our 15 applications (all of which we generated) are shown in Figure 2.8(b). For validation of PICS generation, we ran 3 DARCHR microbenchmarks [102] expecting *one* PC to show up

in the PICS stack for these microbenchmarks. The resultant PCs are shown below verifying the generation. One PC has a large cycle-count, showing that is primarily responsible for performance stalls.

PC	Assembly	kCycles	C Code Line
STL2 causes LSQ Full			
4017f6	mov %eax,(%rsi,%rdx,1)	127878	arr[lfsr].p1=lfsr
CCH_st causes Branch Misspeculation			
401813	jne 4017f8	177	if(randArr[i])
ML2 causes D-Cache Miss			
4017ee	mov %eax,(%rsi,%rdx,1)	58595	lfsr = lfsr +arr[lfsr].p1

Analysis of approximations. This case study has the notion of dropped data - if an event is triggered during the PSV generation window of a previous event, we drop that event. Note that IORegs are designed to hold their "old" data (and drop new data) until the IPU reverts back to AP state. To understand the impact of this, we used our simulation testbed to create PICS with simulating introspection code running in 1 cycle vs per-cycle simulation of the introspection (which can take 8 cycles when two events occur in the same cycle). Our error metric is defined as average relative error (compared to the single-cycle version) of the cycle stack height for each PC for each application. Figure 2.10 shows this in the red bars. Typically the quantitative error is $< 3\%$ while a couple of applications show up to 12% error. In all scenarios the list of PCs and the scale of the cycles contributions to PICS was correct, which is most important for performance optimizations. In some very rare scenarios, we drop entire PCs from the PICS stack - when a PC always appears in the dropped window. The Y-axis shows the percentage of cycles covered by these dropped PC in blue. By definition these are exceedingly rare and unimportant for performance analysis. Across our applications, they cover $< 0.025\%$ of cycles.

Area and Power. The $\text{IPU}_{\texttt{lite}}$ has an area of 0.019 mm^2. Compared to the CPU reference, this is an area overhead of 0.06%. Power consumption is 15.0 mW which is 0.38% of the CPU reference power. On a chip with a single $\text{IPU}_{\texttt{lite}}$ instead of one per core complex, the area and power overheads reduce to 0.015% and 0.095% respectively.

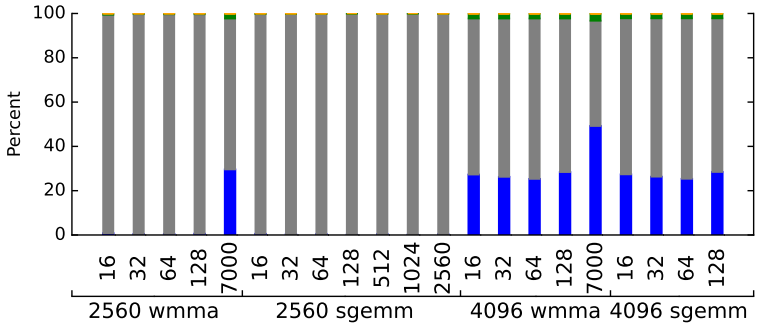


Figure 2.11: Breakdown of how use overlaps amongst the three signals collected shown per gemm shape. Blue is all low. Grey is 1 high and 2 low. Green is 2 high and 1 low. The number under each stack is the n and number in groupings is the m and k. wmma or sgemm are the kernels. The leftmost stack is (2560,16,2560) wmma benchmark.

Takeaway 2. A IPU accurately captures PICS stacks showcasing it address the HW obfuscation problem.

Obfuscated Software

Hardware designers rarely see how real applications behave post-fabrication, and benchmark suites quickly become outdated—especially with fast-evolving ML workloads—leading to a growing mismatch with in-field behavior. In the context of GPUs, as they evolve [109], designers would benefit from cycle-level visibility, which motivates using IPU_{lite} to capture high-rate, low-complexity insights. **IPU_{lite} can generate histograms of fine-grained, cycle-level activity, revealing mutually exclusive usage patterns.** In optimized CUTLASS GEMM kernels, SIMT Core, TensorCore, and Memory Engine operate in largely disjoint cycles, highlighting a clear opportunity for overlap and performance gains without extra hardware or bandwidth.

HIT & Interface. This is a GPU case study, with the HIT for the IPU being an SM scoreboard block. The data signals are 3 one-bit signals indicating whether the SIMT core is active, the TC is active, and the L1 cache subsystem is in a state where it is servicing one or more outstanding requests (MSHRs non-empty status). While GPU hardware is proprietary the performance counters from NVIDIA Nsight Compute CLI (NCU) count aggregates

for these signals indicating they are readily available and not disrputive from a design standpoint. Optionally, the virtual address retiring PC is connected to the ADDR register with IPU's software API used to restrict regions of interest. To isolate to a region within a kernel, PC start and end range be provided to TS and TE. Or it can be run without trigger to capture this activity for the entire kernel.

Introspection code. The introspection code is a single histogram instruction (optimized with a loop directive) that runs for 256 cycles receiving new data every cycle. The output is 3 bytes every 256 cycles, denoting how many active cycles of that unit, which can also be batched across windows.

Performance analysis. Every 256 cycles, we emit three 1-byte values; thus, introspection output data bandwidth is 3 * 108(#SMs) / 256 bytes per cycle = 1.7 GB/second at 1.4 GHz. This frequency of this host traffic can be further reduced by batching in the IPU's data-store. By using longer windows the traffic can be further reduced.

Simulation methodology and results. One representative output is shown in Figure 2.8(c). The top half shows chronologically ordered windows of 256 cycles with the Y-axis denoting % of cycles in that window where that signal was active. We can see the mutually exclusive behavior. The bottom graph shows the same data in a histogram format: the windows are sorted in increasing order of utilization, and we plot the running average up to that window for the signals. Figure 2.11 post processes this data and presents it in a different way. We classify windows of 256-cycles into 4 bins: 2 signals high (green), one signals high (blue), all signals low (grey). Where high mean greater than 25% the cycles in that window, and low meaning less than 25% of the cycles. We can see that large portions of time are spent with at least one component of the SM being idle - pointing to further hardware optimization beyond directions like TMA [109] that have appeared. Other work has also looked at improving such utilization [157, 20].

Analysis of approximations. This case study uses our histogram instruction in a novel way, essentially treating each signal as its own bucket and builds a 3-bucket histogram.

Hence we have very little dropped data - the source of error is the few cycles needed at the end of a window to write 3 bytes of accumulated statistics.

Area and Power. The $\text{IPU}_{\texttt{lite}}$ has an area of 0.019 mm^2 that is 0.6% of the area of the GPU reference. This case study consumes 24.1 mW of power as determined by Section 2.4 methods. The $\text{IPU}_{\texttt{lite}}$ power overhead is around 3% of GPU reference. If instead the chip designer included only one $\text{IPU}_{\texttt{lite}}$ on the chip, the area and power overheads are 0.003% and 0.024% respectively.

Takeaway 3. An IPU enables fine-grained, simulation-level GPU analysis in the field, revealing optimization opportunities in concurrent resource usage.

Value Histograms

Another form of the obfuscated software problem is that ML workloads can perform well on reduced precision. Recent work has shown arithmetic can play a large role in energy efficiency [34]. Therefore, gathering distributions tensor activation values can inform decisions on arithmetic both at design-time and adaptive run-time. In particular, we determine histograms of **values** of tensors in deep learning applications to understand inter- and intra-application differences without needing access to code or model structure. We demonstrate empirically two results: First, an IPU can obtain histograms of distributions. Second, we show that distributions have substantial differences between operators and across operators between applications, showing that data richness is necessary.

HIT & Interface. The HIT is the L1 dCache block. The data signal is 256-bits corresponding to cache-line writes into the L1 cache (exact width not public), of which 64-bits are used with the remaining dropped to conserve power and area. To restrict regions of interest in the program, if necessary, address ranges of tensors which can be extracted from Python's API are set to the `TS` and `TE` registers. The `ADDR` register is set to the address of the L1 cache line write. GPU implementations of GEMMs and other operators in DL do not do partial

accumulations, hence when data is written to the L1, that is the "final" value of the Tensor.

Introspection Code. The introspection code is a single instruction hash because of the inclusion of the hardware histogram unit. It is within a signal-processing-esc loop instruction that ensures no wait-time between hash instructions.

Performance Analysis. Assuming fp16 data-types, we process 4 elements every cycle, and populate a 128-bucket histogram with 18-bit integers. Every 32768 cycles, we write the histogram to host memory, which amounts to 384 bytes. Considering an A100 class GPU with 108 SMs, operating at 1.4 GHz frequency, the total transfer bandwidth to host through PCIe is 1.7 Gbytes/second, roughly 5% of bandwidth. A wider histogram accumulator, would increase area of the YPU marginally, but can substantially reduce the PCIe bandwidth needed.

Simulation methodology and results. For results, we produce a histogram per tensor for the values within around 5000 activation tensors from inference Resnet (batch=16), BERT (tokens=128) and LLama2 (tokens=64). By considering each tensor's histogram as a 128-D vector, we can get inter-tensor cosine similarity. By averaging tensors across an application, we can also get inter-application similarity. Both of these are shown in Figure 2.8(d). These show tensors have dis-similarity between applications in their value distributions, showing application-driven arithmetic optimizations could help.

Analysis of approximations. For high efficiency, we sample only 1/4th of the data, whose error metric can be defined as average relative error of every tensor across every bucket per application between sampled and non-sampled. If H_{all} represents the reference histogram, and $H_{sampled}$ represents the sampled histogram, our error metric is

$$\left(\sum_{n=0}^{M} \sum_{i=0}^{127} abs(H_{all_n}[i] - H_{sampled_n}[i]) / H_{all_n}[i] \right) / (M * 128)$$

for M tensors in an application. For our three applications this error is 5.9%, 2.4%, and 4.8%.

Area and Power. The $\text{IPU}_{\texttt{lite}}$ has an area of 0.019 mm^2 that is 0.6% of the area of the GPU reference. This case study consumes 4.7 mW of power as determined by Section 2.4 methods. The $\text{IPU}_{\texttt{lite}}$ power overhead is around 0.5% of GPU reference. If instead the chip designer included only one $\text{IPU}_{\texttt{lite}}$ on the chip, the area and power overheads are 0.003% and 0.004% respectively.

Takeaway 4. Using an IPU, value histograms are possible without application modification by all entities in the semi-conductor stack. Furthermore, our analysis shows that values distributions are highly varied across applications and within, thus showing data richness is necessary and valuable.

Comparison with Existing Mechanisms

Across all four case studies, the IPU enables runtime, signal-level introspection beyond the reach of fixed-function hardware, tracing tools, or postmortem analysis. We summarize each case study's relation to existing tools. In the first, the IPU emulates a candidate prefetcher via custom logic triggered on memory accesses—something not possible with commercial tools like Intel PCM, CUPTI, or CoreSight, which lack in-field programmable logic. While LBA [23, 24] is conceptually closest, it doesn't support user-defined prefetch emulation in production. In the TEA case, performance counters cannot implement Time-Proportional Event Analysis (TEA) [57], which underpins PICS. Unlike prior work requiring dedicated RTL for RISC-V BOOM, our IPU replicates TEA without RTL changes and with greater flexibility. Finally, the IPU builds histograms of fine-grained, cycle-level signal states, which are impractical to obtain via simulation, emulation, or debug monitors due to visibility limits or high overheads (e.g., 100× slowdown). Profiling co-processors [160] offer partial alternatives but struggle with non-uniform, fine-grained utilization.

2.6 Related Work

We define a four-attribute taxonomy to contextualize the IPU in related work (Table 2.5). **S**peed measures how quickly introspection can run without disrupting execution; **T**ransparency indicates visibility into microarchitectural or gate-level behavior—both crucial for all three key challenges. **P**rogrammability captures flexibility for post-fabrication analysis, enabling A/B testing and handling obfuscated hardware. **A**ccessibility reflects ease of use for hardware/software developers, especially in-field, supporting A/B testing and opaque software contexts.

Software profiling tools like `strace`, `gprof`, and binary instrumentation are well-established, but lack hardware transparency and suffer slowdown as introspection complexity increases (e.g., instruction count is fast with Pin, memory tracing is not). *Performance counters* offer speed and accessibility and are the current state of the art, yet suffer from limited programmability and visibility—restricted to pre-defined events, with no support for A/B testing or richer analytics. Techniques such as PGO driven by perf counters [59, 22, 21, 120, 147, 104, 119, 37, 94, 84, 78] and Intel PT [61] improve program optimization, but address only part of the hardware opacity problem. *Debug monitors* [66], including JTAG and boundary-scan, offer deeper hardware access but require physical connections and cannot capture live software execution efficiently—making them inaccessible for typical users. *HW emulation* platforms like Cadence Palladium [114] and Synopsys Zebu [118] offer full hardware transparency but are limited to chip designers, cost millions of dollars, and run at 2–5× slowdown. RTL simulation lowers cost but is orders-of-magnitude slower and equally inaccessible. *Cycle-level simulators* improve speed slightly, but still suffer 100× or more slowdown, making them impractical for in-field use. Table 2.6 compares IPU to two-SOTA introspection approaches.

The related academic works fall under 3 categories. Along the security angle, DISE [28], FlexCore [40], LBA[23, 24], and PHMon/Nile [38, 39] focus on triggers based on events occurring and often run at-speed, however eschewing programmability, accessi-

Yr	Technique	S	P	T	A
	General Approaches				
–	SW profiling	N	N	N	Y
–	Perf counters	Y	N	p	Y
–	Debug monitors	Y	N	Y	N
–	HW emulation	N	Y	Y	N
–	RTL simulation	N	Y	Y	N
	Security				
'03	DISE [28]	N	p	N	N
'10	FlexCore [40]	Y	p	Y	N
'11	LBA [23, 24]	N	Y	N	N
'20	PHMon/Nile [38, 39]	Y	p	N	N
	Embedded Systems				
'10[1]	ABACUS[3] [90, 133, 42]	Y	p	N	Y
'13	hidICE Verification [10]	Y	Y	N	N
'15	SOF [76, 77]	Y	N	p	N
'16[1]	AIPHS[2] [95, 140, 139]	Y	N	p	Y
'17	Enhanced PMU[2] [128]	Y	N	p	Y
'18	NIRM [130]	Y	N	N	N
	Profiling and PerfMonitors				
'01	Programmable Co-Proc[160, 97]	Y	Y	p	N
'01	Stratified Sampling [127]	N	Y	p	N
'03	ULF [156]	p	Y	N	N
'03	Interval Based Profiling [98]	N	Y	p	N
'05	Owl [129]	N	p	Y	Y
'25	**IPU (ours)**	Y	Y	Y	Y

[1]Year of most relevant work [2]Limited to manipulating event counts
[3]Limited to architectural traces. p means partial.

Table 2.5: Related work in our 4-axes taxonomy. S (Speed); P (Programmability); A (Accessibility); T (HW Transparency)

Feature	CoreSight	Intel PT/PMU	IPU
Granularity	Trace-level	Event/sample	Per-signal
Programmable	No	Configurable	Full
Control interface	Hardwired	MSRs	API

Table 2.6: IPU compared to existing introspection mechanisms

bility, and transparency. Driven by JTAG-like approaches, techniques from embedded systems, ABACUS[90, 133, 42], hidICE Verification [10], SOF [76, 77], AIPHS [95, 140, 139] Enhanced PMU [128], NIRM [130], allow at-speed analysis, in general assuming direct access to the hardware signals - thus lacking accessibility and programmability. Finally, novel uses of performance counters and profiling have been proposed, which suffer from lack of hardware transparency[127, 156, 98, 129, 146, 88].

2.7 Conclusion

This chapter presents a detailed treatment of hardware introspection resolving three key challenges: Lack of A/B Testing for hardware, obfuscated Hardware, and obfuscated Software . We demonstrate an IPU can execute introspection programs in the field at real time speeds. This paper's key contributions are the definitions, system design, and implementation of the introspection processing unit (IPU) down to the level of an RTL implementation, including a comprehensive simulation testbed comprising both CPU and GPU designs. We show an implemented IPU adds $< 1\%$ to area and power. The IPU provides unprecedented insights to hardware designers and software developers.

3 NEUROSCALAR: A DEEP LEARNING FRAMEWORK FOR FAST, ACCURATE, AND IN-THE-FIELD CYCLE-LEVEL PERFORMANCE PREDICTION

The relentless pace of innovation in microprocessor design is fundamentally constrained by the methodologies used to evaluate new architectural ideas. The gold standard, cycle-level simulation [3, 9, 12, 17, 43, 116, 126], provides high-fidelity performance data but suffers from prohibitive performance overheads. For instance, a detailed out-of-order CPU model in a widely-used simulator like gem5 may only achieve a simulation rate of 0.1 million instructions per second (MIPS), several orders of magnitude slower than native hardware execution. This bottleneck is exacerbated by the ever-increasing complexity of modern processors. This, in turn, necessitates a correspondingly larger exploration of the design space. Furthermore, traditional evaluation relies heavily on standardized benchmark suites or curated workload traces. While useful, these benchmarks often fail to capture the dynamic and unpredictable nature of "in-the-field" applications that run on user devices that result from complex interactions. An ideal evaluation methodology would therefore possess three key properties: **speed** approaching native execution, **fidelity** matching that of a cycle-accurate simulator, and **transparency** that allows for evaluation on real user workloads without interference or noticeable slowdown.

No existing technique successfully achieves this trifecta. The previous system, Introspection Processing Units (IPUs), provides these three key properties with a crippling caveat: an IPU cannot evaluate full systems. It can perform A/B testing on the microarchitecture level, one component at a time – not the architecture or system level. This chapter removes this caveat through deep learning keeping an IPU-like system architecture: gather microarchitecture values as inputs, use a co-processor for analysis, and report back. While prior work has explored machine learning for performance prediction, notable approaches fall

short of enabling true in-the-field evaluation. For example, SimNet [80] requires detailed microarchitectural traces as input, which are unavailable on production hardware for a hypothetical design. Other approaches, such as TAO [115], pursue the ambitious goal of learning a general model of processor behavior but focus on "high-level" hardware trends. This is of limited value to a chip designer as these high-level trends are often already known. The true secret sauce of modern microarchitecture lies in extreme feature engineering—subtle choices like prefetcher policies, load-store queue (LSQ) sizing, or the number of load ports. TAO is ill-equipped to capture the impact of such low-level details; its authors concede that accuracy suffers on branch-predictor-dependent applications. Furthermore, its massive model size precludes any practical form of the sampling-based, in-the-field inference necessary for transparent deployment. This leaves a critical gap: a methodology to rapidly evaluate how specific, low-level microarchitectural trade-offs perform on real-world workloads.

This chapter addresses this multifaceted challenge by introducing a novel paradigm for ultra-fast, in-the-field simulation of new microarchitectural concepts on existing production silicon. Our approach is centered on a deep learning (DL) model trained *a priori* to perform cycle-level performance prediction. The key insight is to train this model using only microarchitecture-independent features—that is, features derived solely from the instruction stream and its data dependencies. During deployment, this pre-trained model runs on production chips in the hands of users, ingesting instruction traces from the host processor and predicting the execution latencies of a new, hypothetical microarchitecture. As even the DL model inference is too slow to process constantly produced data at native speed, we employ a sampling-based methodology. A lightweight hardware monitor captures a sample epoch of instructions (e.g., 100,000 instructions), which is then processed by the DL model. Our model achieves an inference rate of ~4 MIPS on desktop class GPUs like RTX4090. By tuning the sampling frequency—for instance, sampling one epoch every 75 billion instructions (a 100,000 instruction epoch every ~25 seconds)—we can scavenge

idle resources from an accompanying GPU, introducing a negligible GPU slowdown of just 0.1%. When using our co-designed 28mW Neutrino in-silicon accelerator, this sampling rate dramatically increases to an epoch every 4.9 seconds. With an 8-tile design, further reducing to an epoch every 0.6 seconds, while consuming 230mW.

This framework, we call NeuroScalar enables powerful new modalities for architectural evaluation. It can be used to gain deep insights into how diverse, real-world applications perform on a proposed design: an *Enterprise Forecaster* like a hyperscalar who wants to understand future chip (their own or from a vendor like AMD, Intel, NVIDIA) on their proprietary workloads. More significantly, for chip designers (who are not hyperscalars - Intel, AMD, NVIDIA) it facilitates large-scale design space exploration by allowing multiple candidate designs (represented by different trained models) to be evaluated concurrently on live workloads (through Ecosystem Partners), creating the hardware equivalent of A/B testing. The primary contributions of this chapter are:

1. The design and implementation of a fast and accurate DL model for cycle-level performance prediction that relies on microarchitecture-independent features.

2. The design of a complete system for in-field evaluation, including a simple hardware module for trace collection and an ultra-lightweight, 28mW accelerator for the DL model. This accelerator enables continuous sampling and achieves an 85$\times$ energy reduction and a 391$\times$ area reduction compared to GPU-based inference, while allowing a 5$\times$ higher sampling rate.

3. A thorough evaluation of the model's predictive accuracy against a cycle-level simulator.

4. A demonstration of the framework's effectiveness for design space exploration, where it achieves 95% accuracy in A/B testing across eight pairwise comparisons of five distinct processor designs.

This chapter is organized as follows. Section 2 provides an overview of our approach. Section 3 details the DL model architecture. Section 4 describes the end-to-end system design, including the hardware components. Section 5 outlines our experimental methodology, and Section 6 presents our results. We discuss related work in Section 7 and conclude.

3.1 System Overview

This section provides a high-level overview of the NeuroScalar system. We frame the core challenges in processor simulation that motivate NeuroScalar and present our key technical insights. We describe the end-to-end flow, defining the roles of its users and the practical use cases it enables.

Challenges and Key Insights

The central challenge in modern microprocessor design is the three-way trade-off between simulation **fidelity**, **speed**, and **workload relevance**. Traditional cycle-level simulators provide high fidelity but are excruciatingly slow (often by a factor of 1,000,000x), rendering large-scale design space exploration impractical. This speed limitation also forces designers to rely on standardized benchmark traces, which fail to capture the rich, data-dependent behaviors of the in-the-field applications that users run on a daily basis. *This chapter directly confronts this challenge: how can we evaluate novel microarchitectural ideas with cycle-level accuracy, at near-native speed, on real user workloads, and with complete transparency to the end-user?*

Our solution, NeuroScalar, is built upon a foundational insight: we can decouple the slow, offline process of detailed simulation from a fast, online prediction phase by using a deep learning (DL) model trained on **microarchitecture-independent features**. This allows a model, trained once by a chip designer in a laboratory setting, to be deployed on existing production silicon to predict the performance of a different, hypothetical processor. This breaks the dependency on slow, monolithic simulation for every design change.

A second key insight enables practical, in-the-field deployment. To be viable, the prediction process must be non-intrusive. While prior work like Simnet and Tao demonstrated the potential of DL for simulation, their models were often too heavyweight for transparent deployment. NeuroScalar is explicitly co-designed to be **ultra-lightweight**. We combine a compact, 1-million-parameter model with an **sampling methodology**. Any high-fidelity simulation requires significantly more computation per instruction than native execution. For instance, even an idealized linear model on a state-of-the-art A100 GPU running at 100% compute utilization, consuming 300W, cannot process instruction features as fast as a single CPU core produces them. Therefore, a principled sampling methodology is not a workaround, but a fundamental requirement for any transparent, in-the-field simulation. NeuroScalar is architected around this reality. For scenarios without a powerful GPU, we also introduce a co-designed **hardware accelerator** that delivers high-speed inference within a tiny power budget of **28mW** at 7nm based on a full RTL and P&R flow. Our cross-cutting insight is: *processor simulation is fundamentally a time-series behavior, which LSTMs are intuitively well suited, for high accuracy they benefit from a context that is substantially larger than the ROB size, and a hardware accelerator can be co-designed to match the embedding dimension of the LSTM to achieve ultra-high efficiency. Specifically feature engineering to preserve address information, specializing layers for long and short sequences, log-transform to handle the wide range of output retirement cycles, a sequence length upto* $3\times$ *larger than an ROB demonstrates small models can accurately learn using microarchitecture-independent features only.*

End-to-End Workflow and Use Cases

The NeuroScalar system operates in a two-phase workflow that involves two key actors: the **chip designer** and the **end-user**. During the offline Training Phase, the **chip designer** performs a one-time, offline training process. Using a conventional cycle-accurate simulator, they generate training data to produce a suite of NeuroScalar models. Each lightweight model is a proxy for a specific microarchitectural idea (e.g., a new cache prefetch algo-

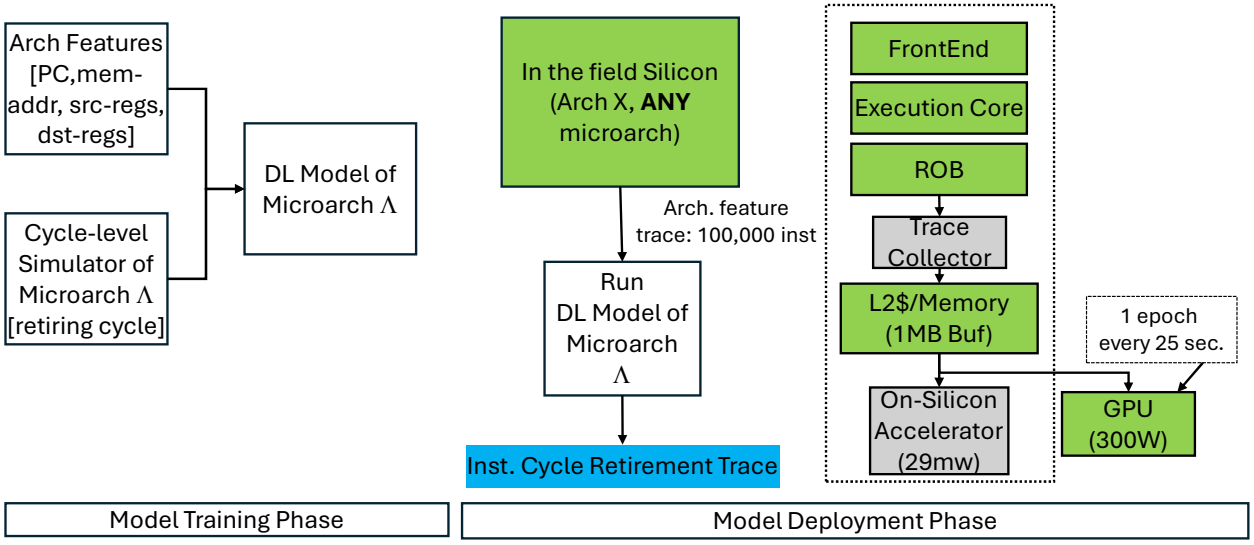


Figure 3.1: The NeuroScalar end-to-end workflow, showing the offline training phase performed by the chip designer and the online inference on the end-user's system.

rithm or branch predictor), effectively encapsulating a complex design into a portable neural network. During the *online Inference Phase*, the trained models are distributed to end-users. A lightweight tracing mechanism on the user's machine—either a minor hardware modification or enhancing existing debug interface (e.g., Intel PT)—captures the required architectural features from running applications. These traces are then fed into the NeuroScalar model to predict cycle counts for the hypothetical hardware, based on a sampling strategy. This workflow enables powerful new use cases tailored to different users.

- **For the Enterprise Forecaster:** A customer, like a hyperscalar vendor, with proprietary workloads can use a NeuroScalar model to get a concrete performance forecast for a future processor on their own applications, all without sharing their sensitive code or data with the chip designer.

- **For the Chip Designer:** The designer can now perform massive, in-the-field A/B testing. By collecting anonymized performance reports from opt-in **Ecosystem Partners**, they can gain unprecedented insight into how their design ideas perform across a

diverse, real-world software ecosystem.

3.2 DL Model Theory

This section details the methodology for our cycle-level prediction framework. The data pipeline encodes each instruction with 13 microarchitecture-independent features. There are six key properties gathered from an instruction: PC, opclass, mem_addr, src_reg1, src_reg2, and dst_reg. The 64-bit PC and mem_addr are represented as 3 features of 2x22bits + 20bits. Registers are represented as a pair (class and number). This accounts for the 13 features. They characterize ground-truth (GT) cycle distributions across benchmarks, and we mitigate severe target skew via clipping, a logarithmic transform, and an auxiliary two-way classifier. We then assess four neural architectures with respect to prediction accuracy, computational efficiency, and memory footprint: LSTM, CNN, SSM and Transformer based designs. We select an LSTM backbone based on its lightweight architecture and performance. An LSTM is a recurrent neural network architecture that utilizes a dedicated cell state and gating structures (input, forget, and output) to selectively manage the flow of sequential information, enabling it to model long-range dependencies [131] – a good fit here. Finally, we describe the predictor's architecture, including a study to identify the optimal number of LSTM layers, and a unified design that jointly trains the regression and classification heads.

Data Representation and Preprocessing

Feature Selection

Designing an effective cycle-level prediction framework requires features that not only reflect instruction-level semantics but also capture execution-relevant microarchitectural patterns. A central challenge lies in constructing a representation that is sufficiently expressive to encode spatial locality, operational semantics, and data dependencies, while

remaining independent of specific microarchitectural configurations. Such a representation must generalize across workloads and hardware designs, avoid overfitting to architecture-specific idiosyncrasies, and still preserve information critical for latency prediction.

To address these requirements, we adopt a set of six carefully selected microarchitecture-independent properties, which together provide a holistic view of each instruction's execution context. These key properties are represented in 13 features. These features fall into two categories: (1) continuous-valued encodings of instruction and memory addresses to preserve spatial locality, and (2) categorical descriptors such as operation type and register usage to convey semantic and dependency information. This combination enables the model to capture both fine-grained numerical patterns and discrete structural relationships, thereby mitigating the limitations of using either type alone.

Feature Processing and Usage

We next consider how each feature is encoded, transformed, and integrated into the model. Registers are encoded as a pair: their class and number.

Address decomposition. Given a 64-bit address a, we preserve locality and expose large jumps by a 3-way split:

$$\phi_{\text{addr}}(a) = \Big(\underbrace{\lfloor a/2^{42} \rfloor}_{\text{upper 22b}}, \underbrace{\lfloor (a \bmod 2^{42})/2^{20} \rfloor}_{\text{middle 22b}}, \underbrace{a \bmod 2^{20}}_{\text{lower 20b}} \Big) \in \mathbb{R}^3.$$

These are encoded as fp32 in order to preserve the entire bit-sequence; fp32 has 23 bits of precision in the mantissa.

Embeddings and projections. Categorical features are embedded; continuous features are linearly projected into the same space:

$$
\begin{aligned}
e_t^{\text{op}} &= \mathbf{E}_{\text{op}}[o_t], \\
e_{t,j}^{\text{reg}} &= \left[\mathbf{E}_{\text{cls}}[c_{t,j}];\ \mathbf{E}_{\text{idx}}[n_{t,j}]\right], \quad j \in \{\text{dst}, \text{src1}, \text{src2}\}, \\
e_t^{\text{pc}} &= \mathbf{W}_{\text{pc}}\phi_{\text{addr}}(a_t^{\text{pc}}), \quad e_t^{\text{mem}} = \mathbf{W}_{\text{mem}}\phi_{\text{addr}}(a_t^{\text{mem}}).
\end{aligned}
$$

The instruction vector is the concatenation

$$
x_t = \left[e_t^{\text{op}};\ e_{t,\text{dst}}^{\text{reg}};\ e_{t,\text{src1}}^{\text{reg}};\ e_{t,\text{src2}}^{\text{reg}};\ e_t^{\text{pc}};\ e_t^{\text{mem}}\right] \in \mathbb{R}^{13}.
$$

Sliding window formation. Training data is generated via a sliding window of arbitrary length N. Given a target segment length $R \leqslant N$, we center the target and use the remaining positions as context. Let $s = \lfloor (N - R)/2 \rfloor$. Then

$$
x_{1:N} = \left[x_{1:s} \parallel x_{s+1:s+R} \parallel x_{s+R+1:N}\right], \quad \mathbf{y} = \mathbf{y}_{s+1:s+R} \in \mathbb{R}^R, \tag{3.1}
$$

so the left context has length s and the right context has length $N - R - s$. The model predicts cycles for the middle segment using both the preceding and succeeding context. All features are normalized before training, and the target cycles are preprocessed as in Section 3.2. This is one of our fundamental insights, creating a very long context, we found $3\times$ of ROB size, provides accuracy while using only **microarchitecture-independent** features.

Ground Truth Distribution

We examine the ground-truth (GT) cycle distribution for ten representative benchmarks: namd, blender, cam4, deepsjeng, exchange2, gcc, imagick, leela, mcf, and nab. The distribution for a given benchmark is calculated as the portion of the instructions of said benchmark

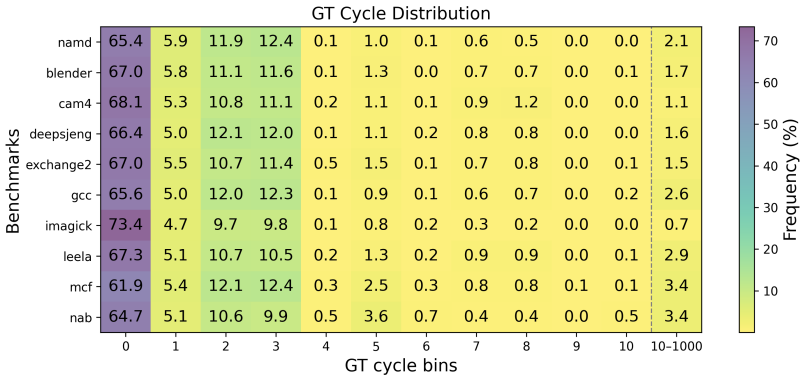


Figure 3.2: GT cycle distributions as percentage of total number of instructions shown per benchmark - the rows.

that have a specified cycle latency. The distributions are visualized as a heatmap (Fig. 3.2) where each row is a benchmark and columns represent integer cycle bins from 0 to 10, plus a long-tail bin covering 10 to 1000 cycles.

Across all benchmarks, the majority of instructions have very small cycle counts (mode at 0 cycles and high density for 1,2, and 3 cycles), with more than 60%–73% of events occurring at zero cycles and another 20%–25% within the first three cycles. Beyond this dense low-latency region, the distributions exhibit a long but thin tail: bins beyond 10 cycles account for only 0.7%–3.4% of number of instructions across benchmarks. Events beyond 1000 cycles are exceedingly rare. This extreme concentration in the low-latency region creates a challenging imbalance: regressors risk being dominated by the abundant short-latency samples while underrepresenting the patterns that characterize rare, high-latency cases.

Ground Truth Processing

To mitigate the effects of this heavy skew and improve regression performance, we apply a logarithmic transformation to the GT values. Concretely, we cap extremes and then

compress the range:

$$\tilde{y}_i = \min(y_i, 1000), \qquad z_i = \log\bigl(1 + \tilde{y}_i\bigr). \tag{3.2}$$

This compresses the dynamic range of the target variable, smooths the distribution, and reduces the disproportionate influence of rare high-latency samples, enabling the model to learn from a more balanced signal.

Furthermore, we modify the regression task with an auxiliary two-way classification task, in which each sample is labeled according to whether its GT cycle count falls below or above a threshold. With a tunable threshold τ (set to τ=10 in our experiments), we define

$$c_i = \mathbb{1}\{y_i > \tau\} \in \{0, 1\}. \tag{3.3}$$

This choice aligns with the natural boundary between the dense low-latency region and the sparse long tail, and the framework allows τ to be adjusted for different application needs. Both the classification and regression tasks are trained jointly, sharing a common feature representation; this joint optimization enables the classification branch to dynamically inform the regression head about the likely latency regime during inference, avoiding the error-propagation risks of a sequential approach and supporting end-to-end learning that is both more accurate and more flexible in practice.

In summary, our main steps are: *Address decomposition, large context, log-transform, and specialized long/short region predictors which are novel yet simple techniques that enable a small LSTM to effectively learn microarchitecture performance, without requiring a detailed microarchitecture trace.*

Choice of Models

We formulate cycle-level prediction as a supervised regression problem with a clear input–output association task. Given a window $x_{1:N} \in \mathbb{R}^{N\times 13}$ (Section 3.1), the model

Table 3.1: Layer-depth study for the BiLSTM backbone. Speed is in million instructions per second (M instr/s).

Layers L	MAE↓	RMSE↓	RAE↓	Acc(round)↑	±1↑	±2↑	rel⩽5%↑	Speed (M)↑	val_loss↓
1	0.3623	4.9636	0.1387	0.7948	0.9512	**0.9856**	0.7998	**4.77**	0.0290
2	**0.3509**	**4.9386**	**0.1340**	**0.7968**	**0.9533**	0.9845	**0.8013**	4.00	**0.0283**
3	0.3579	4.9984	0.1357	0.7895	0.9520	0.9840	0.7938	3.12	0.0287

estimates the GT cycles for the centered R-length target segment (Eq. 3.1). In other words, we learn the mapping

$$f_\theta : \mathbb{R}^{N\times 13} \to \mathbb{R}^R, \qquad \widehat{\mathbf{y}} = f_\theta(x_{1:N}),$$

augmented with an auxiliary two-way classifier that shares the backbone to indicate the low-/high-latency regime (Section 3.1.3).

To instantiate f_θ, we evaluated four backbone families - LSTM, Transformer, CNN, and SSM - covering recurrent, attention-based, convolutional, and state-space designs. Transformers proved too heavy for our lightweight, real-time setting, while CNNs and SSMs underperformed. LSTM offered the best accuracy–efficiency trade-off and was therefore adopted as our backbone. Further analysis is in Section 3.3.

Model Structure

For our regression task – a window of N instruction-level feature vectors serve as the input and the GT cycles for the central R positions are the output. We now ask: what computational scaffold most reliably turns contextual instruction streams into accurate cycle estimates? Our design proceeds in three steps: (i) construct a backbone that faithfully encodes sequential context seen before and after the target region, (ii) attach task heads that express both continuous (regression) and discrete (regime) structure of latency, and (iii) validate capacity via a depth study before finalizing the configuration for deployment.

Backbone

We adopt a stacked bidirectional LSTM (BiLSTM) as the sequence encoder. Since the prediction target is the centered segment within a sliding window, both past and future context are available during training and inference; thus, strict causality is not required in this setting. The input to the backbone is the length-N sequence of instruction embeddings from Section 3.1, linearly projected to a hidden size H:

$$x_{1:N} \in \mathbb{R}^{N\times 13}, \qquad \tilde{x}_t = \mathbf{W}_{in} x_t + \mathbf{b}_{in} \in \mathbb{R}^{H}.$$

The BiLSTM (implemented with `batch_first`) yields contextualized states by concatenating the forward and backward hidden vectors:

$$\mathbf{h}_t = \left[\overrightarrow{\mathrm{LSTM}}(\tilde{x}_t);\ \overleftarrow{\mathrm{LSTM}}(\tilde{x}_t)\right] \in \mathbb{R}^{2H}.$$

Stacking across time gives

$$\mathbf{H} = [\mathbf{h}_1, \ldots, \mathbf{h}_N]^\top \in \mathbb{R}^{N\times 2H}.$$

Then slice the centered target region as the shared representation for prediction, following the window split in Eq. (3.1):

$$\mathbf{H}_{\mathrm{mid}} = \mathbf{H}_{s+1:s+R} \in \mathbb{R}^{R\times 2H}. \tag{3.4}$$

This matches our data formulation (Section 3.1.2), which leverages both left and right context to forecast the middle region.

Heads and unified design

On top of $\mathbf{H}_{\text{mid}}$ (Eq. 3.4), we attach (i) a regression head that maps each hidden state to a scalar cycle prediction, and (ii) a two-way classifier that predicts whether the instruction lies in the short- or long-latency regime (threshold at 10 cycles; Section 3.1.3). Concretely, the classifier applies a two-layer MLP with ReLU to produce logits $\mathbf{p} \in \mathbb{R}^{R\times 2}$. For regression, we employ two lightweight heads, g_{short} and g_{long}, each a single-layer MLP from $\mathbb{R}^{2H} \to \mathbb{R}$, and select per-instruction outputs by a mask induced from the predicted class. This decouples local response surfaces for different latency regimes while preserving a single shared encoder.

Depth study (capacity vs. efficiency)

Under identical training and evaluation settings, we vary $L \in \{1, 2, 3\}$. As shown in Table 3.1, L=2 offers the best accuracy–efficiency trade-off with competitive throughput (4.00M instr/s) and the lowest validation loss (0.0283). By comparison, L=1 is faster (4.77M instr/s) but slightly less accurate, while L=3 yields no accuracy gains and incurs higher cost (3.12M instr/s). We therefore adopt L=2 as the default.

Putting it together

Figure 3.3 shows the final architecture. An input projection aligns heterogeneous features into a common H-dimensional space; an L=2 stacked BiLSTM encodes the window bidirectionally; the central R embeddings feed a unified head block with a two-way classifier and regime-conditioned regressors. Unless stated otherwise, we set H=128.

Training and joint objective

Two practical challenges guide our training design: (a) the regression target is heavy-tailed even after clipping, and (b) the classifier and regressors must share signal without cascading errors. Thus, we adopt *joint training* with a single loss that uses the log-transformed

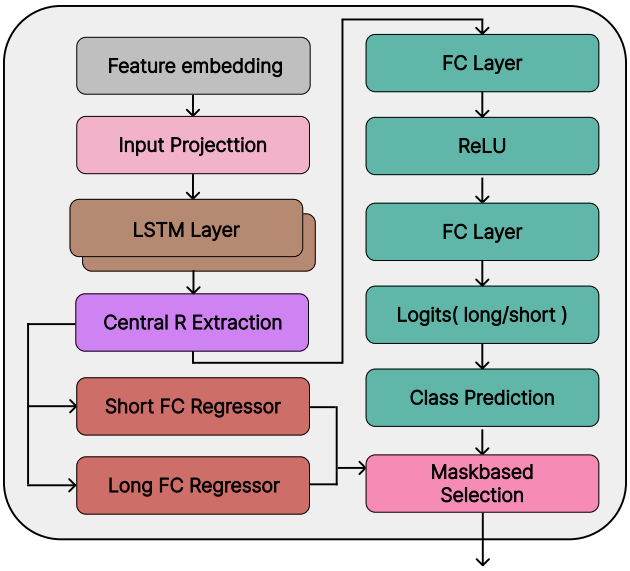


Figure 3.3: Overall architecture of the proposed LSTM-based cycle predictor.

target z_i (Eq. 3.2) and the regime labels c_i (Eq. 3.3):

$$\mathcal{L} = \underbrace{\frac{1}{R}\sum_{i=1}^{R} \mathrm{SmoothL1}\big(\widehat{y}_i,\ \log(1+y_i)\big)}_{\text{regression on log-transformed cycles}} + \lambda \underbrace{\frac{1}{R}\sum_{i=1}^{R} \mathrm{CE}\big(\widehat{\mathbf{p}}_i,\ c_i\big)}_{\text{two-way classification}},$$

The shared encoder lets the classifier shape the representation toward regime-discriminative features, while the regime-conditioned heads prevent a single regressor from being biased toward the dominant short-latency region. We train end-to-end with dropout on recurrent layers and early stopping on validation MAE; unless noted, the joint setting is used for all experimental results.

Accuracy and Downstream Tasks

Our design choices aim to improve predictive performance while remaining mindful of deployment constraints. Similar to vision models that must operate efficiently on edge devices, high "raw accuracy" is desirable but not the sole criterion of success. What ultimately matters is whether the backbone delivers strong performance in the context of

downstream tasks. A model that achieves slightly higher raw accuracy at the expense of significantly greater computational cost may provide only marginal benefits for the end application.

In this sense, our backbone plays a role akin to that of foundational encoders in vision and video processing. The raw accuracy of such models should be interpreted in light of their downstream utility. As our results show, although the backbone achieves 70–85% raw accuracy, it enables downstream tasks—such as selecting between processor configurations across diverse benchmarks—to reach over 95% accuracy. This demonstrates that the backbone strikes an effective balance between efficiency and task-level performance.

3.3 System Design

To realize the vision of in-the-field microarchitectural simulation, a practical and robust system design is paramount. This section details the end-to-end architecture of NeuroScalar, focusing on the two central challenges: the efficient **collection of feature traces** and the **high-speed execution of model inference**. We address these challenges by presenting a complete data and execution pipeline. First, we describe the mechanisms for gathering the necessary microarchitecture-independent features, both from a cycle-level simulator during the offline training phase (Section 3.3) and from live production hardware during online inference (Section 3.3). Next, we detail two distinct deployment targets designed to run the NeuroScalar model with minimal overhead. We present a software-based inference engine optimized for commodity GPUs, designed to scavenge resources transparently. We then describe a co-designed, power-efficient on-chip accelerator for environments where a GPU is not available or where dedicated hardware is preferred.

Collecting Traces for Inference

The first critical component of the NeuroScalar system is an efficient and low-overhead mechanism for collecting instruction traces from a processor running in a production environment. This section defines the composition of these traces, details our proposed hardware solution for capturing them, and addresses the key system-level challenges of integration and security.

Trace and Epoch Definitions

For the NeuroScalar model to predict performance, it requires a stream of microar-chitecture-independent features for each instruction. Our trace format is composed of six fundamental and readily available signals: the **Program Counter** (**PC**), the **full memory address** for load/store operations, the **instruction opcode class**, and the register identifiers for **source register 1**, **source register 2**, and the **destination register**.

To manage data flow and enable batched processing, we group instructions into fixed-size units called *epochs*. An epoch is defined as a contiguous block of **100,000 instructions**. Traces are collected for an entire epoch and stored in a buffer; then it is consumed by the inference engine, which allows for significant weight reuse and improved computational efficiency. The epoch size is a tunable parameter largely determined by feature dedicated memory capacity.

While an epoch has no specific semantic meaning on its own, it is useful to understand the diversity of the code being executed. To this end, we compute a signature for each captured epoch by hashing its sequence of PCs. This signature allows an **Enterprise Forecaster** to analyze workload coverage, observe how different application inputs affect execution paths, and ensure a representative sample is collected over an application's runtime.

Trace Collection Hardware and System Integration

While existing technologies like binary instrumentation or hardware tracing mechanisms (e.g., Intel Processor Trace) can provide this data, they typically incur prohibitive performance overheads, making them unsuitable for a transparent, in-the-field deployment.

Therefore, we propose a simple and minimally invasive hardware **trace collector** (See Figure 3.4). Our design consists of a small FIFO buffer attached to the processor's Re-Order Buffer (ROB). As instructions retire, the six required feature signals—which are readily available at this stage in a modern out-of-order processor—are written into the FIFO. To ensure proper operation in a multitasking environments, the trace collector is managed by the operating system. Its activation is tied to a process ID, ensuring that tracing is automatically paused during context switches and only resumes when the target application is scheduled, thus preventing contamination of the trace from other processes or the kernel.

The FIFO's output is mapped to system memory, allowing it to be drained through the conventional memory hierarchy. A FIFO with 512 entries, with each entry storing features for 5 instructions, results in a total on-chip trace buffer of just **12KB**. The memory storage needed for a 100,000 instruction-long epoch is 2.5MB. We summarize below a complete Power-Performance-Area (PPA) analysis:

- **Area:** The area overhead is a negligible 12KB for the FIFO. We explicitly avoid a large, multi-megabyte dedicated SRAM, which would be prohibitively expensive.
- **Performance:** The processor front-end is stalled only if the buffer becomes full. This is a rare event, as modern L2 cache subsystems are well-equipped to absorb such write streams.
- **Power:** Due to its simple logic and the fact that it is only active during sparsely sampled epochs, the power consumption of the trace collector is negligible.

By writing the trace data to OS-managed system memory, our solution is not only efficient but also simplifies the data pipeline, as the trace becomes directly accessible by the GPU or our custom accelerator for inference.

Security and Privacy Considerations A trace collection mechanism that captures instruction and memory addresses is inherently sensitive. For the **Enterprise Forecaster**, trace confidentiality is paramount, and for **Ecosystem Partners**, data privacy is a primary concern. Our design addresses this by treating the trace buffer as a protected memory region, accessible only by a trusted driver and the inference engine. For heightened security, the trace data can be encrypted on-the-fly by the collector hardware before being written to system memory. This ensures trace opaqueness even to a compromised kernel. Furthermore, we emphasize that only metadata about the execution is captured; at no point is the raw data from memory or registers ever exposed in the trace.

High-Speed Inference on Commodity GPUs

A primary deployment target for NeuroScalar is the commodity GPU owing to its ubiquity and its mature parallel programming ecosystem. We detail our software-based inference engine, its performance characteristics, and the methodology for achieving transparent, low-overhead execution.

Our implementation is intentionally straightforward. The trace data for a sampled epoch, residing in OS-managed memory, is transferred to the GPU via standard user-space driver calls. The NeuroScalar model is then executed using a conventional deep learning inference framework. The only optimization we use is the quantization of model weights to **FP16**, which halves the memory footprint and bandwidth requirements without any measurable impact on prediction accuracy for this task.

The performance of our model on a GPU is dictated by the deliberate lightweight design. Table 3.2 shows measurements for inference speed of 1000 epochs on six different

GPUs, measured in inferred instructions per second. As detailed earlier, NeuroScalar is an LSTM-based model with a hidden dimension of only 256. This means that the core computation is a series of relatively small matrix-matrix multiplications (GEMMs). For an epoch of 100,000 instructions, processed with a batch size of 100, the key operation is a matrix multiplication of roughly (100x1024) $\times$ (1024x1024) (applying a single matrix multiplication for the four f, g, i, o LSTM gates [60]). As shown by prior work analyzing GPU kernel efficiency, such as [36], GEMM operations with these dimensions are too small to saturate the memory bandwidth or fully utilize the TensorCores of a modern desktop GPU, sustaining only about 4% of theoretical peak FLOPs.

However, this underutilization is a consequence of our goal of a lightweight model, and the resultant absolute performance is more than sufficient for our needs. On a commodity NVIDIA 4090 GPU, we achieve a simulation speed of **4 to 5 million instructions per second (MIPS)**. To translate this to user-felt overhead, we perform the following analysis:

- Time to run GPU inference on one epoch (100,000 instructions) is: $100,000$ inst$/4,000,000$ inst/sec $= 0.025$s.
- To maintain a system overhead of 0.1%, the CPU must be allowed to execute for $0.025\text{s}/0.001 = 25s$ in the time it takes to process one epoch.
- Assume a host CPU executes instructions at a rate of 3,000 MIPS (3 GHz, IPC=1).
- This means a sampling rate of one epoch for every ~0.75 million epochs executed, which aligns with our system goals.

Finally, to ensure true transparency, the inference engine is designed to *scavenge* GPU resources. The inference kernels are launched at a low OS priority, ensuring that any user-facing, latency-sensitive applications (e.g., gaming, UI rendering) are always given precedence. This prevents our background processing from introducing any noticeable stutter or lag. Once inference is complete, the resulting cycle predictions are either stored

Table 3.2: GPU inference speed (instructions/second).

GPU	Mean	Std. Dev.
H100	4,846,724	522,613
A100	4,420,457	340,531
L40	5,309,333	294,540
L40S	5,793,294	287,928
RTX 4090	5,916,285	362,703
RTX 5000	4,233,920	174,764

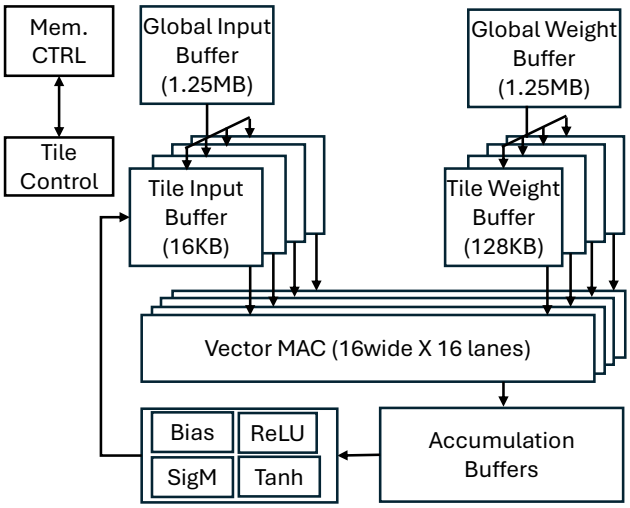


Figure 3.4: Neutrino Inference Accelerator.

locally for the **Enterprise Forecaster's** internal use or aggregated and sent back to the chip designer. Whether the DL weights might leak some information of the microarchitecture is a valid concern. The DL weights themselves can be protected by encrypting and decrypting before use in a TEE environment and secure attestation modern GPUs provide. Finally, simply the availability of performance projection, might allow microbenchmark-based reverse engineering: to protect from this, the system stack can be designed such that the inference outputs are not visible to users.

High-Speed Inference with an On-Chip Accelerator

While commodity GPUs offer a flexible and high-performance inference platform, they are not always available, particularly in power-constrained environments. To provide a

dedicated, ultra-low-power alternative, we co-designed a hardware accelerator, dubbed Neutrino, specifically tailored to the NeuroScalar DL model. This section describes the accelerator's architecture, its dataflow, and the performance benefits derived from this holistic design approach.

Accelerator Architecture and Dataflow

The design of Neutrino is informed by the principles of successful deep learning hardware like NVDLA [107], MAGNET [141], and Eyeriss [25]. The core of the accelerator is a 256-wide **INT8 vector engine**, structured as 16x16 lanes as shown in Figure 3.4. This is supported by a hierarchical memory system designed to eliminate off-chip memory access during inference. An epoch's entire feature trace (quantized to 1MB) is first loaded into a **1.25MB on-chip SRAM buffer** from CPU memory. Weights are loaded once at boot-time into the global weight buffer of **1.25MB buffer**. From there, weights and activations are staged into smaller, local buffers: a **128KB weight buffer** and a **16KB input buffer** directly feeding the vector engine.

This on-chip memory hierarchy allows the accelerator's execution to be completely **statically scheduled**. Once an epoch's input features are loaded into the main SRAM buffer, the entire inference process proceeds without any stalls for DRAM access, enabling predictable, high-performance execution. The dataflow consists of staging weights from the larger weight buffer into the local accelerator buffers, performing vector operations, and accumulating partial sums.

Co-Design for Maximum Utilization

The novelty of this design lies in the tight **co-design of the DL model, the accelerator's microarchitecture, and the execution dataflow**. The model's hidden dimension of 256 was deliberately chosen to perfectly match the 256-wide vector engine. The dominant computation in our LSTM model is the vector-matrix multiplication required for the input,

	1 Tile	8 Tiles
Total Area	2.04mm^2	3.15mm^2
Tile Area	0.16 mm^2	1.26mm^2
Power	28mW	226 mW
Speed	0.02 million inst/sec	0.157 million inst/sec
Sampling rate	1/152642	1/19080

Table 3.3: Accelerator Metrics. Total area includes area of the Global Input Buffer and Global Weight Buffer large SRAMs.

output, forget, and cell gates. For a single instruction trace, this becomes a $(1, 256) \times (256, 256)$ operation, which is executed on the 256-wide vector unit in 256 cycles. This precise matching allows the vector lanes to achieve nearly 100% utilization during these matrix operations. The application of the bias, and non-linear sigmoid and tanh activation functions, which are orders of magnitude less computationally intensive, are temporally pipelined after the MATMUL. A single LSTM layer finishes in 8264 cycles, achieving 99% MAC utilization.

Performance and Efficiency

To verify the design and obtain accurate performance and power figures, we developed a cycle-level simulator for the accelerator and a simple compiler to map the LSTM operations onto its instruction set. The simulator's performance was validated against a full RTL implementation of the accelerator, synthesized using a commercial 16nm P&R flow. Results are in Table 3.3, for a single-tile and an 8-tile design.

Our analysis shows that Neutrino can process traces at a rate of **0.02 million instructions per second** (**MIPS**) while consuming only **28 milliwatts** of power. This represents a nearly **85× improvement in energy efficiency** (inferences/sec/watt) compared to the NVIDIA 4090 GPU solution. This result demonstrates the main benefit of our co-design approach: by tailoring the hardware and software together, we can create a solution that is orders of magnitude more efficient than a general-purpose programmable one. This design can be trivially scaled-out in an 8-tile design, each tile processing a different batch, with no

inter-tile communication, while sharing of the global input buffer SRAM and weights SRAM.

Collecting Traces For Training

The foundation of NeuroScalar is a model trained on data generated from a trusted, high-fidelity simulator. This section briefly outlines the process of collecting this training data.

The training process requires paired examples of key properties and output labels. We instrument a conventional cycle-level simulator—in our case, **Gem5**—to generate these pairs. For each instruction executed in a simulation, we collect the six key properties defined in Section 4.1 (PC, memory address, etc.). It is critical that the properties extracted from the simulator are **semantically identical** to those captured by the hardware tracer to prevent any train-serve skew.

The corresponding ground-truth label for each instruction is its **retirement latency**: the number of cycles elapsed since the previous instruction retired. This cycle count is a direct output of the simulator's detailed pipeline model. To create a robust and generalizable foundation model, we generate our training corpus by running a diverse set of workloads, including 16 applications from the SPEC CPU 2017 benchmark suite, through our instrumented simulator.

While our core methodology relies on microarchitecture-independent features to enable in-the-field deployment on existing hardware, the framework is extensible. For scenarios where a designer wishes to model features of a new ISA or a specific hardware mechanism, those microarchitecture-dependent features could be added to the training input to potentially increase model accuracy further.

Deployment and Model Management

A successful in-the-field simulation system requires more than just hardware and models; it needs a robust infrastructure for software control, model distribution, and data

reporting. This section outlines the complete deployment and management framework for NeuroScalar.

Software Interface and Control

End-users interact with the NeuroScalar system through a dedicated driver and a user-space API. This interface provides the necessary control to manage the tracing and inference process, critical for the **Enterprise Forecaster** user. The driver is responsible for configuring the hardware trace collector, managing the secure memory buffer, and scheduling the inference tasks on the GPU or accelerator, as described in Section 4.2. It ensures that tracing is properly synchronized with the OS scheduler to handle context switches and maintain process isolation. If application slowdown is permitted, sampling frequency can be tuned as well.

Model Distribution and Updating

The chip designer makes NeuroScalar models, each representing a different microarchitectural design, available through a secure **model repository**. When a user requests a specific model, it is downloaded, cryptographically verified, and stored locally. This repository-based approach allows for seamless updates and version control. If a designer retrains a model with more data to improve its accuracy, end-users can be prompted to update to the latest version, ensuring they are always working with the most current and accurate performance predictions.

Telemetry and Reporting Framework

For the **Ecosystem Partner** use case, a secure and privacy-preserving telemetry pipeline is essential for returning performance data to the chip designer. The process is designed to be fully transparent and opt-in. When inference for a sampled epoch is complete, the result (a predicted IPC or cycle count) is aggregated locally. Periodically, this aggregated,

anonymized data—stripped of any personally identifiable information—is sent back to a central collection server managed by the chip designer. The data format includes the model version, the application signature (SHA256 hash of PCs), and the performance prediction. This allows designers to build a large-scale, real-world dataset of how their architectural ideas perform across thousands of applications and usage scenarios.

This ecosystem can extend existing logging frameworks like those developed by hyperscalars such as Google's OpenTelemetry [55] and Meta's Dynolog [30], or commercial solutions from providers like Datadog [35] and Splunk [136]. While these platforms are typically geared towards software and infrastructure monitoring, our hardware-derived performance predictions could be integrated as a novel telemetry source, providing chip designers with direct, in-the-field feedback through established and trusted data pipelines. NVIDIA's GeForce Telemetry [106], Intel's Platform Monitoring Technology (PMT) and Telemetry Interface Specification (TIS) allow telemetry data to be collected already with a full-fledged system infrastrcture in place - performance projection is simply another source of telemetry data.

3.4 Experimental Methodology

Model construction To construct the DL model we used the GEM5 simulator which was annotated to produce the feature trace and the ground truth retirement cycle count for each instruction. We did this on 16 SPEC2017 benchmarks. We fast-forwarded by 2 billion instructions and created a dataset comprising of 100 million instructions for each application. We also studied four additional configurations besides our baseline 8wconfiguration. The details of the configurations are shown in Table 3.4. For inference testing, 10 million instructions from this dataset are set aside and never used for training. The model training and experiments were done on a mix of A100, L40, L40S, and A10. The final unified model trained on the 8wconfiguration, across all 16 benchmarks took a total

Table 3.4: Microarchitectural parameters for the five processor configurations evaluated. The baseline is an 8-wide out-of-order processor. L2 cache is 8MB across the board. Variations explore different trade-offs in memory and core resources.

Config Abbr.	**Base 8-wide** (*8w*)	**6-wide+ LS Unit** (*6w+ls*)	**Large ROB** (*rob*)	**Large LSQ** (*lsq*)	**More Memory** (*4w+mem*)
Width	8	6	8	8	4
LS Units	1	2	1	2	2
LSQ Entries	32	32	32	64	32
Num Regs	256	256	512	256	256
ROB Size	192	192	384	192	192
L1D$ Size	64KB	64KB	64KB	64KB	128KB
L1I$ Size	64KB	64KB	64KB	64KB	64KB

of 8 hours to train on an A100.

Inference To study inference we run the model on a family of commodity GPUs and measure speed (inferences/second). The platforms we considered are A100, H100, A6000, 4090, and an RTX5000. Our accelerator was implemented in RTL and synthesized with a full P&R flow at 16nm. Recall that our accelerator implements a complete static schedule, and its execution is deterministic in the number of cycles for an epoch (with any potential delays only arising from the time to transfer from the processor's memory into the accelerator's Global Input Buffer SRAM). The accelerator's main results are summarized in Table 3.3.

Downstream tasks We examine two downstream tasks for a chip designer as case studies. Starting from an 8w processor, the chip designer seeks to understand how performance changes for 4 other configurations. In particular, they seek to understand at the instruction level. We report two studies where the 5 processors are ranked for each retiring instruction for each benchmark. Second, we present pairwise comparison for each benchmark for each instruction. This allows extremely low-level analysis on in-the-field workloads. Both of these push the model's accuracy and fidelity to the limit - the changes between the processor configurations is intentionally set to subtle features like LSQ size, # ports etc.

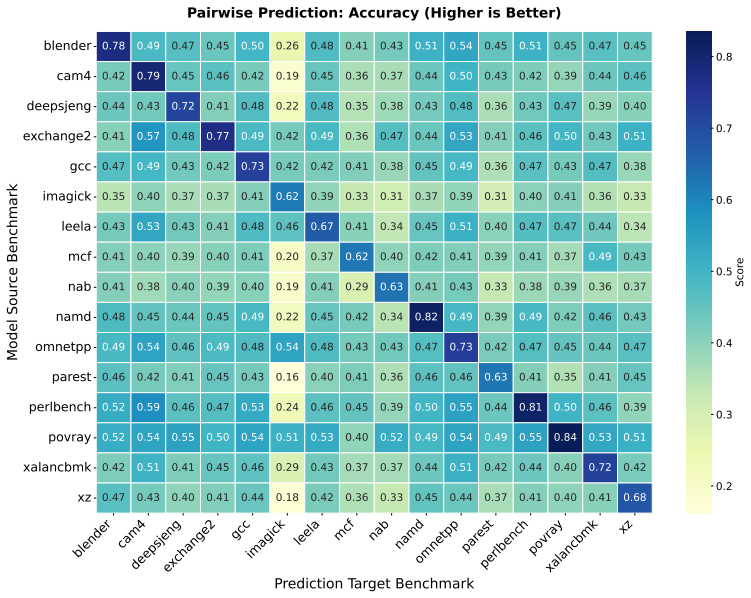


Figure 3.5: Pairwise prediction **Acc(round)**

3.5 Results

First, we show that individual benchmark models work well and can transfer to other models. Second, a model is examined that uses information from every benchmark. Finally, we test the accuracy in downstream tasks like pairwise ordering and best-configuration.

Benchmark-wise Error and Accuracy Heatmaps

We visualize cross-benchmark behavior with two heatmaps. In both plots, *rows* denote the **source** benchmark used to train/fit the predictor and *columns* denote the **target** benchmark on which we evaluate. Each cell is annotated with the corresponding score for that (source, target) pair.

Prediction accuracy. Figure 3.5 shows *Acc*(*round*) results. In-domain accuracies exceed 70% (often 80%+), while cross-dataset evaluations remain around 50%, demonstrating reasonable generalization. Overall, $\sim$ 80% of instructions are predicted exactly and over 95% within ± 1 cycle, with a detailed error figure deferred to Appendix A.1.

Table 3.5: Accuracy-centric evaluation of the foundation model (**TraceFusion-13**)

Benchmark	$\pm 1 \uparrow$	$\pm 2 \uparrow$	Acc(round) (ref)$\uparrow$
`xz`	0.8388	0.9159	0.6782
`gcc`	0.8607	0.9378	0.7174
`xalancbmk`	0.8739	0.9415	0.7231
`nab`	0.7802	0.8947	0.6232
`deepsjeng`	0.8579	0.9430	0.7102
`parest`	0.8002	0.8999	0.6246
`namd`	0.9391	0.9725	0.8116
`povray`	0.9545	0.9821	0.8318

Foundation Model on the Concatenated Corpus

We train a single backbone on the union of all benchmark splits (**TraceFusion-13**) and evaluate it on held-out data. The model achieves consistently strong accuracy across diverse workloads, generalizing well to both short-latency and long-tail benchmarks. Table 3.5 summarizes representative results, with full details in Appendix A.1.

Summary: Neuroscalar is highly accurate and at fine-granularity of individual instructions.

Downstream Tasks: Processor-Config Comparisons

Pairwise ordering accuracy For each configuration pair $(i \leqslant j)$, we evaluate whether the predicted ordering preserves the ground-truth ordering. Table 3.6 reports aggregated statistics across all benchmarks. In addition to the overall correct/total counts dubbed match rate, we also include the fraction of cases where the ground truth indicates i is strictly better than j (`GT-better`) and the fraction of samples with non-zero ground-truth cycles (`Non-zero`). These richer statistics provide a more complete picture of pairwise ordering fidelity, showing that pairwise orderings are preserved at roughly 90% rates depending on the configuration pair. The 3rd column shows the "irregular" nature of OOO processors: comparing 4w+mem$\leqslant$ 8w, for 15% of instructions, the 4w+mem processors retires earlier than the 8w. Neuroscalar is able to learn and capture this behavior as well.

Table 3.6: Pairwise ordering results aggregated across benchmarks. Columns: match rate, proportion of ground-truth cases where the left config is strictly better (`GT-better`), and proportion of non-zero ground-truth samples (`Non-zero`).

Pair			Match rate	GT-better	Non-zero
4w+mem	$\leqslant$	8w	91.11%	15.0%	39.2%
4w+mem	$\leqslant$	rob	91.08%	15.0%	39.3%
4w+mem	$\leqslant$	lsq	91.06%	15.0%	39.4%
4w+mem	$\leqslant$	6w+ls	91.04%	15.1%	39.5%
8w	$\leqslant$	rob	91.06%	15.0%	39.4%
8w	$\leqslant$	lsq	91.08%	15.0%	39.3%
8w	$\leqslant$	6w+ls	91.11%	15.0%	39.3%
rob	$\leqslant$	lsq	91.14%	15.0%	39.2%
rob	$\leqslant$	6w+ls	91.17%	15.0%	39.2%
lsq	$\leqslant$	6w+ls	91.20%	15.0%	39.1%

Table 3.7: Five-config ranking per benchmark. Metrics over held-out instructions: Kendall τ (higher is better), full permutation match, and best-config match.

Benchmark	**Kendall τ**	**Full Match(%)**	**Best Match(%)**
xz	0.8657	56.93	96.09
gcc	0.8449	52.31	96.24
xalancbmk	0.8036	48.35	96.46
nab	0.8188	50.88	94.67
deepsjeng	0.8360	53.87	96.21
parest	0.7741	43.92	94.24
namd	0.9150	64.45	98.19
povray	0.9355	64.20	98.61

Five-config ranking: full-rank match, best-config match, and Kendall τ For each instruction, we rank five configs by rounded cycles and report: (i) full permutation match, (ii) whether the best (lowest-cycle) config is matched, and (iii) Kendall τ over all $\binom{5}{2}$ pairs with ties handled by group ranks. Representative end-of-run summaries is show in Table 3.7, where Kendall τ typically falls between 0.8 and 0.95, and best-config match consistently exceeds 94%.

Summary: Neuroscalar can extract detailed per-instruction differences between different microarchitectures accurately.

3.6 Related Work

Prior work in applying machine learning to performance prediction has critical limitations for in-the-field deployment. Early efforts like Ithemal [93] focused on predicting the latency of static basic blocks, but by ignoring dynamic execution phenomena like memory access and branch prediction outcomes, they cannot model realistic, full-program behavior. More recent full-program simulators fall into two categories, neither of which is suitable for our goals. SimNet [80] achieves accuracy by using microarchitectural "context" features—such as which level of the cache hierarchy served a request. This approach is fundamentally incompatible with in-the-field prediction, as these features are the *output* of a specific microarchitecture and cannot be collected from production hardware for a hypothetical design. Conversely, TAO [115] builds a large, monolithic model to explore a predefined space of *basic* parameters (e.g., ROB size). This makes it too computationally heavyweight for lightweight, sampling-based deployment and, more importantly, too inflexible to evaluate the novel, complex architectural ideas (like a new prefetcher) that designers need to test. *In summary, previous methods are either too limited in scope, depend on features unavailable on production hardware, or are too monolithic to evaluate novel design trade-offs.* NeuroScalar provides a lightweight, extensible framework specifically designed to fill this gap.

ML-based performance models. ML-based performance models apply/use ideas from linear regression and interpolation, to extrapolate performance of new microarchitecture from sampling the space. They lack the ability in general for fine-grained microarchitecture policies and typically fail to accurately model the run-time complex dynamic interaction between the program and hardware [79, 158, 62, 64, 65, 74, 75]. ML models have also been developed for application performance prediction across ISAs and machine architecture [158, 159, 8, 11, 112]. Concorde is a state-of-art hybrid ML/analytical model [99] - for in-the-field simulation it exposes too much processor information.

3.7 Conclusion

This chapter studies a fundamental bottleneck in computer architecture research: the inability to evaluate novel microarchitectural ideas with both high fidelity and high speed on real-world, "in-the-field" workloads. Traditional cycle-accurate simulation is too slow, and existing ML-based approaches are either incompatible with in-field deployment or too high-level to capture the detailed design trade-offs that matter to practitioners. Our solution is a deep learning framework built on the key insight of using **microarchitecture-independent features** to train an ultra-lightweight DL model. This approach decouples performance prediction from the underlying hardware, allowing a model to run on existing silicon, extract features of a workload, and accurately forecast it's performance on a hypothetical design. We presented a complete system, including a lightweight hardware tracer, a sampling methodology, commodity GPU-based inference and an ultra-low-power accelerator, that performs this analysis with negligible overhead. Our evaluation shows that this framework can differentiate between competing designs with **95% accuracy** in A/B testing scenarios, providing architects with a powerful new tool for data-driven design. This chapter paves the way for a future where hardware design cycles are dramatically accelerated through large-scale, continuous feedback from live user workloads.

4 SAHM: STATE-AWARE HETEROGENEOUS MULTICORE FOR SINGLE-THREAD PERFORMANCE

Improving single-thread performance remains a critical and ongoing challenge in modern processor design. Although multicore scaling and accelerators have dominated recent advances in throughput, the performance of individual threads—often measured as Instructions Per Cycle (IPC)—remains vital for latency-sensitive workloads, legacy software, and many user-facing applications. Traditional approaches to boost IPC have focused on deepening speculation, widening pipelines, and complex out-of-order execution; however, these techniques face diminishing returns in the face of increasing complexity and energy constraints. Research ideas abound on various ways to increase IPC with better prefetch/caching for targeted program behaviors [105, 46, 100, 125, 70, 122, 148, 134].

Recent product trends illustrate this plateau: for example, AMD's Ryzen CPU generations have shown incremental single-thread performance improvements of roughly 5–10% per generation, with much of the performance gain coming from improved clock speed, not fundamentally better per-cycle efficiency. These marginal gains highlight the need for new architectural strategies that go beyond conventional microarchitectural tuning. This chapter proposes such a strategy to directly tackle the problem of single-thread performance.

Our approach is grounded in a set of key empirical observations. First, we find that applications exhibit substantial behavioral heterogeneity across time. This observation was seen time and time again as we developed Introspection Processing Units (IPUs) (see Chapter 2); it is even the topic of one of the IPU case studies: Obfuscated Software where GPU utilization histograms are produced and found to stress a single component at a time. We choose to focus on single-threaded CPU applications for this chapter due to their simpler nature. In particular, they vary widely in how they stress four key microarchitectural components: **the branch predictor, L1 data cache, L1 instruction cache, and L2 cache.** By collecting real hardware performance counter data from a state-of-the-art CPU across

a wide range of applications, we demonstrate that this behavioral diversity is not only common, but pronounced. This constitutes the first contribution of the chapter: an empirical characterization of fine-grained behavioral variability in modern workloads.

Second, we show that applications do not remain fixed in one behavioral mode. Instead, they transition between modes—or what we call **behavioral states**—throughout execution. We define a behavioral state as a 4-bit encoding derived from four key performance metrics, each binarized into HIGH or LOW categories based on observed distribution thresholds. This results in a total of 16 unique behavioral states. Different applications not only visit different subsets of these states, but also dwell in them for variable intervals and transition between them at different rates. These findings suggest that single-threaded performance could be enhanced by adapting the underlying hardware to the current behavioral state.

This observation leads to our third and central contribution: the design of a new type of processor architecture that we call **SAHM**—*State-Aware Heterogeneous Multicore*. In SAHM, we construct a multicore processor where each core is specialized for one or more of the 16 behavioral states. Rather than attempting to build a one-size-fits-all core that includes all possible optimizations (e.g., a deep branch predictor, aggressive prefetchers, large instruction and data caches), we partition these features across cores based on their relevance to specific states. At runtime, we monitor the application's state and migrate it to the core best suited for the current behavior. This strategy enables composability of microarchitectural enhancements without incurring the area, power, and *complexity penalties of incorporating all features into a single core*. The key design questions then become: how to detect the current state, when to trigger migration, and how to manage the cost of migration.

We further extend this idea to a realistic multi-programmed scenario, which addresses concerns about core underutilization. In a modern cloud environment, it is common to co-locate multiple independent workloads on the same processor. In such settings, SAHM's specialized cores are always busy, as different workloads occupy different behavioral states

Aspect	big.LITTLE	SAHM
Core Design Goal	Energy-efficiency vs. Performance	Performance specialization by behavior
Granularity	Coarse (application phase)	Fine (100ms epoch)
State Awareness	None or static hints	Performance counter-driven
Migration Trigger	OS heuristics	Runtime behavioral state detection
Core Utilization	Some idle in steady state	Fully utilized in multiprogrammed workloads

Table 4.1: Comparison of SAHM with big.LITTLE

at different times. We show how an intelligent runtime scheduler can orchestrate the migration of threads across cores to maximize system-wide performance, without leaving specialized cores idle.

We also recognize practical considerations such as virtualization, cloud isolation, and security. SAHM is compatible with modern virtualization frameworks and can be integrated into hypervisors or OS-level schedulers. Migration decisions can be made within tenant boundaries to maintain isolation guarantees. Furthermore, performance counters used for state detection are already widely virtualized in commodity processors, and migration policies can be designed to respect container and VM boundaries.

Relationship to Prior Work. SAHM differs fundamentally from prior heterogeneous processor designs such as ARM's big.LITTLE and related single-ISA heterogeneous multicore architectures. big.LITTLE focuses on trading performance for energy efficiency: LITTLE cores are deliberately simplified to save power, while big cores are designed for high performance. In contrast, SAHM is designed to *improve* performance across the board by targeting microarchitectural behaviors rather than coarse-grained application phases. Table 4.1 summarizes the key differences.

In summary, this chapter makes the following contributions:

- We introduce a new behavioral state taxonomy derived from performance counter measurements and show that applications exhibit diverse and time-varying microarchitectural demands. Section 4.1 and 4.2

- We propose SAHM, a heterogeneous multicore architecture in which each core is

specialized for one or more behavioral states, and threads migrate at runtime to the best-fit core. Section 4.3

- We analyze current state of the art of the components within our study and the opportunity for speed up with a SAHM system. Section 4.4
- We evaluate SAHM both for single-threaded applications and for multi-programmed systems, showing substantial performance benefits under realistic assumptions. Section 4.6
- We discuss implementation considerations, including migration policies, prediction strategies, and system-level integration.

4.1 Overview and Motivation of Heterogeneity of Applications

In this section we present motivation for state awareness and an overview of a state aware heterogeneous multicore system.

Understanding and Quantifying the Opportunity

To understand the opportunity for improving single-thread performance through microarchitectural specialization, we begin by studying how modern applications behave at runtime. Our investigation is motivated by the hypothesis that even standard benchmark applications—such as those from the SPEC suite—*impute different demands on the processor during different parts of execution*. If these demands can be captured and used to inform hardware behavior, we can potentially design architectures that are dynamically tailored to the application's needs.

We use performance counters available on modern CPUs to measure four key microarchitectural metrics: Using modern tools (e.g., Linux `perf`, `likwid`[138]), we collect all four

Metric	PMC Values
Branch Misprediction Ratio	Branch Misprediction Count / Branch Instruction Count
L1I Cache Miss Rate	L1I Cache Miss Count / Instruction Count[1]
L1D Cache Miss Ratio	L1D Cache Miss Count / L1D Cache Access Count
L2 Cache Miss Ratio	L2 Cache Miss Count / L2 Cache Access Count

Table 4.2: Performance monitoring metrics and the values used to calculate them. [1]This is a non-programmable counter.

metrics simultaneously with negligible overhead (~0.5% application slowdown), enabling fine-grained tracking without affecting program behavior.

To structure this data, we classify each metric as either HIGH or LOW, based on cutoffs derived intuitively or from the empirical distribution (e.g., below the 50th percentile is LOW, above the 50th is HIGH). This results in a 4-bit encoding of the application's current behavior, defining one of 16 possible **behavioral states**. This state space is determined by static cut-offs unlike related work that uses performance metric stability during execution to define application phases. We sample the application at regular 100ms intervals—*epochs*—and label each epoch with its corresponding behavioral state. Thus, an application's execution trace can be viewed as a time series over this 16-state space.

We analyze the behavioral state traces (details in Section 3) of a variety of single-threaded applications and make several key observations:

(**a**) **State Coverage Across Applications.** Different applications occupy different subsets of the state space. For each application, we plot the percentage of time spent in each of the 16 states, showing that state distribution is highly non-uniform and workload-specific. Some applications are dominated by branch-heavy phases, while others exhibit memory intensity or front-end bottlenecks.

(**b**) **State Transition Dynamics.** Applications do not remain in a single state but transition across states at different frequencies. Using per-application heatmaps of state-

to-state transitions, we observe a variety of transition behaviors—some applications flip rapidly between states, while others maintain longer intervals within a dominant state.

(**c**) **Interval Length Variability.** We define a state interval as a consecutive sequence of epochs during which the application remains in the same behavioral state. We compute the average interval length per state for each application, showing that some states are stable over hundreds of milliseconds, making them strong candidates for state-specific optimization.

(**d**) **Opportunity for Specialization.** For each state, we analyze how much time each application spends in it. This cross-application view reveals three insights:

- No single state dominates execution across all applications. This reinforces the idea that different workloads stress different microarchitectural components.
- State 0—the ideal state where all four metrics are LOW—accounts for only 8% of runtime on average. This highlights the potential for performance gain: even modest speedups in the remaining states can lead to meaningful improvements.
- For instance, if 30% speedup is possible in the non-state-0 regions, overall application speedup could approach 22%—a significant figure in the context of single-thread IPC improvements.

Each of these findings is expanded to its own subsection at the end of this section. These findings strongly motivate the idea that instead of designing one general-purpose core to handle all scenarios, we should build multiple specialized cores, each tuned to perform well for a subset of the behavioral states. SAHM implements this vision through a practical and scalable hardware-software co-design approach.

SAHM overview

There are two central ideas to systems using State Aware Heterogeneous Multicore: 1) A CMP with diverse specialized cores and 2) A performance monitoring mechanism within

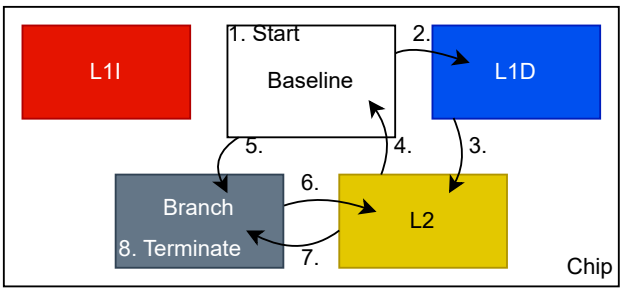


Figure 4.1: The canonical configuration of a SAHM system. Each core except the baseline is specialized for the listed component. An example program migration pattern is included.

the scheduler.

Point 1 is that a each core allocates more area and power to one component - different components for each core. Figure 4.1 depicts a canonical configuration in which four of the five cores have been specialized to a different component each. Due to the higher allocation, more complex designs are possible providing speed up versus the baseline component. Yet, this speed up is attainable only if the program is placed on the core matching the stressed component.

Thus, point 2 is a scheduler that uses performance monitoring to influence which program-core assignment. The scheduler first gathers the characteristic metrics from the previous OS timestep. Next, it decides based on the metrics if the program should migrate. If so, then migration occurs to a suitable core if load-balancing conditions are met. Finally, the scheduler chooses a program from the core-local queue and launches it. Figure 4.1 also shows an example program migrating between cores over time.

Alleviating Concerns

In proposing SAHM, it is important to address several potential concerns or misconceptions that readers may have:

Isn't this just adding complexity for no gain? Not at all. The goal is to reduce system-wide complexity by avoiding the integration of all expensive features (e.g., large branch predictors, deep prefetchers, massive caches) into every core. By distributing these features

across specialized cores and migrating applications to the best-fit core based on current behavior, we achieve better performance-per-area and performance-per-watt.

Why not just make all cores smarter? Adding every optimization into every core is cost-prohibitive in terms of area, power, and verification effort. Worse, many optimizations conflict: an aggressive data prefetcher may hurt icache locality, or a large predictor may increase access latency. SAHM allows for decoupling and composability.

Aren't migrations expensive? Migration is rare and coarse-grained (every 100ms epoch), and we assume a modest 5ms cost. Our results show that state intervals are often long enough to amortize this cost, and even reactive migration policies are effective.

How is this different from big.LITTLE? big.LITTLE trades performance for energy efficiency using static cores. SAHM uses all high-performance cores, each tuned for different runtime behavior. Migration is driven by microarchitectural signals, not OS policies. The goal is performance, not energy reduction.

What about OS, VMs, or security concerns? SAHM is compatible with virtualization. Modern OSes already support thread migration. Migration policies can be constrained within VM or container boundaries. Performance counters are virtualizable and can be used without breaking isolation.

By proactively addressing these issues, we aim to clarify the feasibility and relevance of the SAHM architecture for real-world deployment.

4.2 Characterization

The section delves into the program characterization including data gathering method, analysis of the collected data, and discussion of the analysis.

Methodological Details

Our measurements are collected on a Golden Cove microarchitecture, representative of recent Intel client-class CPUs. This platform supports concurrent tracking of up to seven programmable performance values using hardware performance counters. To overcome the hardware limitation on simultaneously monitoring more than seven values, modern tools like Linux `perf` use a technique called *multiplexing*, wherein sets of counters are time-sliced during execution. However, in our study, we restrict ourselves to seven key values, see table 4.2, which can all be captured simultaneously without needing multiplexing, thereby ensuring high fidelity of measurement [101, 83, 145, 155]. We measure all of the SPEC CPU 2017 benchmark suite.

We choose a sampling interval of 100ms after empirical calibration. While finer-grained intervals (e.g., 10ms) are possible, we observed that they introduce measurement instability and noise due to context switches, sampling overheads, and limitations of event delivery latency. At 100ms, the counter values are both stable and precise, yielding nearly 100% accuracy as validated through cross-referencing with aggregate counter values and controlled experiments. Some benchmarks execute in less time than 100ms and are excluded from further study. This interval is also coarse enough to amortize migration costs yet fine enough to capture meaningful behavioral variation.

While our experiments are limited to Golden Cove due to platform availability and infrastructure readiness, our methodology is entirely portable. ARM and AMD processors offer comparable performance monitoring capabilities, and the SAHM approach—classifying behavior and specializing hardware accordingly—can be directly adapted to those platforms. We view this study as a proof-of-concept for a broader architectural direction that is applicable across vendors.

Metric	25%	50%	Intuitive
branch mispred.	0.03%	0.34%	1%
L1I miss (MPKI)	0.004	0.009	1
L1D miss	0.5%	0.99%	2%
L2 miss	3.64%	18.47%	10%

Table 4.3: Three candidate cut-offs for metric binning.

State Coverage Across Applications

Goal Show that applications occupy different subsets of the behavioral state space statically determined by cut-offs. Higher variability between applications decreases the likelihood that they stress the same component at the same time when in a workload together. As the process of defining the state space includes binning, which states applications occupy is dependent on the cut-offs that determine the binning.

Experiment First, analyze the gathered traces to determine a few empirical cut-offs and include one set of cut-offs from intuition based on prior architecture experience. These are listed in Table 4.3. The empirical cut-offs are percentiles from the gathered traces - i.e. 50% is the medians, 50th percentile, of each metric.

Then, calculate the portion of time spent in each state on average for each set of cut-offs. Figure 4.2 shows this breakdown. Each state is labeled by the combination of components stressed in that state; for example, the "L2+Branch" state has both the L2 miss ratio and the branch mispredict ratio as HIGH which identifies these components as stressed. One expectation of this state space is mostly mutually exclusive behavior - L2 HIGH makes sense to overlap either L1 cache metrics being HIGH. This is the case for the intuitive cut-offs; however, both of the others have a large portion of highly combined component stresses. The 25% cut-offs have more than half of an average application spent stressing the entire core – this does not reflect reality. Another expectation is some amount of the Low state due to the cache starting cold and paging in larger data sets. This is confirmed as 8% of an average application is spent in the Low state for the intuitive cut-offs.

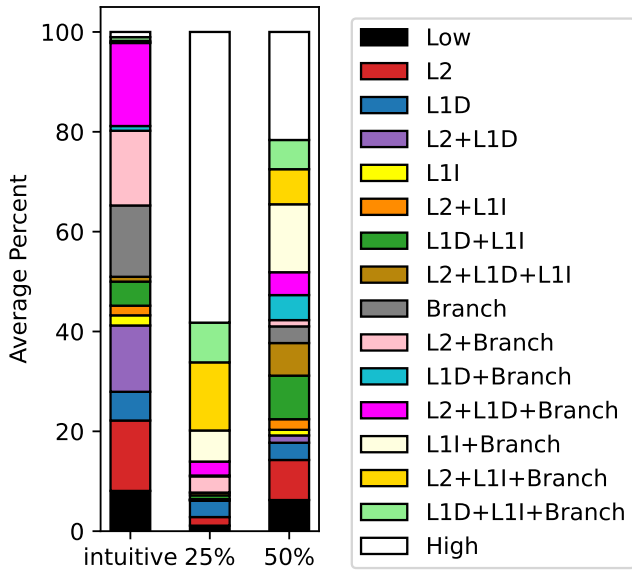


Figure 4.2: The portion of an application that the application is in a state averaged across the SPEC 2017 benchmarks. The intiutive cut-offs captures the most diversity.

Figure 4.3 provides another look at the intuitive column by unstacking the bars. The most occupied states are much easier to see: Low, L2, L2+L1D, Branch, L2+Branch, and L2+L1D+Branch. The overlap between data memory and branch prediction could be that branch prediction is causing cache pollution or that the data brought in from cache misses does not align with the current pattern learned by the branch predictor.

Figure 4.4 breaks the intuitive column into a stack for each benchmark. It is apparent that the programs show a wide range of state occupations. Physics simulations like pop2, roms, and cactuBSSN all utilize a large working set that does not fit into the first level of cache. This is reflected in our results, as each stresses the data memory via L1D cache and L2 cache. Imagick spends the majority of its time in the LOW state - making it a poor candidate for SAHM.

Key takeaways *i) Different applications occupy different states in the statically determined state space. ii) The intuitive cut-offs show the most diverse set of state occupation and more closely match how a core functions. iii) Almost all of the execution time is spent in states where one of the 4 key metrics is HIGH. iv) For the rest of this chapter, the intuitive cut-offs are used to define the state*

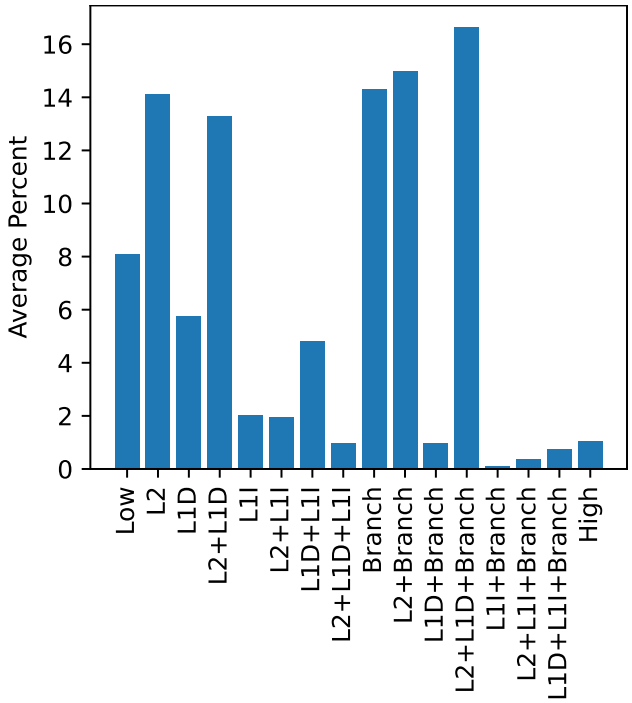


Figure 4.3: The portion of an average application spent in a state with intuitive cut offs. The majority of states are visited.

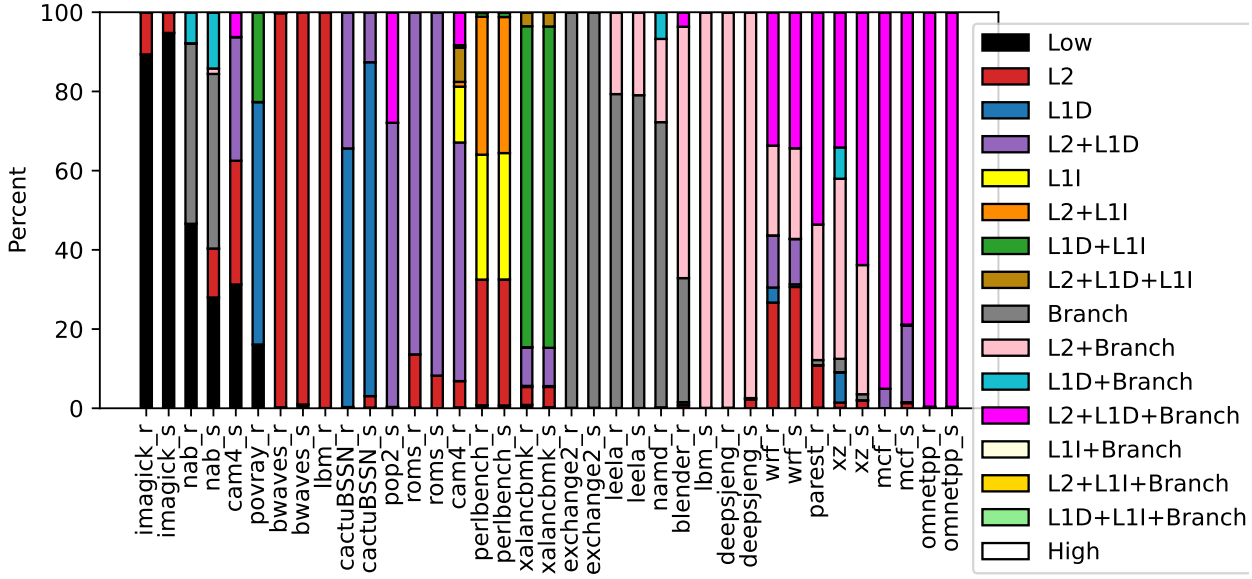


Figure 4.4: The percent of runtime spent in each state by application. The breadth of behaviors stands out.

space.

State Transitions

Goal: We want to understand how long applications stay in a state, when they transition, are transitions highly correlated with the originating state (i.e. out-degree of a state in this graph of transitions), and how application-dependent this behavior is. The more dynamic and application-dependent the more difficult for compilers, software transformations, or static hardware policies to exploit this.

Experiment: Analyze gathered traces for state transitions - for each epoch in a trace, examine the previous epoch and record the edge between states. Count the total for each such edge. Figure 4.5 depicts the heatmap of the average percent of the total number of transitions for each transition. If a transition never occurred during our analysis, then the cell for the transition is white. The diagonal, the state transitions that stay in the same state, has been removed from the figure. Transitions that stay in the same state dominate with 84% of epochs staying in the same state as the previous epoch.

Analysis: First, this confirms the expectation that transitions do occur. Programs usually go through phases and prior works have confirmed this. Next, more used transitions are between states that have more time spent in them. This is beneficial as it means that capturing the states most resided in also captures the most frequently used transitions. Furthermore, the figure is not completely symmetric across the diagonal. This indicates that order that states occur is non-trivial: states do not merely return to Low. More rigorously: the set of states and their likelihood to transition to a given state is different than the set of states and likelihood of transitions from that given state. One example of this is Low to L1I HIGH does not occur yet L1I HIGH to Low does occur.

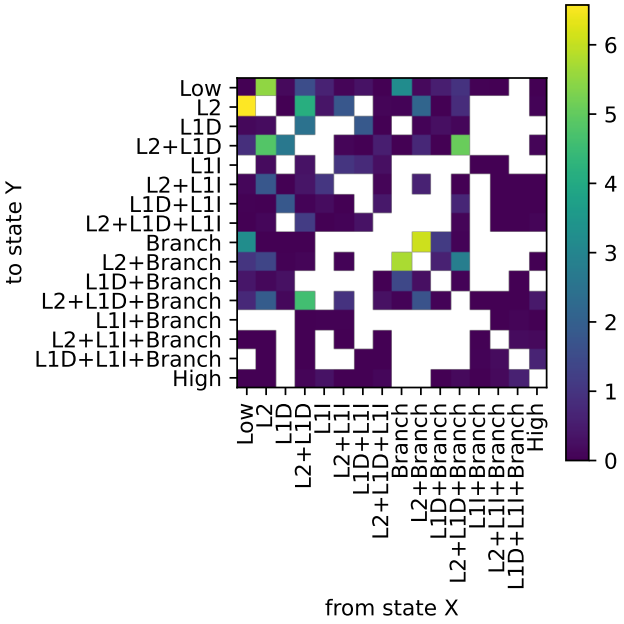


Figure 4.5: The portion of the total number of transitions on average. The diagonal has been removed and white cells indicate transitions that were not seen in our study. There is no overwhelming outliers that should be designed for.

Key findings: *State Transitions occur and are not uniform between applications* (*i.e. how an application enters state L2 HIGH is not always from L1D HIGH*)

Interval Length Variability

Goal: We have seen transitions do occur; what is the length of the intervals between transitions that change state? We define a state interval as a consecutive sequence of epochs during which the application remains in the same behavioral state. Shorter intervals are more difficult to exploit for performance gain - the transient nature means that any adjustment to the state could take so long the state has passed by the time the adjustment is in place.

Experiment: Using the gathered traces, we calculate the interval length through examining the current epoch to the previous epoch like in the transition experiment. Figure 4.6 presents this calculation breaking down both the total count and the total time for an

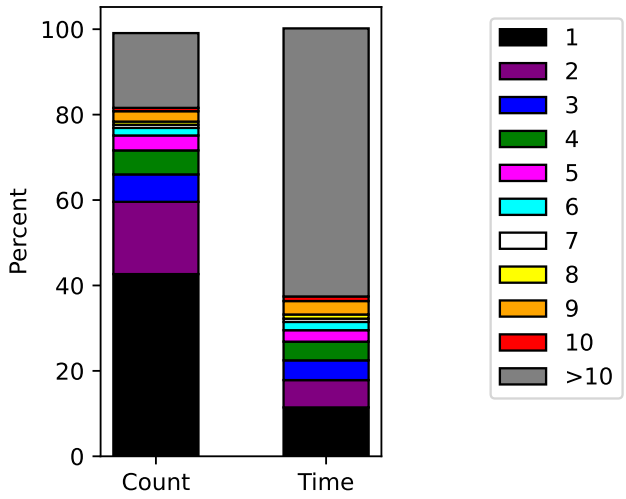


Figure 4.6: Analysis of intervals - strings of the same behavioral state. In count, 1 epoch intervals dominate; on the other hand, long intervals provide ample time to amortize migration overhead.

average application. The extremes cover the majority of the data: The short intervals make up most of the overall number of intervals. On the other hand, long intervals make up the majority of the time spent. This alerts us to a pitfall for exploiting these intervals - a program may ping-pong between two specialized cores if the program is only short intervals. Yet, there is plenty of time to amortize overhead when a long interval is reached. These factors indicate that some sort of inertia, keeping migrated programs on the assigned core for a period of time, should be included in a system that adjusts for state behavior.

Key Findings: *The average interval duration provides ample opportunity to migrate based on the dynamic behavioral state, therefore extracting efficiency and/or speedup.*

Summary

This characterization has shown that even though the state space is determined by the cut-offs, applications occupy a swath of the space. This occupation is highly dependent on the application. Transitions between states are uniformly used and usually occur either immediately or distantly. The long intervals have applications reside in a handful of states

for more than half their runtime. It is clear that specializing for these states will improve performance.

4.3 SAHM Design

This section details the designs required for SAHM. These include the architecture of a heterogeneous multi-core system, the created configuration space, and the scheduling policy.

Single Core Architecture

Current designs focus on two points: designed for performance, big or P cores, or designed for energy efficiency, LITTLE or E cores. Focusing on just performance, the monolith cores are generalists - do everything mediocre. In this era of multicore, this is wasteful. Individual cores can specialize, contributing their slice of the behavioral state space to the CMP as a whole. In a SAHM system, each core attributes most of the area and power budget to one component. The next Section 4.3 provides examples of components that lead to speed up when there is budget allocated for them.

SAHM Architecture

Individual specialized cores by themselves cannot compete with the monolith cores. In this case, the system is greater than the sum of its parts: the specialized cores cover the deficiencies of each other. Figure 4.1 depicts one conservative general purpose CMP. It includes a general core and specialty cores for each component. As a program executes it will be placed onto the core that best suits its current operating needs. A SAHM system does not necessarily have a specialized core for each component; some may be left out so others may be duplicated.

This introduces a new architecture space: the number and type of specialized cores on a CMP. A chip designer can use analysis akin to Figure 4.4 and knowledge of the workload the chip will run, if known, to pick the most apt configuration within the space. For example, cactuBSSN, pop2, and roms are all physics modeling programs and all stress the data memory. A chip designer would place more cores specialized to handle large working sets into a CMP that will execute these programs. This chapter examines hypothetical configurations as a primary exploration: those with 1 baseline core and a number of uniquely specialized cores:

- 1 specialty core results in 4 configurations
- 2 specialty cores results in 6 configurations
- 3 specialty cores results in 4 configurations
- and 4 specialty cores results in 1 configuration

Furthermore, we model potential speedups gained by each specialty core of 10%, 20%, and 30%. The total number of configurations is 256 including a baseline chip with no specializations. These modeled speedups are seen in many recent microarchitecture works. We provide an example for each key metric that achieves between 10% and 30% speed up.

Branch Misprediction: Eyerman et al. achieved an average 29% increase in performance through selectively flushing instructions after mispredicted branches [46].

L1 instruction cache miss: The entangling prefetcher achieves 10% speedup in less size than the compared instruction prefetchers [122]. Also, increasing the size of the L1 instruction cache is viable.

L1 data cache miss: Berti: an Accurate Local-Delta Data Prefetcher [100] and Register file prefetching [134] both provide 5% improvements. Berti accomplishes this by selecting the

best local-deltas. Register file prefetching orchestrates the data pipeline in an out-of-order system to prefetch nearly half of load requests to the register file.

L2 cache miss: Wu et al. achieve approximately 30% speed up by the proposed Triage prefetcher that filters for important meta-data and sometimes uses last level cache to store additional meta-data [148]. Saglam et al. achieve 30% to 130% speed up in HBM2 memory systems by using a proposed aggressive prefetcher [125]. In DDR systems, the prefetcher switches to a conservative mode with no performance gain.

Re-implementing these published microarchitecture techniques and running them at cycle-level simulation is infeasible and unnecessary for 100ms epochs. Instead, SAHM builds on these established works, composing them in a novel way making it bigger than the sum of its parts. Essentially, our system can be thought of as implementing each of those specific ideas into their own core with only that specialization.

Furthermore, these works were invented and evaluated through the lens of generalist cores: all applications execute their whole duration on the proposed components. Our deployment of them in targeted phases - the most suitable portions of applications execute on each component - should mean our speedups are easily viable; see Section 4.4 for a deep analysis of these works.

Software: Scheduling

A chip using only specialized cores theoretically gains performance. Unfortunately, chips do not exist in a vacuum: current schedulers, like CFS [113], do not acknowledge asymmetry nor specialization [124, 153]. All parts of the system must work together to improve performance. Thus, we have designed a simple greedy scheduler; the algorithm is seen in Listing 4.1. Three pieces enable the scheduler to use specializations. The first is a multi-queue structure that includes a mapping of specializations to cores. The second is a performance analysis section to determine behavioral state and suitable cores. The

third is an inertia component to ensure that the numerous short intervals do not cause continuous migrations of a program. The inertia prevents migration for a certain number of schedulings after migration to a new core. An oracle version of this scheduler has knowledge of which states the program will enter in the next execution timespans.

Listing 4.1: SAHM scheduler incorperating state analysis and program-core inertia

```
if current_program.inertia > 0:
  # do nothing as the program has
  # migrated too recently
  decrement current_program.inertia
  launch_from_local_queue()

perf_counts = get_PMU_counts()
state = calculate_state(perf_counts)
if not specialty_matches(
    current_core.specialty, state):
  # choose a new core
  core = is_core_available_for(state)
  program_handle = current_core. \
                dequeue(current_program)
  if core is null:
    # load balance if not specialty
    core = most_idle_core()
  core.queue(program_handle)
  # queueing also sets program inertia
launch_from_local_queue()
```

4.4 Microarchitecture Specialization

This section serves as a literature survey for the state of the art in each component discussed in this chapter: branch prediction, instruction cache, data cache, and last level cache (LLC). The goal is to demonstrate that the modeled speed ups are achievable in each domain through a combination of the state of the art and trivial techniques like expanding the cache size.

Each subsection is split up by policy category and details the state of the art in that area. We group the LLC discussion into the relevant category, instruction or data, because the memory system is stressed in combination. Table 4.4 summarizes the works, reports average speed up based on all applications included in the work, and details the expected speedup from the proposed technique in a SAHM system. In rows (papers) that do not show any change in speed up from All App to SAHM System, the papers already narrow the evaluation to the program regions that SAHM would apply the techniques.

Component	Work	All App Speedup	SAHM System Speedup
Branch Prediction	CMA-BP [81]	2%	2%
	Selective Flushing [46]	29%	40%
L1I Cache	iTP+xPTP [19]	18.9%	20%
	Entangling Prefetcher [122, 123]	10.1%	10.1%
L1D Cache	Register File Prefetching [134]	3.1%	5%
	Berti [100]	10%	13%
L2 Cache	PACIPV [96]	3.4%	20%
	Mockingjay [132]	4%	20.1%
	Triage [149]	23%	23%
	Runtime Selection [5]	5.8%	5.8%

Table 4.4: Summary of works and their contribution to overall speed up in a SAHM system because of the curated use only when the component is heavily stressed.

Branch Prediction

This subsection covers all design aspects surrounding branch prediction. Of course, the prediction algorithm is included; yet, of more consequence, is the handling of mispredictions. Combining the results of these specializations would result in up to a 40% speed up; more probable the system will achieve less since a better prediction algorithm will mean less mispredictions to gain from. Furthermore, these designs are not currently realistic for implementation into a monolithic core due to their high parameter count or alterations of a base component: more design analysis is necessary.

Prediction Algorithm

Clustered Multi-task Learning and Branch Attention Mechanism Based Branch Predictor applies a CNN model that is augmented with an attention network to the branch history to extract features for different branch types [81]. This enables the model to aggregate similar branches and thereby lower the total parameter count. CMA-BP achieves 97.9% accuracy improving on BranchNet by 1.4% while using an order of magnitude less parameters (6k vs 44k). CMA-BP also has 0.05 - 0.19 MPKI less than BranchNet; that being 4.10 to 4.15 MPKI at the low end and 7.54 to 7.73 MPKI at the high end. This modest reduction tends toward less than 1% improvement of IPC when compared to BranchNet as well as roughly 2% on average improvement against the baseline TAGE-SC-L predictor. These improvements are calculated via my interpolation of BranchNet results, the MPKI per %IPC improvement, on the CMA-BP reported results [154, 81]. This demonstrates one of the key points acknowledged in the SAHM chapter: branch prediction is thoroughly researched so high improvements are hard to achieve.

Handling Misprediction

Examining branch prediction in a different way lends to a new solution; instead of working toward more accurate prediction, lessen the effect that mispredictions have. Eyerman et

al. take this perspective in Enabling Branch-Mispredict Level Parallelism by Selectively Flushing Instructions [46]. In the paper, they recognize that the bodies of many branches are short and reconverge quickly. This means that correct path instructions after reconvergence can already be in the ROB before the misprediction is identified. So, selectively flush solely the wrong-path instructions. The ROB is adapted into a linked list structure enabling correct path instructions post-mispredict to be inserted after the already in the ROB correct path instructions that are post-reconvergence. The analysis of graph applications showed a 29% improvement in performance.

Instruction Caches

There are three main design considerations for an L1 instruction cache: size, replacement policy, and prefetch policy. Size will be passed over because programs' working sets are growing and size is usually constrained to easy numbers making increases harder to justify. Overall, the discussion below combine to provide a 33.3% speed up in the best case. Due to the nature of the components, a 28.9% is a conservative estimate: the replacement policies conflict due to both updating the LLC policy while the prefetcher is orthogonal.

Replacement Policy

As for cache replacement, PACIPV is currently the best for the LLC beating mockingjay with less hardware for instruction heavy workloads [96]. It modifies insertion/promotion vectors to function along side re-reference interval prediction. Furthermore, by using different vectors for demand vs prefetch accesses, the algorithm is cognizant of the data differences. The improved hardware efficiency is due to implementing a static algorithm. The speedup over LRU is 3.4% with a fraction of a percent being above mockingjay.

For the support components, Instruction Translation Prioritization and extended Page Table Prioritization (iTP+xPTP) trade better TLB hit rate for more page table walking [19]. Both mechanisms alter the replacement policy for their component: the TLB and L2

cache respectively. The choice for the TLB is to keep more instruction translations because those translations are on the critical path. This results in more page table walks for data translations. They alter the L2 cache to prioritize keeping these page table entries within the cache; thereby offsetting the negative effect of the TLB change. Single thread results show a geometric mean speed up of 18.9% over LRU policies. A sensitivity study shows that iTP+xPTP increases performance by 1.6% when compared with mockingjay; a better improvement than PACIPV.

Prefetching Policy

For prefetching, the Entangling Prefetcher that pairs cache misses to early enough PCs so as to trigger a prefetch with time to spare for the response to arrive[122, 123]. In other words, a miss is analyzed and a prefetch trigger is inserted at a point early enough that the latency of the miss is covered. If a miss occurs from multiple branches of a control flow, then the miss is associated with multiple prefetch triggers. This method achieves a 10.1% speed up over a hashed perceptron. This is also for the largest proposed area: 77.44KB. Their results show decreasing returns approaching the ideal speed up of 11.8%. Doubling from 40 to 80 KB resulted in less than 1% gain so increasing the size more is not worthwhile.

Data Caches

There are three main design considerations for an L1 data cache: size, replacement policy, and prefetch policy. Size will be passed over because programs' working sets are growing and size is usually constrained to easy numbers making increases harder to justify. Overall, the discussion below combine to provide a 45% speed up in the best case. Due to the nature of the components, a 29% is a conservative estimate: all of the prefetch policies provide the same benefit - closer data when it matters.

Replacement Policy

Mockingjay is the state of the art currently: up to 20% speed up over LRU [132]. It uses temporal difference learning to learn to predict reuse distance in an effort to mimic Belady's MIN policy. In other words, it uses the predicted reuse distance in place of oracle knowledge and employs the MIN policy.

Prefetching Policy

These policies are discussed from nearest to the core to the furthest. Starting in the core, Register File Prefetching gains 3.1% speed up by coordinating data movement all the way to the register file instead of stopping at L1 cache [134]. It does so by reworking the ROB and Out of Order scheduling methods. Working our way outward, the Berti prefetcher applies the same concept of the entangled prefetcher: pair misses with an PC that occurs more than the miss latency amount of time in the past [100]. This sets up the prefetch trigger so that it has enough time to be timely. The reported speed up is over 10% for a Berti+SPP-PPF multi-level prefetcher that takes 41.8KB of storage. Berti by itself achieves 9% of this 10% with a storage requirement of less than 3KB.

Now, shifting focus to the LLC: Triage from Practical Temporal Prefetching achieves 23% speed up over best offset prefetching [149]. This is through a thorough treatment of metadata in the LLC to guide the prefetcher. This can be combined with many other prefetchers through a methodology proposed in Characterizing Machine Learning Based Runtime Prefetcher Selection [5]. The study results in a 5.8% speed up for a reactive sample based model (a realistic model to implement). The prefetchers they are selecting from do not include the state of the art from literature; instead the systems currently implemented on a consumer chip: best offset, second best offest, next line, and adjacent sector. Altering these to include some of the best prefetchers could improve the reported results.

4.5 Evaluation Framework

This evaluation is based on a first-order model and a simulator. For the model, a chip has n specialized cores, each defined by the specialized component and the provided speed up. The workload is selected from the SPEC 2017 CPU benchmark suite. A program is represented by a trace of its gathered performance values. Once a benchmark concludes it restarts; this keeps the mix of applications and behaviors diverse over time. A test executes for a fixed time: the length of the longest benchmark in the executing workload.

This model is developed into a trace driven simulator in order to test the configurations, the scheduler, and the impact of core contention and migration cost. The simulator first initializes the chip and the scheduler data structures. Each benchmark starts at the beginning of its execution trace and has its progress tracker set to zero. Then, the simulator enters the main loop: first, invoke the scheduler on all cores individually making migrations as necessary. Then, perform a timestep of 10ms for each program currently assigned to a core - not waiting in the queue for a core. If a program is assigned to a core whose specialization matches the current state, then the program advances further than on a baseline core; it is sped up. Due to the representation of the workload, non-uniform scheduling events like system calls are not included.

Metrics

During evaluation, we compute:

- **System-level Speedup**: total throughput relative to a baseline with an equal number general-purpose cores.
- **Per-Application Speedup**: how each app benefits or suffers under the scheduling policy.
- **Migration Rate**: average number of swaps per second across the system.

- **Epoch Utilization**: fraction of epochs where apps run on matching specialized cores.

4.6 Results

SAHM's results can be viewed in two lights: per benchmark and per configuration. First, a limited study into the per benchmark speed up is presented. Second, we delve into the breadth of potential speed ups per benchmark. Third, a general chip and a few specialized chips are examined. Fourth, we factor in realistic constraints like migration cost into simulation. The results demonstrate the SAHM system provides speed up even in extreme cases. Finally, included in an appendix is the whole design space of configurations and their speed ups. There are too many data points in the space to lead the reader through; we select the most interesting for this section.

Single Application Limit Study

Goal: What is the limit of the SAHM system in an idealized scenario?

Experiment: We model 5 cores in the canonical configuration: 1 baseline core and 4 specialized cores. The specialized cores provide 30% speed up when an application is in a behavioral state with the specialized component being stressed. Each benchmark is individually run on the system with no migration cost. This means there is no contention for cores. The speed up achieved by each benchmark is shown in Figure 4.7.

Key Takeaway: *The majority of benchmarks achieve high speed up and return on investment.*

Analysis: These results mirror the portion of time outside of the Low state in the characterization results. Imagick has poor speedup as the vast majority of its time is spent in the Low state. Similarly for nab and cam4-s. Overall, this confirms that there is opportunity to increase performance.

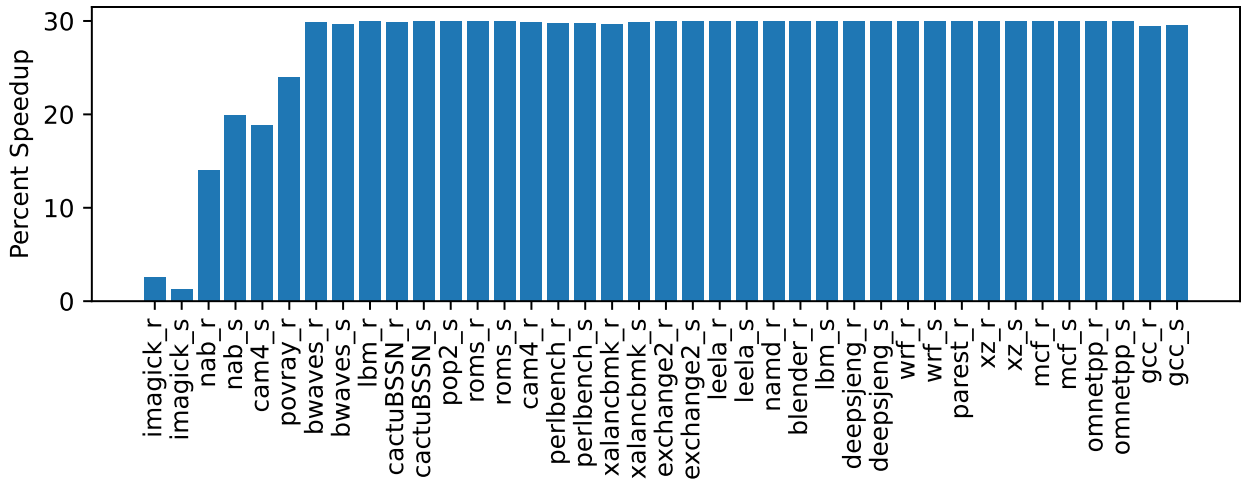


Figure 4.7: The maximum possible idealized speed up by benchmark

Benchmark Breadth

Goal: The size of the design space is difficult to grapple. First, we use the lens of application speedup to understand the distribution provided by other chip configurations that are composed using different number of cores, and different speedups provided by the cores.

Experiment: Each benchmark is simulated on a range of configurations. There is no contention or migration cost so that we can understand the potential of the design space. The configurations are as described in Section 4.3 that results in 255 configurations - we do not include a baseline configuration here in order to tell if an application does not achieve performance from a configuration that includes specialization. The distribution per benchmark is displayed in Figure 4.8. The whiskers are the entire range which explains how the top whisker extends to the ideal speed up as seen in Figure 4.7. The green triangles show the mean and the orange line is the median. These along with the box of the 25th and 75th percentile show the skew of the distribution.

Key Finding: *The majority of configurations provide speed up to the majority of applications.*

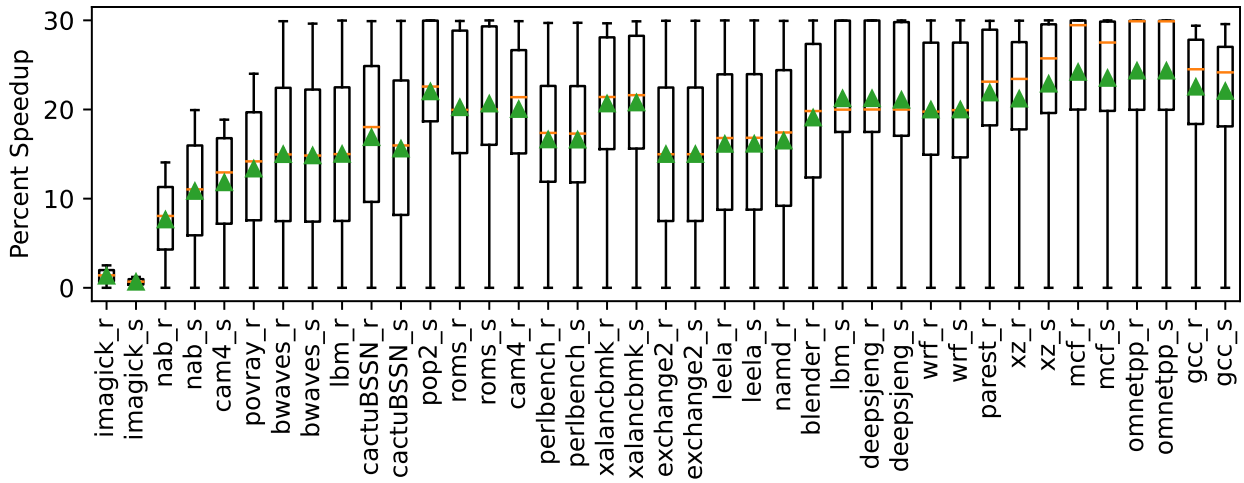


Figure 4.8: Distribution of percent speed up by application; the configurations create the distribution. The green triangle is the average while the orange line is the median. The box is the 25% and 75% percentile. The whiskers cover the entire range.

Analysis: The average speed up is 15% across applications. Additionally, the distributions are skewed toward higher speed up seen through the longer distance to the top of the box compared to the bottom of the box from the median line. This indicates that the majority of configurations outperform the average speed up. Furthermore, the average 25th percentile is over 10% speed up. This demonstrates that even in poorly aligned configurations - configurations that are missing specialization for the behavioral state the program resides in most - applications still benefit from specialization.

Generalist vs Specialized

Goal: Considering the vast design space, does incremental progress on a single core outperform a system of specialized cores in single-threaded performance?

Experiment: We simulate a representative sample of configurations as well as a single core that has 5% improvement overall. Each benchmark is individually simulated on a configuration with no migration cost. This provides the best opportunity for single-threaded speed up to every configuration. The representative configurations are canonical

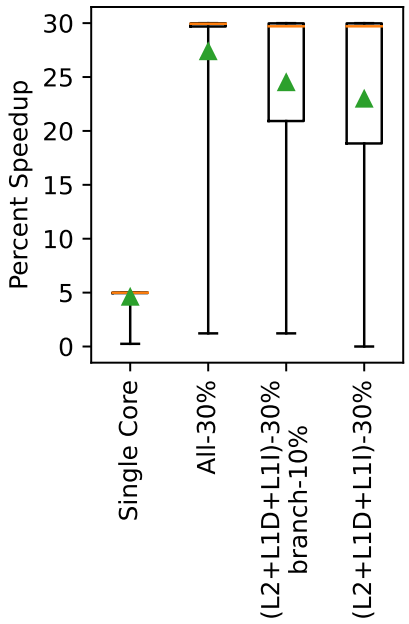


Figure 4.9: Distribution of speed up by configuration. Applications create distribution. Green triangle is mean while orange line is median. Box is the 25% and 75% percentile. Whiskers are the entire range.

5 core systems with 30% speed up for each specialized core except for the core specializing the branch predictor. In Figure 4.9 we provide the configuration with 30% speed up branch core for context under 'All-30%' while also showing systems with the branch core providing 10% and 0% speed up.

Key Finding: *Specialized systems for the majority of applications outperform the best an incremental approach can provide.*

Analysis: The 25th percentile for each of the specialized systems is, at minimum, 3x the best speed up the single core achieves. Even for the bottom quarter of applications where specialization is not fully compatible, the speed up enabled by specialization outperforms the single core.

Realistic Simulation

Goal: Now that we understand the theoretical speed ups of the design space, how effective is the system under real world constraints like migration cost and contention?

Experiment: We simulate a scaled up version of the canonical configuration: it has 7 baseline cores and 8 of each specialized core. 39 cores is chosen as there are 39 benchmarks to be run on the system. This drastically simplifies results to one workload that captures the entire diversity of application behavior instead of needing multiple plots explaining and showing the results for a dozen workload mixes. These results scale to a 32 or 64 core system with equal numbers of specialized cores because the workload mixes for these systems will comprise the same behaviors. Therefore, the effects of core contention will not change. We analyze the same specialized configurations presented in the previous Section 4.6. From an ideal system using an oracle scheduler, we introduce migration cost, realistic scheduling, and the inertia component to the scheduler. Figure 4.10 shows the speed ups for each of 8 constraint configurations. 'Oracle' indicates employ of a scheduler with oracle knowledge of the application states. 'Ideal' has no migrations cost and thus, exhibits the effect of solely contention for cores. Migration cost is 1 millisecond in the 'Cost' and 'Inertia' configurations. This cost exceeds reported costs on big.LITTLE machines [68] and recent work [52, 51]. The 'Inertia' configuration has a scheduler that uses inertia: applications that migrate are held to the assigned core for 5 schedulings. Additionally, we study the impact of higher migration costs to exhibit the stability of achieved speedups in 'Inertia+5ms' and 'Inertia+9ms', respectively having 5ms per migration and 9ms per migration.

Key Finding: *In each system configuration, Inertia achieves close to Ideal speed up. Additionally, the system is robust against high migration cost achieving within 1.5% of the ideal speed up in extreme migration cost scenarios.* **This means, for single threaded applications, that specialization**

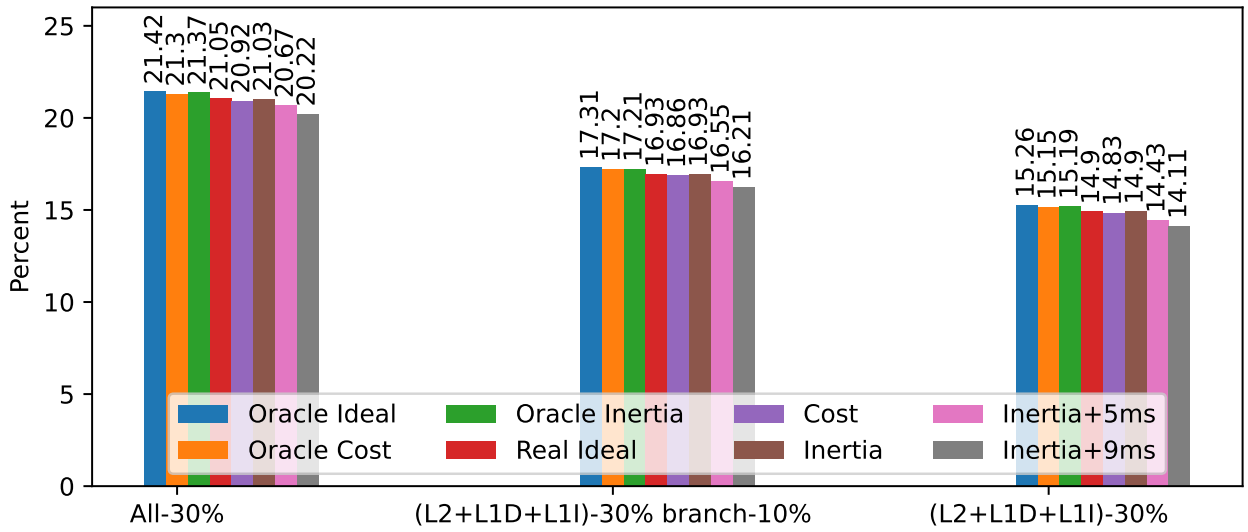


Figure 4.10: Average application speed up achieved by each configuration under various constraints. 'Oracle' uses an oracle scheduler. 'Ideal' has no migration cost. 'Cost' is 1ms per migration. 'Inertia' is 1ms per migration with the inertia scheduler. Inertia+time uses the inertia scheduler while increasing the migration cost to the list time.

is a road ahead for performance improvement.

Analysis: First, realistic scheduling loses less than 1% performance compared to oracle scheduling. The schedules where oracle knowledge outperforms realistic scheduling are at the state transitions. These account for 2% of the total schedules. In all configuration and constraint pairs, applications execute on a core that matches their current stressed component approximately 70% of the time. This provides explanation for how each constraint achieves relatively equal speed up. Furthermore, the Inertia configuration recovers more than half of the loss from introducing migration overheads. This is the goal of including inertia in the scheduler.

4.7 Related Work

This section details the related work along two avenues. First, how this chapter compares to prior phase analysis work. Second, we list some works that already find performance

gain by adjusting systems for heterogeneous hardware.

Phase Analyses

The classification and prediction of program phases has been a combed through topic applied to power management [63, 14, 72] and security [103, 67]. Machine learning has sparked new research especially for prediction [87]. We refer the reader to recent surveys for a compendium of techniques [32, 2]. This section presents the limitations of these works and how SAHM differs.

Isci et al. classifies two hardware performance counters through static windows identifying different boundedness phases: from CPU-bound to memory-bound. These phases lie on one axis and cannot represent as rich a space as is presented in this chapter. The phases inform prediction, which in turn informs DVFS. The system improves energy-delay product over the baseline by 18% on average. Similarly, Bui and Kim examined super fine grain program phases with application to DVFS [14]. The characterization is at cycle-count granularity. Intervals can be 1 to 100K cycles which is 1ns to 100µs assuming a 1GHz frequency: much finer granularity than our analysis in this chapter; however, it is not feasible to collect entire traces from large benchmarks like those in the SPEC CPU 2017 suite using their methodology. They ran MiBench benchmarks on a processor model on an FPGA. They find that short duration super fine granularity phases detect and adjust V/F levels for more power savings: between 1.5 to 2x savings. Overall, these works provide benefit via DVFS configuration; neither migrates programs to cores more suitable for the performance/watt expected in the detected phase. In contrast, a core idea of SAHM is migration based on phase behavior.

More complex classifiers and predictors have been proposed. Alcorta et al. use hardware performance counters to train a phase classifier and phase predictor both in the single core [4] and multicore cases [87, 6]. The single core work achieved higher accuracy than table based methods in phase classification. It also determined that for the combined

classification and prediction task, two-level k-means clustering was the best classifier for the majority of predictors. In the multicore case, classification is done to separate data into multiple predictors - each specialized to a phase. The results shows that phase-aware LSTM had the best prediction with 23% mean absolute percent error averaged across the benchmarks studied. Due to their complexity, these phase analysis tools increase analysis overhead therefore increasing the difficulty in initial implementation in a system setting. This chapter demonstrates that simple methods are enough to substantially improve performance.

Support for Heterogeneous CMP

Various areas have gained from exploiting heterogeneous CMP. Cao et al. analyzed VM services and found opportunity for power and performance gains through placing JIT, garbage collection, and the interpreter on a different smaller core [16]. Alcorta et al. applied phase analysis and machine learning to choosing the L2 prefetcher in multicore chips [5]. The dynamic selection of the prefetcher improved IPC by 5.8% on average. Both demonstrate that using heterogeneous CMP offers opportunity for improvements when the systems built on top of the architecture acknowledge the heterogeneity.

4.8 Conclusion

SAHM introduces a principled, empirical approach to identifying opportunities for architectural specialization based on fine-grained performance counter monitoring. By mapping runtime behavior to a discrete set of states and designing specialized cores accordingly, SAHM pushes the frontier of single-thread performance without requiring speculation or instruction-level parallelism. Our exhaustive modeling shows that even modest specializations can yield substantial gains, offering a compelling path forward for future CPU designs.

5 CONCLUSION

Throughout this work strides are taken toward a better architectures and systems. The IPU can assist both architects and developers to create a more cohesive computer thereby improving single thread performance. NeuroScalar combats the growing problem of non-representative benchmarks for testing new systems. In the long run, these tests will create better architectures. SAHM is a direct approach to improving single thread performance by diversification and careful informed management. Each of these contribute to a larger overall goal: improving single thread performance.

Each proposed system enlightens a new method toward this goal. IPU and Neuroscalar take a unique perspective by running tests in the field. The IPU, meant as a flexible unit, uses an FPGA and a simple core to dramatically increase the availability of analysis at levels here-to unknown. The unit is powerful enough to perform complex component testing prior to fabrication while using off-the-shelf hardware enabling developers easy access to create PICS for example. In a different vein, Neuroscalar proposes using machine learning to estimate latencies of different hardware configurations in real-time, enabling A/B testing. It uses either commodity parts for mass general deployment or a specialized accelerator for focused in-house testing. Both options increase the speed of design space exploration recouping and saving more than the initial training costs. Finally, SAHM proposes a new system informed by program state. Our experiments using the current state of the art as modeled speed up show a high return on investment for the diverse specializations.

These three systems provide enormous potential for improving single thread performance as well as informing the architecture design cycle at a more rapid pace.

A APPENDICES

The following appendices list which chapter they are for. Much of the material is expansions or supporting details of the work presented that does not add to the overall argument.

A.1 NeuroScalar Appendix

We provide results that expand on those presented in the chapter. These are provided for completeness.

Table A.1: Detailed parameter shapes and counts of the proposed model.

Parameter	**Shape**	**#Params**
input_proj.weight	[256, 13]	3,328
input_proj.bias	[256]	256
rnn.weight_ih_l0	[1024, 256]	262,144
rnn.weight_hh_l0	[1024, 256]	262,144
rnn.bias_ih_l0	[1024]	1,024
rnn.bias_hh_l0	[1024]	1,024
rnn.weight_ih_l1	[1024, 256]	262,144
rnn.weight_hh_l1	[1024, 256]	262,144
rnn.bias_ih_l1	[1024]	1,024
rnn.bias_hh_l1	[1024]	1,024
cls_fc1.weight	[64, 256]	16,384
cls_fc1.bias	[64]	64
cls_fc2.weight	[2, 64]	128
cls_fc2.bias	[2]	2
output_layer_short.weight	[1, 256]	256
output_layer_short.bias	[1]	1
output_layer_long.weight	[1, 256]	256
output_layer_long.bias	[1]	1
Total	–	1,073,348

Table A.2: Foundation model (**TraceFusion-13**) evaluated per benchmark (held-out splits). Error-centric metrics are emphasized; *Acc*(*round*) is reported for reference. All numbers are fractions except MAE/RMSE (cycles).

Benchmark	rel⩽5%↑	RAE↓	MAE↓	RMSE↓	±1 ↑	±2 ↑	Acc(round) (ref)↑
`blender`	0.7791	0.1862	0.5795	4.0245	0.9158	0.9646	0.7774
`cam4`	0.7941	0.1367	0.3762	4.9745	0.9490	0.9803	0.7913
`deepsjeng`	0.7104	0.2580	0.9147	6.3028	0.8579	0.9430	0.7102
`exchange2`	0.7614	0.2016	0.6695	3.9367	0.8960	0.9579	0.7599
`gcc`	0.7189	0.2650	0.9408	7.5068	0.8607	0.9378	0.7174
`imagick`	0.6098	0.3425	0.8163	2.2620	0.7827	0.9043	0.6098
`leela`	0.6714	0.3358	1.2239	6.1895	0.8163	0.9160	0.6704
`mcf`	0.6174	0.3795	2.4035	23.5354	0.7768	0.8997	0.6171
`nab`	0.6240	0.3844	1.4612	6.7714	0.7802	0.8947	0.6232
`namd`	0.8156	0.1415	0.4563	4.2321	0.9391	0.9725	0.8116
`omnetpp`	0.7263	0.2539	0.9170	5.9410	0.8637	0.9377	0.7244
`parest`	0.6252	0.5029	3.5813	34.1094	0.8002	0.8999	0.6246
`perlbench`	0.8064	0.1347	0.3520	4.8300	0.9570	0.9836	0.8054
`povray`	0.8333	0.1140	0.3143	1.7281	0.9545	0.9821	0.8318
`xalancbmk`	0.7263	0.2454	0.9135	8.8675	0.8739	0.9415	0.7231
`xz`	0.6790	0.3386	1.2454	6.5660	0.8388	0.9159	0.6782

Table A.3: Pairwise ordering per benchmark (Part I). Each cell shows *Match rate* / *GT-better* / *Non-zero*.

Benchmark	4w+mem⩽ 8w	4w+mem⩽ rob	4w+mem⩽ lsq	4w+mem⩽ 6w+ls	8w⩽ rob
blender	92.75 / 13.80 / 39.21	92.73 / 13.81 / 39.30	92.72 / 13.81 / 39.39	92.72 / 13.83 / 39.48	92.73 / 13.76 / 39.41
cam4	96.46 / 16.28 / 35.99	96.45 / 16.29 / 36.05	96.47 / 16.29 / 36.12	96.49 / 16.30 / 36.18	96.48 / 16.25 / 36.14
deepsjeng	90.49 / 14.75 / 39.38	90.45 / 14.77 / 39.47	90.41 / 14.76 / 39.57	90.38 / 14.79 / 39.66	90.41 / 14.74 / 39.60
exchange2	92.74 / 14.70 / 39.19	92.75 / 14.71 / 39.28	92.76 / 14.69 / 39.37	92.77 / 14.70 / 39.46	92.80 / 14.62 / 39.39
gcc	90.00 / 16.00 / 38.79	89.99 / 16.03 / 38.87	89.97 / 16.02 / 38.95	89.97 / 16.03 / 39.02	89.98 / 15.95 / 38.98
imagick	81.62 / 15.33 / 34.47	81.40 / 15.31 / 34.61	81.18 / 15.29 / 34.75	81.01 / 15.30 / 34.88	81.14 / 15.27 / 34.79
leela	89.80 / 15.12 / 38.53	89.77 / 15.12 / 38.64	89.74 / 15.11 / 38.74	89.72 / 15.13 / 38.83	89.75 / 15.07 / 38.77
mcf	89.36 / 15.99 / 43.49	89.29 / 16.05 / 43.61	89.22 / 16.08 / 43.73	89.16 / 16.14 / 43.85	89.17 / 16.04 / 43.83
nab	89.00 / 14.85 / 34.70	88.98 / 14.89 / 34.81	88.96 / 14.87 / 34.91	88.94 / 14.89 / 35.02	89.01 / 14.82 / 35.05
namd	94.37 / 15.04 / 40.05	94.37 / 15.06 / 40.15	94.37 / 15.06 / 40.24	94.38 / 15.09 / 40.32	94.38 / 15.02 / 40.26
omnetpp	89.45 / 15.83 / 37.66	89.45 / 15.86 / 37.74	89.46 / 15.86 / 37.83	89.47 / 15.89 / 37.91	89.48 / 15.82 / 37.87
parest	88.52 / 16.38 / 43.81	88.50 / 16.40 / 43.86	88.48 / 16.41 / 43.92	88.47 / 16.45 / 43.97	88.48 / 16.35 / 43.92
perlbench	96.29 / 15.16 / 37.91	96.28 / 15.16 / 38.02	96.28 / 15.13 / 38.13	96.28 / 15.15 / 38.24	96.27 / 15.10 / 38.17
povray	95.09 / 14.78 / 33.47	95.09 / 14.78 / 33.56	95.08 / 14.77 / 33.64	95.08 / 14.79 / 33.72	95.08 / 14.75 / 33.66
xalancbmk	89.72 / 15.50 / 34.52	89.72 / 15.50 / 34.64	89.72 / 15.47 / 34.76	89.73 / 15.47 / 34.88	89.76 / 15.39 / 34.88
xz	92.00 / 14.58 / 41.66	91.95 / 14.59 / 41.75	91.92 / 14.59 / 41.83	91.89 / 14.65 / 41.91	91.91 / 14.56 / 41.85

Table A.4: Pairwise ordering per benchmark (Part II). Each cell shows *Match rate* / *GT-better* / *Non-zero*.

Benchmark	8w⩽ lsq	8w⩽ 6w+ls	rob⩽ lsq	rob⩽ 6w+ls	lsq⩽ 6w+ls
blender	92.76 / 13.72 / 39.34	92.79 / 13.75 / 39.28	92.82 / 13.72 / 39.21	92.86 / 13.75 / 39.15	92.89 / 13.78 / 39.08
cam4	96.46 / 16.22 / 36.11	96.47 / 16.25 / 36.07	96.46 / 16.22 / 36.03	96.47 / 16.25 / 36.00	96.50 / 16.28 / 35.92
deepsjeng	90.44 / 14.71 / 39.54	90.46 / 14.74 / 39.48	90.50 / 14.72 / 39.42	90.52 / 14.75 / 39.36	90.55 / 14.77 / 39.29
exchange2	92.82 / 14.60 / 39.32	92.85 / 14.62 / 39.25	92.88 / 14.60 / 39.18	92.92 / 14.63 / 39.11	92.94 / 14.63 / 39.01
gcc	89.99 / 15.92 / 38.93	90.00 / 15.95 / 38.89	90.02 / 15.93 / 38.84	90.04 / 15.95 / 38.80	90.07 / 15.97 / 38.71
imagick	81.25 / 15.24 / 34.71	81.34 / 15.26 / 34.62	81.47 / 15.26 / 34.53	81.61 / 15.29 / 34.43	81.73 / 15.32 / 34.35
leela	89.76 / 15.04 / 38.70	89.78 / 15.08 / 38.64	89.81 / 15.06 / 38.57	89.83 / 15.09 / 38.51	89.87 / 15.11 / 38.42
mcf	89.17 / 16.00 / 43.82	89.17 / 16.02 / 43.81	89.18 / 15.98 / 43.80	89.18 / 15.99 / 43.78	89.18 / 16.01 / 43.75
nab	89.04 / 14.80 / 35.09	89.07 / 14.82 / 35.12	89.11 / 14.79 / 35.14	89.14 / 14.82 / 35.17	89.21 / 14.83 / 35.12
namd	94.40 / 14.98 / 40.21	94.42 / 15.00 / 40.15	94.44 / 14.97 / 40.10	94.47 / 15.00 / 40.04	94.49 / 15.01 / 39.97
omnetpp	89.51 / 15.79 / 37.82	89.55 / 15.82 / 37.78	89.58 / 15.79 / 37.73	89.62 / 15.82 / 37.69	89.66 / 15.84 / 37.62
parest	88.47 / 16.30 / 43.87	88.47 / 16.32 / 43.82	88.48 / 16.28 / 43.77	88.50 / 16.30 / 43.72	88.52 / 16.33 / 43.65
perlbench	96.27 / 15.09 / 38.10	96.28 / 15.12 / 38.03	96.30 / 15.12 / 37.96	96.32 / 15.16 / 37.89	96.34 / 15.16 / 37.79
povray	95.09 / 14.73 / 33.60	95.11 / 14.77 / 33.55	95.12 / 14.74 / 33.50	95.14 / 14.77 / 33.45	95.15 / 14.81 / 33.37
xalancbmk	89.79 / 15.36 / 34.88	89.83 / 15.38 / 34.88	89.87 / 15.35 / 34.88	89.90 / 15.37 / 34.88	89.95 / 15.37 / 34.85
xz	91.96 / 14.48 / 41.78	91.97 / 14.53 / 41.72	92.03 / 14.45 / 41.65	92.05 / 14.51 / 41.59	92.06 / 14.56 / 41.53

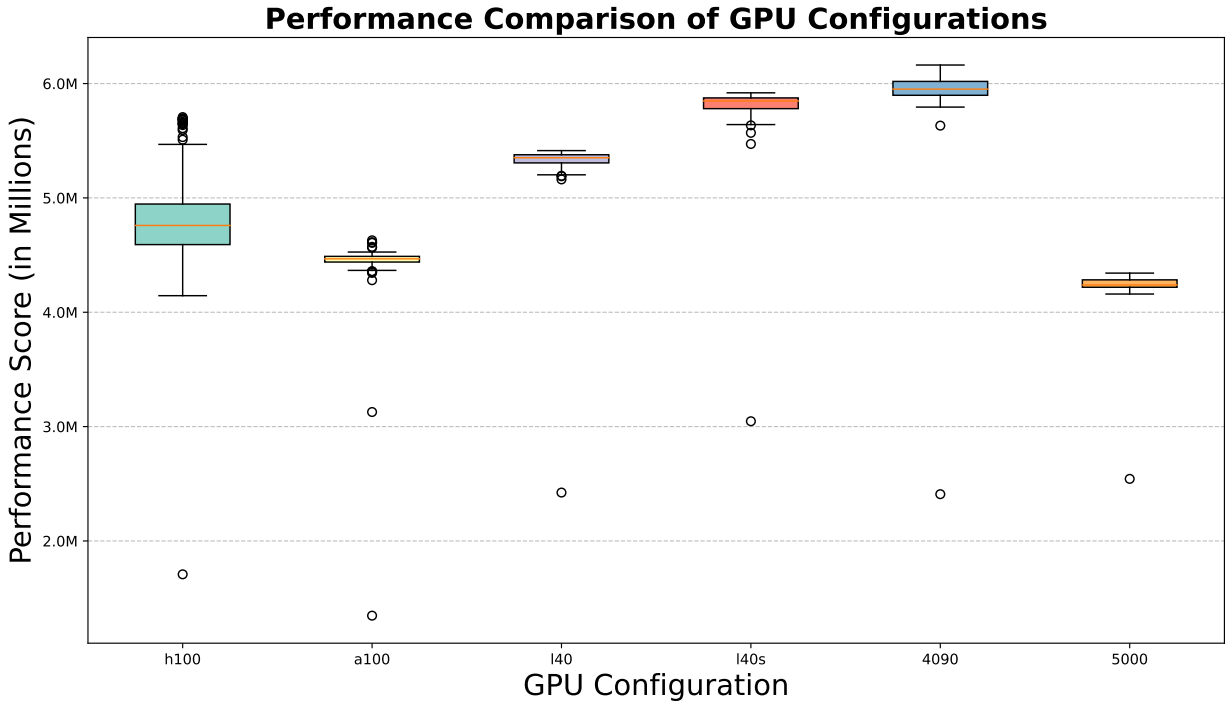


Figure A.1: Distribution of GPU throughput across repeated runs. Boxplot visualization complements Table 3.2, showing variability and stability of inference speeds.

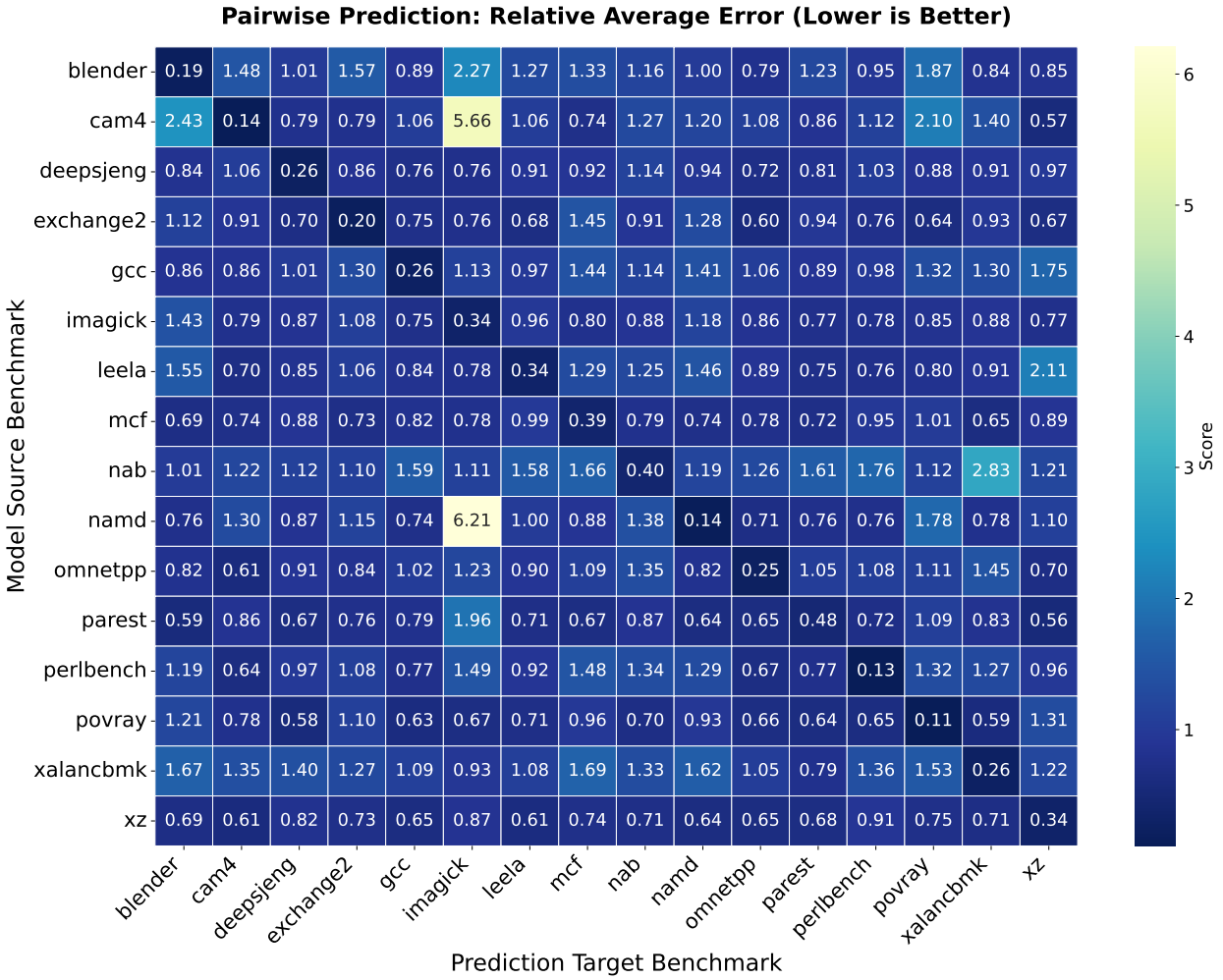


Figure A.2: Pairwise prediction **Relative Average Error**.

A.2 SAHM Appendix

This figure shows the distribution of potential speed ups given a configuration. Each graph is associated with a branch configuration while each vertical gridline has the rest of the configuration listed on the x-axis. For example, the leftmost boxplot of the bottommost graph is the configuration with a baseline core and a branch specialty core that provides a speed up of 30%.

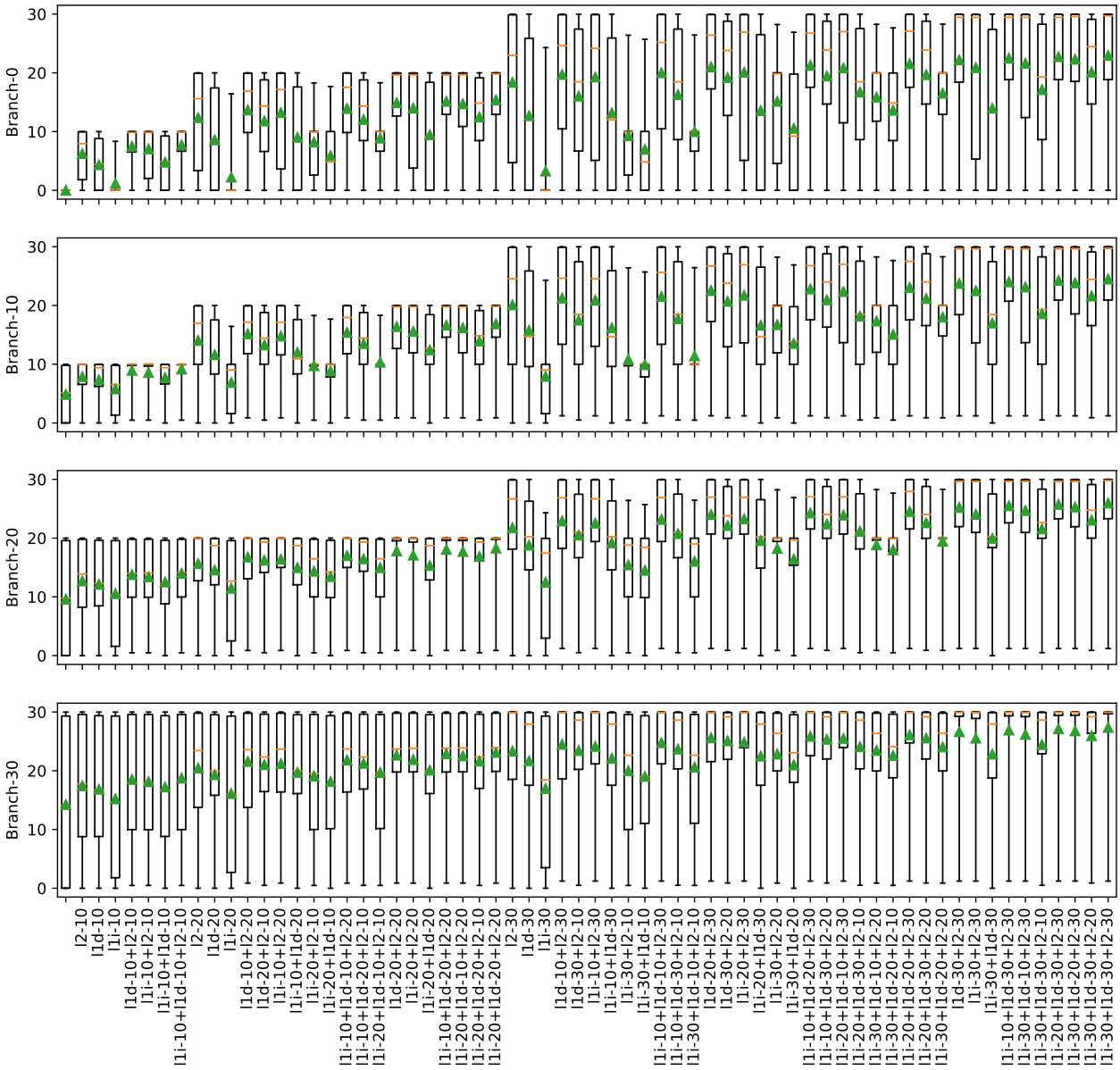


Figure A.3: Distribution of application speed up by configuration. The whole design space is listed with the branch specialized core broken out into a chart each while the the rest of the cores are listed on the x-axis in ascending amount of speed up. Green triangle is mean while orange line is median. Box is 25% and 75% percentiles and the whiskers are the entire range.

BIBLIOGRAPHY

[1] ios app signing. `https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf`.

[2] Cristinel Ababei and Milad Ghorbani Moghaddam. A survey of prediction and classification techniques in multicore processor systems. *IEEE Transactions on Parallel and Distributed Systems*, 30(5):1184–1200, 2019.

[3] Ayaz Akram and Lina Sawalha. A survey of computer architecture simulation techniques and tools. *Ieee Access*, 7:78120–78145, 2019.

[4] Erika S. Alcorta and Andreas Gerstlauer. Learning-based workload phase classification and prediction using performance monitoring counters. In *2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6, 2021.

[5] Erika S. Alcorta, Mahesh Madhav, Richard Afoakwa, Scott Tetrick, Neeraja J. Yadwadkar, and Andreas Gerstlauer. Characterizing machine learning-based runtime prefetcher selection. *IEEE Computer Architecture Letters*, 23(2):146–149, 2024.

[6] Erika S. Alcorta, Pranav Rama, Aswin Ramachandran, and Andreas Gerstlauer. Phase-aware cpu workload forecasting. In Alex Orailoglu, Matthias Jung, and Marc Reichenbach, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 195–209, Cham, 2022. Springer International Publishing.

[7] Android app signing. `https://developer.android.com/studio/publish/app-signing`.

[8] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 725–737, 2015.

[9] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[10] Rico Backasch, Christian Hochberger, Alexander Weiss, Martin Leucker, and Richard Lasslop. Runtime verification for multicore soc with high-quality trace data. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 18(2):1–26, 2013.

[11] Ioana Baldini, Stephen J Fink, and Erik Altman. Predicting gpu performance from cpu runs using machine learning. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, pages 254–261. IEEE, 2014.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[13] 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. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011.

[14] Van Bui and Martha Allen Kim. Analysis of super fine-grained program phases. *Columbia University Technical Report*, 2017.

[15] Martin Burtscher. Vpc3: A fast and effective trace-compression algorithm. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, page 167–176, New York, NY, USA, 2004. Association for Computing Machinery.

[16] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. *SIGARCH Comput. Archit. News*, 40(3):225–236, June 2012.

[17] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[18] Tsung-Yung Jonathan Chang, Yen-Huei Chen, Wei-Min Chan, Hank Cheng, Po-Sheng Wang, Yangsyu Lin, Hidehiro Fujiwara, Robin Lee, Hung-Jen Liao, Ping-Wei Wang, Geoffrey Yeap, and Quincy Li. A 5-nm 135-mb sram in euv and high-mobility channel finfet technology with metal coupling and charge-sharing write-assist circuitry schemes for high-density and low-vmin applications. *IEEE Journal of Solid-State Circuits*, 56(1):179–187, 2021.

[19] Dimitrios Chasapis, Georgios Vavouliotis, Daniel A. Jiménez, and Marc Casas. Instruction-aware cooperative tlb and cache replacement policies. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 619–636, New York, NY, USA, 2025. Association for Computing Machinery.

[20] Binghao Chen, Han Zhao, Weihao Cui, Yifu He, Shulai Zhang, Quan Chen, Zijun Li, and Minyi Guo. Maximizing the utilization of gpus used by cloud gaming through adaptive co-location with combo. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 265–280, New York, NY, USA, 2023. Association for Computing Machinery.

[21] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 12–23, 2016.

[22] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples

for fdo compilation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 42–52, 2010.

[23] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, and Todd C. Mowry. Log-based architectures: Using multicore to help software behave correctly. *SIGOPS Oper. Syst. Rev.*, 45(1):84–91, feb 2011.

[24] Shimin Chen, Michael Kozuch, Phillip B. Gibbons, Michael Ryan, Theodoros Strigkos, Todd C. Mowry, Olatunji Ruwase, Evangelos Vlachos, Babak Falsafi, and Vijaya Ramachandran. Flexible hardware acceleration for instruction-grain lifeguards. *IEEE Micro*, 29(1):62–72, jan 2009.

[25] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016.

[26] Albert Cheu1(B), Adam Smith, Jonathan Ullman1, David Zeber3, and Maxim Zhilyaev. Distributed differential privacy via shuffling. In *Proceedings of Eurocrypt*, 2019.

[27] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. Asap7: A 7-nm finfet predictive process design kit. *Microelectronics Journal*, 53:105–115, 2016.

[28] M.L. Corliss, E.C. Lewis, and A. Roth. Dise: a programmable macro engine for customizing applications. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 362–373, 2003.

[29] Graham Cormode, Somesh Jha, Tejas Kulkarni, Ninghui Li, Divesh Srivastava, and Tianhao Wang. Privacy at scale: Local differential privacy in practice. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1655–1658, 2018.

[30] ByBrian Coutinho. Dynolog: Open source system observability. https://developers.facebook.com/blog/post/2022/11/16/dynolog-open-source-system-observability/.

[31] Tdp and power draw: No real surprises https://www.anandtech.com/show/16214/amd-zen-3-ryzen-deep-dive-review-5950x-5900x-5800x-and-5700x-tested/8.

[32] Keeley Criswell and Tosiron Adegbija. A survey of phase classification techniques for characterizing variable application behavior. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):224–236, 2020.

[33] Deeksha Dangwal, Weilong Cui, Joseph McMahan, and Timothy Sherwood. Safer program behavior sharing through trace wringing. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 1059–1072, New York, NY, USA, 2019. Association for Computing Machinery.

[34] Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, et al. Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point. *Advances in neural information processing systems*, 33:10271–10281, 2020.

[35] Datadog. Datadog continuous profiler. https://www.datadoghq.com/product/code-profiling/.

[36] Michael Davies, Ian McDougall, Selvaraj Anandaraj, Deep Machchhar, Rithik Jain, and Karthikeyan Sankaralingam. A journey of a 1,000 kernels begins with a single step: A retrospective of deep learning on gpus. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 20–36, New York, NY, USA, 2024. Association for Computing Machinery.

[37] Jeffrey Dean, James E Hicks, Carl A Waldspurger, William E Weihl, and George Chrysos. Profileme: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 292–302. IEEE, 1997.

[38] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. PHMon: A programmable hardware monitor and its security use cases. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 807–824. USENIX Association, August 2020.

[39] Leila Delshadtehrani, Schuyler Eldridge, Sadullah Canakci, Manuel Egele, and Ajay Joshi. Nile: A programmable monitoring coprocessor. *IEEE Computer Architecture Letters*, 17(1):92–95, 2018.

[40] Daniel Y. Deng, Daniel Lo, Greg Malysa, Skyler Schneider, and G. Edward Suh. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 137–148, 2010.

[41] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context*, SSR '01, page 266–277, New York, NY, USA, 2001. Association for Computing Machinery.

[42] Nicholas C Doyle, Eric Matthews, Graham Holland, Alexandra Fedorova, and Lesley Shannon. Performance impacts and limitations of hardware memory access trace collection. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 506–511. IEEE, 2017.

[43] Muhammad ES Elrabaa, Ayman Hroub, Muhamed F Mudawar, Amran Al-Aghbari, Mohammed Al-Asli, and Ahmad Khayyat. A very fast trace-driven simulation

platform for chip-multiprocessors architectural explorations. *IEEE Transactions on Parallel and Distributed Systems*, 28(11):3033–3045, 2017.

[44] U. Erlingsson, V. Pihur, and A. Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of CCS 2014*, 2014.

[45] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[46] Stijn Eyerman, Wim Heirman, Sam Van Den Steen, and Ibrahim Hur. Enabling branch-mispredict level parallelism by selectively flushing instructions. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 767–778, New York, NY, USA, 2021. Association for Computing Machinery.

[47] Dror G Feitelson, Eitan Frachtenberg, and Kent L Beck. Development and deployment at facebook. *IEEE Internet Computing*, 17(4):8–17, 2013.

[48] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189, 2017.

[49] Linux Foundation.

[50] Sotiria Fytraki, Evangelos Vlachos, Onur Kocberber, Babak Falsafi, and Boris Grot. Fade: A programmable filtering accelerator for instruction-grain monitoring. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 108–119, 2014.

[51] Giorgis Georgakoudis, Dimitrios S. Nikolopoulos, and Spyros Lalis. Fast dynamic binary rewriting to support thread migration in shared-isa asymmetric multicores. In *Proceedings of the First International Workshop on Code OptimiSation for MultI and Many Cores*, COSMIC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[52] Giorgis Georgakoudis, Dimitrios S. Nikolopoulos, Hans Vandierendonck, and Spyros Lalis. Fast dynamic binary rewriting for flexible thread migration on shared-isa heterogeneous mpsocs. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 156–163, 2014.

[53] Badih Ghazi, Pasin Manurangsi, Pritish Kamath, and Ravi Kumar Ravikumar. Anonymized histograms in intermediate privacy models. In *NeurIPS 2022*, 2022.

[54] Nathan Gober, Gino Chacon, L. Wang, Paul V. Gratz, Daniel A. Jiménez, Elvira Teran, Seth H. Pugsley, and Jinchun Kim. The championship simulator: Architectural simulation for education and competition. *ArXiv*, abs/2210.14324, 2022.

[55] Open telemetry. `https://cloud.google.com/learn/what-is-opentelemetry`.

[56] Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. Tip: Time-proportional instruction profiling. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 15–27, New York, NY, USA, 2021. Association for Computing Machinery.

[57] Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. Tea: Time-proportional event analysis. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[58] Joseph L Greathouse, Hongyi Xin, Yixin Luo, and Todd Austin. A case for unlimited watchpoints. *ACM SIGPLAN Notices*, 47(4):159–172, 2012.

[59] Wenlei He, Hongtao Yu, Lei Wang, and Taewook Oh. Revamping sampling-based pgo with context-sensitivity and pseudo-instrumentation. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 322–333, 2024.

[60] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[61] Intel. Hardware-based profile guided optimization (pgo) from intel `https://www.intel.com/content/www/us/en/developer/articles/technical/hwpgo.html`, 2024.

[62] Engin Ïpek, Sally A McKee, Rich Caruana, Bronis R de Supinski, and Martin Schulz. Efficiently exploring architectural design spaces via predictive modeling. *ACM SIGOPS Operating Systems Review*, 40(5):195–206, 2006.

[63] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 359–370, 2006.

[64] PJ Joseph, Kapil Vaswani, and Matthew J Thazhuthaveetil. Construction and use of linear regression models for processor performance analysis. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 99–108. IEEE, 2006.

[65] PJ Joseph, Kapil Vaswani, and Matthew J Thazhuthaveetil. A predictive performance model for superscalar processors. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 161–170. IEEE, 2006.

[66] Tutorial: The role of jtag in system debug & test throughout the embedded system development lifecycle. `https://www.embedded.com/tutorial-the-role-of-jtag-in-system-debug-test-throughout-the-embedded-system-deve`

[67] Sai Praveen Kadiyala, Akella Kartheek, and Tram Truong-Huu. Program behavior analysis and clustering using performance counters. In *Proceedings of the 2020 Work-*

shop on DYnamic and Novel Advances in Machine Learning and Intelligent Cyber Security, DYNAMICS '20, New York, NY, USA, 2022. Association for Computing Machinery.

[68] Shubham Kamdar and Neha Kamdar. big. little architecture: Heterogeneous multi-core processing. *International Journal of Computer Applications*, 119:35–38, 06 2015.

[69] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G Rogers, Tor M Aamodt, and Nikos Hardavellas. Accelwattch: A power modeling framework for modern gpus. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 738–753, 2021.

[70] Anirudh Mohan Kaushik, Gennady Pekhimenko, and Hiren Patel. Gretch: A hardware prefetcher for graph analytics. *ACM Trans. Archit. Code Optim.*, 18(2), February 2021.

[71] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accelsim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020.

[72] Saman Khoshbakht and Nikitas Dimopoulos. A new approach to detecting execution phases using performance monitoring counters. In Jens Knoop, Wolfgang Karl, Martin Schulz, Koji Inoue, and Thilo Pionteck, editors, *Architecture of Computing Systems - ARCS 2017*, pages 85–96, Cham, 2017. Springer International Publishing.

[73] Dirk Koch, Nguyen Dao, Bea Healy, Jing Yu, and Andrew Attwood. Fabulous: An embedded fpga framework. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 45–56, New York, NY, USA, 2021. Association for Computing Machinery.

[74] Benjamin C Lee and David M Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *ACM SIGOPS operating systems review*, 40(5):185–194, 2006.

[75] Benjamin C Lee and David M Brooks. Illustrative design space studies with microarchitectural regression models. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 340–351. IEEE, 2007.

[76] Jong Chul Lee, Faycel Kouteib, and Roman Lysecky. Event-driven framework for configurable runtime system observability for soc designs. In *2012 IEEE International Test Conference*, pages 1–10. IEEE, 2012.

[77] Jong Chul Lee and Roman Lysecky. System-level observation framework for non-intrusive runtime monitoring of embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 20(3):1–27, 2015.

[78] Roy Levin, Ilan Newman, and Gadi Haber. Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In *High Performance Embedded Architectures and Compilers: Third International Conference, HiPEAC 2008, Göteborg, Sweden, January 27-29, 2008. Proceedings 3*, pages 291–304. Springer, 2008.

[79] Jiangtian Li, Xiaosong Ma, Karan Singh, Martin Schulz, Bronis R de Supinski, and Sally A McKee. Machine learning based online performance prediction for runtime parallelization and task scheduling. In *2009 IEEE international symposium on performance analysis of systems and software*, pages 89–100. IEEE, 2009.

[80] Lingda Li, Santosh Pandey, Thomas Flynn, Hang Liu, Noel Wheeler, and Adolfy Hoisie. Simnet: Accurate and high-performance computer architecture simulation using deep learning. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(2), June 2022.

[81] Ming Li, Rucong Xu, Hexu Zhang, Lin Li, and Yun Li. Cma-bp: A clustered multi-task learning and branch attention based branch predictor. In *2024 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1370–1376, 2024.

[82] Yehuda Lindell. Secure multiparty computation (mpc). Cryptology ePrint Archive, Paper 2020/300, 2020. https://eprint.iacr.org/2020/300.

[83] Tong-Yu Liu, Jianmei Guo, and Bo Huang. Efficient cross-platform multiplexing of hardware performance counters via adaptive grouping. *ACM Trans. Archit. Code Optim.*, 21(1), January 2024.

[84] XH Liu, Yuan Peng, and JY Zhang. A sample profile-based optimization method with better precision. In *Proc. Int. Conf. Artif. Intell. Comput. Sci*, pages 340–346, 2016.

[85] Daniel Lo, Tao Chen, Mohamed Ismail, and G. Edward Suh. Run-time monitoring with adjustable overhead using dataflow-guided filtering. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 662–674, 2015.

[86] Locuza. Nvidia's ada lineup, configurations, estimated die sizes and a comparison with other chips https://locuza.substack.com/p/nvidias-ada-lineup-configurations.

[87] Erika Susana Alcorta Lozano and Andreas Gerstlauer. Learning-based phase-aware multi-core cpu workload forecasting. *ACM Trans. Des. Autom. Electron. Syst.*, 28(2), December 2022.

[88] Yirong Lv, Bin Sun, Qingyi Luo, Jing Wang, Zhibin Yu, and Xuehai Qian. Counterminer: Mining big performance data from hardware counters. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 613–626, 2018.

[89] Marketwatch. Moore's law is dead. marketwatch. https://www.marketwatch.com/story/moores-laws-dead-nvidia-ceo-jensen-says-in-justifying-gaming-card-price-hike

[90] Eric Matthews, Lesley Shannon, and Alexandra Fedorova. A configurable framework for investigating workload execution. In *2010 International Conference on Field-Programmable Technology*, pages 409–412. IEEE, 2010.

[91] Ian McDougall, Shayne Wadle, Harish Batchu, and Karthikeyan Sankaralingam. Ipu: Flexible hardware introspection units, 2025.

[92] Joseph McMahan, Michael Christensen, Kyle Dewey, Ben Hardekopf, and Timothy Sherwood. Bouncer: Static program analysis in hardware. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 711–722, 2019.

[93] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on machine learning*, pages 4505–4515. PMLR, 2019.

[94] Matthew C Merten, Andrew R Trick, Christopher N George, John C Gyllenhaal, and Wen-mei W Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proceedings of the 26th annual international symposium on Computer architecture*, pages 136–147, 1999.

[95] Andrea Moro, Fabio Federici, Giacomo Valente, Luigi Pomante, Marco Faccio, and Vittoriano Muttillo. Hardware performance sniffers for embedded systems profiling. In *2015 12th International Workshop on Intelligent Solutions in Embedded Systems (WISES)*, pages 29–34. IEEE, 2015.

[96] Saba Mostofi, Setu Gupta, Ahmad Hassani, Krishnam Tibrewala, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. Light-weight cache replacement for instruction heavy workloads. In *Proceedings of the 52nd Annual International Symposium on Computer*

Architecture, ISCA '25, page 1005–1019, New York, NY, USA, 2025. Association for Computing Machinery.

[97] Shashidhar Mysore, Banit Agrawal, Navin Srivastava, Sheng-Chih Lin, Kaustav Banerjee, and Tim Sherwood. Introspective 3d chips. *SIGOPS Oper. Syst. Rev.*, 40(5):264–273, oct 2006.

[98] Satish Narayanasamy, Timothy Sherwood, Suleyman Sair, Brad Calder, and George Varghese. Catching accurate profiles in hardware. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 269–280. IEEE, 2003.

[99] Arash Nasr-Esfahany, Mohammad Alizadeh, Victor Lee, Hanna Alam, Brett W. Coon, David Culler, Vidushi Dadu, Martin Dixon, Henry M. Levy, Santosh Pandey, Parthasarathy Ranganathan, and Amir Yazdanbakhsh. Concorde: Fast and accurate cpu performance modeling with compositional analytical-ml fusion. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ISCA '25, page 1480–1494, New York, NY, USA, 2025. Association for Computing Machinery.

[100] Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibáñez, Víctor Viñals-Yúfera, and Alberto Ros. Berti: an accurate local-delta data prefetcher. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 975–991, 2022.

[101] Richard Neill, Andi Drebes, and Antoniu Pop. Fuse: Accurate multiplexing of hardware performance counters across executions. *ACM Trans. Archit. Code Optim.*, 14(4), December 2017.

[102] Hao Nguyen and Vertical Research Group. microbench. `https://github.com/darchr/microbench`, 2019.

[103] Junaid Nomani and Jakub Szefer. Predicting program phases and defending against side-channel attacks using hardware performance counters. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '15, New York, NY, USA, 2015. Association for Computing Machinery.

[104] Diego Novillo. Samplepgo-the power of profile guided optimizations without the usability burden. In *2014 LLVM Compiler Infrastructure in HPC*, pages 22–28. IEEE, 2014.

[105] Tony Nowatzki and Karthikeyan Sankaralingam. Analyzing behavior specialized acceleration. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 697–711, New York, NY, USA, 2016. Association for Computing Machinery.

[106] NVIDIA. Geforce experience. `https://www.nvidia.com/en-us/geforce/geforce-experience/`.

[107] NVIDIA. Nvdla. `https://nvdla.org/`.

[108] NVIDIA. Nvidia ampere architecture in-depth.`https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/`.

[109] NVIDIA. Nvidia hopper architecture in-depth.`https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/`.

[110] NVIDIA. Overview of nvtx. `https://docs.nvidia.com/nvtx/overview/index.html`.

[111] NVIDIA Corporation. NVIDIA Hopper Architecture In-Depth. `https://resources.nvidia.com/en-us-gtc-2022/nvidia-hopper-architecture`, 2022. GTC 2022 Talk S41455.

[112] Kenneth O'neal, Philip Brisk, Ahmed Abousamra, Zack Waters, and Emily Shriver. Gpu performance estimation using software rasterization and machine learning. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–21, 2017.

[113] Chandandeep Singh Pabla. Completely fair scheduler. *Linux Journal*, 2009(184):4, 2009.

[114] Palladium emulation. high-performance hardware verification and debug of complex socs and systems. `https://www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/palladium.html`.

[115] Santosh Pandey, Amir Yazdanbakhsh, and Hang Liu. Tao: Re-thinking dl-based microarchitecture simulation. *Proc. ACM Meas. Anal. Comput. Syst.*, 8(2), May 2024.

[116] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. Marss: A full system simulator for multicore x86 cpus. In *Proceedings of the 48th Design Automation Conference*, pages 1050–1055, 2011.

[117] Dylan Patel and Afzal Ahmad. Tsmc's 3nm conundrum, does it even make sense? – n3 & n3e process technology & cost detailed. `https://www.semianalysis.com/p/tsmcs-3nm-conundrum-does-it-even`.

[118] Synopsys zebu emulation. the industry's fastest emulation systems `https://www.synopsys.com/verification/emulation.html`.

[119] Vinodha Ramasamy, Paul Yuan, Dehao Chen, and Robert Hundt. Feedback-directed optimizations in gcc with estimated edge profiles from hardware event sampling. In *Proceedings of GCC Summit*, pages 87–102. Citeseer, 2008.

[120] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, pages 65–79, 2010.

[121] Alberto Ros and Alexandra Jimborean. A cost-effective entangling prefetcher for instructions. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 99–111, 2021.

[122] Alberto Ros and Alexandra Jimborean. A cost-effective entangling prefetcher for instructions. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 99–111, 2021.

[123] Alberto Ros and Alexandra Jimborean. Wrong-path-aware entangling instruction prefetcher. *IEEE Transactions on Computers*, 73(2):548–559, 2024.

[124] Juan Carlos Saez, Adrian Pousa, Fernando Castro, Daniel Chaver, and Manuel Prieto-Matias. Acfs: a completely fair scheduler for asymmetric single-isa multicore systems. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, page 2027–2032, New York, NY, USA, 2015. Association for Computing Machinery.

[125] Berk Saglam, Nam Ho, Carlos Falquez, Antoni Portero, Fabian Schätzle, Estela Suarez, and Dirk Pleiter. Data prefetching on processors with heterogeneous memory. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '24, page 45–60, New York, NY, USA, 2024. Association for Computing Machinery.

[126] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer architecture news*, 41(3):475–486, 2013.

[127] S Subramanya Sastry, Rastislav Bodik, and James E Smith. Rapid profiling via stratified sampling. *ACM SIGARCH Computer Architecture News*, 29(2):278–289, 2001.

[128] Tobias Scheipel, Fabian Mauroner, and Marcel Baunach. System-aware performance monitoring unit for risc-v architectures. In *2017 Euromicro Conference on Digital System Design (DSD)*, pages 86–93. IEEE, 2017.

[129] Martin Schulz, Brian S. White, Sally A. McKee, Hsien-Hsin S. Lee, and Jürgen Jeitner. Owl: Next generation system monitoring. In *Proceedings of the 2nd Conference on Computing Frontiers*, CF '05, page 116–124, New York, NY, USA, 2005. Association for Computing Machinery.

[130] Minjun Seo and Roman Lysecky. Non-intrusive in-situ requirements monitoring of embedded system. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 23(5):1–27, 2018.

[131] Omer Berat Sezer and Jeffery Zhuang. Lstm and rnn tutorial with demo. `https://github.com/omerbsezer/LSTM_RNN_Tutorials_with_Demo`, 2018.

[132] Ishan Shah, Akanksha Jain, and Calvin Lin. Effective mimicry of belady's min policy. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 558–572, 2022.

[133] Lesley Shannon, Eric Matthews, Nicholas Doyle, and Alexandra Fedorova. Performance monitoring for multicore embedded computing systems on fpgas. *arXiv preprint arXiv:1508.07126*, 2015.

[134] Sudhanshu Shukla, Sumeet Bandishte, Jayesh Gaur, and Sreenivas Subramoney. Register file prefetching. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 410–423, New York, NY, USA, 2022. Association for Computing Machinery.

[135] Teja Singh, Sundar Rangarajan, Deepesh John, Russell Schreiber, Spence Oliver, Rajit Seahra, and Alex Schaefer. 2.1 zen 2: The amd 7nm energy-efficient high-performance x86-64 microprocessor core. In *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, pages 42–44, 2020.

[136] Splunk. Splunk alwayson profiling. `https://docs.splunk.com/observability/en/apm/profiling/intro-profiling.html`.

[137] Aaron Stillmaker and Bevan Baas. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration*, 58:74–81, June 2017.

[138] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th International Conference on Parallel Processing Workshops*, pages 207–216, 2010.

[139] Giacomo Valente, Tiziana Fanni, Carlo Sau, Tania Di Mascio, Luigi Pomante, and Francesca Palumbo. A composable monitoring system for heterogeneous embedded platforms. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5):1–34, 2021.

[140] Giacomo Valente, Vittoriano Muttillo, Luigi Pomante, Fabio Federici, Marco Faccio, Andrea Moro, Serenella Ferri, and Carlo Tieri. A flexible profiling sub-system for reconfigurable logic architectures. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 373–376. IEEE, 2016.

[141] Rangharajan Venkatesan, Yakun Sophia Shao, Miaorong Wang, Jason Clemons, Steve Dai, Matthew Fojtik, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. Magnet: A modular accelerator generator for neural networks. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[142] Shayne Wadle and Karthikeyan Sankaralingam. Sahm: State-aware heterogeneous multicore for single-thread performance, 2025.

[143] Shayne Wadle, Yanxin Zhang, Vikas Singh, and Karthikeyan Sankaralingam. Neuroscalar: A deep learning framework for fast, accurate, and in-the-wild cycle-level performance prediction, 2025.

[144] T Wang, J Blocki, N Li, and S Jha. Locally differentially private protocols for frequency estimation. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[145] Vincent M. Weaver and Sally A. McKee. Can hardware performance counters be trusted? In *2008 IEEE International Symposium on Workload Characterization*, pages 141–150, 2008.

[146] Paul E West, Yuval Peress, Gary S Tyson, and Sally A McKee. Core monitors: monitoring performance in multicore processors. In *Proceedings of the 6th ACM conference on Computing frontiers*, pages 31–40, 2009.

[147] Baptiste Wicht, Roberto A Vitillo, Dehao Chen, and David Levinthal. Hardware counted profile-guided optimization. *arXiv preprint arXiv:1411.6361*, 2014.

[148] Hao Wu, Krishnendra Nathella, Matthew Pabst, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Practical temporal prefetching with compressed on-chip metadata. *IEEE Transactions on Computers*, 71(11):2858–2871, 2022.

[149] Hao Wu, Krishnendra Nathella, Matthew Pabst, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Practical temporal prefetching with compressed on-chip metadata. *IEEE Transactions on Computers*, 71(11):2858–2871, 2022.

[150] Shien-Yang Wu, CH Chang, MC Chiang, CY Lin, JJ Liaw, JY Cheng, JY Yeh, HF Chen, SY Chang, KT Lai, et al. A 3nm cmos finflex™ platform technology with enhanced power efficiency and performance for mobile soc and high performance computing applications. In *2022 International Electron Devices Meeting (IEDM)*, pages 27–5. IEEE, 2022.

[151] Zhiyao Xie, Xiaoqing Xu, Matt Walker, Joshua Knebel, Kumaraguru Palaniswamy, Nicolas Hebert, Jiang Hu, Huanrui Yang, Yiran Chen, and Shidhartha Das. Apollo: An automated power modeling framework for runtime power introspection in high-volume commercial microprocessors. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 1–14, New York, NY, USA, 2021. Association for Computing Machinery.

[152] Geoffrey Yeap, S. S. Lin, Y. M. Chen, H. L. Shang, P. W. Wang, H. C. Lin, Y. C. Peng, J. Y. Sheu, M. Wang, X. Chen, B. R. Yang, C. P. Lin, F. C. Yang, Y. K. Leung, D. W. Lin, C. P. Chen, K. F. Yu, D. H. Chen, C. Y. Chang, H. K. Chen, P. Hung, C. S. Hou, Y. K. Cheng, J. Chang, L. Yuan, C. K. Lin, C. C. Chen, Y. C. Yeo, M. H. Tsai, H. T. Lin, C. O. Chui, K. B. Huang, W. Chang, H. J. Lin, K. W. Chen, R. Chen, S. H. Sun, Q. Fu, H. T. Yang, H. T. Chiang, C. C. Yeh, T. L. Lee, C. H. Wang, S. L. Shue, C. W. Wu, R. Lu, W. R. Lin, J. Wu, F. Lai, Y. H. Wu, B. Z. Tien, Y. C. Huang, L. C. Lu, Jun He, Y. Ku, J. Lin, M. Cao, T. S. Chang, and S. M. Jang. 5nm cmos production technology platform featuring full-fledged euv, and high mobility channel finfets with densest 0.021μm2 sram cells for mobile soc and high performance computing applications. In *2019 IEEE International Electron Devices Meeting (IEDM)*, pages 36.7.1–36.7.4, 2019.

[153] Teng Yu, Pavlos Petoumenos, Vladimir Janjic, Hugh Leather, and John Thomson. Colab: a collaborative multi-factor scheduler for asymmetric multicore processors. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '20, page 268–279, New York, NY, USA, 2020. Association for Computing Machinery.

[154] Siavash Zangeneh, Stephen Pruett, Sangkug Lym, and Yale N. Patt. Branchnet: A convolutional neural network to predict hard-to-predict branches. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 118–130, 2020.

[155] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. Accuracy of performance counter measurements. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 23–32, 2009.

[156] Ming Zhang, Xubin He, and Qing Yang. A unified, low-overhead framework to support continuous profiling and optimization. In *Conference Proceedings of the 2003*

IEEE International Performance, Computing, and Communications Conference, 2003., pages 327–334. IEEE, 2003.

[157] Han Zhao, Weihao Cui, Quan Chen, Jieru Zhao, Jingwen Leng, and Minyi Guo. Exploiting intra-sm parallelism in gpus via persistent and elastic blocks. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 290–298, 2021.

[158] Xinnian Zheng, Lizy K John, and Andreas Gerstlauer. Accurate phase-level cross-platform power and performance estimation. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[159] Xinnian Zheng, Pradeep Ravikumar, Lizy K John, and Andreas Gerstlauer. Learning-based analytical cross-platform performance prediction. In *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 52–59. IEEE, 2015.

[160] C.B. Zilles and G.S. Sohi. A programmable co-processor for profiling. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 241–252, 2001.

[161] Matej Špeťko, Ondřej Vysocký, Branislav Jansík, and Lubomír Říha. Dgx-a100 face to face dgx-2—performance, power and thermal behavior evaluation. *Energies*, 14(2), 2021.