Post-Silicon CPU Adaptation Made Practical Using Machine Learning

Stephen J. Tarsa, Rangeen Basu Roy Chowdhury, Julien Sebot, Gautham Chinya
Jayesh Gaur, Karthik Sankaranarayanan, Chit-Kwan Lin
Robert Chappell, Ronak Singhal, Hong Wang
{stephen.j.tarsa,rangeen.basu.roy.chowdhury,julien.sebot,gautham.n.chinya}@intel.com
Intel Corporation
Santa Clara, California

ABSTRACT

Processors that adapt architecture to workloads at runtime promise compelling performance per watt (PPW) gains, offering one way to mitigate diminishing returns from pipeline scaling. State-of-the-art adaptive CPUs deploy machine learning (ML) models on-chip to optimize hardware by recognizing workload patterns in event counter data. However, despite breakthrough PPW gains, such designs are not yet widely adopted due to the potential for systematic adaptation errors in the field.

This paper presents an adaptive CPU based on Intel SkyLake that (1) closes the loop to deployment, and (2) provides a novel mechanism for post-silicon customization. Our CPU performs *predictive cluster gating*, dynamically setting the issue width of a clustered architecture while clock-gating unused resources. Gating decisions are driven by ML adaptation models that execute on an existing microcontroller, minimizing design complexity and allowing performance characteristics to be adjusted with the ease of a firmware update. Crucially, we show that although adaptation models can suffer from *statistical blindspots* that risk degrading performance on new workloads, these can be reduced to minimal impact with careful design and training.

Our adaptive CPU improves PPW by 31.4% over a comparable non-adaptive CPU on SPEC2017, and exhibits two orders of magnitude fewer Service Level Agreement (SLA) violations than the state-of-the-art. We show how to optimize PPW using models trained to different SLAs or to specific applications, e.g. to improve datacenter hardware *in situ*. The resulting CPU meets real world deployment criteria for the first time and provides a new means to tailor hardware to individual customers, even as their needs change.

CCS CONCEPTS

• **Computer systems organization** → **Superscalar architectures**; • **Computing methodologies** → *Batch learning*; *Computational control.*

ISCA '19, June 22–26, 2019, PHOENIX, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322267

KEYWORDS

Adaptive hardware, machine learning, clustered architectures, runtime optimization

ACM Reference Format:
Stephen J. Tarsa, Rangeen Basu Roy Chowdhury, Julien Sebot, Gautham Chinya, Jayesh Gaur, Karthik Sankaranarayanan, Chit-Kwan Lin, and Robert Chappell, Ronak Singhal, Hong Wang. 2019. Post-Silicon CPU Adaptation Made Practical Using Machine Learning. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, PHOENIX, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3307650.3322267

1 INTRODUCTION

Demand for CPUs with higher performance per watt (PPW) is strong [43, 47], but slowing design productivity and growing workload diversity have made gains harder to achieve. For example, between 2003 and 2013, processor design complexity and team size tripled industry-wide, while project duration doubled [14]. Meanwhile, significant new classes of workloads appeared, including social networking, mobile apps, big data analytics, healthcare IoT, autonomous vehicles, and augmented reality. Scaling processor width and depth in this environment has produced diminishing returns, and fundamental new approaches to CPU design are needed.

Substantial optimization opportunities exist in the long tail of customer-specific workloads, but they are difficult to capture with a single static CPU. High-volume customers have turned to design-time customization, e.g., to improve PPW across data centers or differentiate consumer devices [33, 48]. However, a more sustainable and scalable approach is to make high-performance CPUs flexible enough to adapt to changing runtime needs in the field, after deployment.

Such adaptive architectures are well-studied in the literature [13], and include, e.g., tile-based clock gating [41], heterogeneous core scheduling [30], and pipeline lane gating [38]. Each design employs an *adaptation model* to recognize patterns in telemetry data, and to tune microarchitecture parameters to fine-grained workload demands. State-of-the-art models use machine learning (ML) to extract complex predictive patterns offline from example data, and then deploy the resulting predictor in hardware to adapt configuration at runtime. ML adaptation models have achieved breakthrough accuracy, issuing predictions at sub-100k instruction granularities to produce projected PPW gains on par with process technology scaling [15, 30, 41]. However, adaptive CPUs are not yet widely deployed, largely because they run the risk of unexpected behavior when workloads in the field generate unanticipated telemetry.

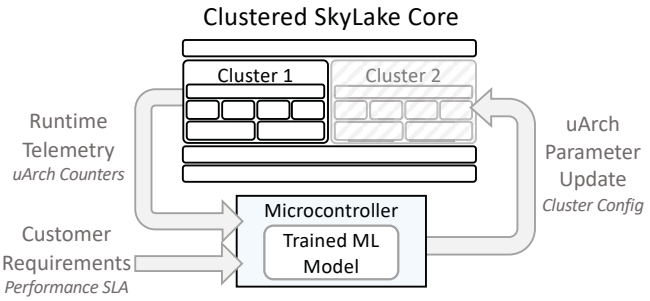

Figure 1: We implement machine learning adaptation models in microcontroller firmware to predictively set cluster configuration based on observed workload patterns.

A primary focus of this work is on explaining and removing this obstacle to adoption.

Our ML-driven adaptive CPU is a wide, out-of-order, superscalar machine derived from the Intel Skylake microarchitecture that implements *predictive cluster gating*. As shown in Figure 1, our design links three existing subsystems: a core with two out-of-order execution clusters, a telemetry subsystem, and a microcontroller with available computation budget. At runtime, the core either steers instructions to both clusters, or operates in a low-power mode that uses a single cluster and clock-gates the other. The telemetry system captures architecture event counters, which are snapshot on a regular interval and routed to an *existing* microcontroller. Meanwhile, powerful ML models run on this microcontroller to predict the best cluster configuration for upcoming runtime from telemetry data. This design implements adaptation at minimal incremental design complexity, while providing the flexibility to adjust CPU behavior post-silicon through a simple firmware update.

To show how our CPU meets real world deployment criteria, we demonstrate that statistical blindspots [3, 28] are the primary driver of service level agreement (SLA) violations. This is because adaptation errors that occur systematically, for example on new workloads, can cause CPU performance to degrade measurably, whereas spurious mistakes have comparatively imperceptible impact at the system level. We introduce training procedures that mitigate blindspots by quantifying the statistical diversity of training data, selecting counters that maximize telemetry information content, and screening models for those that perform most consistently on unseen workloads. Together, our training methods and architecture produce a clustered CPU that selects an optimal configuration every 40k instructions to improve PPW by 21.9-31.4% over a comparable non-adaptive core, at a $< 0.3\%$ rate of SLA violations. This represents a two orders of magnitude decrease in SLA violations over the best neural network adaptation model [41], and meets performance guarantees across all SPEC2017 applications with *higher* PPW gains.

The forward-looking advantages of our design extend beyond PPW improvements. Implementing adaptation models in firmware creates new *post-silicon* customization opportunities. For example, we show that it is possible to train and install models that meet different SLA guarantees to provide power and performance differentiation for the *same* physical chip design. We also demonstrate that general adaptation models can be further tuned to improve PPW for an individual application, a capability of particular interest in high-performance data centers where the same applications execute repeatedly across thousands of machines.

Our contributions are summarized as follows:

- We establish that a variety of ML adaptation models—e.g. neural networks and random forests—can generate predictions within the computation budgets of existing microcontrollers, providing powerful new optimization capabilities at zero incremental cost;
- We introduce interpretable metrics and model training procedures for measuring and eliminating model blindspots, which we identify as the main cause of post-deployment SLA violations;
- We present an ML-driven adaptive CPU that implements predictive cluster gating to improve PPW up to 31.4% on SPEC2017 at a $\leq 0.3\%$ rate of SLA violation, making deployment practical for the first time;
- We leverage post-silicon model customization to adjust CPU power and performance characteristics, boosting application-specific PPW by up-to 8.5% on certain applications.

2 BACKGROUND

2.1 Clustered Architectures

Clustered architectures scale execution pipeline width and depth, while keeping wire delays and component complexity low along critical timing paths. They achieve this by partitioning instruction queues, register files, and functional units into multiple clusters [5, 6, 10, 11, 17, 19, 24, 27, 35], and distributing instructions among them. By simplifying each component and communicating locally within a cluster when possible, clustering yields higher clock rates and more instructions per cycle (IPC) than monolithic scaling.

Our CPU implements *cluster gating*, a mechanism that dynamically adjusts the issue width of a clustered core by temporarily disabling and clock-gating clusters. When compute intensity is low, this reduces power dissipation without performance loss [4, 18, 21, 22]. This approach is attractive because toggling architecture parameters requires just a few cycles' transition time and produces a significant impact on PPW.

Cluster gating is one of many adaptation methods for optimizing PPW at different levels of granularity. Most recently, tile-based clock gating [41], microengine switching in heterogeneous big/little cores [30], and microarchitecture resource adaptation [15] have been proposed in the literature. Dynamic voltage and frequency scaling (DVFS) [25, 31, 32, 42, 44, 46] has also been applied at both system and core levels [25, 42], and we note that cluster gating is a complementary technique that can further reduce power at V_{min}.

2.2 Adaptation Models

All adaptive hardware implementations use runtime telemetry data such as architecture and microarchitecture event counts to inform configuration adjustments. The body of prior work has collectively articulated several requirements for these mechanisms. They must:

(1) **Adapt at a fine granularity**, operating at timescales below 100K instructions to capture the bulk of opportunities;

(2) **Predict, not react**, to configure hardware for *upcoming* workload demands rather than current demands, which may not remain the same;
(3) **Capture non-linear predictive relationships**, e.g., due to sudden transitions between distinctive workload phases;
(4) **Generalize** to produce desired system behaviors on workloads that were unknown at design time.

Existing adaptation models fall into several categories:

Heuristic models [4, 12, 29, 40, 44] exploit expert knowledge and intuition to analyze telemetry and derive adaptation rules. Due to the complexity of high-dimensional data analysis, these methods usually rely on a small number of event counters with known relationships. While they are inherently human-interpretable, heuristics are only effective at capturing coarse-grained adaptation opportunities, and have been supplanted by data-driven models.

First-order analytical models extrapolate current architecture states to predict upcoming states, e.g. using linear or polynomial regression [15]. However, since telemetry data includes non-Gaussian noise and discontinuities due to phase behaviors, the efficacy of prediction by extrapolation is limited.

Machine learning models can extract predictive signatures from noisy data with complex, compositional patterns [9, 23, 30, 38, 41]. For example, neural networks represent the current state-of-the-art in adaptation models [41]. Recent work in cache replacement [8] has shown that ML models implemented on a microcontroller can meet tight computation constraints to make timely microarchitectural decisions. We take a similar approach.

2.3 Machine Learning Adaptation Models

ML models operate in two modes: (1) *training*, in which model parameters are tuned to available data; and, (2) *inference*, in which trained models generate predictions on new data. Most training methods operate on batches of recorded data, so we assume that training is performed *offline*, after which model parameters do not change. On-chip, at runtime, models run in inference mode.

In this work, we develop ML adaptation models by (1) selecting appropriate telemetry data streams, (2) recording a training dataset over diverse workloads, (3) choosing and configuring an ML model, and (4) running a known training algorithm to tune model parameters. These procedures are explained in detail in Section 6. Once trained, we implement and optimize the model's inference procedure for our microcontroller. Examples and code sizes are shown in Section 5.

3 A PREDICTIVE CLUSTER GATING ARCHITECTURE

Our CPU is a scaled redesign of the Intel SkyLake architecture with two 4-wide clusters (Figure 2). The instruction cache is split per cluster, and each cluster is given its own Memory Execution Unit (MEU) and Execution Units (EUs). The CPU supports two modes: a *high-performance mode* offering 8-wide execution by steering instructions to both clusters (resolving source-sink dependencies via an inter-cluster communication channel); and a *low-power* mode with one cluster active and the other gated.

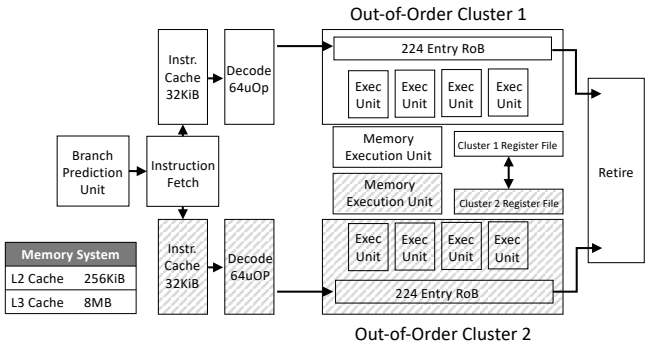

Figure 2: Our CPU is a scaled Intel SkyLake core with two out-of-order clusters. With both enabled, it operates in high-performance mode. With Cluster 2 gated, it operates in low-power mode, consuming 35% less power.

The CPU implements *cluster gating*, and can toggle modes at a configurable interval of 10-100k instructions. To switch from high-performance to low-power mode, the scheduler is first notified to stop steering instructions to Cluster 2. A custom microcode flow then copies register states from Cluster 2 to Cluster 1. Finally, Cluster 2 is clock gated. Mode-switching requires one micro op to be inserted in Cluster 1's execution stream to transfer each register dependency, up-to 32 in the worst case. This means that configuration changes take in the low tens of cycles to complete, while execution continues on Cluster 1. Worst-case power and energy overheads are on the order of 0.1% when adapting at 10k-instruction granularity, though we find average overheads considerably lower, on the order of 0.01%. To return from low-power to high-performance mode, we need only ungate Cluster 2 and update the scheduler to steer instructions to both clusters, incurring negligible overhead.

While other ML-driven adaptative designs in the literature have relied on new, dedicated hardware to model telemetry data and make configuration decisions, we show that modern machine learning algorithms can be deployed to great effect *using existing resources, with minimal design or hardware overhead.* We capitalize on an available telemetry system and microcontroller, standard subsystems carried forward and updated from existing designs. The telemetry system is comprised of a set of architecture and microarchitecture event counters and routing circuitry, originally designed to expose runtime data for debugging and code optimization at a single on-chip convergence point. The microcontroller operates at 500 MIPS and supports integer and floating point instructions, but currently no vectorized instructions. We note that 50% of its cycles are safely available to generate adaptation predictions without interfering with existing real-time deadlines.

We present data from an in-house, cycle-accurate simulator and use the SkyLake power model of Haj-Yihia *et al.* [20], an event-based model that was trained using data from an Intel-proprietary power simulator. On average, low-power mode consumes 35% less power than high-performance mode.

3.1 Service Level Agreements

The goal of predictive cluster gating is to improve PPW without perceptible performance loss. To formalize this, we adhere to guarantees defined in a service level agreement (SLA) with customers. Throughout this paper, excepting Section 7.3 where we evaluate different SLAs, we target an SLA guarantee that the core in low-power mode must perform within $P_{SLA} = 90\%$ of the high-performance mode, measured in instructions per cycle (IPC) over a window of $T_{SLA} = 1$ms. We guarantee the SLA to 99%, i.e., allowing no more than one window in violation of the performance threshold for every 100 runtime windows measured. While the final CPU design will implement a fail-safe guardrail, we present all results assuming none; instead, we focus on minimizing SLA violations so that guardrails may be set as permissively as possible.

3.2 New Optimization Scenarios

Our CPU is designed for diverse customer use cases, and ML-driven adaptation both eases the complexity of meeting existing needs and creates new opportunities for hardware differentiation. For example, present-day CPUs are often customized at design time for specific high-volume customers [33, 48]; a data center leasing computation time to third parties may seek better PPW, while a telecommunications system with strict real-time deadlines will demand peak performance. Capitalizing on this diversity represents an enormous opportunity in general purpose computing, but increases design complexity and cost as team sizes and product lines grow to meet individualized needs. However, an adaptive CPU whose performance characteristics can be adjusted post-silicon with just a firmware patch significantly reduces the difficulty of customizing hardware for individual customers. We demonstrate this capability in Section 7.3 by updating adaptation models to optimize PPW under different SLA guarantees.

Our firmware-based approach also creates new opportunities for post-silicon and even post-deployment optimization. For example, that same data center may require peak performance when demand spikes during the holiday season, but otherwise optimize for PPW throughout the rest of the year to minimize total cost of ownership (TCO) [7]. Unlike static CPUs, or other adaptive CPUs that have model-specific circuitry, we can meet this need by installing different firmware models via existing, well-supported datacenter infrastructure management (DCIM) software [1, 2]. This greatly increases the granularity at which customers can optimize TCO, improving a critical metric for data center operators, and providing a major advantage over other CPUs.

This approach also supports a new way to adapt hardware to individual applications deployed at scale. In Section 7.3, we show that telemetry recorded from initial application executions can be used to retrain ML adaptation models, boosting PPW in subsequent executions on different input data. This result is not only technical—it suggests a novel optimization-as-a-service usage model: each CPU ships with firmware pre-trained on a large and diverse trace repository; customers can trace new applications they wish to further optimize on-site; these traces are replayed on real hardware to generate telemetry and labels for retraining; and finally, after retraining, the updated models are pushed back to the customer to boost PPW *in situ*.

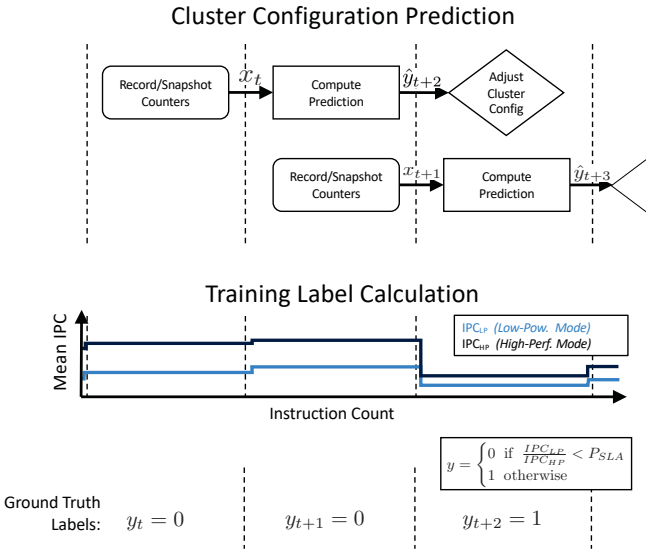

Figure 3: Telemetry is divided into intervals by instruction count. Counters from the first interval are used in the second to predict CPU configuration for the third. For training, ground truth labels are computed by comparing the IPC ratio of both configurations to an SLA threshold.

4 DATASETS, METRICS, & METHODS

4.1 Datasets

Our datasets capture counter and IPC values over a range of workloads. We define *workload* as an execution of an application on a unique input. As a workload runs, we record portions of its instruction stream in *traces* for later playback in a cycle-accurate simulator. This simulator outputs simulated telemetry, which we save as our dataset. In this work, we collect two such datasets, one for design-time training and validation, and another hold-out test set for evaluating our adaptive CPU after deployment.

The first dataset represents a large, diverse set of applications known by architects to be important to customers. We capture 2,648 traces of 593 client and server applications spanning many categories: high-performance computing and performance benchmarks; cloud and security; AI and data analytics; web browsers and productivity tools; multimedia; and, games, rendering, and augmented reality (Table 1). We denote this the high-diversity *training set* (HDTR). HDTR traces are significantly shorter than evaluation traces, and average 5M instructions each, recorded after warming caches and microarchitecture structures.

The second dataset mimics applications that are unknown during training but later encountered in the field. We use SPEC2017 to generate this *test set*, and ensure that no SPEC applications, including those from earlier releases (e.g., SPEC2006), are present in the HDTR set. SPEC2017 is a natural choice because it broadly stresses CPUs and is easy to extend with new workloads for most of its benchmarks (Table 2). We record our test applications for multiple inputs to characterize executions over time or across machines in the field. For each workload, we trace 200M-instruction SimPoints, after warming caches for 500M instructions and microarchitecture

structures for 5M instructions. Our test dataset is generated from 571 such SimPoint traces extracted from 118 workloads that span the 20 SPEC2017 applications.

To generate data, we simulate constituent traces in both high-performance and low-power modes, snapshotting IPC and telemetry values every 10k instructions. We then normalize counter values by the number of cycles in each interval, finding that it improves model accuracy. When coarser granularity is desired, we simply sum over successive intervals and re-normalize as appropriate. We initially record all 936 available event counters, but select just a subset of these for use with our adaptation models (see Section 6.2). In the end, for each mode and for each trace, we have a counter matrix $X = [x_1, \ldots, x_T]$, where x_t is a vector of the normalized counter values at interval t.

For each x_t we need a corresponding ground truth label y_{t+2} that specifies the best cluster configuration for interval $t+2$. Note that the ground truth label is for future interval $t+2$ rather than current interval t, allowing a full interval after counters are sent to the microcontroller to compute a prediction (see Figure 3). When building datasets, we set $y_{t+2} = 1$ by observing whether IPC in the low-power mode simulation meets the SLA performance threshold, indicating that Cluster 2 should be gated during interval $t+2$, and $y_{t+2} = 0$ otherwise. We ultimately train two models that operate alongside each other, one from the high-performance mode telemetry and the other from the low-power mode telemetry. At inference time, given a counter snapshot x_t and a flag indicating the CPU mode when that data was recorded, the appropriate model is used to produce a prediction $\hat{y}_{t+2}$.

4.2 Metrics

During development, evaluating adaptation models requires more than just measuring prediction accuracy, since the impact of a mistake depends on the cluster configuration chosen. For example, incorrect low-power mode predictions may lead to an SLA violation, while incorrect high-performance mode predictions represent a missed gating opportunity. Many metrics exist for comparing predictors; we use those that map easily to system behaviors so we can reason about how modeling choices affect CPU operation.

First, we categorize predictions by their correctness and the predicted cluster configuration. This defines true positive, false positive, true negative, and false negative metrics, as expressed below (where $\mathbb{1}$ is the indicator function):

	Correct	Incorrect
Low-power Mode	$(\mathbb{1}_{\hat{y}_t=y_t})y_t$ *True Positive*	$(\mathbb{1}_{\hat{y}_t\neq y_t})(1-y_t)$ *False Positive*
High-perf. Mode	$(\mathbb{1}_{\hat{y}_t=y_t})(1-y_t)$ *True Negative*	$(\mathbb{1}_{\hat{y}_t\neq y_t})y_t$ *False Negative*

Second, we track the **Percentage of Gating Opportunities Seized (PGOS)**, since these directly drive PPW gains. This metric, known as the *recall* or *true positive rate*, is defined in our context as:

$$\textbf{PGOS} = \frac{\sum_{t=1}^{T}(\mathbb{1}_{\hat{y}_t=y_t})y_t}{\sum_{t=1}^{T}\mathbb{1}_{y_t=1}} = \frac{\#\text{ correct low-power predictions}}{\#\text{ ground truth low-power intervals}} \quad (1)$$

Next, we introduce an SLA violation metric expressed in terms of prediction errors. We first compute whether predictions are likely to violate the SLA *in expectation*, based on a sample of size W:

$$E[(\mathbb{1}_{\hat{y}\neq y})(1-y)] = \frac{1}{W}\sum_{i=1}^{W}(\mathbb{1}_{\hat{y}_i\neq y_i})(1-y_i) \quad (2)$$

We say that a *model* is likely to violate the SLA for a measurement window of duration T_{SLA} when the expectation that its predictions will do so is > 50%. This threshold means that a randomly recorded IPC measurement is more likely to be found violating the SLA than not. We set $W = R * T_{SLA} * L$, where R is the peak instruction throughput and L the prediction interval (e.g., W = (16B Ins/s) $*$ (0.001s) $*$ (1 pred/10,000 Ins) = 1600 predictions). We then define an indicator that a model is likely to violate the SLA:

$$V = \begin{cases} 1 \text{ if } E[(\mathbb{1}_{\hat{y}\neq y})(1-y)] > 0.5 \\ 0 \text{ otherwise.} \end{cases} \quad (3)$$

We report a model's **Rate of SLA Violations (RSV)** over a set Ω of samples:

$$\textbf{RSV} = \frac{1}{|\Omega|}\sum_{\omega\in\Omega} V_\omega. \quad (4)$$

For our results, we compute RSV across the complete set of samples spanning a trace.

We employ RSV for two reasons. First, it directly reflects the customer's objective of minimizing *perceptible* performance degradation due to many false-positive gating decisions within a window of time. Second, it captures the effect of systematic mispredictions within a workload phase—periods of time during which repetitive execution behaviors yield statistically stationary telemetry data. Significant RSV values indicate that a model is consistently making mistakes on a particular region of the telemetry data distribution, reflecting a statistical *blindspot* [3, 28].

4.3 Methods

When evaluating trained models at development time, our goal is to assess how they will perform on future, unseen workloads in the field. We therefore use k-fold cross validation to characterize the distribution of possible model behaviors on workloads not present in the HDTR set.

k-fold cross validation partitions training data into tuning and validation sets. The tuning set is used to learn model parameters, while the validation set is held out and used to calculate performance metrics. Each distinct partitioning corresponds to a single *fold* of the training data; we do this for k folds, training and validating a model for each. Throughout this paper, we use randomized 80/20% partitions and $k = 32$ folds. From the resulting empirical distributions, we calculate averages and standard deviations to quantify expected values and variations for each metric, using these to guide design-time model selection.

Note that when partitioning the HDTR dataset, we assign all telemetry data and ground truth labels from one application to either a tuning or a validation set. This is done so that telemetry reflecting common sections of code does not appear in both sets, which would cause validation metrics to overestimate model performance on applications not available during development.

We also emphasize that we use cross validation statistics only at design-time, and *not* in our evaluation (Section 7.1), which measures performance on unseen applications post-deployment. There, we

Server Apps			Client Apps		
HPC & Perf. Benchmarks	Cloud & Security	AI & Analytics	Web & Productivity	Multi-media	Games, Rendering & Aug. Reality
176	75	34	171	80	57

Table 1: We train on 2,648 traces from 593 applications.

SPEC2017 Integer Benchmark	# App. Inputs *(Workloads)*	SPEC2017 Float Benchmark	# App. Inputs *(Workloads)*
600.perlbench_s	4	603.bwaves_s	5
602.gcc_s	7	607.cactuBSSN_s	6
605.mcf_s	7	619.lbm_s	3
620.omnetpp_s	9	621.wrf_s	1
623.xalancbmk_s	2	627.cam4_s	1
625.x264_s	12	628.pop2_s	1
631.deepsjeng_s	12	638.imagick_s	12
641.leela_s	10	644.nab_s	5
648.exchange2_s	5	649.fotonik3d_s	5
657.xz_s	5	654.roms_s	5

Table 2: Our test set consists of SPEC2017 benchmarks traced over multiple application inputs.

instead report metrics obtained directly by executing models on the SPEC2017, which is entirely held out during design.

5 MACHINE LEARNING INFERENCE IN FIRMWARE

Our microcontroller allows a great deal of flexibility to deploy different types of ML adaptation models. Here, we consider the inference computations of different model classes and show how they can fit within its safely-available computation budget (see Table 3; for performance reference, PGOS calculated according to Section 6.3 is included).

Multi-Layer Perceptrons (MLPs) are neural networks that stack multiple pattern matching layers, passing the output of one through an *activation* function to become the input to the next. During inference, a *linear* pattern-matching layer calculates the inner-product between a data vector and several filter vectors, whose coefficients reflect patterns learned from training data. Activation functions then damp or threshold this output to cull weakly expressed patterns, e.g., to denoise, before computing the next layer.

Inner products and activation functions (e.g., the Rectified Linear Unit (ReLU)) require standard integer and floating point operations, as shown in the example firmware code in Listing 1. When generating a prediction, the total number of these operations depends on the number of layers an MLP has and the number of filters per layer. Table 3 shows costs for different MLP configurations, including the state-of-the-art model of Ravi *et al.*, which feeds 8 telemetry counters to a one layer network with 10 filters.

Random Forests (RFs) perform inference by evaluating an ensemble of binary decision trees and taking a majority vote across them. Prior to training, an RF is configured by choosing the number of decision trees to include in the ensemble and the maximum depth of each tree. During tree traversal, an architecture counter is compared against a threshold at each tree node, directing traversal to the left or right. Traversal continues recursively until reaching a leaf node, which contains a prediction. Each node's specified counter and threshold are learned from data at training time.

Listing 2 shows example firmware code for evaluating one decision tree in an RF with depth 2. We hand-optimize firmware code to remove conditional branch instructions and to insert trivial comparisons to balance unbalanced trees. These steps ensure that inference executes efficiently on the microcontroller and that each prediction requires the same computational cost, simplifying budgeting. Table 3 shows that RFs require more storage than MLPs but have similar computation costs and perform best on average.

Logistic Regression (LR) uses a probabilistic model to separate data into two classes by fitting coefficients of a logistic function to training data. At inference time, an inner product between trained coefficients and a data vector yields a positive value if one class is predicted, and a negative value for the other. To produce a probability of class membership, the inner product score can be scaled by evaluating the exponentiation function. The computation requires a single inner product, but when probabilities are desired, exponentiation can introduce significant computation complexity (e.g., `exp()` in `math.h` requires up to 60 operations with 12 branches).

Support Vector Machines (SVMs) classify data by finding a hyperplane that maximizes the distance between classes of data, as defined by a *kernel* distance metric. Any valid distance metric can be used, permitting kernels designed for specific data domains. SVMs sidestep fitting an optimal separator by finding the training data points that sit closest to it—the so-called support vectors. At inference time, incoming data is compared to support vectors to predict a class.

We include SVMs in our analysis due to their popularity, but find insufficient accuracy to warrant implementing on our microcontroller. While the simplest linear kernel SVMs need just one inner product for inference, sophisticated kernels (e.g., χ^2) are cost prohibitive. Table 3 shows that, while χ^2 SVMs perform well, inference requires an order of magnitude more computations than the largest MLP we consider.

6 BLINDSPOT MITIGATION

Once deployed, the performance and PPW of our adaptive CPU will depend on how well adaptation models can meet SLA guarantees on any potential customer workload. We next show that careful training can minimize statistical blindspots and ensure that adaptation decisions generalize to workloads not available to designers.

6.1 Training Set Diversity

One source of adaptation errors is insufficient data diversity in the training set: when its statistics do not represent an application behavior encountered in the field, PGOS and RSV may behave unexpectedly. We show this is mitigated by increasing the number of distinct applications included in the tuning set, and that using hundreds of applications makes a significant impact. Here, we use a 3-layer MLP with 32, 32, and 16 filters per layer, respectively (this relatively large network factors out the impact of restricting network topology, which we analyze in Section 6.3). We train on

CPU: 2.0 GHz, 8-Wide, 16,000 MIPs Microcontroller: 500 MHz, 1-Wide, 500 MIPs		
Prediction Granularity (# CPU Inst)	Max Micro-controller Ops	Prediction Ops Budget
10k	312	156
20k	625	312
30k	937	468
40k	1,250	625
50k	1,562	781
60k	1,875	937
...		
100k	3,125	1,562

Model Class	Topology/Configuration	# Input Counters	# Ops. per Prediction	Memory Footprint	% Gating Opps. Seized
Multi Layer Perceptron	3 Layers, 32/32/16 Filters, ReLU Activation	12	6,162	640B	81.38%
Decision Tree	Max Depth 16	12	133	655.36KB	77.78%
Support Vector Machine	χ^2 Kernel, Max Support Vectors 1,000	12	121k	48.88KB	67.54%
Random Forest	16 Trees, Max Depth 8	12	1,074	40.48KB	66.67%
Random Forest	8 Trees, Max Depth 8	12	538	20.48KB	65.68%
Multi Layer Perceptron	3 Layers, 8/8/4 Filters, ReLU Activation	12	678	160B	60.99%
Multi Layer Perceptron	1 Layer, 10 Filters, ReLU Activation (∝Ravi *et al.*)	8	292	80B	57.90%
Support Vector Machine	Linear Kernel, 5 SVM Ensemble	12	412	484B	54.50%
Regression	Logistic	12	158	8B	38.33%

Table 3: The computation ratio between microcontroller and CPU is 1:32, and 50% of microcontroller cycles are available for inference. Random Forests give best results for fine-grained predictions.

```
float mlp_evalFilter(float *u, float *w, float b)
{
    float sum = b;

    // Inner product, e.g., for 4 inputs/counters:
    for (unsigned int i = 0; i < 4; i++)
      sum += u[i]*w[i];

    sum *= float(sum > 0.0f);   // ReLU

    return sum;
}
```

```
fld     dword ptr [ecx]         ;Inner product
fmul    dword ptr [eax]
fadd    dword ptr [esp + 12]
fld     dword ptr [ecx + 4]
fmul    dword ptr [eax + 4]
faddp   st(1)
fld     dword ptr [ecx + 8]
fmul    dword ptr [eax + 8]
faddp   st(1)
fld     dword ptr [ecx + 12]
fmul    dword ptr [eax + 12]
faddp   st(1)
fldz
fxch    st(1)                   ;ReLU
fucomi  st(1)
fld1
fxch    st(2)
fcmovnbe st(0), st(2)
fstp    st(2)
fmulp   st(1)
ret
```

Listing 1: Each MLP filter evaluation invokes just a short assembly routine, belying their strong statistical power.

low-power mode telemetry, which poses the harder prediction problem, and train MLPs using different tuning set sizes, ranging from 1 application to 440. Throughout, our validation set size is fixed to use 20% of HDTR applications.

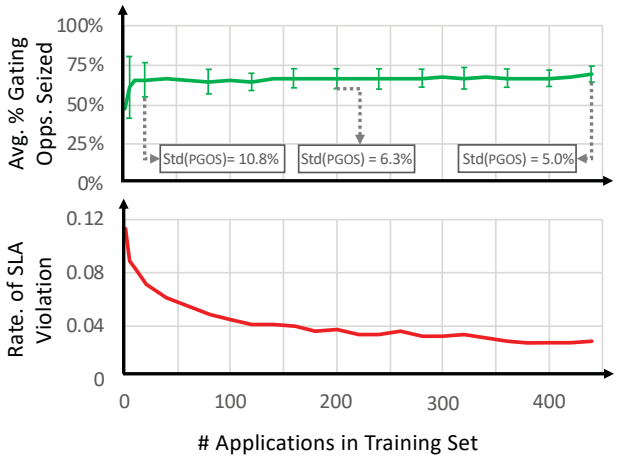

Figure 4: Training on a large number of applications mitigates model blindspots, reducing SLA violations.

Figure 4 shows two important results from this experiment. First, though a relatively small number of applications are needed to seize most of the gating opportunities—20 are sufficient—adding more applications drops the standard deviation of PGOS in half, from 10.8% at 20 applications to 6.3% at 200 applications, and eventually 5.0% at 440 applications. This shows that increasing training data diversity produces models that will yield more stable PPW across workloads. Second, we see a more pronounced effect on the reduction of RSV: scaling from 20 applications to 440 reduces SLA violations 2.5-fold, from 7.1% to 2.8%. In contrast to prior works that train on limited data (e.g., just the SPEC benchmark suite), we demonstrate here that at least hundreds of different applications are necessary to thoroughly defend against model blindspots.

```
bool RF_evalTree(float *curCtrs, RFNode *RF)
{
    int nodeIdx     = 0;
    int nmask       = 0;

    //e.g. Tree with depth 2:
    nmask = (curCtrs[RF[nodeIdx].featureId]
             < RF[nodeIdx].val);
    nodeIdx = nmask*RF[nodeIdx].idxLeft
             + (1-nmask)*RF[nodeIdx].idxRight;

    nmask = (curCtrs[RF[nodeIdx].featureId]
             < RF[nodeIdx].val);
    nodeIdx = nmask*RF[nodeIdx].idxLeft
             + (1-nmask)*RF[nodeIdx].idxRight;

    return (RF[nodeIdx].val>0);
}
```

```
mov     edx, dword ptr [ecx + 4]
lea     esi, [ecx + 12]
fld     dword ptr [eax + 4*edx]
fld     dword ptr [ecx]
fucompi st(1)
lea     edx, [ecx + 8]
fstp    st(0)
cmova   esi, edx
mov     edx, dword ptr [esi]
shl     edx, 4
mov     esi, dword ptr [ecx + edx + 4]
fld     dword ptr [eax + 4*esi]
fld     dword ptr [ecx + edx]
fucompi st(1)
lea     eax, [ecx + edx + 8]
lea     edx, [ecx + edx + 12]
fstp    st(0)
cmova   edx, eax
mov     eax, dword ptr [edx]
shl     eax, 4
fld     dword ptr [ecx + eax]
fldz
fxch    st(1)
fucompi st(1)
fstp    st(0)
seta    al
pop     esi
ret
```

Listing 2: Evaluating a decision tree in a random forest invokes a simple assembly routine.

6.2 Telemetry Information Content

A second source of systematic adaptation errors can arise from disparity in the predictive value of event counters across different applications. As an extreme example, differences in the number of instruction cache misses may provide a strong characterization of workload behavior when code footprint is large, but near zero information when footprint is small and entirely cached. We therefore take an information-theoretic approach to counter selection, and choose those counters that maximize *information content* throughout the HDTR dataset. This constrasts with prior works that select telemetry to maximize a specific model's accuracy. It also allows us to support different models without changing hardware, including models that support new core adaptations.

To choose counters, we collect all 936 available at design time, recording their data streams for the entire HDTR data set. We then apply two heuristic screens to cull low-information counters. First, we remove counters that often provide no information for substantial portions of runtime, flagging any that read '0' for more than 15% of a trace and removing those that are flagged in more than 5% of traces. Second, we remove the bottom 50% of counters by their standard deviations; these have the lowest signal-to-noise ratio, assuming Gaussian variations post-silicon. This procedure leaves 308 counters.

We next apply spectral clustering [45] to select a subset that maximizes information content. Our algorithm is based on the observation that many counters are redundant to each other—e.g., branch mispredictions and pipeline flushes—and maximizing information content corresponds to excluding redundant counters. We adapt the Perona-Freeman algorithm [37] (Alg. 1), an extension of Principle Component Analysis (PCA), to identify and rank groups of interchangeable counters. First, we compute a 308×308 matrix of pairwise counter covariance. We then compute its eigendecomposition and apply a threshold to screen for similar large magnitude coefficients in its second eigenvector. These coefficients correspond to a cluster of statistically interchangeable counters that has highest mutual information to the full counter dataset. We pick one counter, remove the group from data, and repeat the process until r counters are chosen. We call this method **PF Counter Selection**.

To show the effect of choosing counters for information content, we repeat the experiment from Section 6.1 with the tuning set size constant at 80% of HDTR applications and vary the number of counters from 2 to 32. First, Figure 5 shows that 8 counters is the minimum needed to consistently provide a high PGOS. This observation highlights a fundamental shortcoming of heuristic adaptation policies—that the minimum number of distinct telemetry streams needed for consistent performance exceeds that which can be manageably analyzed by experts. Second, when we compare to counters used in the state-of-the-art MLP adaptation model (see Section 7) to the 12 counters identified by PF Counter Selection, we find that maximizing information content drops RSV significantly, and reduces variation. On validation folds, our counters improve average RSV from 3.6% to 2.4% over model-specific counters, while standard deviation drops from 1.6% to 1.0%. In Section 7.1, we will see that RSV drops even more significantly on held-out SPEC2017 applications when we choose counters using PF Counter Selection. For the rest of this paper, we use these 12 counters, listed in Table 4.

6.3 Model Hyperparameters

Finally, we demonstrate how different model configurations affect how well adaptations generalize beyond training data. Using a high throughput screening approach [39], we repeat the same experiment as in prior sections, but hold data size and the counter set

Algorithm 1 Perona-Freeman Counter Selection

Require: $X \in \mathbb{R}^{936\times T}$ a matrix of counter data; r a desired number of counters; τ_s a similarity threshold.

function PFScreening(X)
  $R \leftarrow \{\}$
  $C \leftarrow 1 \ldots n$
  $X \leftarrow$ FilterLowActivityCounters(X)
  $\mu \leftarrow$ rowMean(X)
  $A \leftarrow (X - \mu 1'_m)$
  for $i \leftarrow 1$ to r **do**
    $E \leftarrow$ eig(AA^T)
    $R \leftarrow R \cup C_{\text{argmax}(|E_{*,2}|)}$
    $J \leftarrow \{j : |E_{j,2}|/|E_{R_i,2}|) > \tau\}$
    $A \leftarrow A_{\overline{J},\overline{J}}$
    $C \leftarrow C_{\overline{J}}$
  end for
return R
end function

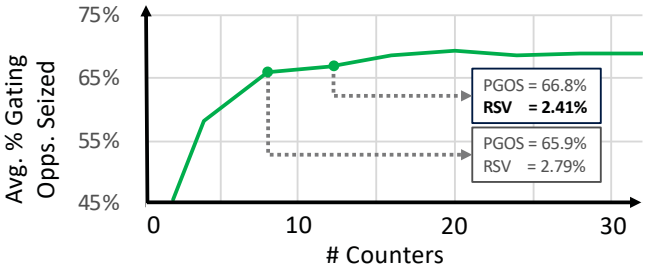

Figure 5: 12 telemetry counters seize most gating opportunities at minimum RSV on validation data.

Counter Name	
1. Micro Op Cache Misses	7. Physical Register Ref. Count
2. L2 Silent Evictions	8. Loads Retired
3. Wrong-Path uOps Flushed	9. L1 Data Cache Hits
4. Store Queue Occupancy	10. Micro Op Cache Hits
5. L1 Data Cache Reads	11. Micro Ops Stalled on Dep.
6. Stall Count	12. Micro Ops Ready

Table 4: PF Counter Selection identifies this set of 12 counters as having highest information content.

constant while varying MLP hyperparameters.[1] We evaluate networks with 1 to 3 linear layers, ReLU activations, and 4 to 32 filters per layer. After training each network, we adjust its sensitivity—the prediction threshold required to choose low-power mode—to keep SLA violations below 1.0% on the tuning set.

The full set of networks is shown in Figure 6 (top), where we plot average versus standard deviation of PGOS. Adding layers improves PGOS: most 3-layer networks (green) lie above the 1-layer networks (purple), and the networks with highest PGOS overall

[1] An MLP's filter weights are referred to as parameters, while settings such as the number and types of layers are "hyperparameters".

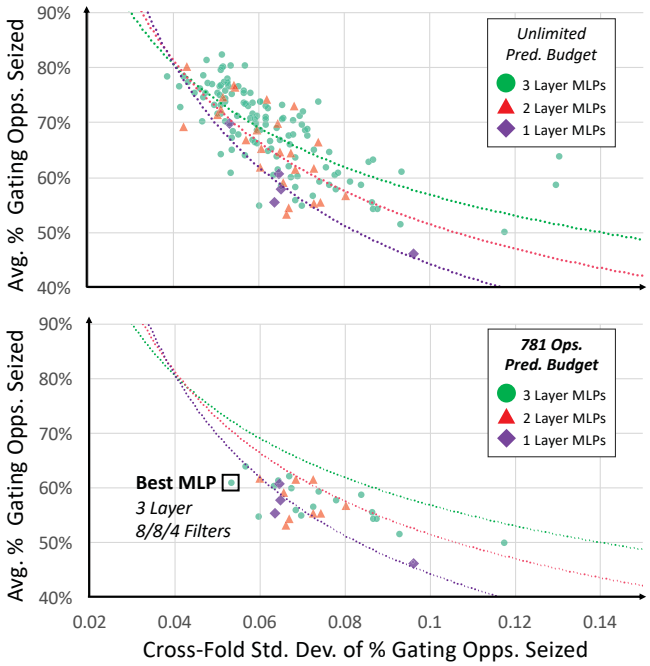

Figure 6: We choose MLP hyperparameters that minimize standard deviation in PGOS but maintain a high average to keep future variations on unseen data low.

have three layers. When we restrict models to fit the computation budget of a 50k-instruction prediction interval (Figure 6, bottom), 3-layer networks still have the lowest PGOS standard deviation. We observe the same trends in RSV.

From these results, we identify the best model hyperparameters by choosing those that minimize standard deviation but maintain a high average PGOS – here a three layer network with 8, 8, and 4 filters per layer, respectively. Searching the hyperparameter space is a general process for any ML model, and so for random forests, we search over the number of decision trees and decision tree depth, using the same criteria to choose a best RF with 8 trees of depth 8.

7 EVALUATION

Using the training procedures of Section 6 and enforcing the computation budgets of Section 5, we next compare our CPU when adaptations are driven by our models versus existing state-of-the-art models.

Ravi *et al.* propose **CHARSTAR** [41] , an architecture that uses a 10-filter single-layer MLP trained on eight counters to drive tile-based clock gating. Authors assume the MLP runs on a hardware neural network accelerator, and adapt the CPU every 10k instructions. As compared to, e.g., the linear regression model used in the Composite Cores architecture [30], this method gives superior accuracy and error characteristics.

We implement an equivalent neural network with the same topology for cluster gating. Three of CHARSTAR's input counters are specific to tile-based clock gating, so we select replacements using the analysis of Eyerman *et al.* [16], which informed the choice of

CHARSTAR's other five counters; the counters we use are: Branch Mispredictions, Instruction Cache Misses, Data Cache Misses, L2 Cache Misses, IPC, I-TLB Misses, D-TLB Misses, and Stall Count. We also observe better performance and more efficient firmware code using ReLU activations, and incorporate them accordingly. On our microcontroller, the resulting network requires 292 operations for inference, allowing us to predict on a 20k-instruction interval, since we assume no specialized hardware acceleration. We train via backpropagation, using an open source implementation of the Adam optimizer [26, 36].

Dubach *et al.* [15] propose a general framework for ML adaptation models to optimize architecture parameters that take on multiple values. This method encodes counter data as a histogram over a window of time, and trains by first sampling IPC and telemetry over a range of parameter values, and then fitting a softmax regression to predict the highest performing configuration. We call this **Softmax Regression on Counter Histograms (SRCH)**. When there are only two parameter values—e.g., gated and ungated—this model reduces to a logistic regression on counter histograms.

Per their method, we histogram each counter into 10 buckets and update histogram tallies by sampling counters every 10k instructions. Since the 15 counters used in their original work are not readily available in our telemetry system, we use the top 15 counters chosen by PF Counter Selection. Inference requires 572 ops, so to compare fairly we implement this method both at the 40k-instruction interval supported on our microcontroller, and its originally proposed 10M-instruction interval. We train by fitting a logistic regression using an open source implementation [36] of the L-BFGS algorithm.

Best MLP is our 3-layer neural network chosen in Section 6.3, with 8, 8, and 4 filters per layer respectively, fit to the 12 counters in Table 4. We train using the same open source implementation of Adam used for training the CHARSTAR network discussed above. Per Table 3, predictions require 678 ops, enabling us to gate on a 50k-instruction interval.

Best RF is our best Random Forest chosen in Section 6.3. It has 8 trees that have a maximum depth of 8, and takes the same set of 12 counters in Table 4 as input. We train using an open source implementation [36] of the CART algorithm that greedily grows trees by partitioning tuning samples into groups to minimize label entropy. Predictions require 538 ops (Table 3), so we gate on a 40k-instruction interval.

We train all models using the HDTR dataset and execute at the finest temporal granularity our microcontroller supports for each model, observing that this maximizes PPW as in the literature [34, 41]. We then deploy trained models and evaluate resulting CPU characteristics on our SPEC2017 test set. Figure 7 shows that, at baseline, SPEC2017 applications would ideally spend 45.7% of time in our CPU's low-power mode on average.

For reference, our 12-counter HDTR telemetry spans 2,648 traces of 593 applications and is 626MB. On an Intel 3.3GHz Core i9-7900X, Best-RF trains on one core in 9s, and Best-MLP in 87s.

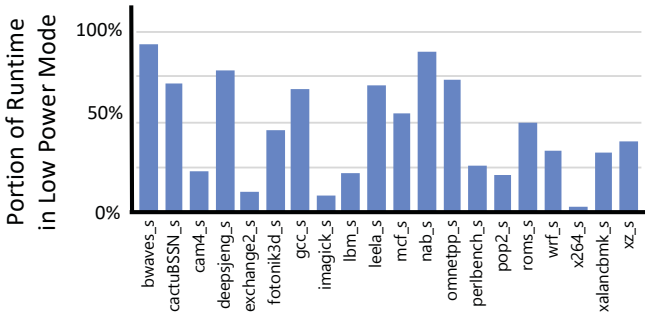

Figure 7: Across SPEC2017, applications ideally run in low power mode for 45.7% of runtime on average.

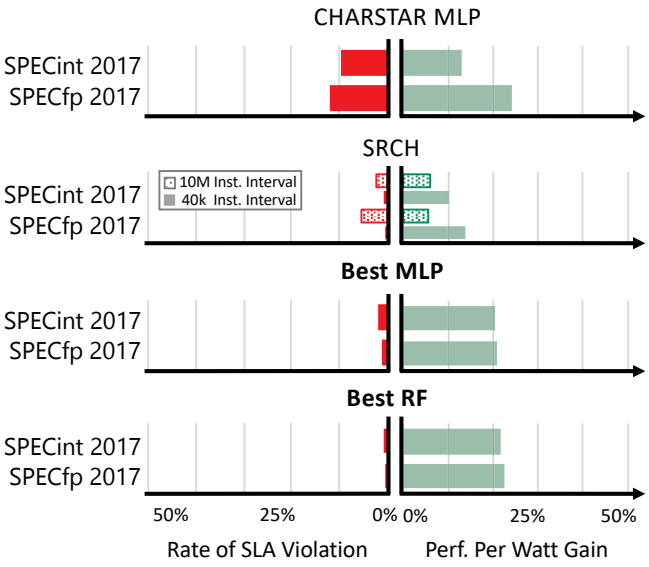

Figure 8: Compared with existing methods, our Best MLP and Best RF models further improve PPW and exhibit orders-of-magnitude lower RSV.

7.1 SPEC2017 Performance Per Watt

Figure 8 summarizes PPW gains and the observed RSV of predictive cluster gating when driven by these four models on SPEC2017. SRCH's original model is most conservative, improving PPW by 5.8% on average, at a 3.8% rate of SLA violations. When run at 40k-instruction granularity, its PPW gain doubles to 11.8% and RSV drops to 0.3%, demonstrating the importance of fine-grained adaptation. CHARSTAR's network is much more aggressive, and improves PPW by 18.4% on average, but runs a 10.9% RSV. In comparison, our Best MLP and Best RF models improve PPW *and* significantly reduce RSV. Best MLP improves PPW by 20.6% on average, at a 1.5% RSV. This represents an order of magnitude decrease in SLA violations over CHARSTAR with 2.2% greater PPW. Best RF performs even better, improving PPW by 21.9% at a 0.3% RSV, a further order-of-magnitude reduction in RSV. We also observe that both Best MLP and Best RF yield far more consistent PPW and RSV between the SPEC2017 int and fp suites.

Figure 9 breaks out these metrics for each SPEC2017 benchmark, showing CHARSTAR's neural network alongside Best RF, our best performing model. This data demonstrates the severe negative impact that model blindspots can have on some applications. Here, `roms_s` exhibits 77.8% RSV, meaning its users will experience marked performance degradation. Worse, we cannot know which applications fall into blindspots *a priori*, creating precisely the kind of unexpected behavior our training procedures seek to eliminate. In contrast, we observe *no* such behaviors for Best RF across all 20 benchmarks, while it nets 3.5% additional gain in PPW.

7.2 Evaluating Blindspot Mitigation

We next isolate the contribution of each technique in Section 6 to blindspot mitigation, building up step-by-step from CHARSTAR's MLP to our Best MLP. We first establish the value of increased training set diversity (Section 6.1) by comparing the baseline MLP when trained on only SPEC2017 data[2] versus the full HDTR training set. Figure 10 shows an average RSV of 16.5% when the baseline MLP is trained solely on SPEC2017 data. Training on the more diverse HDTR data reduces this by 5.6%, to 10.9% RSV.

Next, we measure the effect of maximizing telemetry information content (Section 6.2). By switching from the expert-chosen counters of Eyerman *et al.* to those chosen by our PF Counter Selection algorithm, we see a further 6.6% reduction in RSV to 4.3%.

Finally, we look at the impact of hyperparameter screening (Section 6.3), which produces a different topology (3-layer MLP) from the baseline (1-layer MLP). This further reduces RSV by another 3.1% to 1.2%. Combined, these techniques reduce RSV by 15.3%.

7.3 Post-Silicon Adjustments

Implementing adaptation models in firmware lets us adjust PPW post-silicon and thus serve the new target scenarios described in Section 3.2. This is accomplished by simply installing new models as firmware updates, for example via existing data center infrastructure management software [1, 2].

We first use this capability to improve PPW by modifying our SLA guarantee to be more permissive. With the same telemetry data, we revise ground truth labels to select low-power mode under two new SLA guarantees (P_{SLA} = 80% and P_{SLA} = 70%). We then retrain the Best RF model on the HDTR set and reevaluate the two resulting models on the SPEC2017 test set. Table 5 shows that reducing P_{SLA} to 80% improves PPW by another 6.3% to a 28.2%, at a 0.2% RSV. While the model sets a 10% lower minimum performance threshold, we see that *average* performance changes by only a small amount—just 2.4%. Setting P_{SLA} to 70% and training a third version of Best RF improves PPW gains to 31.4%, with no observed SLA violations. We underscore that *these three models effectively produce three CPUs with distinctive power and performance characteristics at zero additional design complexity or cost.*

Finally, we showcase a new capability by retraining adaptation models to boost PPW for a specific application using the multi-workload SPEC2017 dataset (Table 2). This is particularly attractive to data center customers who repeatedly execute applications across many thousands of machines. We capture application-specific telemetry in each cluster configuration over multiple inputs to the target application. We then retrain a new version of Best RF to P_{SLA} = 90% that combines the blindspot-mitigation techniques of Section 6 with application-specific training. We build this model by combining one RF (4 trees, 8 deep) trained on HDTR data, with a second RF (also 4 trees, 8 deep) trained on the target application to form a single RF (8 trees, 8 deep; as in Best RF). We find that combining high-diversity and application-specific trees reduces SLA violation rates significantly over just application-specific trees.

To measure PPW and RSV, we assume that models are deployed only for future workloads with different inputs. Considering only those applications with five or more workloads, and for which the general-use Best RF seized <95% of gating opportunities (ensuring sufficient headroom to measure improvement), we test on 11 applications in total. Given the relatively small number of workloads available per application, we report average behaviors using leave-one-out crossfold validation.[2] Table 6 shows that application-specific training improves PPW over high-diversity training in 8 of 11 applications, with gains of >2.9% for half of them, and up to 8.5% in the case of `fotonik3d_s`. Furthermore, we highlight that the PPW gain over a non-adaptive version of our scaled Skylake core is >50% for `nab_s` and `bwaves_s`.

Based on the analysis of Section 6.1, we expect these gains to grow and RSV to fall further when 100's of workloads are available for application-specific training. Presently, benchmark datasets of this size and diversity are unavailable, but in a datacenter with many thousands of machines, data collection at this scale is straightforward. We earmark building this dataset as important future work.

SLA Perf. Loss Tolerance	SLA Viol. Rate	Perf. per Watt Gain	Avg. Performance Relative to High Perf Mode
0.90	0.3%	21.9%	98.2%
0.80	0.2%	28.2%	95.8%
0.70	<0.1%	31.4%	93.4%

Table 5: Adaptation models can be trained post-silicon to tune CPU characteristics for customers with different SLAs.

	Perf. per Watt Gain			SLA Viol. Rate	
Benchmark	General RF	App. Specific RF		General RF	App. Specific RF
fotonik3d_s	**12.8%**	**21.3%**	**+8.5%**	**0.0%**	**0.0%**
bwaves_s	**47.6%**	**53.5%**	**+5.9%**	**0.2%**	**0.2%**
mcf_s	**25.2%**	**30.1%**	**+4.9%**	**0.0%**	**0.7%**
gcc_s	*29.4%*	*32.6%*	*+3.2%*	*0.4%*	*2.8%*
nab_s	**48.4%**	**51.3%**	**+2.9%**	**0.0%**	**0.0%**
cactuBSSN_s	*39.2%*	*41.4%*	*+2.2%*	*0.0%*	*1.8%*
x264_s	*0.3%*	*1.0%*	*+0.7%*	*0.0%*	*1.6%*
omnetpp_s	41.4%	42.0%	+0.6%	0.0%	0.0%
imagick_s	2.2%	2.2%	–	0.0%	0.0%
roms_s	21.0%	20.9%	-0.1%	0.7%	0.2%
exchange2_s	4.7%	3.2%	-1.5%	0.0%	0.0%

Table 6: Retraining our Best RF model to be application-specific improves PPW for 8 out of 11 benchmarks.

[2]To train on SPEC2017 we apply leave-one-out cross-validation, assigning 19 applications to the tuning set, computing metrics on 1 held out application, and averaging over all 20 training folds.

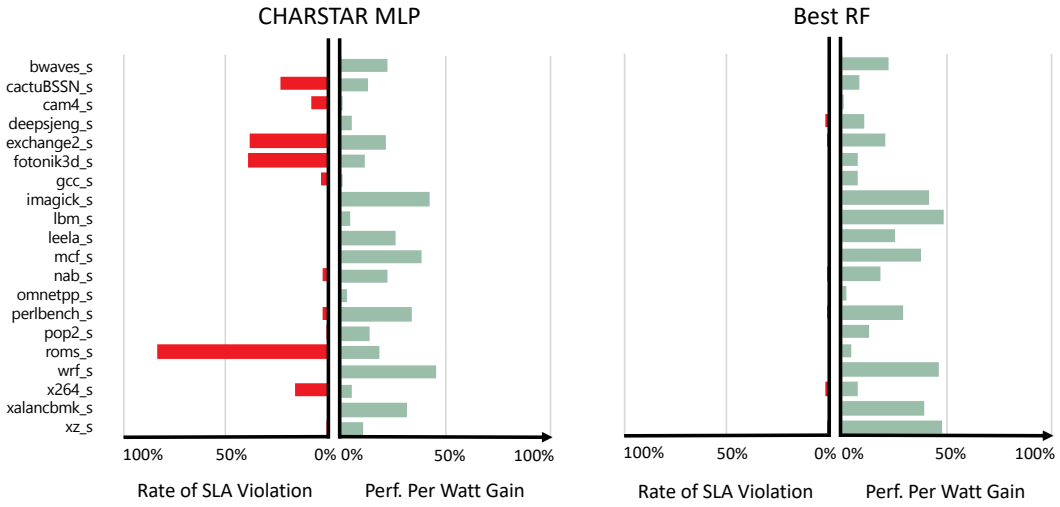

Figure 9: The MLP adaptation model introduced for CHARSTAR improves PPW by 18.4% on SPECint 2017, but with RSV as high as 77.8% on some applications. Our Best RF model improves PPW by 21.9% with RSV < 1% across the board, meeting all performance guarantees.

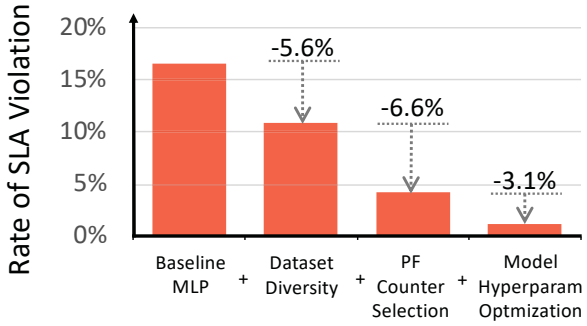

Figure 10: Increasing data diversity, choosing counters for information content, and optimizing model hyperparameters mitigates blindspots to reduce RSV by 15.3%.

8 CONCLUSION

This paper shows that adaptive CPU designs are ready for commercial deployment—the need to optimize hardware to long-tail application behaviors is acute, the gains from adaptation are compelling, and the technology is mature. Specifically, we showed that the risk of unexpected adaptation behaviors on new workloads is low when ML models are trained to mitigate statistical blindspots. We also showed that adaptation can be integrated into existing CPU designs at near-zero incremental cost. Finally, we showed that ML-driven adaptations not only address existing needs for better PPW, but also create new opportunities to extract value by customizing CPUs *post-silicon*. Taken together, this demonstrates that offline-trained ML adaptation is no longer an architecture technique of purely academic interest, but one that will be integrated into many future commercial processors.

REFERENCES

[1] [n. d.]. Dell PowerEdge Updates Best Practices Guide. https://tinyurl.com/y7ce6758.
[2] [n. d.]. Hewlett-Packard Enterprise SUM Best Practices Implementation Guide. https://tinyurl.com/y85tp8a6.
[3] Joshua Attenberg, Panos Ipeirotis, and Foster Provost. 2015. Beat the machine: Challenging humans to find a predictive model's âĂIJunknown unknownsâĂİ. *Journal of Data and Information Quality (JDIQ)* (2015).
[4] R Iris Bahar and Srilatha Manne. 2001. Power and energy reduction via pipeline balancing. In *ISCA*. IEEE.
[5] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H Albonesi. 2003. Dynamically managing the communication-parallelism trade-off in future clustered processors. In *ISCA*. IEEE.
[6] Amirali Baniasadi and Andreas Moshovos. 2000. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *MICRO*. IEEE.
[7] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture* (2013).
[8] Nathan Beckmann and Daniel Sanchez. 2017. Maximizing cache Perf. under uncertainty. In *HPCA*. IEEE.
[9] Ramazan Bitirgen, Engin Ipek, and Jose F Martinez. 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO*. IEEE Computer Society.
[10] Ramon Canal, J-M Parcerisa, and Antonio Gonzalez. 1999. A cost-effective clustered architecture. In *Parallel Architectures and Compilation Techniques*. IEEE.
[11] Ramon Canal, Joan-Manuel Parcerisa, and Antonio González. 2000. Dynamic cluster assignment mechanisms. In *HPCA*. IEEE.
[12] Pedro Chaparro, Jose Gonzalez, and Antonio Gonzalez. 2004. Thermal-aware clustered microarchitectures. In *Intrnl. Conf on Computer Design: VLSI in Computers and Processors*. IEEE.
[13] Rangeen Basu Roy Chowdhury, Anil K Kannepalli, Sungkwan Ku, and Eric Rotenberg. 2016. Anycore: A synthesizable rtl model for exploring and fabricating adaptive superscalar cores. In *ISPASS*. IEEE.
[14] Ryan Collett and Dorian Pyle. 2013. What Happens When Chip-Design Complexity Outpaces Productivity?
[15] Christophe Dubach, Timothy M Jones, Edwin V Bonilla, and Michael FP O'Boyle. 2010. A predictive model for dynamic microarchitectural adaptivity control. In *MICRO*. IEEE Computer Society.
[16] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E Smith. 2006. A Perf. counter architecture for computing accurate CPI components. In *SIGOPS Operating Systems Review*. ACM.
[17] Keith I Farkas, Paul Chow, Norman P Jouppi, and Zvonko Vranesic. 1997. The multicluster architecture: Reducing cycle time through partitioning. In *MICRO*. IEEE Computer Society.

[18] Soraya Ghiasi, Jason Casmira, and Dirk Grunwald. 2000. Using IPC variation in workloads with externally specified rates to reduce power consumption. In *In Workshop on Complexity Effective Design*.
[19] Linley Gwennap et al. 1996. Digital 21264 sets new standard. *Microprocessor report* (1996).
[20] Jawad Haj-Yihia, Ahmad Yasin, Yosi Ben Asher, and Avi Mendelson. 2016. Fine-grain power breakdown of modern out-of-order cores and its implications on skylake-based systems. *Trans. on Arch. and Code Opt. (TACO)* (2016).
[21] Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, and Pradip Bose. 2004. Microarchitectural techniques for power gating of execution units. In *Symposium on Low power electronics and design*. ACM.
[22] Michael C Huang, Jose Renau, and Josep Torrellas. 2003. Positional adaptation of processors: application to energy reduction. In *ISCA*. IEEE.
[23] Engin Ipek, Onur Mutlu, José F Martínez, and Rich Caruana. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *SIGARCH Computer Architecture News*. IEEE Computer Society.
[24] Richard E Kessler. 1999. The alpha 21264 microprocessor. (1999).
[25] Wonyoung Kim, Meeta S Gupta, Gu-Yeon Wei, and David Brooks. 2008. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *HPCA, 2008*. IEEE.
[26] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[27] Christos Kozyrakis and David Patterson. 2003. Overcoming the limitations of conventional vector processors. *ACM SIGARCH Computer Architecture News* (2003).
[28] Himabindu Lakkaraju, Ece Kamar, Rich Caruana, and Eric Horvitz. 2017. Identifying Unknown Unknowns in the Open World: Representations and Policies for Guided Exploration.. In *AAAI*.
[29] Hai Li, Swarup Bhunia, Yiran Chen, TN Vijaykumar, and Kaushik Roy. 2003. Deterministic clock gating for microprocessor power reduction. In *HPCA*. IEEE.
[30] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M Sleiman, Ronald Dreslinski, Thomas F Wenisch, and Scott Mahlke. 2012. Composite cores: Pushing heterogeneity into a core. In *MICRO*. IEEE Computer Society.
[31] Peter Macken, Marc Degrauwe, Mark Van Paemel, and Henri Oguey. 1990. A voltage reduction technique for digital systems. In *Solid-State Circuits Conf.* IEEE.
[32] Diana Marculescu. 2000. On the use of microarchitecture-driven dynamic voltage scaling. In *Workshop on Complexity-Effective Design*.
[33] Cade Metz. 2012. Ultimate Silicon Valley Perk: Custom Chips from Intel and AMD. Available at https://www.wired.com/2012/09/intel-amd-custom-chips/.
[34] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. 2013. Trace based phase prediction for tightly-coupled heterogeneous cores. In *MICRO*. ACM.
[35] S Palacharla, NP Jouppi, and JE Smith. 1997. Complexity-Effective Superscalar Processors. In *ISCA*. IEEE.
[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* (2011).
[37] Pietro Perona and William Freeman. 1998. A factorization approach to grouping. In *European Conf on Computer Vision*. Springer.
[38] Paula Petrica, Adam M Izraelevitz, David H Albonesi, and Christine A Shoemaker. 2013. Flicker: A dynamically adaptive architecture for power limited multicore systems. In *SIGARCH computer architecture news*. ACM.
[39] Nicolas Pinto, David Doukhan, James J DiCarlo, and David D Cox. 2009. A high-throughput screening approach to discovering good forms of biologically inspired visual representation. *PLoS computational biology* (2009).
[40] Dmitry Ponomarev, Gurhan Kucuk, and Kanad Ghose. 2006. Dynamic resizing of superscalar datapath components for energy efficiency. *IEEE Trans. on Computers* (2006).
[41] Gokul Subramanian Ravi and Mikko H Lipasti. 2017. CHARSTAR: Clock hierarchy aware resource scaling in tiled architectures. *SIGARCH Computer Architecture News* (2017).
[42] Tajana Simunic, Luca Benini, Andrea Acquaviva, Peter Glynn, and Giovanni De Micheli. 2001. Dynamic voltage scaling and power management for portable systems. In *Design Automation Conf*. ACM.
[43] Leendert van Dorn. 2017. Enabling cloud workloads through innovations in Silicon. Available at https://azure.microsoft.com/en-us/blog/.
[44] Augusto Vega, Alper Buyuktosunoglu, Heather Hanson, Pradip Bose, and Srinivasan Ramani. 2013. Crank it up or dial it down: coordinated multiprocessor frequency and folding control. In *MICRO*. IEEE.
[45] Yair Weiss. 1999. Segmentation using eigenvectors: a unifying view. In *Intrnl. Conf on Computer vision*. IEEE.
[46] Qiang Wu, Philo Juang, Margaret Martonosi, and Douglas W Clark. 2005. Voltage and frequency control with adaptive reaction time in multiple-clock-domain processors. In *HPCA, 2005*. IEEE.
[47] Hanan Youssef, Sami Iqram, and Scott Van Woudenberg. 2017. Compute Engine updates bring Skylake GA, extended memory and more VM flexibility. Available at https://cloud.google.com/blog/products/.
[48] Jason Zander. 2015. Building the Intelligent Cloud: Announcing New Azure Innovations to Transform Business. Available at https://azure.microsoft.com/en-us/blog/.