Characterizing Machine Learning-based Runtime Prefetcher Selection

Erika S. Alcorta*†, Mahesh Madhav†, Richard Afoakwa†, Scott Tetrick†,
Neeraja J. Yadwadkar*‡, Andreas Gerstlauer*
*The University of Texas at Austin. †Ampere Computing. ‡VMWare Research.

***Abstract*—Modern computer designs support composite prefetching, where multiple prefetcher components are used to target different memory access patterns. However, multiple prefetchers competing for resources can sometimes hurt performance, especially in many-core systems where cache and other resources are limited. Recent work has proposed mitigating this issue by selectively enabling and disabling prefetcher components at runtime. Formulating the problem with machine learning (ML) methods is promising, but efficient and effective solutions in terms of cost and performance are not well understood. This work studies fundamental characteristics of the composite prefetcher selection problem through the lens of ML to inform future prefetcher selection designs. We show that prefetcher decisions do not have significant temporal dependencies, that a phase-based rather than sample-based definition of ground truth yields patterns that are easier to learn, and that prefetcher selection can be formulated as a workload-agnostic problem requiring little to no training at runtime.**

I. Introduction

Modern processors combine multiple prefetcher components that cover a wider range of memory access patterns than monolithic prefetchers [2], [7], [8]. This, however, increases the number of prefetches that compete with each other, which can lead to contention and pollution of shared resources like memory bandwidth and cache space [4], [8]. This is especially critical in multi-core systems, where enabling prefetching can sometimes hurt performance depending on the workload [5]. Consequently, modern processors offer users the ability to enable/disable prefetcher components through registers [3], [8], but determining the optimal prefetcher configuration for any application is a challenging task.

Previous research has explored various techniques for tuning prefetcher component selection at runtime using system measurements typically obtained from hardware counters. Some studies use heuristics to repeatedly iterate over and rank prefetcher configurations during program execution [6], [8], [9]. However, exhaustively exploring configurations at runtime to make decisions misses performance opportunities and is not scalable [4]. More recently, the use of machine learning (ML) methods has shown promise [2]–[4]. However, these studies lack an in-depth analysis to show whether their formulations are efficient and well-suited for this problem. Additionally, proposed models, e.g. using deep reinforcement learning, are expensive to deploy on a real-world platform.

In this work, we investigate fundamental characteristics of ML-based techniques to manage prefetcher selection on a state-of-the-art many-core SoC. Our goal is to provide data-driven insights that help designers find the right ML formulation for their specific needs. Results of our study can inform the design of novel prefetcher selection frameworks, e.g., in terms of guiding the formulation of the problem as either supervised or reinforcement learning.

TABLE I: List of workloads.

Benchmark suite	Workload names
SPEC CPU 2017 (multi-programmed)	perlbench (S0), gcc (S1), mcf (S2), omnetpp (S3), xalancbmk (S4), x264 (S5), deepsjeng (S6), leela (S7), exchange2 (S8), xz (S9)
DaCapo (multi-threaded)	avrora (D0), biojava (D1), cassandra (D2), graphchi (D3), h2 (D4), h2o (D5), lusearch (D6), tomcat (D7), xalan (D8)
Renaissance (multi-threaded)	als (R0), finagle-http (R1), fj-kmeans (R2), gauss-mix (R3), naive-bayes (R4), stm-bench7 (R5), scrabble (R6)

TABLE II: List of hardware counter features.

Instr. per cycle (IPC)	Cache miss per 1k instr.
Branch miss per 1k instr.	L2I refills to branch miss ratio
Mem. acc. per 1k instr.	Pref. refills to reqs. ratio
Cache miss to mem. acc. ratio	L2D refills to miss ratio
Pref. refills ratio	Pref. refills to cache miss ratio
Pref. reqs. to mem. acc. ratio	

We study *temporal causality*, *selection decision frequency*, and *policy generalization*. The characterization of *temporal causality* measures the duration of the performance impact of a prefetcher selection decision. This is important to study since the complexity of the problem increases with longer temporal dependencies. We further study the optimal prefetcher *selection decision frequency*. We analyze the trade-off between changing selection decisions rapidly to maximize performance and finding stable behaviors that are easier to learn. Lastly, the study of *policy generalization* investigates whether the problem is workload-agnostic, i.e., a model's expected performance is consistent for any unseen workload or whether a selection model needs updates during runtime as workloads change.

II. Experimental Setup

1) System: Our baseline system is AmpereOne™, a state-of-the-art cloud-scale many-core platform that is representative of the high-end server SoCs available in the market in 2023. All modern server CPUs are built with multiple prefetchers, along with controls that allow them to be enabled and disabled. Our platform has 160 cores, and runs Fedora Linux 36 for aarch64. Each core has an L2 prefetcher subsystem that consist of four distinct prefetchers: a Best Offset Prefetcher (BOP), a Second-Best Offset Prefetcher (SBOP), a Next Line Prefetcher (NLP), and a Spatial or Adjacent-Sector Prefetcher (SP). All prefetchers can be enabled independently except SBOP, which can only be enabled if BOP is enabled. As such, there are 12 valid prefetcher configurations. The default configuration on the platform is to only enable BOP.

2) Data Collection: We collected data from various multi-programmed and multi-threaded benchmark suites used in

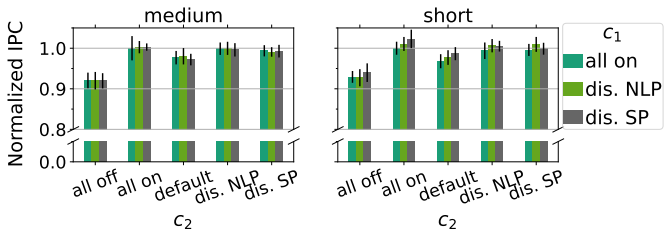

Fig. 1: Impact of the choice of prefetcher configuration c_1 in the first region of *500.perlbench* on the first 10 samples (left) and first sample (right) of the second region for different values of c_2.

cloud-scale performance evaluations to exercise multiple cores (see Table I). For each benchmark we collected one trace per prefetcher configuration. Each trace consists of 9 hardware counters sampled every 100 ms with Linux's *perf* tool. The absolute values of the hardware counters are transformed into 11 features, listed in Table II.

3) Problem Formulation: The goal of a prefetcher selection policy is to minimize the execution time of a workload. We reformulate this goal into smaller sub-goals that maximize the IPC of each sample, since IPC is easily available for offline explorations and correlates with execution time for single-threaded benchmarks. Note that for multi-threaded results, however, IPC is not necessarily indicative of observable performance differences. We use ρ_t to represent the IPC of a sample at time t. At each time step t, a prefetcher selection model should select an output that will be set in the next step, y_{t+1}, with the goal of maximizing the IPC of future samples $\rho_{t+k}, k \geq 1$. The output is a one-hot encoded vector, $y_{t+1} \in \{0,1\}^N$, where N is the number of prefetchers, and each element in the vector indicates whether the prefetcher should be on or off.

III. Characterization of Temporal Causality

We study the impact of prefetcher selection decisions on the performance of future samples. Formally, we consider that the selection y_{t+1} made at time t might not only influence the performance of the next sample, ρ_{t+1}, but also the performance in future samples, $\rho_{t+k}, k > 1$. This may be true if, for instance, the prefetched lines brought to the cache are used in more than one future sample. Therefore, it is important to design an ML formulation that considers the extent of the impact of prefetcher selection decisions across time.

Methodology: We partition a workload into two regions and assign a different prefetcher configuration to each region. We refer to the configuration used in the first and second regions as c_1 and c_2, respectively. We then explore whether the choice of c_1 impacts the short-, medium-, or long-term performance in the second region and how this impact varies with different values of c_2. Each combination is executed 10 times to reduce noise in our results. We analyze the short-, medium-, and long-term impact in the second region by measuring the performance of the first sample, the first 10 samples, or the full second region, respectively.

We show representative results using one benchmark, *500.perlbench*. We performed the same experiment on other benchmarks and observed overall similar results. The best static prefetcher configuration for *500.perlbench* is *all on*, followed by enable all except SP (*dis. SP*) and enable all except NLP (*dis. NLP*). We selected these three configurations for c_1, as it is likely that they will be selected by a good model. We also explored these three configurations in addition to *all off* and the platform's default configuration for c_2.

Results: Fig. 1 shows the medium- and short-term IPC results for each combination of c_1 and c_2 normalized to c_1 = *all on* and c_2 = *all on*. Results for long-term effects were very similar to medium-term effects; thus, we only show medium-term results in our plot. Each group of bars shows how the choice of configuration c_1 in the first region affects medium- or short-term performance in the second region for different choices of c_2. Higher IPC variations within bar groups indicate a stronger dependency of second region performance on the choice of c_1, indicating a temporal impact of decisions. Medium-term results do not show any significant variations across c_1 values, while short-term measurements do. This suggests that there is only a short-term impact of the selection for the current sample on the performance of the immediately following sample. However, even this relationship is weak, and the best selection of c_2 is the same regardless of which configuration was selected for c_1.

Takeaway: *Given the current sample, the selection of a prefetcher does not depend on past history, suggesting that models can disregard temporal dependencies and make decisions looking at samples in isolation.*

IV. Characterization of Decision Frequency

We next study how often to adjust the prefetcher selection. Optimally selecting prefetcher configurations requires adapting to changes in workloads, which represent dynamic behaviors at different frequencies. Short-term effects result in changes on a sample-by-sample basis, whereas long-term effects occur due to phase changes and may last multiple samples [1]. This section explores trade-offs between per-sample and per-phase decision frequencies.

Selecting the prefetcher that achieves the highest performance in each sample yields the highest expected performance. However, if the model can only react to changes, its decisions will lag by at least one sample, and per-sample selections may lead to sub-optimal decisions when a workload exhibits high short-term variations. In the worst case, a model will always make wrong decisions when presented with a rapid change rate. Additionally, per-sample decisions may display complex relationships between inputs and outputs that are harder to discern for models.

By contrast, a model can only adjust the prefetcher configuration when a phase change occurs, where the configuration that yields the highest average performance throughout each phase is selected. Since phases are typically stable and last multiple samples, the change rate is lower and per-phase labels may present better separability (in terms of mapping inputs to outputs), making it easier for ML models to learn patterns. However, a per-phase selection frequency may miss short-term performance optimization opportunities.

Methodology: We characterize selection frequency by comparing the expected performance of phase- and sample-based decisions. To obtain estimations of such expected performance, we align the samples of the collected traces, compare the IPC

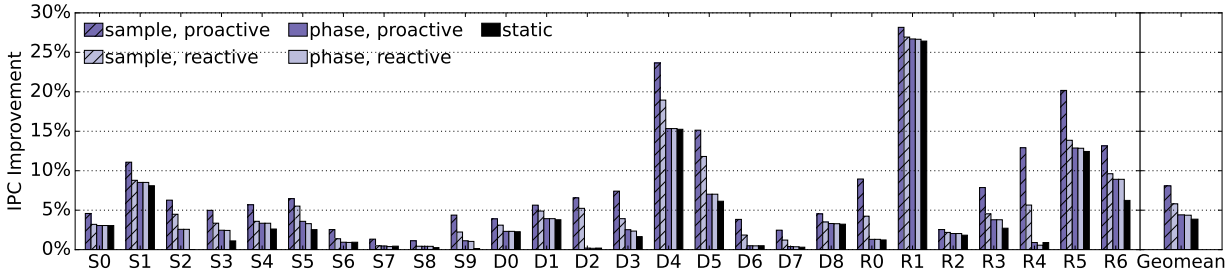

Fig. 2: Comparison of average IPC improvement of sample-based and phase-based proactive and reactive predictions.

of each sample to obtain the per-sample labels, and generate workload phases to obtain per-phase labels as follows:

1) Trace Alignment: The selection of prefetchers directly affects the workload's runtime and, hence, the length of different traces collected for the same workload. We align the samples of all traces from the same workload to be able to compare each sample's performance across prefetcher configurations. We align traces based on the number of executed instructions, such that all traces will have the same number of samples T', where all aligned samples with the same index t', $X^n_{t'}$ $\forall n \in 1,..,N$, can be compared to each other. We use superscript, n, to index a trace of the same workload but with different prefetcher configurations.

2) IPC Estimation: Once the traces are aligned, we compute the IPC of each sample in each aligned trace, $\rho^i_{t'}$. We can then generate an oracle prefetcher selection sequence by selecting the prefetcher configuration with the highest IPC for each sample, $\phi^*_{t'} = \max_i \rho^i_{t'}$. Aligned traces also provide a rapid way to estimate the expected performance of a prefetcher selection policy. Given a sequence of prefetcher selections for a workload, $\phi_{t'}$, we can estimate the average IPC of that sequence as $\bar{\rho} = \frac{1}{T'} \sum_{t'=1}^{T'} \rho^{\phi_{t'}}_{t'}$.

3) Phase Generation: We aim to find stable phases that maximize the average IPC. Recent work proposed filtering and clustering-based methods to identify stable workload phases [1]. This requires tuning multiple hyperparameters for: (1) selecting a clustering method between k-means or gaussian mixture models, (2) selecting the method's input features, (3) determining the filter size, and (4) determining the number of clusters. We formulated the tuning process as an optimization problem that selects the hyperparameters that maximize the average phase-based IPC, and used Monte-Carlo tree search to find these parameters.

4) Predictions: We study proactive and reactive predictors for both sample-based and phase-based selection frequencies. The proactive predictor assumes oracle knowledge about future workload behaviors, while the reactive predictor makes a future decision with only the information available at any given time. Specifically, at time t, the proactive predictor has information about X_{t+1} and can therefore select the best configuration for the next period, y_{t+1}. By contrast, a reactive predictor at time t only has information about X_t and assumes that the best selection for y_t is also the best for y_{t+1}.

Results: Fig. 2 illustrates the results of phase-based and sample-based selections compared to a static oracle that selects the best fixed prefetcher for each program, with all results normalized to the platform's default configuration. Results show that depending on diversity of workload behavior, dynamic prefetcher selection can improve IPC over the best static selection by up to 2x on average. Sample-based models generally perform better than phase-based ones since they can select the best configuration for each individual sample. The best-performing model is a proactive sample-based model with an average IPC improvement of 8%. However, this drops to 5.8% for a reactive sample-based model in which predictions lag behind the ground truth in every sample. By contrast, phase-based models achieve similar average IPC improvements of 4.3-4.4% in both their reactive and proactive configurations.

Takeaway: *A sample-by-sample selection approach yields higher performance, but to maximize its performance, an accurate prediction of future hardware counter values is needed. A phase-based approach can achieve comparable performance without the need for a model to predict future behavior.*

V. Characterization of Policy Generalization

When designing and deploying the prefetcher selection model on a real-world platform, it is important to measure its capability to generalize its policy (the action it takes given a set of inputs) to unseen samples and workloads. Studying generalizability helps to determine whether the model requires updates and training at runtime.

Methodology: We characterize policy generalization by designing workload-specific and workload-agnostic data sets and training various reactive ML models with each data set. To generate the data sets, we use a 70/30 split that partitions the data of each benchmark into train and test sets. Workload-specific models are trained with the training set of one workload and evaluated with the test set of that workload. Workload-agnostic models are trained with the training set from all but one workload and evaluated with the test set from the workload that was not used for training.

We chose to train decision trees (DT) for their simplicity and support vector machine classifiers (SVC) with a radial basis function kernel for their powerful input space separation abilities. We evaluated simple models with low overhead to meet the resource and latency constraints of runtime prefetcher selection deployments. Note that the specific models we used were chosen with the purpose of evaluating prefetcher selection generalizability and should not necessarily be taken as a final solution for this problem. We used three-fold cross-validation of the training sets for hyperparameter tuning. We explore decision tree depths between 1 and 30, and the SVC's regularization parameter with values between 10^{-5} and 10^5 increased in powers of 10.

Results: Fig. 3 shows the results normalized to the default prefetcher. On average, DT models perform better than

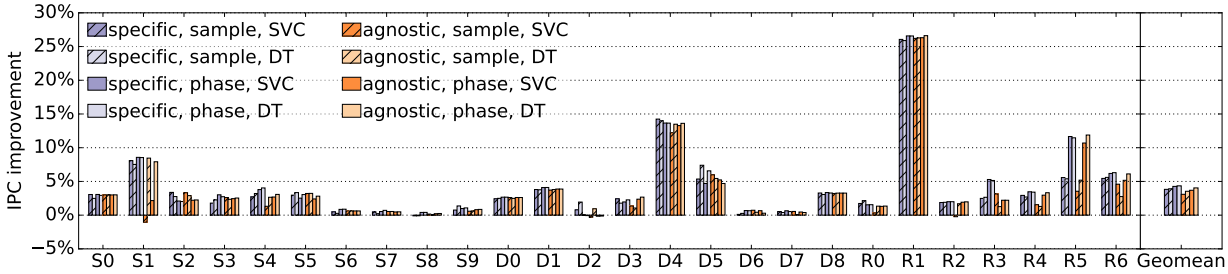

Fig. 3: Average IPC improvement of workload-specfic and workload-agnostic experiments with different models and selection frequencies.

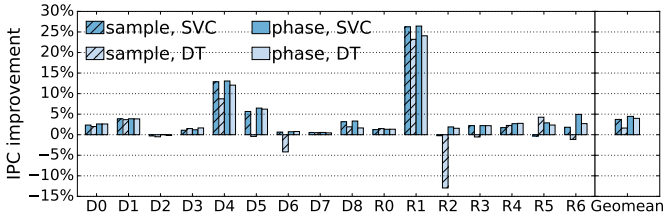

Fig. 4: Average IPC improvement of ML models trained with SPEC CPU workloads and tested on other suites.

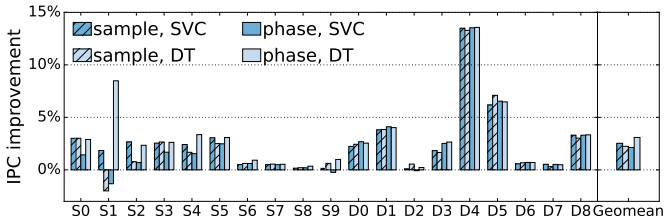

Fig. 5: Average IPC improvement of ML models trained with Renaissance workloads and tested on other suites.

SVC models. Comparing workload-specific and workload-agnostic DT models, only two out of the 26 benchmarks exhibited a performance loss of more than 1%. We observe that workload-agnostic models exhibit similar performance gains to workload-specific ones, suggesting that they can generalize what they learned to new workloads.

Contrary to the results shown in Sec. IV, models trained with phase-based labels generally yielded higher performance than those trained with sample-based labels. The average accuracy of sample-based and phase-based models is 45% and 85%, respectively. This suggests that sample-based models struggled to learn the per-sample patterns. Future work can investigate whether more complex ML models can achieve closer to expected sample-based performance.

We further evaluate the performance of a model trained with one benchmark suite and tested with workloads from other benchmark suites. This evaluates the ability of ML models to generalize what they learned across different application domains, where we consider each benchmark suite as a different domain. Similar to workload-agnostic models before, these models are also presented with unseen workloads during testing; the difference is that the models are now trained with less data. Fig. 4 and Fig. 5 show the results after training models with either the SPEC CPU or Renaissance suite and evaluating them on the DaCapo and Renaissance or SPEC CPU and DaCapo suites, respectively. When comparing phase-based to sample-based in each figure, we observe that there is no clear winner for all workloads. However, the best-performing models on average use a phase-based definition. This is consistent with earlier results. When comparing these suite-agnostic results to the workload-agnostic results in Fig. 3, we observe that their performance is very similar, with a few exceptions. In particular, *R5* had the largest drop in performance from more than 10% to less than 5%. This highlights the importance of training coverage when designing runtime prefetcher selection models.

Takeaway: *Given a statistically representative dataset, prefetcher selection models can generalize well to new workloads, allowing them to be trained offline without runtime adaptations.*

VI. Summary and Conclusions

We studied three fundamental characteristics of ML-based runtime prefetcher selection for many-core systems to help designers find the right ML formulation for their needs. We assessed the temporal causality of prefetcher selection and found no evidence to suggest that selections significantly impact future samples' performance. We further showed that a sample-based decision frequency has the most potential to improve performance if it has accurate information about future workload behavior, but that a phased-based formulation is easier to learn and can achieve better performance with simple reactive models, a trade-off that may be considered in future designs. Finally, our experiments comparing workload/suite-specific and -agnostic models showed that a prefetcher selection model may not need online updates when trained with a statistically representative dataset, and if updates are needed, they may be infrequent. Overall, our results suggest that prefetcher selection can be feasibly implemented at runtime with simple offline-trained supervised learning models, without the need for complex formulations. Future work includes deploying a corresponding prefetcher selection approach, and exploring more complex prefetchers.

References

[1] E. S. Alcorta and A. Gerstlauer. Learning-based Phase-aware Multi-core CPU Workload Forecasting. *ACM TODAES*, 28(2):23:1–23:27, 2022.
[2] F. Eris et al. Puppeteer: A random forest based manager for hardware prefetchers across the memory hierarchy. *ACM TACO*, 20(1), 2022.
[3] J. Hiebel et al. Machine Learning for Fine-Grained Hardware Prefetcher Control. In *ICPP*, 2019.
[4] M. Jalili and M. Erez. Managing Prefetchers With Deep Reinforcement Learning. *IEEE CAL*, 21(2):105–108, 2022.
[5] H. Kang and J. Wong. To hardware prefetch or not to prefetch? A virtualized environment study and core binding approach. In *ASPLOS*, 2013.
[6] M. Khan et al. AREP: Adaptive Resource Efficient Prefetching for Maximizing Multicore Performance. In *PAC*, 2015.
[7] S. Kondguli and M. Huang. Division of Labor: A More Effective Approach to Prefetching. In *ISCA*, 2018.
[8] C. Navarro et al. Bandwidth-Aware Dynamic Prefetch Configuration for IBM POWER8. *IEEE TPDS*, 31(8):1970–1982, 2020.
[9] C. Ortega et al. Intelligent Adaptation of Hardware Knobs for Improving Performance and Power Consumption. *IEEE TC*, 70(1):1–16, 2021.