Puppeteer: A Random Forest Based Manager for Hardware Prefetchers Across the Memory Hierarchy

FURKAN ERIS, MARCIA LOUIS, and KUBRA ERIS, Boston University
JOSÉ ABELLÁN, Universidad Católica de Murcia
AJAY JOSHI, Boston University

Over the years, processor throughput has steadily increased. However, the memory throughput has not increased at the same rate, which has led to the memory wall problem in turn increasing the gap between effective and theoretical peak processor performance. To cope with this, there has been an abundance of work in the area of data/instruction prefetcher designs. Broadly, prefetchers predict future data/instruction address accesses and proactively fetch data/instructions in the memory hierarchy with the goal of lowering data/instruction access latency. To this end, one or more prefetchers are deployed at each level of the memory hierarchy, but typically, each prefetcher gets designed in isolation without comprehensively accounting for other prefetchers in the system. As a result, individual prefetchers do not always complement each other, and that leads to lower average performance gains and/or many negative outliers. In this work, we propose Puppeteer, which is a hardware prefetcher manager that uses a suite of random forest regressors to determine at runtime which prefetcher should be ON at each level in the memory hierarchy, such that the prefetchers complement each other and we reduce the data/instruction access latency. Compared to a design with no prefetchers, using Puppeteer we improve IPC by 46.0% in 1 one-core, 25.8% in four-core, and 11.9% in eight-core processors on average across traces generated from SPEC2017, SPEC2006, and Cloud suites with ~11-KB overhead. Moreover, we also reduce the number of negative outliers by more than 89%, and the performance loss of the worst-case negative outlier from 25% to only 5% compared to the state of the art.

CCS Concepts: • **Computing methodologies** → **Classification and regression trees**; **Ensemble methods**; • **General and reference** → **Performance**; • **Computer systems organization** → *Architectures;*

Additional Key Words and Phrases: Prefetching, runtime management, machine learning

ACM Reference format:
Furkan Eris, Marcia Louis, Kubra Eris, José Abellán, and Ajay Joshi. 2022. Puppeteer: A Random Forest Based Manager for Hardware Prefetchers Across the Memory Hierarchy. *ACM Trans. Arch. Code Optim.* 20, 1, Article 19 (December 2022), 25 pages.
https://doi.org/10.1145/3570304

This work was supported by project grant PID2020-112827GB-I00 funded by MCIN/AEI/10.13039/501100011033.
Authors' addresses: F. Eris, S. Louis, K. Eris, and A. Joshi, ECE, Boston University, 340, 8 St. Mary's Street, Boston, MA 02215; emails: {fe, marcia93, kubra, joshi}@bu.edu; J. Abellán, CS, Universidad Católica de Murcia, Campus de Los Jerónimos, Av. de los Jerónimos, 135, Guadalupe, Murcia, Spain, 30107; email: jlabellan@ucam.edu.

1544-3566/2022/12-ART19 $15.00
https://doi.org/10.1145/3570304

1 INTRODUCTION

Instruction and data prefetching [18] are commonly used in today's processors to overcome the memory wall problem [70]. The key idea behind prefetching is identifying the current memory access pattern and predicting addresses to proactively fetch instructions and data into the cache to avoid cache misses.[1] Prefetching hides the large memory access latency and, in turn, improves processor performance. As a result, modern processors employ multiple prefetchers to cover a wide range of applications. Consequently, existing prefetchers do not improve the performance of all applications; in some cases they hurt application performance by prefetching the wrong memory addresses [36]. These incorrect prefetches use up the precious memory bandwidth and the limited space in the cache hierarchy. This increases data and instruction access latency, which hurts application performance.

To evaluate a prefetcher's performance, we can use scope and accuracy as the metrics [6, 34]. A prefetcher with high prefetching accuracy usually has limited scope—that is, it is very good at identifying a limited number of memory access patterns and can accurately prefetch data/instructions if those specific memory access patterns exist. However, such a prefetcher fails to identify other memory access patterns. Conversely, a prefetcher with broad scope caters to a wide variety of memory access patterns, but it has low accuracy.

Effectively, we need to find a balance between the accuracy and scope of the prefetcher. One way to balance scope and accuracy is to use multiple high accuracy prefetchers at each level of the memory hierarchy, as is the case in AMD and Intel processors [1–3, 7, 21–27, 65]. Each prefetcher is customized to identify a specific type of memory access pattern and make a prefetching prediction. However, having multiple prefetchers operating at each level in the memory hierarchy can lead to the following issues:

- Given that each prefetcher is trained independently to track a specific type of traffic and simultaneously share microarchitectural resources, prefetchers can sabotage each other during runtime. A prefetcher may trigger prefetch requests that evict cache lines that another prefetcher has accurately prefetched. This behavior leads to a loss in performance, wasted memory access bandwidth, and increased power consumption.
- Different prefetchers (either at the same level or across levels in the memory hierarchy) latch onto memory access patterns at different speeds. So a prefetcher's prediction can be influenced by the traffic generated by other prefetchers. These differences in temporal behavior can cause faulty synchronization among prefetchers and lead to a drop in application performance.

Essentially, prefetchers compete for resources and, at times, sabotage each other. To validate our argument, we use traces[2] generated from SPEC2017, SPEC2006, and Cloud benchmark suites and run these traces on an OoO processor that uses a different prefetcher at each level of the memory hierarchy (details of the particular evaluation methodology are provided in Section 4). We execute these traces using two state-of-the-art prefetcher solutions, IPCP [45] and EIP [55], which were the winners of prior prefetching competitions [49–51]. These prefetcher solutions use different prefetchers at each level of the memory hierarchy—that is, different **prefetcher system**

[1]This is a new article, not an extension of a conference paper.
[2]A trace is a group of instructions that represent a specific behavior. One or more traces can be used to represent the behavior of a benchmark. For example, a benchmark with consistent looping behavior can be represented by one trace corresponding to a single iteration of the loop. One or more unique representative traces are generated from each benchmark [48].

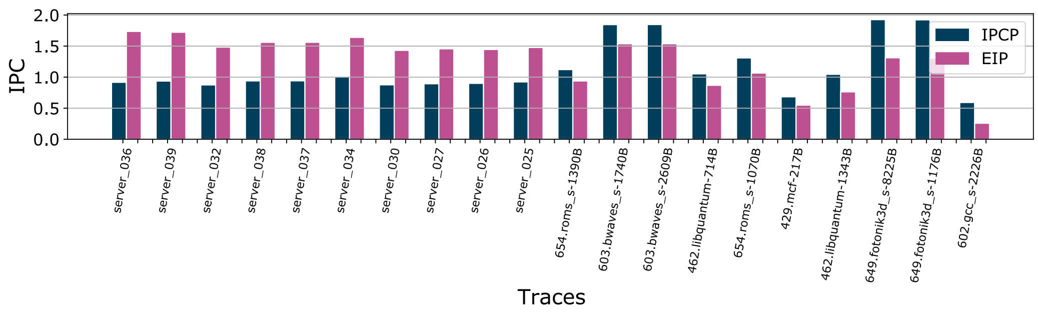


Fig. 1. Motivational example. Performance of a processor when using IPCP and EIP prefetchers. Here we show the 20 traces with the highest difference in performance out of 232 traces that we evaluated.

configurations (PSCs).[3] Both prefetcher solutions improve performance compared to the no prefetching case. However, IPCP (*no-ipcp-ipcp-nl*) (further details on the types of prefetchers are given later in Table 5) shows better performance over EIP in 100 out of 232 traces with the largest performance gain of 56% in `602.gcc_s-2226B`. EIP (*EIP-nl-spp-no*) shows better performance over IPCP in the remaining 132 traces and has the largest performance gain of 89% in `server_036`.

In Figure 1, we show a subset of traces that have a significant difference in their IPC when using IPCP and EIP prefetcher solutions. The standard methodology would have us select EIP because it has on average 7.8% performance gain over IPCP, but that would come at the cost of having a performance loss for 100 out of the 232 traces.

One way to address this problem is to have multiple different prefetchers at each cache level and switch ON a prefetcher based on the current program phase. For example, one prior work that has attempted to leverage multiple prefetchers in the *same* level of the memory hierarchy is by Kondugli and Huang [34]. The authors propose a "composite prefetcher,"which uses a priority queue as the control algorithm to select a prefetcher at a single level in the memory hierarchy. Although this approach provides benefits, the composite prefetcher priority queue is designed offline for a given set of applications. For previously unseen applications, we will not necessarily see any performance improvement. Furthermore, the control algorithm will not scale well as we increase the number of prefetchers in the system and target *different* levels in the memory hierarchy.

What is really needed is a *manager* that can successfully manage multiple prefetchers at each level in the memory hierarchy and has a low overhead. This manager will choose the PSC that is suitable for the current phase of an application. The chosen PSC should have prefetchers that complement each other for the current phase, reduce memory access overhead, and, in turn, improve application performance. Given that multiple prefetchers are available at each level in the memory hierarchy, this "manager" will effectively determine which prefetcher should be ON/OFF at each level in the memory hierarchy, both across different phases of an application and across applications. In this article, we propose a **machine learning (ML)**-based hardware manager called *Puppeteer* that selects the PSC at runtime. Contrary to the prior work [37, 52, 71] that focuses on training the ML model to improve the prefetch address prediction accuracy of the ML model, we train the ML model of Puppeteer to increase the overall system performance (quantified as IPC). Using our unique training strategy, we are able to specifically target training for application phases where the swing in IPC is much higher than in other regions. Thereby, we tailor Puppeteer for

[3]A PSC specifies which prefetcher is switched ON at each level of the memory in the system. We denote a PSC using the following format `<prefetcher-in-L1I$>-<prefetcher-in-L1D$>-<prefetcher-in-L2$>-<prefetcher-in-LLC$>`.

these phases and achieve high targeted-application-phase accuracy instead of just high overall model prediction accuracy.

To manage the prefetchers across multiple cache levels at runtime, we propose a multi-regression ML-based approach. We use the observed IPC of the various PSCs for different phases of each application to train our ML model. We train a unique random forest regressor per PSC (in our case, we have 5 different PSCs, which we pruned down from the 300 possible PSCs—more details about this are provided in Section 4), to create a suite of random forests regressors. For features used in the ML model, we use events that can be tracked using **hardware performance counters (HPCs)** and whose behavior does not change with the choice of PSC (i.e., PSC-invariant events). An example of such an event is the number of branch instructions in an application. The branch instruction count does not change with the choice of PSC. Using only PSC-invariant events, we can limit the number of executions per trace we must account for during training, making it easier to train the ML model in Puppeteer (more details about the training approach are presented in Section 3.3). In summary, the contributions of our work are as follows:

- We propose a novel ML-based runtime hardware manager called *Puppeteer* to manage the various prefetchers across the memory hierarchy to improve processor performance.
- We design Puppeteer to use a set of PSC-invariant events (which can be tracked using HPCs) as inputs and predict the PSC for the next instruction window.[4] We co-optimize the hardware design and the ML model of Puppeteer with the goal of maximizing the overall application performance while minimizing the area overhead. At runtime, at the end of an instruction window, Puppeteer predicts the IPC for each PSC and selects the PSC with the highest predicted IPC for the succeeding instruction window.
- We train Puppeteer to maximize processor performance instead of the prefetch address prediction accuracy. We use only 20% of the data for training to prevent overfitting and design Puppeteer with a low hardware overhead (~11 KB). For the 232 traces we experiment with, Puppeteer achieves an average performance gain of 46.0% in **one core (1C)**, 25.8% in **four core (4C)**, and 11.9% in **eight core (8C)** processors on average compared to a system with no prefetching. Moreover, we ensure that Puppeteer reduces negative outliers. When using Puppeteer, we observe an 89% reduction in the number of outliers, down to 8 outliers from 53, and a 20% reduction in the IPC loss of the worst-case outlier compared to the state-of-the-art prior prefetcher managers.
- Finally, to the best of our knowledge, Puppeteer is the only manager that targets all cache levels in the memory hierarchy.

2 BACKGROUND AND RELATED WORK

2.1 Heuristic-Based Prefetcher Managers

When a processor executes an application, there is a diverse set of interactions among the compute, memory, and communication components of a computing system. These interactions are dependent on the processor (micro)architecture and the application, thus effectively making the processor a big finite state machine with an extremely large state space. This makes it difficult to use deterministic techniques, which consider all possible states, for hardware management. As a result, over the past 20 to 30 years, heuristic algorithms have been used for hardware management, including for prefetcher management such as static priority queue-based approaches [34, 54]

[4]An instruction window is group of instructions that are executed sequentially at runtime. We experimented with different instruction window sizes and did not see a large change in the performance of Puppeteer. We set the size to 100,000 instructions.

Table 1. Overview of the Related Work on Prefetcher Managers

Work	# Pf	# Cache Levels Managed	IPC Gain over Baseline PSC	IPC Gain over No Prefetching	Heuristic or ML	Overhead
[54]	1	1	Only reports accuracy		Heuristic	Software
[34]	1	1	5%	41%	Heuristic	~4.6KB
[29]	1	1	7.8% (6% from 1 application)	*NA*	Heuristic	Software
[31]	0	1	11%	No Pf baseline	Heuristic	Software
[37]	4	2	Only reports ML accuracy		ML	Software
[52]	1	2	Only reports ML accuracy		ML	Software
[6]	1	1 Pf targets 2 levels	2.27%	15.24%	ML	~39 KB
[24]	4	1	−1% to 3.6%	−1% to 8.5%	ML	Software
[39]	1	1	−5% to 1%	23%	ML	Software
[28]	8	3	−1%	~10%	ML	171.8 KB
Puppeteer	**7**	**All (4)**	**14.7%**	**46%**	**ML**	~11 **KB**

We use *NA* when the paper lacks information about the respective metric. In the last column, "software" indicates the software prefetcher. For IPC gains, we report average IPC gain in 1C.

and rule-based prefetcher throttling approaches [14–17, 23, 25, 38, 62]. These algorithms have low memory/area overhead and improve processor performance for the average case. However, with the ever-increasing complexity of processors and the diversity of applications, heuristic algorithms are no longer effective. Heuristic algorithms are extremely dependent on the hardware/operating conditions and cannot easily adapt to variations at runtime.

2.2 ML-Based Prefetcher Managers

ML methods have been gaining traction in place of heuristic methods for achieving superior prefetcher management performance [6, 24, 28, 29, 37, 39]. ML algorithms can extract the non-intuitive interactions between the different prefetchers. Prior methods on prefetcher management configure or train the manager, which typically predicts which PSC to use for a given application, using values of hardware events collected for a single fixed PSC (generally, the default PSC) [6, 29, 37, 52]. Using an ML model trained using only a single fixed PSC would make sense if the prefetcher system always uses that single fixed PSC at runtime. However, at runtime, the PSC changes. If the values of the hardware events are highly dependent on which PSC is being used, using a dataset generated using a fixed PSC for training leads to a low-accuracy ML model for the prefetcher adaptation. To address this concern, we train our ML model using PSC-invariant hardware events (i.e., events that are not dependent on the PSC, e.g., the number of branch instructions).

We observe a wide variation in the complexity of the ML algorithms used in prior work. We list the prior work in Table 1. Some of the algorithms are simple and either use small datasets or use datasets that do not accurately portray the runtime environment as they use PSC-variant events and only train data of one fixed PSC. As a result, these algorithms cannot achieve good accuracy at runtime [29, 37, 52]. Other algorithms, such as neural networks, are too complex and their size increases prohibitively with the size of the dataset [6, 28]. Moreover, some prior works focus on hardware adaptation only from the perspective of accuracy without worrying about the hardware implementation [24, 29, 37, 52].

Contrary to the prior work, we jointly account for accuracy of the ML model and hardware overhead when designing Puppeteer. Puppeteer is complex enough to provide good accuracy on a wide variety of applications. At the same time, Puppeteer is not too complex (as demonstrated in Section 3.4) to be implemented in hardware and scales well with the size/complexity of the dataset. Furthermore, Puppeteer is agnostic of the underlying internal mechanics of the prefetchers and can be easily retrained for a new prefetcher that is introduced in a new system. In addition, to the best of our knowledge, Puppeteer is the only manager that targets all cache levels in the memory hierarchy.

2.3 Overview of Existing Prefetchers

The focus of our work is on a prefetcher manager, and our manager can work with any type of prefetcher. Broadly, we can split prefetching techniques into several categories. We can have regular-pattern-based (i.e., stride-based prefetchers) [30, 68], irregular-pattern-based (i.e., stream-based prefetchers) [9, 61], prefetchers that track both regular and irregular patterns [32, 58], and region-based prefetchers [20, 66]. A very good comprehensive survey on prefetchers can be found in Falsafi and Wenisch [18].

ML-based algorithms can be used for predicting the addresses of the instructions/data that should be prefetched at each cache level in the memory hierarchy. The ML-based solutions include table-based reinforcement learning [5, 46], linear model-based reinforcement learning [72], perceptron-based neural networks [19, 47], Markov chain model [41] and LSTM-based neural networks [8, 22, 43, 54, 59, 60, 63, 71, 73] to predict memory access patterns. As ML algorithms get more powerful, and the techniques to compress these algorithms become more sophisticated, ML-based prefetchers will become commonplace in processors.

3 PUPPETEER DESIGN

3.1 Puppeteer System-Level Overview

In Figure 2, we show the system-level design of an example prefetcher system that uses Puppeteer. The prefetcher system consists of eight different prefetchers (Pf1–Pf8), which is typical in modern high performance processors such as Intel i9 [26], AMD Ryzen7 [1, 3], and AMD EPYC [2]. These eight prefetchers track different memory access patterns and prefetch data from main memory to **last-level cache (LLC)**, from LLC to L2\$, and from L2\$ to L1\$. Pf1 and Pf2 target instruction lines to bring into L1I\$, Pf3 and Pf4 prefetch data into L1D\$, and Pf5 and Pf6 prefetch data into L2\$, whereas Pf7 and Pf8 target data to bring into LLC. These prefetchers compete among them for cache and memory resources. Even across memory levels, a wrong prefetch request from lower levels of memory can harm the performance of prefetchers at higher levels of memory. These prefetchers can sometimes act overly aggressive, and can adversely affect each other, in turn leading to loss of application performance. There are many heuristics-based algorithms that use simple inputs such as accuracy[5] of the prefetchers or memory bandwidth utilization to throttle prefetchers in such adverse scenarios [14–17, 23, 25, 62]. These heuristic algorithms are designed to have low overhead and hence are highly optimized for the prefetchers in a given prefetcher system.

Puppeteer works as a manager of all prefetchers and complements the heuristic algorithms used in the given prefetcher system. At runtime, Puppeteer periodically updates the PSC—that is, it sets which prefetcher should be ON and which should be OFF at each level in the memory hierarchy. To update the PSC, Puppeteer uses an ML model with the PSC-invariant hardware events, collected from HPCs as inputs. Although low-overhead heuristic algorithms are still required to make extremely low latency decisions at the cycle level, Puppeteer provides additional adaptability by leveraging the power of ML and thereby increasing the performance. Effectively, heuristic algorithms such as throttling are used in the system to constantly regulate the short-term behavior of the prefetchers, whereas Puppeteer controls the longer-term system-level behavior (across hundreds of thousands of cycles).

[5]Here, accuracy of a prefetcher is quantified as the fraction of the total prefetches that were actually useful, and is calculated as #prefetches referenced by the program divided by total #prefetches. Note, this prefetcher accuracy is not the same as the ML model accuracy. Scope is calculated as total #misses eliminated by the prefetcher divided by the total #misses when prefetching is disabled.

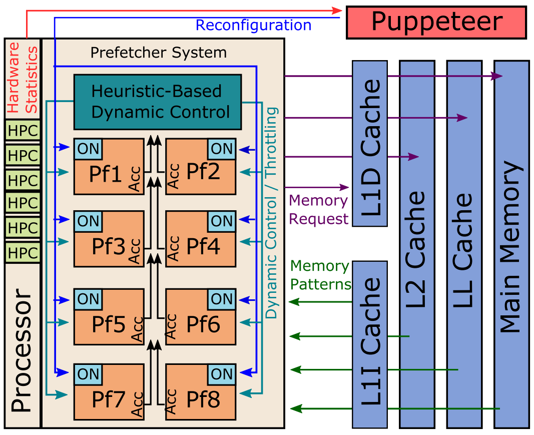


Fig. 2. Overview of a Puppeteer-based system. Here, Pf = prefetcher. Acc = Accuracy of the prefetchers. Pf1 and Pf2 target instructions to bring into L1I\$, Pf3 and Pf4 prefetch data into L1D\$, and Pf5 and Pf6 prefetch data into L2\$, whereas Pf7 and Pf8 target data to prefetch into LLC. The Heuristic-Based Dynamic Control block is a heuristic algorithm that controls the low-level cycle behavior of the prefetchers. Puppeteer controls the longer-term behavior. HPC values are fed into Puppeteer as inputs at runtime.

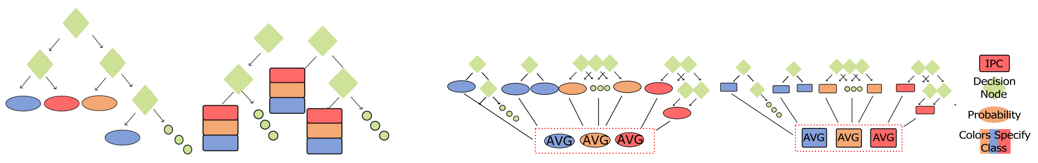
3.2 Puppeteer Algorithm

For the ML-based Puppeteer algorithm, we considered a classification-based approach and a regression-based approach. The classification-based approach has been used by prior works because it is relatively easy to train offline and has lower hardware overhead compared to regression. The regression-based approach has potential for higher performance and has better tolerance to variability compared to classification during runtime when trained offline.

Classification vs. regression. To train Puppeteer, as a classification problem we created a dataset using thresholding method similar to prior works [6, 29, 37, 39]. Here, a trace is run using all available PSCs. A PSC is given a label of "1" if the IPC when using that PSC is within some threshold (in the case of the prior work, the threshold is 0.5%) of the IPC when using the ideal PSC. Multiple PSCs can pass the chosen threshold for a given program phase, and we then end up using the PSC that is predicted as "1" with the highest probability. Otherwise, the PSC receives a label of "0." Using such a classification approach leads to sub-optimal results.

As an example, consider that we have four different traces. Let us say that we classify the first three out of the four traces correctly and the fourth one incorrectly—that is, we are able to identify the correct PSC for the first three traces but not the fourth trace. So, our classification accuracy is 75%. This means that the first three traces will have performance that is within 0.5% of their "ideal"performance. However, the performance of the fourth trace could be 100% worse or just 0.51% worse than the ideal performance. This variation in the performance is not accounted in the classification-based model. Furthermore, this problem is not unique to the given example. Any classification-based method would have a similar issue because all labeling methods used to create the dataset for the classification algorithm will certainly lose some amount of information.

In contrast, a regression-based approach accounts for the *value* of IPC gain/loss, and not just if there is IPC gain/loss, when deciding the PSC. Given that the regression algorithm is trained on the IPC values directly, the quantitative information of IPC gain/loss is not lost, and the regression algorithm can learn the magnitude of a good or bad prediction. In particular, the regression algorithm will be used to predict an IPC value for each PSC for a given instruction window. Then, we choose the PSC with the highest predicted IPC value.



(a) Single Classifier **(b)** Single Regressor **(c)** Suite of Classifiers **(d)** Suite of Regressors

Fig. 3. Algorithm options. A decision tree for a singular classifier, a random forest for singular regressor, a suite of decision trees for a suite of classifiers, and a suite of random forest regressors for a suite of regressors (i.e., Puppeteer).

Suite of regressors vs. single regressor. When using regression algorithms, we have two options: (i) use a single regressor, where all data collected from all PSCs are used to train that single regressor, or (ii) use a suite of regressors, where each PSC will have a dedicated regressor. Using a suite of regressors leads to a more customized solution that has higher accuracy as compared to using a single regressor. Conceptually, this is because in a single regressor, we maximize the accuracy across all PSCs instead of maximizing the accuracy of each PSC separately, whereas in a suite of regressors, we train a dedicated regressor for each PSC. In this way, we indirectly jointly increase the scope and accuracy of the overall prefetching system.

In our work, we use a suite of regressors where we implement each regressor using random forest (i.e., one random forest trained per PSC) due to its simple implementation, its robustness to noise in the dataset, its lower overhead (compared to other ML algorithms e.g. neural networks), its resilance to overfitting, and its higher accuracy (compared to other ML algorithms e.g. decision trees) [35, 44, 53].

We have multiple trees per forest, and we allow each tree to split at locations that are unique to the PSC associated with the forest. The leaves of each tree in the forest specify the predicted IPC value for the PSC. For each forest (i.e., each PSC), we calculate the average of the predicted IPC values obtained from all trees in the forest, then choose the PSC with the highest average predicted IPC. Given that each forest has multiple decision trees, our method has higher tolerance to wrong decisions by the trees, where even if some of the trees give wrong decisions, other trees can compensate.

In Figure 3, we conceptually show the differences between the four different options discussed earlier: (a) single classifier, (b) single regressor (c) a suite of classifiers, and (d) a suite of regressors. Respectively, the leaf nodes in (a) contain the PSC choice directly, (b) have the predicted highest IPC among all the PSCs, (c) the probability value of a given instruction window belonging to the given PSC (of which we choose the highest one), and (d) the predicted IPC value for a given PSC that will be averaged per tree in a given RF and compared with the other averaged predicted PSC values from the other RFs (of which we will choose the highest). For (a) and (b), the PSCs are traversed simultaneously, since the PSCs share a single algorithm, whereas for (c) and (d), each PSC has a unique algorithm we traverse.

3.3 Puppeteer Training

To train Puppeteer, we need to generate a representative dataset. Consider the case where we have a single prefetcher, Pf, at only one level in the memory hierarchy. Here, the number of PSCs $(N_{psc}) = 2$ (i.e., Pf = OFF and Pf = ON). For two consecutive instruction windows, we will have $N_{psc}^2 = 4$ possible scenarios: (i) Pf = OFF $\rightarrow$ Pf = OFF, (ii) Pf = ON $\rightarrow$ Pf = OFF, (iii) Pf = OFF $\rightarrow$ Pf = ON, and (iv) Pf = ON $\rightarrow$ Pf = ON. With N number of instruction windows and N_{trace} number of traces, the number of different possible scenarios will then be $N_{trace} \times N_{psc}^N$. When N increases, the number of different scenarios will increase exponentially, hence including each unique scenario in the dataset for training is not feasible. To handle this problem, we propose to use only PSC-invariant events as our features. An example of a PSC-invariant event is the number of conditional branches, which is not affected by the choice of PSC. We check the variance of each hardware event value (for 180 total hardware events that we can track) for each PSC. We identify 59 events whose values vary by less than ±10% from their mean value across all PSCs. We further reduce the number of events by eliminating the redundant events that track similar behavior and have high correlation with each other. Table 2 shows the final six events we choose to track trace behavior. After we have identified our PSC-invariant events that will be the features and the PSCs that will be the choices of our ML model, we collect an IPC value per PSC for each instruction window as our ground truth.

Using the features and IPC values we have collected, we then form our suite of random forest regressors wherein we train a separate forest for each PSC using CART (classification and regression trees) [64]. CART is a greedy recursive search algorithm that maximizes information by splitting the data at each node using one feature. Each child node is split recursively until there is no information gain from splitting a child node. We limit the total number of decision nodes in Puppeteer to keep the size of Puppeteer smaller than L1$. With this limitation in mind, we conduct a hyper-parameter search and determine that the number of estimators (trees per random forest) should be 5 and the number of max nodes should be 100 per tree.

3.4 Puppeteer Microarchitecture Design

Figure 4 shows the microarchitecture details of Puppeteer. We use a single-port SRAM array called *Node MEM* to store information about the nodes that form the trees of each random forest in Puppeteer. We train Puppeteer offline on a group of applications of interest. After the training is complete, we load the random forest based ML model of Puppeteer into the *Node MEM* at startup using firmware. If the processor unexpectedly executes a completely different set of applications than those already considered, the ML model might need to be retrained by including these new applications. This retraining process is performed offline, and the model is reloaded using firmware. As more and more applications are included, we will need to retrain the ML model less often. Each entry of *Node MEM* corresponds to one node in one of the random forests, and it consists of the following fields. First, it consists of a 3-bit `HPC ID` field that specifies which PSC-invariant event (i.e., which HPC) is used by that node to make a decision. The 3-bit encoding enables the node to use one of six different PSC-invariant events (see Table 2). Second, it consists of a 16-bit `Threshold` field (threshold value is determined during training), which is employed by the node to decide if the decision path should branch left or right. In our problem, 16 bits provide enough precision for the ML model weight values. Third, it consists of a 12-bit (for 2,250 node addresses) `Left Node Value(LNV)` field, and fourth, a 12-bit `Right Node Value(RNV)` field. These `LNV` and `RNV` fields represent child node indices for internal nodes of a tree. For the leaf nodes of a tree, we use these `LNV` and `RNV` fields to indicate the predicted IPC value of a PSC. We differentiate between child node index and predicted IPC using, fifth, a 1-bit `Type` field. We use a separate 1-bit `Type` field for `LNV` and `RNV`.

At the end of every instruction window, Puppeteer calculates the predicted IPC for each PSC in the next instruction window by traversing the trees of the associated forest and using the PSC-invariant event values for the current window as inputs. For each forest, the controller in Puppeteer reads the *Node MEM* index of the root node for the first tree from *Root Index Table* and loads the *Node MEM* entry for the root node using a *Load Unit* into a register. Next, the `HPC ID` in the loaded *Node MEM* entry is used to load the corresponding PSC-invariant event value into a second register. Then the `Threshold` value, stored in the first register, and PSC-invariant event value stored in the second register are compared using the *Comparator*.

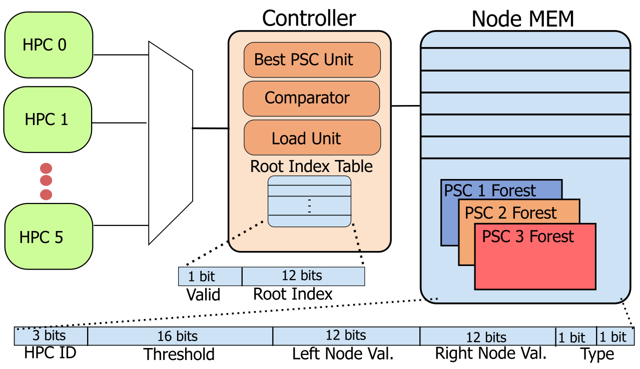


Fig. 4. Puppeteer hardware design. Puppeteer is made up of a Node MEM-SRAM array, a max logic unit, and several register files.

Based on the *Comparator* output, we choose to traverse down to the left child or the right child. The *Controller* then uses the corresponding index value from LNV or RNV to find the next node in *Node MEM*. The *Controller* continues traversing the tree until it loads a predicted IPC value corresponding to a leaf from the *Node MEM*. The preceding steps are repeated for the remaining trees in the forest, then we calculate the average of the predicted IPC values obtained from all trees in that forest. The *Best PSC Unit* in the *Controller* stores the ID of the PSC with the highest predicted IPC value. Every time the *Controller* finishes traversing a forest, the predicted IPC value of that forest (i.e., PSC) is compared with the predicted IPC value stored in the *Best PSC Unit* using the *Comparator*. If the new predicted IPC value is higher than the current value, the *Best PSC Unit* updates the predicted IPC value and the ID of the PSC. Once all forests have been traversed (i.e., all PSCs have been evaluated), Puppeteer chooses the entry stored in the *Best PSC Unit* as the PSC for the next instruction window.

We determined that a maximum depth of 10 per tree is more than sufficient to accurately determine the best PSC. In our evaluation, we use a prefetcher system with $N_{psc} = 5$ (given later in Table 6 and discussed in detail in Section 4). We need a total of 2,250 nodes to design the trees in Puppeteer, and these nodes require a 10.75-KB-sized *Node MEM* (compared to a typical L1$ of 32 KB). Other than *Node MEM*, we require a $5 \times N_{psc}$-entry *Root Index Table* where each entry is 13-bit wide (12 bits for the root node index and 1 valid bit), a 12-bit comparator containing comparison logic and two registers, a load unit, and a register to store the best PSC information in the *Controller*. We discuss the hardware overhead in more detail in Section 5.

4 EVALUATION METHODOLOGY

We use ChampSim [50] for our analysis, where we model 1C, 4C, and 8C processors to have multiple prefetchers at each level of the cache—private L1I$, L1D$, private L2$, and shared LLC (more details are provided in Table 2). We train Puppeteer using data collected from a 1C and 4C OoO processor and then evaluate it on 1C, 4C, and 8C OoO processors. In the 4C and 8C processors, we have a private Puppeteer per core. Each private Puppeteer utilizes the six PSC-invariant events per core and makes independent decisions. For our evaluation, we use a diverse set of 232 traces generated from SPEC2017 [12], SPEC2006 [11], and Cloud [51] benchmarks. Here, a trace is a group of instructions that represent a specific behavior. For example, a benchmark with consistent looping behavior can be represented by one trace corresponding to a single iteration of the loop. One or more unique representative traces are generated from each benchmark using clustering algorithms [48] to represent the different behaviors of a benchmark. The traces we use for our evaluation were generated by the organizers of the prior prefetching competitions [49, 51].

Table 2. Simulated System Parameters

Component	Simulated Parameters
Core	One to four cores, 4 GHz, 4-wide, 256-entry ROB
TLBs	64 entries ITLB, 64 entries DTLB, 1,536 entry shared L2 TLB
L1I$	32 KB, 8-way, 3 cycles, PQ: 8, MSHR: 8, 4 ports
L1D$	48 KB, 12-way, 5 cycles, PQ: 8, MSHR: 16, 2 ports
L2$	512 KB, 8-way, 10 cycles, PQ: 16, MSHR: 32, 2 ports
LLC	2 MB/core, 16-way, 20 cycles, PQ: 32×cores, MSHR: 64×cores
DRAM	4 GB 1 channel/1-core, 8 GB 2 channels/multi-core, 1,600 MT/sec

Hardware Event	Properties
L1I_PAGES_READ_LOAD	L1I$ pages read on load
L1D_PAGES_READ_LOAD	L1D$ pages read on load
L1D_RFO_ACCESS	L1D$ store accesses
BRANCH_RETURN	Branch returns
NOT_BRANCH	Not branches
BRANCH_CONDITIONAL	Conditional branches

Table 3. Static PSCs and Managers Evaluated

Algorithm Notation	Explanation	Static PSC or Manager	Training Dataset
NO	No prefetching is used; PSC = *no-no-no-no*	Static PSC	Not trained
IPCP	Winner of DPC3 [49]; PSC = *no-ipcp-ipcp-nl*	Static PSC	Not trained
EIP	Winner of IPC1 [51]; PSC = *EIP-nl-spp-no*	Static PSC	Not trained
PY	RL-based algorithm named Pythia [5]	ML-based prefetcher	Online
J3	Heuristic-based algorithm by Jiménez et al. [29]	Manager	Not trained
NN	Multi-layer perceptron similar to Bhatia et al. [6]	Manager	1C
B1C	Decision tree algorithm by Liao et al. [37]	Manager	1C
B4CS	Decision tree algorithm by Liao et al. [37]	Manager	4CS
B4CM	Decision tree algorithm by Liao et al. [37]	Manager	4CM
P1C	Puppeteer	Manager	1C
P4CS	Puppeteer	Manager	4CS
P4CM	Puppeteer	Manager	4CM

Note: We list the static PSCs from the prior prefetching competitions, the manager algorithms from prior work, and the different flavors of Puppeteer.

In Table 3, we list the notations used for all static PSCs and the runtime managers (that change PSC at runtime) that we have evaluated. In our evaluation, we normalize all IPC values to the same state-of-the-art baseline as PPF [6]—that is, SPP [32] (*no-no-spp-no*). We compare Puppeteer against the best static PSCs from competitions (i.e., IPCP (*no-ipcp-ipcp-nl*) and EIP (*EIP-nl-spp-no*), as well as managers such as the final version of the algorithm developed by Jiménez et al. [29] that uses trial periods to latch onto the PSC (J3), Pythia that is an RL-based algorithm developed by Bera et al. [5] (PY),[6] a multi-layer perceptron similar to Bhatia et al. [6] (NN), and a binary-tree (BT) based algorithm [37], where Liao et al. tried several different ML methods (e.g., decision trees and NN) and concluded that decision trees are the best choice.

4.1 Training Approaches for ML-Based Prefetcher Managers

When evaluating any new idea, a widely used approach in the industry is to use all available evaluation data. We are using a ML-based approach and are cognizant of the fact that we do not want to overfit our model using all available data for training the ML model. So here we compare three different approaches for constructing the training dataset for Puppeteer. For training, we have

[6]Note that Bera et al. used SHiP [69] for their cache replacement policy and perceptron as their branch predictor. We tested with their settings as well as with using Pythia with hashed-perceptron and least recently used (LRU), and observed on average that Pythia with hashed-perceptron with LRU has 12% higher IPC. Hence, we use hashed-perceptron and LRU similar to all other comparison points in this article.

Table 4. Dataset Training Approaches

Approach	%Benchmarks	%Traces	%Inst. Win.	Total % of all Inst. Win. Used for Training
# 1	ALL	ALL	10%	10%
# 2	ALL	80%	60%	48%
# 3	80%	ALL	60%	51%

Note: Here, %Benchmarks indicates the percentage of benchmarks used during training, %Traces indicates the percentage of traces from the available benchmarks used during training, and %Inst.
Win. indicates the percentage of instructions windows from the available traces used during training. The last columns indicates overall the percentage of instruction windows using during training.

232 traces ×10,000 instruction windows per trace = 2,320,000 instructions windows. For each approach, we limit the training data in different ways. Note that we are still training at the instruction window-level and performing 10-fold cross validation on the training set for all approaches. We give the percentage of data used to construct each training dataset in Table 4.

In the *first approach*, we severely limit the data and randomly select 20% of the instruction windows at random time intervals to form the training dataset (i.e., 80% of the instruction windows are not seen during training). The idea behind this approach is that we are training the algorithm to recognize low-level memory access behavior. This approach does not overfit the model because the behavior of the benchmarks when collecting the training data would be different from the behavior of the benchmarks when we use the trained Puppeteer. This difference is because when collecting training data we do not change the PSC, whereas when using Puppeteer the PSC can potentially change for each instruction window. In the *second approach*, we use 80% of the traces for testing. We consider all instruction windows of the remaining 20% traces, and perform a 60–40 split where 60% of the windows are used for training and 40% of the windows are used for validation. This avoids overfitting. Given that traces are constructed to represent unique behaviors in each application, our training set will not have traces corresponding to all of the unique behaviors of a given benchmark. This means that some of the traces from other benchmarks are assumed to be from the same distribution of these missing traces or else there will be no way for the algorithm to train for these regions. In the *third approach*, we generate the training dataset using 20% of the benchmarks and for those benchmarks we use a 60–40 split of the instruction windows. The remaining 80% of the benchmarks are used for testing. In this approach, whole benchmarks are not considered during training. Here, too, we choose a 60–40 split of the instruction windows to avoid overfitting the model to the 20% of benchmarks in the training set.

In a 1C processor, when using Puppeteer on unseen benchmarks, for the first, second, and third approaches, we observe an average IPC gain of 11.3%, 9.8%, and 6.6%, respectively, over SPP. In the second and third approaches, we see that Puppeteer has lower performance gain than the first approach. Despite having 3% more instruction windows for training in the third approach compared to the second approach, there is a 3.2% drop in performance. Furthermore, in the first approach, we use only 10% of the total number of instruction windows for training (5 × less instruction windows than the other approaches), yet this approach performs 1.5% and 4.7% better than the other two approaches. This implies that the second and third approaches are trained on an insufficient number of unique memory accesses patterns, leading to lower performance gain. Therefore, for the rest of our evaluation, we train all ML-based prefetcher managers using the first approach. As mentioned earlier, Puppeteer can always be retrained and updated via firmware.

Table 5. Initial Prefetcher Options

L1I$	L1D$	L2$	LLC
No prefetcher	No prefetcher	No prefetcher	No prefetcher
Next-line [18]	Next-line	Next-line	Next-line
FNL+MMA [56]	IPCP [45]	IPCP	
DJOLT [42]	MLOP [57]	SPP [32]	
EIP [55]	Bingo [4]	KPCP [33]	
		Ip-Stride [18]	

Note: We show the regular and irregular prefetcher options we used at each cache level to construct our set of 300 PSCs. We reduce the number of PSCs down to five PSCs before training.

4.2 1C, 4C, and 8C Workload Formulation

We run 1C and 4C experiments using a 200M instruction warmup phase and 1B instruction detailed simulation phase, whereas for 8C we use a 200M instruction warmup phase and 250M instruction detailed simulation phase. We generate a total of 232 traces from SPEC2017, SPEC2006, and Cloud benchmarks. We create two flavors of trace sets for the 4C and 8C experiments: a single-type trace set where each core runs the same trace, and a mixed-type trace set where each core runs a unique trace. Therefore, we have five data suites in total: 1C, 4C single-type trace set (4C STS), 4C mixed-type trace set (4C MTS), 8C single-type trace set (8C STS), and 8C mixed-type trace set (8C MTS). We use hashed-perceptron for branch predictor and least recently used policy for cache replacement policy provided by ChampSim. To construct our mixed-type trace sets, we first split the 232 traces into six groups based on ascending execution latency. Then we randomly select a trace from a given group per core and construct a mixed-type trace group. We cover all permutations of the various latency groups. This way, we have diversity in the traces running on the cores. For example, in a 4C processor, we can assign a trace from group 2 to core0, a trace from group 3 to core1, a trace from group 4 to core3, and a trace from group 5 to core3. This can be represented as $2 - 3 - 4 - 5$. Therefore, we have 360 experiments (six options for core0 $\times$ 5 options for core1 $\times$ 4 options for core2 $\times$ 3 options for core3) in 4C MTS for six latency groups and four selected traces (one for each core). For 8C MTS, since the number of experiments increases and the experiments take quite long, we replicate the same latency group for four of the cores. For example, we can use $1 - 1 - 1 - 1 - 4 - 4 - 4 - 4$ for an 8C experiment. However, instead of 6 groups as in the 4C system, we use 10 groups for the 8C system, to cover a more granular variety of behavior and instead of permutations we use combinations with replacement. Therefore, we have 55 (two traces selected from 10 latency groups with replacement, i.e., $C^R(10, 2)$) experiments for 10 groups and two selections. Note that even if the latency group is the same, since we choose a trace randomly from a latency group, the trace can still be different for each core.

4.3 Generating ML Models and PSC Pruning

To generate the 1C dataset (PSC and associated IPC values), we run the 1C experiments using all possible PSCs generated from the prefetcher options that are available in the ChampSim repository, the first place (IPCP [45]), second place (Bingo [4]), and third place (MLOP [57]) winners of the Third Data Prefetching Competition (DPC3) [49], and the first place (EIP [55]), second place (FNL+MMA [56]), and third place (DJOLT [42]) winners of the First Instruction Prefetching Competition (IPC1) [51]. To reduce the hardware overhead of Puppeteer, we avoid including multiple PSCs that cover the same traces. To this end, we initially ran 20 traces for 20M instructions with all possible PSCs (5 prefetching options in L1I$ $\times$ 5 prefetching options in L1D$ $\times$ 6 prefetching options in L2$ $\times$ 2 prefetching options in LLC = 300 PSCs). We summarize the different prefetchers that we evaluated in Table 5.

Table 6. PSCs and Their Prefetching Options Used at Each $ Level

Final PSCs Used	**L1I$**	**L1D$**	**L2$**	**LLC**
djolt-bingo-nl-nl	DJOLT [42]	Bingo [4]	Next-line [18]	Next-line
djolt-bingo-no-no	DJOLT	Bingo	No prefetcher	No prefetcher
fnl-bingo-spp-nl	FNL+MMA [56]	Bingo	SPP [32]	Next-line
fnl-bingo-spp-no	FNL+MMA	Bingo	SPP	No prefetcher
no-nl-spp-no	No prefetcher	Next-line	SPP	No prefetcher
Overhead	221 KB	48.06 KB	6 KB	0.6 KB

Note: These PSCs are used by all manager algorithms, not just Puppeteer.

For each trace, we sort the PSCs based on the corresponding IPC values in descending order. We generate a new table for each trace, where the table contains the top 10 PSC entries for the trace and combine these tables to form a super-table that contains the top 10 PSCs for all traces. Note that a PSC may be in the top 10 for more than one trace. We sort the PSCs in descending order based on the number of traces for which the PSC is in the top 10. Starting from the top, we select just enough PSCs to improve performance of all 20 traces. We picked the PSCs that have the best performance with minimal coverage overlap while reducing the number of unique prefetchers (to reduce the hardware overhead). In Table 6, we show the final 5 PSCs that we selected. These PSCs give good performance for the maximum number of traces. We show the prefetcher used at each cache level. It is interesting to note that the best prefetchers from DPC3 and IPC1—IPCP and EIP—are not used in the top 5 choices for PSCs. This shows that the state-of-the-art prefetchers designed in isolation may not be the best choice of prefetchers when used with other prefetchers.

We collect the 4C STS and 4C MTS datasets in the same way as the 1C dataset—the only difference is that the data is collected per core. We then train BT and Puppeteer using datasets collected from 1C, 4C STS, and 4C MTS. We denote the different flavors of the two algorithms as P1C, P4CS, and P4CM for Puppeteer, and B1C, B4CS, and B4CM for BT.

5 EVALUATION RESULTS

5.1 Puppeteer in a 1C Processor

In this section, we present the processor performance analysis when using Puppeteer in 1C, 4C, and 8C processors, a sensitivity analysis of how Puppeteer's performance varies with cache size, and we explore the trade-off between performance improvement and hardware overhead when using Puppeteer. We evaluate Puppeteer using scope, accuracy, and power metrics. For the analysis presented in Sections 5.1, 5.2, 5.3, and 5.4, we assume a total hardware overhead budget of 11 KB for storing the Puppeteer model, and due to its simplicity (as explained in Section 2), we assume negligible evaluation overhead for the computing logic.

In Figure 5, we show the IPC distribution for various static PSCs and prefetcher managers that change the PSC at runtime. We begin our discussion with P1C, which is Puppeteer trained using the 1C data suite. Broadly, compared to a processor with no prefetchers, P1C provides an average performance gain of 46.0% (peak value of 613%). The true benefit of Puppeteer is observed when we look closer into the performance loss in Figure 6. We observe that when using B1C, 53 traces lose performance and 6 out of those 53 traces lose more than 10% performance. This performance loss would make this solution non-acceptable. Puppeteer has a worst-case loss of only 5%, and only 8 traces in total have lower performance than SPP. In Figure 7, we show the performance loss/gain of the 10 traces that have the worst performance for Puppeteer, when using five different managers: NN, B1C, J3, PY, and P1C (Puppeteer). Across these 10 traces, Puppeteer, NN, B1C, J3, and PY have an average performance loss of 1.9%, 6%, 6.6%, 4.8%, and 4.9%, respectively.

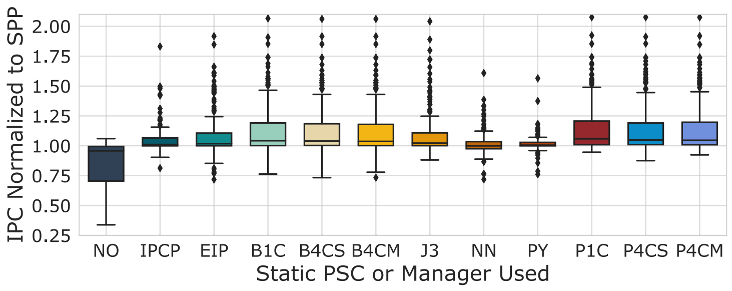


Fig. 5. Normalized performance of a 1C processor. Performance distribution of Puppeteer and prior work normalized to *SPP* [32]. See Table 3 for the notations used along the *x*-axis.

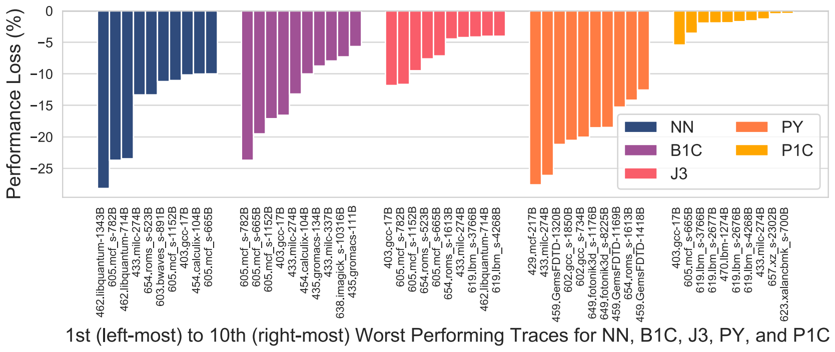


Fig. 6. Bottom 10 performance outliers of all managers on 1C. Performance normalized to *SPP*. Each group shows the worst-performing traces for NN, B1C, J3, and PY, P1C ordered 10th worst (right-most) to 1st worst (left-most).

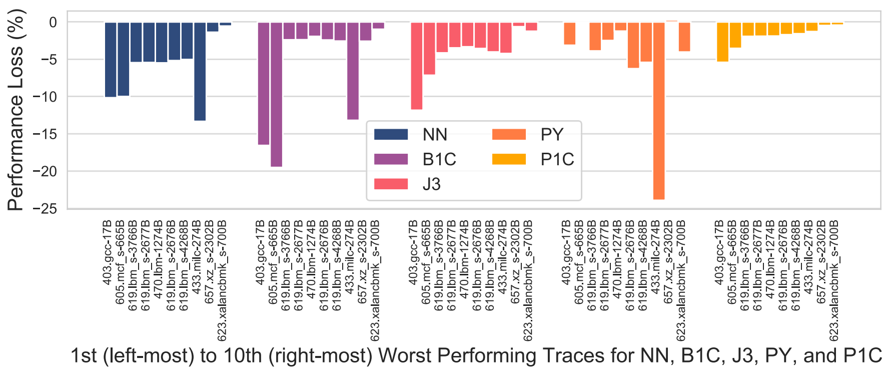


Fig. 7. Bottom 10 performance outliers of P1C. Performance normalized to *SPP*. Each group shows the performance of the bottom 10 traces for P1C when using NN, B1C, J3, and PY.

This clearly illustrates that Puppeteer provides a win-win situation, whereby we not only see a better average performance gain but also see a reduction in the maximum performance loss and the number of traces that have performance loss. For the other prior works, we observe that P1C provides 4.65%, 5.8%, 12.2%, 15.3%, and 5.1% average IPC gain over IPCP, EIP, NN, PY, and J3, respectively.

We also trained two more flavors for Puppeteer and BT using 4C STS and 4C MTS datasets. P4CS and P4CM achieve 0.5% lower average performance gain than P1C. This is because Puppeteer trained using 1C data is better suited for a 1C processor. However, the performance improvement when using Puppeteer, which is trained with 4C data, is still quite large at 45% (average of P4CS and P4CM) compared to a system with no prefetching. This means that Puppeteer generalizes quite well and has learned the underlying architectural phenomena. Compared to B1C for B4CS and B4CM, we observe 1% and 2% lower average performance, respectively. Furthermore, the performance loss for the worst case outlier increases to 30%. Finally, one thing to note about J3 is that its average IPC gain is 2.1% lower than B1C, yet the negative outliers are less in both quantity and magnitude. This is because J3 makes more conservative changes compared to BT since the algorithm cycles through the PSCs as part of a constantly ongoing trial phase. This means that in a production capable system, J3 might have been more viable compared to BT. This is because processors have to ensure that all applications retain or increase their performance across new processor generations.

5.2 Puppeteer in a 4C Processor

Here we discuss the use of Puppeteer trained using 1C, 4C STS, and 4C MTS datasets in a 4C processor. These cases are represented by P1C, P4CS, and P4CM, respectively. In Figure 8, we show the IPC distribution of Puppeteer and the various prior works in a 4C system while running a 4C STS data suite. We observe that the average IPC gain of P1C over the no prefetching case is 25.8%. Compared to B1C, P1C has 4.2% higher average performance. The worst-case performance loss in B1C has gone up to 45%, whereas the worst-case loss of P1C is at only 19%. The number of traces that lose performance is 61 for B1C, whereas P1C has just 24 traces that lose performance. Compared to IPCP, EIP, NN, PY, and J3, our P1C achieves 10.8%, 4.8%, 8.7%, 6.8%, and 3.7% average IPC gain, respectively. One key observation here is that although P1C has only been trained on 1C data, it is still applicable to the 4C case and provides a clear advantage over prior work. This is important given that as the number of cores increases, the number of unique combinations of different traces that we will need to run in the multi-core processor increases exponentially (for a 4C processor, we need to cover $232^4 = 2.89$ billion trace combinations). Therefore, generalization using only 1C data is an important aspect to consider when comparing ML-based algorithms. To further test Puppeteer and BT, we train both algorithms with 4C STS and 4C MTS data suites. For Puppeteer, we observe that P4CS has 1.2% and P4CM has 2.2% better performance compared to P1C (see Figure 8). For BT, B4CS has 4.0% and B4CM has 2.9% better performance compared to B1C. This means that Puppeteer is better at learning the underlying (micro)architectural behavior with a variety of traces running concurrently compared to the same trace running on all cores. In contrast, BT requires data collected specifically from the same set of experiments to achieve better performance.

In Figure 9, we show the IPC distribution of Puppeteer and the various prior work when running 4C MTS. P1C achieves 23% average performance gain over no prefetching. P1C's average performance gain is also very close (<0.6% difference) to the average performance gain of P4CS and P4CM. Once again, we observe that Puppeteer is superior to BT in this regard, with B1C achieving 5% lower performance gain than P1C. B1C's performance gain is also 4% lower than B4CS and 4.5% lower than B4CM. When we observe the outliers, B1C has 23 outliers while P1C has only 3. The worst-case outlier in B1C has 7% performance loss compared to SPP, whereas P1C has only 0.4% loss. Compared to IPCP, EIP, NN, PY, and J3, our P1C (and also P4CS and P4CM) achieves 14.5%, 5%, 12.8%, 11.9%, and 4.4% average IPC gain, respectively.

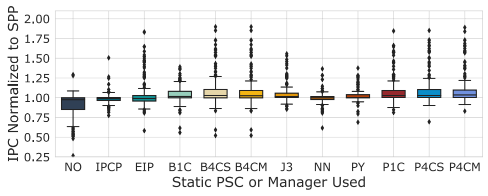


Fig. 8. Normalized performance of a 4C processor running STS. Performance distribution of Puppeteer and prior work normalized to *SPP*. Here, the reported performance is average performance across all cores. See Table 3 for the notations used along the *x*-axis.

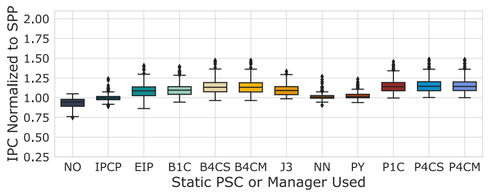


Fig. 9. Normalized performance of a 4C processor running MTS. Performance distribution of Puppeteer and prior work normalized to *SPP*. Here, the reported performance is average performance across all cores. See Table 3 for the notations used along the *x*-axis.

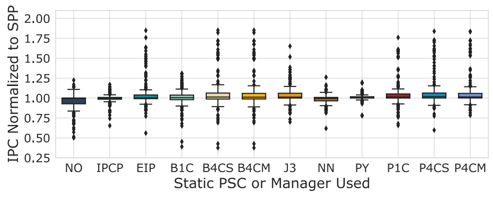


Fig. 10. Normalized performance of an 8C processor running STS. Performance distribution of Puppeteer and prior work normalized to *SPP*. Here the reported performance is average performance across all cores. See Table 3 for the notations used along the *x*-axis.

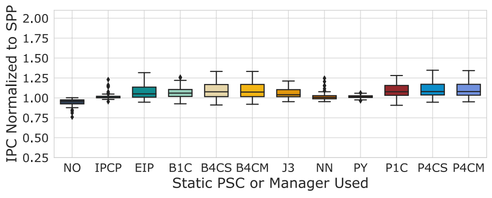


Fig. 11. Normalized performance of an 8C processor running MTS. Performance distribution of Puppeteer and prior work normalized to *SPP*. Here the reported performance is average performance across all cores. See Table 3 for the notations used along the *x*-axis.

5.3 Puppeteer in an 8C Processor

Here we discuss the use of Puppeteer trained using 1C, 4C STS, and 4C MTS datasets in an 8C processor. These cases are represented by P1C, P4CS, and P4CM, respectively. We conduct this 8C processor analysis to test if Puppeteer scales to a larger number of cores. For an 8C processor running STS, we observe several interesting trends (Figure 10). P1C has 11.9% average IPC gain over no prefetching, which is 4.8% higher than B1C. With P4CS and P4CM, our IPC gain is 12.7% and 12.9%, respectively, over the no prefetching case. This means that to train Puppeteer for a multi-core processor, we should have at least some data corresponding to a multi-core processor in our dataset. It should be noted that compared to the no prefetching case, P1C, P4CS, and P4CM have lower IPC gain in the 8C processor than the 4C processor, which in turn has lower performance gain than the 1C processor. This is because prefetching is much harder to do in multi-core processors. The overall benefit of prefetching goes down in multi-core processors, and Puppeteer has lower possible peak performance. Comparatively, B1C has almost 0% performance gain over SPP. Even a static PSC, namely EIP, has 5.5% higher average IPC compared to B1C. P1C also gives comparable performance to B4CS and B4CM even though P1C was not trained using any data from a multi-core processor. When we observe the outliers, B1C has 83 outliers with worst-case performance loss of 62%, whereas P1C has 47 outliers with worst-case performance loss of 35%. Compared to IPCP, EIP, NN, PY, and J3, our P1C achieves 5.9%, 0.2%, 6.5%, 4.4%, and 1.2% average IPC gain, respectively.

When using the 8C MTS (Figure 11), we observe similar trends to when using 8C STS. P1C has 2% better average IPC than B1C and comparable performance to B4CS and B4CM. P4CS and P4CM

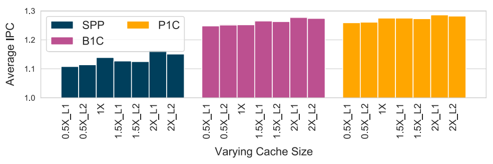


Fig. 12. Cache size sensitivity study. Average performance of B1C, SPP, and P1C for 0.5×, 1×, 1.5×, and 2× the nominal L1$ and L2$ sizes. For each bar, we change only the L1$ size or the L2$ size.

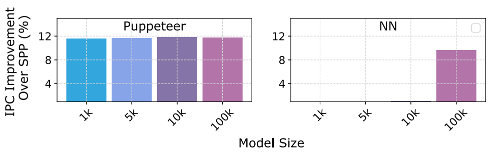


Fig. 13. Model size scaling. Average IPC improvement on P1C and NN for different model sizes.

achieve 4% and 4.4% better average IPC compared to B1C. Compared to IPCP, EIP, NN, PY, and J3, our P1C achieves 6.9%, 1.6%, 6.6%, 7.6%, and 3.1% average IPC gain, respectively.

5.4 Puppeteer Performance for Different Cache Sizes

In this section, we discuss how the performance of BT, SPP, and Puppeteer varies with cache size. We train both models only using data collected on a 1C processor with 32-KB L1I$, 48-KB L1D$, 512-KB L2$, and 2-MB LLC. In Figure 12, we show the average IPC values of BT (i.e., B1C), SPP, and Puppeteer (i.e., P1C) for 0.5×, 1×, 1.5×, and 2× the nominal L1$ and L2$ sizes. P1C is affected by L1$ size slightly more than L2$ size, but the difference is small (at most 0.4% more performance at 2× L1$ compared to 2× L2$). At 0.5× L1$ size, P1C has 1.5% lower performance compared to the performance of P1C at nominal cache size. At 2× L1$ size, P1C has 1.1% better performance compared to the performance of P1C at nominal cache size. At all cache sizes, P1C performs at least 1% better than B1C with the largest performance difference of 2.3% at nominal cache size. SPP performance swings by 5.5% between the 2× L1$ and the 0.5× L1$, and by 3.7% between the 2× L2$ and the 0.5× L1$. In contrast, P1C has only a 2.9% swing. This means that with P1C, we are already operating close to the possible peak performance, and so increasing the cache size does not have a significant effect on the performance.

5.5 Performance Improvement for Different Model Sizes

In Figure 13, we show the average IPC for 1C for the differently sized models. As is immediately apparent, Puppeteer provides good performance improvement even when using a small model that fits within 1 KB, whereas NN does not provide any performance improvement until we use a model that requires 11 KB. At 11-KB model size, a 1C processor with an NN manager has 11.8% lower average IPC as compared to a 1C processor with Puppeteer. The NN also has ~ 3× larger area and 11× larger power as compared to Puppeteer. If we compare 100-KB NN to 1-KB Puppeteer, using Puppeteer provides 11.6% performance improvement, whereas NN provides 9.7% performance improvement while the area and power of 100-KB NN is 76× and 185×, respectively, that of Puppeteer. Therefore, NN is not suitable for hardware prefetcher adaptation.

5.6 Puppeteer Overheads

To understand the tradeoff between the overheads for selecting PSC using Puppeteer and the performance improvement using Puppeteer, in this section we design Puppeteer when we have four different hardware overhead budgets: 1 KB, 5 KB, 11 KB, and 100 KB. Here we train a different Puppeteer model for each hardware overhead budget.

PSC selection. To select a PSC, Puppeteer just traverses through the random forest for each PSC. If we evaluate all five forests in series, where we will require a maximum 500 comparison operations (5 forests × 5 trees per forest × 10 comparisons = 250 comparison for 1 KB and 5 KB; 5 forests × 5 trees per forest × 20 comparisons = 500 comparisons for 11 KB and 100 KB), it will take less

Table 7. Power and Area for Total Compute and Storage of Puppeteer and NN

Algorithm	Area (μm^2)	Area Norm.	Power ((μW))	Power Norm.
Puppeteer 1KB	5,000	1×	4.7	1×
Puppeteer 5KB	10,000	2×	8	1.7×
Puppeteer 11KB	18,000	3.6×	9	1.9×
Puppeteer 100KB	140,000	28×	39	8.3×
NN 1KB	17,000	3.4×	14	3×
NN 5KB	40,000	8×	65	13.8×
NN 11KB	55,000	11×	100	21×
NN 100KB	380,000	76×	870	185×

Note: Here we use the SRAM compiler for designing node MEM and design the compute logic using RTL, then synthesize it using Cadence Genus for GF22FDX® [10].
Average power is given over one instruction window. *Norm.* values are w.r.t. to Puppeteer 1-KB values.

Table 8. Node MEM Size, Area, Power, and Delay of Different Puppeteer Designs

SRAM Size, Model	#Bits/Word, #Words/SRAM Array	# SRAM Arrays Required	Area Norm.	Read Energy Norm.	Delay (Cycles)
1-KB Puppeteer	38, 256	1	1×	1×	1
5-KB Puppeteer	42, 1,024	1	2×	1.3×	1
11-KB Puppeteer	44, 2,048	1	4.2×	1.5×	1
100-KB Puppeteer	50, 16,384	1	31×	3.5×	1

Note: Other than the node MEM, Puppeteer only needs a single comparator unit.
Write energy is not reported, as it is a low one-time cost. Note that values are normalized to 1-KB Puppeteer for proprietary reasons.

than 0.5% of the total time required to execute the 100K instructions in the instruction window (assuming each instruction takes on average a clock cycle). Thus, we end up using the chosen PSC for 99.5% of the instruction window for Puppeteer. We would like to note that the hardware for determining the best PSC is not on the critical path, and it runs in parallel to the normal application execution without stopping the prefetchers.

Power and area overheads. In Table 7, we show the total area and power required for Puppeteer. All designs in Table 7 are such that they need less than 0.5% of the instruction widow time to choose a PSC. We calculate the power, area, and delay values using Cadence Genus and the SRAM array compiler for 22-nm GLOBALFOUNDRIES® (GF22FDX) [10]. These area and power overheads are much smaller compared to the area and power of the overall processor. In Table 8, we provide more details about the Puppeteer design. We show the number of each type of logic component and SRAM sizes for different configurations of Puppeteer. Note that the SRAM word length changes for the different-sized Puppeteer configurations because we require additional bits to address a larger number of nodes.

We would like to note that in most modern processors, the memory subsystem has multiple prefetchers at each level of the cache hierarchy [1–3, 7, 21, 26, 27, 65]. In Table 6, we show the overhead due to the prefetchers (not including Puppeteer) at each level of the memory hierarchy. This overhead is not unusual for today's server class of CPU architectures that use multiple prefetchers to target a variety of memory patterns.

5.7 Puppeteer Versus Other Managers

Power in caches. In Figure 14, we show the average power consumed in the caches for various static PSCs and the prior managers compared to Puppeteer. We calculate the power using Cadence Genus and the SRAM array compiler for GF22FDX® [10]. Compared to the no prefetching case,

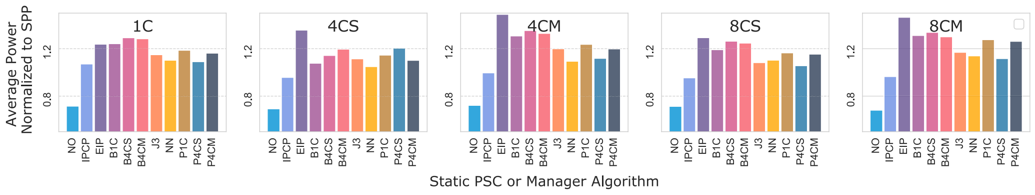


Fig. 14. Average power consumed in the caches (normalized to *SPP*) when using Puppeteer and prior work. Here, 1C = 1C data suite, 4CS = 4C single-type trace set data suite, 4CM = 4C mixed-type trace set data suite, 8CS = 8C single-type trace set data suite, and 8CM = 8C mixed-type trace set data suite.

Puppeteer has 65% higher power consumption overall. Compared to BT (average of B1C, B4CS, and B4CM), Puppeteer (average of P1C, P4CS, and P4CM) has 9% lower average power. NN and J3 have 6.3% and 2.2% lower average power than Puppeteer, and as we have discussed, they also have significantly lower average IPC. For the static PSC, EIP is extremely power hungry, with the highest average power being 20.4% more than Puppeteer. IPCP and NO have lower power consumption than Puppeteer, but they also have the worst performance among all options we have considered.

Scope. In Figure 15, we compare the prefetching scope of the prior works and Puppeteer. Here we determine scope as the number of total misses reduced by the prefetcher or prefetching system, divided by the total number of misses with prefetching disabled. We observe very limited scope across the four levels of the memory hierarchy for NN and J3. This is probably one of the reasons these options have lower performance than BT and Puppeteer. Comparing BT (average of B1C, B4CS, and B4CM) and Puppeteer (average of P1C, P4CS, and P4CM), in L1D$ and L2$, BT achieves a marginal difference with 0.5% broader scope than Puppeteer. In L1I$ cache, BT actually has 3.3% broader scope compared to Puppeteer. However, Puppeteer has 4.8% broader scope compared to BT in LLC. Intuitively, since LLC penalties are more important than L1I$, L1D$, and L2$ penalties, this is a better outcome. Experimentally, our results also support this outcome since Puppeteer has better performance than BT.

Accuracy. In Figure 16, we compare the prefetching accuracy of Puppeteer and the prior works. Here, accuracy of a prefetcher is measured as the number of misses that a prefetcher or prefetcher system has reduced compared to the case when prefetching is disabled, divided by the number of misses caused by the prefetcher. Puppeteer is better than the other options in L1I$ and L1D$, but comparable in L2$. In LLC, J3 fairs better than the other options with only 1.5% lower accuracy than Puppeteer. In case of BT and NN, Puppeteer achieves 21% better accuracy compared to BT and 28% better accuracy compared to NN. It is also of note that since the accuracy of Puppeteer in L1I$ is 6.7% better compared to BT, this better accuracy plays a role in compensating for the broader L1I$ scope of BT compared to Puppeteer.

5.8 Temporal Variations in PSC When Using Puppeteer

In Figure 17, we show the temporal behavior of P1C, B1C, and J3 while running 429.mcf-217B as an example. Figure 17(a) shows the percentage of time for which each PSC was used by each manager algorithm when executing 1.2B instructions of 429.mcf-217B. In Figure 17(b), we show the IPC values and the PSC used in each instruction window for a small slice of the same trace. We would like to note three interesting observations. First, from Figure 17(a), both B1C and P1C use *fnl-bingo-spp-no* for the 80% of the instruction windows, yet the performance difference between the two is around 20% over the whole trace. This means that the PSC chosen in the remaining 20% of the instruction windows have a larger influence on the overall performance. Second, in

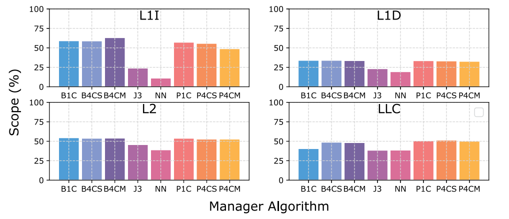


Fig. 15. Average scope of Puppeteer and the prior works in a 1C Processor. See Table 3 for the notations used along the *x*-axis.

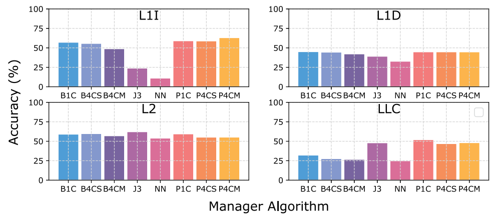
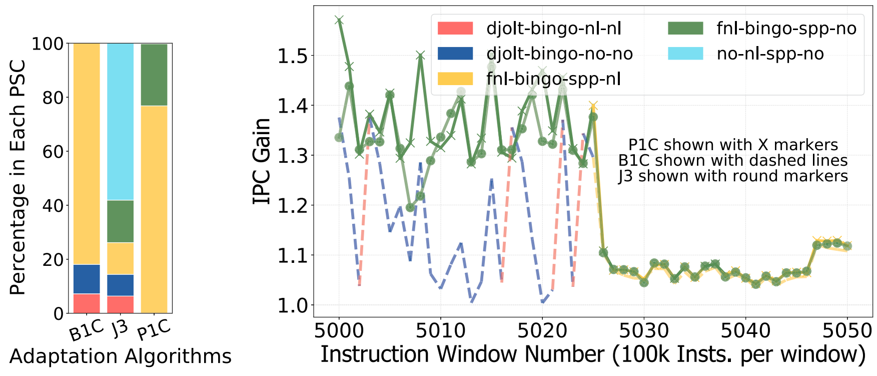


Fig. 16. Average prefetching accuracy of Puppeteer and the prior works in a 1C processor. See Table 3 for the notations used along the *x*-axis.



(a) PSC Percentage **(b)** Runtime Behavior

Fig. 17. Temporal behavior. Percentage usage of each PSC for the P1C, B1C, and J3 (a) and IPC gain when using P1C, B1C and J3 across 50 instruction windows and running 429.mcf-217B (b). For each plot line, for an instruction window we use color coding to indicate the PSC choice. The PSC descriptions are provided in Table 6.

the first 25 instruction windows after the 5,000th instruction window, there is a large variation in IPC gain when using B1C as compared to P1C, whereas all three algorithms converge to the same performance and same PSC during the last 25 instruction windows. This shows that P1C does a better job at predicting the PSC in different regions of an application. Third, J3 uses the same PSC as P1C but has lower performance in the first 25 instruction windows. This illustrates that changing the PSC has a cumulative effect on IPC. The choice of PSC made by P1C in prior instruction windows allowed P1C to gain more performance in the given instruction windows compared to J3.

5.9 Random Forest Regressors Versus Other Regressors

In this section, we compare (see Figure 18) the random forest regressor, which is our choice of the regression algorithm of Puppeteer, with three other regression algorithms: linear regressor [67], passive aggressive regressor [13], and gradient boosted regressor [40]. We compared the four regression algorithms for the 1C system. When used as part of Puppeteer, the random forest regressor has 1.7%, 1.2%, and 0.55% higher average IPC compared to linear regressor, passive aggressive regressor, and gradient boosted regressor, respectively. Furthermore, we observe 6.7%, 14.9%, and 8.2% worst-case IPC loss for linear regressor, passive aggressive regressor, and gradient boosted regressor, respectively, compared to the 5% worst-case IPC loss for random forest regressor. The

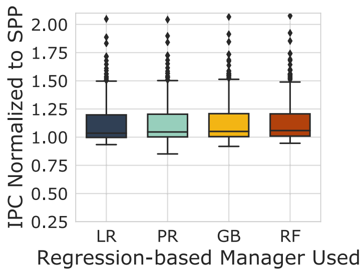


Fig. 18. Normalized performance of a 1C processor. Performance distribution of four different regression algorithms when used in Puppeteer. Here, LR stands for linear regressor, PR stands for passive-aggressive regressor, GB stands for gradient-boosted regressor, and RF stands for random forest regressor.

performance of Puppeteer could conceivably be further increased by mixing and matching different regressors for each PSC.

6 CONCLUSION AND FUTURE WORK

In this work, we introduce Puppeteer, a novel ML-based prefetcher manager designed using custom-tailored random forests. We train a dedicated random forest for each PSC, which allows the random forest to retain more information in a smaller amount of hardware. For the 232 traces that we evaluated, Puppeteer achieves an average performance gain of 46.0% in 1C, 25.8% in 4C, and 11.9% in 8C compared to a system with no prefetching. Puppeteer also reduces the number of negative outliers by 89%. As future work, we will explore a unified design of an ML-based manager that selects from an array of ML-based prefetchers.

ACKNOWLEDGMENTS

We want to thank the reviewers for their comments during the various iterations of this work. Thanks to Dr. Paul Keltcher for being an invaluable resource and providing great feedback.

REFERENCES

[1] AMD. 2017. AMD Ryzen Processor. Retrieved November 9, 2022 from https://www.amd.com/en/ryzen.

[2] AMD. 2020. Software Optimization Guide for AMD EPYC^{TM} 7001 Processors. Retrieved November 9, 2022 from https://developer.amd.com/wordpress/media/2013/12/55723_SOG_Fam_17h_Processors_3.00.pdf.

[3] AMD. 2022. GDC 2022 - AMD Ryzen^{TM} Processor Software Optimization. Retrieved November 9, 2022 from https://youtu.be/helEx02HN_I.

[4] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Bingo spatial data prefetcher. In *Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA'19)*. IEEE, Los Alamitos, CA, 399–411.

[5] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. 2021. Pythia: A customizable hardware prefetching framework using online reinforcement learning. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-54)*. 1121–1137.

[6] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. 2019. Perceptron-based prefetch filtering. In *Proceedings of the 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA'19)*. IEEE, Los Alamitos, CA, 1–13.

[7] Advanced Micro Devices Bios. 2010. Kernel Developer Guide (BKDG) for AMD Family 10h Models 00h-0fh Processors. Available at https://www.amd.com.

[8] Peter Braun and Heiner Litz. 2019. Understanding memory access patterns for prefetching. In *Proceedings of the International Workshop on AI-Assisted Design for Architecture (AIDArc) Held in Conjunction with ISCA*.

[9] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. 2006. Stealth prefetching. *ACM SIGPLAN Notices* 41, 11 (2006), 274–282.

[10] R. Carter, J. Mazurier, L. Pirro, J. U. Sachse, P. Baars, J. Faul, C. Grass, et al. 2016. 22nm FDSOI technology for emerging mobile, Internet-of-Things, and RF applications. In *Proceedings of the 2016 IEEE International Electron Devices Meeting (IEDM'16)*. IEEE, Los Alamitos, CA, 2.

[11] Standard Performance Evaluation Corporation. 2006. SPEC CPU® 2006. Retrieved November 9, 2022 from https://www.spec.org/cpu2006/

[12] Standard Performance Evaluation Corporation. 2017. SPEC CPU® 2017. Retrieved November 9, 2022 from https://www.spec.org/cpu2017/

[13] Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. 2006. Online passive aggressive algorithms. *Journal of Machine Learning Research* 7 (2006), 551–585.

[14] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2010. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. *ACM SIGPLAN Notices* 45, 3 (2010), 335–346.

[15] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2011. Prefetch-aware shared resource management for multi-core systems. *ACM SIGARCH Computer Architecture News* 39, 3 (2011), 141–152.

[16] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. 2009. Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 316–326.

[17] Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2009. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *Proceedings of the 2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, Los Alamitos, CA, 7–17.

[18] Babak Falsafi and Thomas F. Wenisch. 2014. A primer on hardware prefetching. *Synthesis Lectures on Computer Architecture* 9, 1 (2014), 1–67.

[19] Vladyslav Fedchenko, Giovanni Neglia, and Bruno Ribeiro. 2019. Feedforward neural networks for caching: N enough or too much? *ACM SIGMETRICS Performance Evaluation Review* 46, 3 (2019), 139–142.

[20] Ilya Ganusov and Martin Burtscher. 2005. Future execution: A hardware prefetching technique for chip multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. IEEE, Los Alamitos, CA, 350–360.

[21] Intel. 2011. Intel® 64 and IA-32 Architectures Software Developer‘s Manual. Available at https://www.intel.com.

[22] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning memory access patterns. *arXiv preprint arXiv:1803.02329*.

[23] Wim Heirman, Kristof Du Bois, Yves Vandriessche, Stijn Eyerman, and Ibrahim Hur. 2018. Near-side prefetch throttling: Adaptive prefetching for high-performance many-core processors. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 1–11.

[24] Jason Hiebel, Laura E. Brown, and Zhenlin Wang. 2019. Machine learning for fine-grained hardware prefetcher control. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–9.

[25] Ibrahim Hur and Calvin Lin. 2006. Memory prefetching using adaptive stream detection. In *Proceedings of the 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, Los Alamitos, CA, 397–408.

[26] Intel. 2017. Intel i9. Retrieved November 9, 2022 from https://www.intel.com/content/www/us/en/products/details/processors/core/i9.html

[27] Intel. 2017. Tuning Intel Xeon. Retrieved November 9, 2022 from https://community.intel.com/t5/Software-Tuning-Performance/How-to-control-the-four-hardware-prefetchers-in-L1-and-L2-more/td-p/1104586.

[28] Majid Jalili and Mattan Erez. 2022. Managing prefetchers with deep reinforcement learning. *IEEE Computer Architecture Letters* 21, 2 (2022), 105–108.

[29] Victor Jiménez, Roberto Gioiosa, Francisco J. Cazorla, Alper Buyuktosunoglu, Pradip Bose, and Francis P. O'Connell. 2012. Making data prefetch smarter: Adaptive prefetching on POWER7. In *Proceedings of the 2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. IEEE, Los Alamitos, CA, 137–146.

[30] A. Kagi, James R. Goodman, and Doug Burger. 1996. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA'96)*. IEEE, Los Alamitos, CA, 78–78.

[31] Hui Kang and Jennifer L. Wong. 2013. To hardware prefetch or not to prefetch? A virtualized environment study and core binding approach. *ACM SIGPLAN Notices* 48 (2013), 357–368.

[32] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A. L. Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path confidence based lookahead prefetching. In *Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE, Los Alamitos, CA, 1–12.

[33] Jinchun Kim, Elvira Teran, Paul V. Gratz, Daniel A. Jiménez, Seth H. Pugsley, and Chris Wilkerson. 2017. Kill the program counter: Reconstructing program behavior in the processor cache hierarchy. *ACM SIGPLAN Notices* 52, 4 (2017), 737–749.

[34] Sushant Kondguli and Michael Huang. 2018. Division of labor: A more effective approach to prefetching. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, Los Alamitos, CA, 83–95.

[35] Miron Bartosz Kursa. 2014. Robustness of random forest-based gene selection methods. *BMC Bioinformatics* 15, 1 (2014), 1–8.
[36] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When prefetching works, when it doesn't, and why. *ACM Transactions on Architecture and Code Optimization* 9, 1 (2012), 1–29.
[37] Shih-Wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu, and Hucheng Zhou. 2009. Machine learning-based prefetch optimization for data center applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage, and Analysis*. 1–10.
[38] Peng Liu, Jiyang Yu, and Michael C. Huang. 2016. Thread-aware adaptive prefetcher on multicore systems: Improving the performance for multithreaded workloads. *ACM Transactions on Architecture and Code Optimization* 13, 1 (2016), 1–25.
[39] Pranita Maldikar. 2014. *Adaptive Cache Prefetching Using Machine Learning and Monitoring Hardware Performance Counters*. Ph.D. Dissertation. University of Minnesota.
[40] Llew Mason, Jonathan Baxter, Peter Bartlett, and Marcus Frean. 1999. Boosting algorithms as gradient descent. In *Proceedings of the 12th International Conference on Neural Information Processing Systems (NIPS'99)*. 512–518.
[41] Francis B. Moreira, Matthias Diener, Philippe O. A. Navaux, and Israel Koren. 2017. Data mining the memory access stream to detect anomalous application behavior. In *Proceedings of the Computing Frontiers Conference*. 45–52.
[42] Tomoki Nakamura, Toru Koizumi, Yuya Degawa, Hidetsugu Irie, Shuichi Sakai, and Ryota Shioya. 2020. D-JOLT: Distant Jolt Prefetcher. Retrieved November 9, 2022 from https://research.ece.ncsu.edu/wp-content/uploads/sites/19/2020/05/D-JOLT.pdf.
[43] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. 2018. DeepCache: A deep learning based framework for content caching. In *Proceedings of the 2018 Workshop on Network Meets AI and ML*. 48–53.
[44] Thais Mayumi Oshiro, Pedro Santoro Perez, and José Augusto Baranauskas. 2012. How many trees in a random forest? In *Proceedings of the International Workshop on Machine Learning and Data Mining in Pattern Recognition*. 154–168.
[45] Samuel Pakalapati and Biswabandan Panda. 2020. Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In *Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)*. IEEE, Los Alamitos, CA, 118–131.
[46] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. 2015. Semantic locality and context-based prefetching using reinforcement learning. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. 285–297.
[47] Leeor Peled, Uri Weiser, and Yoav Etsion. 2019. A neural network prefetcher for arbitrary memory access patterns. *ACM Transactions on Architecture and Code Optimization* 16, 4 (2019), 1–27.
[48] Erez Perelman, Greg Hamerly, and Brad Calder. 2003. Picking statistically valid and early simulation points. In *Proceedings of the 2003 12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, Los Alamitos, CA, 244–255.
[49] Seth Pugsley et al. 2019. DPC3. Retrieved November 9, 2022 from https://dpc3.compas.cs.stonybrook.edu/.
[50] Seth Pugsley et al. 2020. ChampSim. Retrieved November 9, 2022 from https://github.com/ChampSim/ChampSim.
[51] Seth Pugsley et al. 2020. IPC1. Retrieved November 9, 2022 from https://research.ece.ncsu.edu/ipc/.
[52] Saami Rahman, Martin Burtscher, Ziliang Zong, and Apan Qasem. 2015. Maximizing hardware prefetch effectiveness with machine learning. In *Proceedings of the 2015 IEEE 17th International Conference on High Performance Computing and Communications, the 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and the 2015 IEEE 12th International Conference on Embedded Software and Systems*. IEEE, Los Alamitos, CA, 383–389.
[53] Marko Robnik-Šikonja. 2004. Improving random forests. In *Proceedings of the European Conference on Machine Learning*. 359–370.
[54] Joseph Rogers. 2019. Effects of an LSTM Composite Prefetcher. Retrieved November 9, 2022 from https://www.diva-portal.org/smash/get/diva2:1369282/FULLTEXT01.pdf.
[55] Alberto Ros and Alexandra Jimborean. 2020. The entangling instruction prefetcher. *IEEE Computer Architecture Letters* 19, 2 (2020), 84–87.
[56] André Seznec. 2020. The FNL+MMA Instruction Cache Prefetcher. Retrieved November 9, 2022 from https://research.ece.ncsu.edu/wp-content/uploads/sites/19/2020/05/FNLMMA-final.pdf.
[57] Mehran Shakerinava, Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Multi-lookahead offset prefetching. In the *Third Data Prefetching Championship (DPC3), in Conjunction with the International Symposium on Computer Architecture (ISCA'19)*. 1–4.
[58] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H. Pugsley, and Zeshan Chishti. 2015. Efficiently prefetching complex address patterns. In *Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*. IEEE, Los Alamitos, CA, 141–152.

[59] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. 2019. A Neural Hierarchical Sequence Model for Irregular Data Prefetching. Retrieved November 9, 2022 from https://www.cs.utexas.edu/~lin/papers/mlsys19.pdf.

[60] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. 2021. A hierarchical neural model of data prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 861–873.

[61] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2006. Spatial memory streaming. *ACM SIGARCH Computer Architecture News* 34, 2 (2006), 252–263.

[62] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2007. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, Los Alamitos, CA, 63–74.

[63] Ajitesh Srivastava, Angelos Lazaris, Benjamin Brooks, Rajgopal Kannan, and Viktor K. Prasanna. 2019. Predicting memory accesses: The road to compact ML-driven prefetcher. In *Proceedings of the International Symposium on Memory Systems*. 461–470.

[64] Dan Steinberg. 2009. CART: Classification and regression trees. In *The Top Ten Algorithms in Data Mining*. Chapman & Hall/CRC, 193–216.

[65] Gongjin Sun, Junjie Shen, and Alexander V. Veidenbaum. 2019. Combining prefetch control and cache partitioning to improve multicore performance. In *Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS'19)*. IEEE, Los Alamitos, CA, 953–962.

[66] Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. 2003. Guided region prefetching: A cooperative hardware/software approach. In *Proceedings of the 2003 30th Annual International Symposium on Computer Architecture*. IEEE, Los Alamitos, CA, 388–398.

[67] Sanford Weisberg. 2005. *Applied Linear Regression*. Vol. 528. John Wiley & Sons.

[68] Thomas F. Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2009. Practical off-chip meta-data for temporal memory streaming. In *Proceedings of the 2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, Los Alamitos, CA, 79–90.

[69] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely Jr., and Joel Emer. 2011. SHiP: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 430–441.

[70] W. A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News* 23, 1 (1995), 20–24.

[71] Yuan Zeng and Xiaochen Guo. 2017. Long short term memory based hardware prefetcher: A case study. In *Proceedings of the International Symposium on Memory Systems (MEMSYS'17)*. 305–311.

[72] Naifu Zhang, Kaibin Zheng, and Meixia Tao. 2018. Using grouped linear prediction and accelerated reinforcement learning for online content caching. In *Proceedings of the 2018 IEEE International Conference on Communications Workshops (ICC Workshops'18)*. IEEE, Los Alamitos, CA, 1–6.

[73] Pengmiao Zhang, Ajitesh Srivastava, Benjamin Brooks, Rajgopal Kannan, and Viktor K. Prasanna. 2020. RAOP: Recurrent neural network augmented offset prefetcher. In *Proceedings of the International Symposium on Memory Systems*. 352–362.

Received 29 January 2022; revised 28 July 2022; accepted 17 October 2022