Custom Tailored Suite of Random Forests for Prefetcher Adaptation

Furkan Eris[1], Marcia Sahaya Louis[1], Sadullah Canakci[1], Jose L. Abellan[2], and Ajay Joshi[1]
[1]Department of ECE, Boston University, USA, [2]Department of CS, Universidad Católica de Murcia, Spain
{fe, marcia93, scanakci, joshi}@bu.edu, {jlabellan}@ucam.edu

***Abstract*—To close the gap between memory and processors, and in turn improve performance, there has been an abundance of work in the area of data/instruction prefetcher designs. Prefetchers are deployed at each level of the memory hierarchy, but typically, each prefetcher gets designed without comprehensively accounting for other prefetchers in the system. As a result, individual prefetchers do not always complement each other, and that leads to low average performance gains and/or many negative outliers. In this work, we propose Puppeteer, which is a hardware prefetcher adapter 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. Compared to a design with no prefetchers, using Puppeteer we improve IPC by 34.6% on average across traces generated from SPEC2017, SPEC2006, and Cloud suites with $\sim$ 12KB overhead. Moreover, we also reduce the number of negative outliers by over 89%, and the performance loss of the worst-case negative outlier from 25% to only 5% compared to the state-of-the-art.**

I. INTRODUCTION

Instruction and data prefetching [11] are commonly used in today's processors to overcome the memory wall problem [26]. The key idea behind prefetching is to identify a memory access pattern and predict addresses to proactively fetch instructions and data into the cache. This hides the large memory access latency, and in turn improves processor performance.

To evaluate a prefetcher's performance, we can use scope and accuracy as the metrics. A prefetcher with high prefetching accuracy has limited scope, i.e., this prefetcher is very good at identifying a limited number of memory access patterns and can accurately prefetch instruction/data if those specific memory access patterns exist. However, such a prefetcher fails to identify other memory access patterns. Conversely, a prefetcher with broad scope, i.e, a prefetcher that caters to a variety of memory access patterns, has low accuracy [8], [16]. Given the diversity in the memory access patterns across applications, using a single prefetcher does not improve performance of all applications; in fact, in some cases it hurts application performance by predicting the wrong memory addresses for prefetching. These incorrect prefetches use up the precious memory bandwidth as well as the limited space we have in the L1 and L2 caches. This increases data and instruction access latency, which hurts application performance.

Effectively, we need to find a balance between accuracy and scope of the prefetcher. One way to balance scope and accuracy is to use multiple prefetchers, as it is in AMD and Intel processors [9], [12], where each prefetcher is customized to identify a specific type of memory access pattern and make a prefetching prediction. We can broaden the scope by having multiple different high accuracy prefetchers with limited scope for each prefetcher. 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 that they share microarchitectural resources, prefetchers can sabotage each other during runtime. A prefetcher may trigger prefetch requests that evict cache lines that have been accurately prefetched by another prefetcher. This leads to 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, and so a prefetcher's predictions can be influenced by the traffic generated by other prefetchers. These differences in temporal behavior can cause faulty synchronization among prefetchers and can lead to a drop in application performance.

One way to address these problems is to use only one prefetcher at a time at each memory level. This prefetcher is chosen based on the current traffic pattern. Kondugli et al. [16] proposed such a 'composite prefetcher', which uses a simple priority queue as their control algorithm to target a single memory level. Given that the priority queue of the prefetchers is designed offline using a given set of applications, introduction of unseen applications could lead to loss of performance. What is really needed is a 'coordinator' that chooses a set of prefetchers (one prefetcher per level) that are suitable for the current phase of an application and complement each other, and in turn improve performance. This 'coordinator' will effectively be responsible for determining 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 paper, we propose a machine learning (ML)-based hardware coordinator called Puppeteer to selectively switch ON/OFF prefetchers at each level in the memory hierarchy at runtime. Contrary to the prior work [17] that focuses on training the ML model to improve the prediction accuracy of the ML model, we train the ML model of Puppeteer to increase the overall system performance (quantified as instructions committed per cycle (IPC)). To coordinate the prefetchers across multiple cache levels at runtime, we propose a multi-regression ML-based approach. We use the observed IPC of the various prefetcher system configurations (PSCs)[1] for different phases of each application, to train our ML model. We train a unique random forest regressor per PSC to create a suite of

[1]A PSC specifies which prefetcher is switched ON at each level of the memory hierarchy in the system.

random forests regressors. For features used in the ML model, we use events whose behavior does not change with the choice of PSC, i.e PSC-invariant events. For example, the number of branch instructions in an application would not change with the choice of PSC. This limits the number of executions per trace we must account for during training, which makes it easier to train the ML model in Puppeteer. We design the ML model of Puppeteer with the goal of maximizing IPC while minimizing the area overhead required by the ML model. In summary, the contributions of our work are as follows:

- We propose a novel ML-based hardware coordinator called Puppeteer to improve the system performance. At runtime, at the end of an instruction window[2], Puppeteer predicts the IPC for each PSC and selects the PSC with the highest predicted IPC for the next instruction window.
- We design Puppeteer to use a set of PSC-invariant events, which can be tracked using hardware performance counters as inputs and predict the PSC for the next instruction window. Unlike prior work, Puppeteer chooses a prefetcher for each level in the cache hierarchy such that the prefetchers complement each other.
- 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.

II. Background and Related Work

Over the past 20-30 years, heuristic algorithms have been used for hardware adaptation, including for prefetcher throttling [10]. Heuristic algorithms are generally simple rule-based approaches with very low memory/area overhead and have fairly good performance for the average case. Unfortunately, the performance of heuristic algorithms has not scaled at the same rate as the complexity of the processor and the overlying applications. ML methods have been gaining traction in place of heuristic methods for prefetcher adaptation [8], [13], [14], [17]. The versatility and tolerance to variability of ML algorithms makes them a prime candidate to replace heuristic algorithms. ML algorithms can extract the non-intuitive interactions between the different prefetchers. Prior methods on prefetcher adaptation configure or train the adapter using values of hardware events collected for a single fixed PSC (generally, the default PSC) [8], [14], [17], [21]. However, at runtime the PSC changes. If the values of the hardware events are highly dependent on the PSC, 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, for e.g., the number of branch instructions) as features.

We observe a wide variation in the complexity of the ML algorithms used in prior work. Some of the algorithms are simple and either use small datasets or use datasets that do not accurately portray the runtime environment. As a result, these algorithms cannot achieve good accuracy at runtime [14], [17], [21]. Other algorithms, such as neural networks, are too complex and to achieve high accuracy, their size increases prohibitively with the size of the dataset [8]. Moreover, some prior works focus on hardware adaptation only from the perspective of accuracy without worrying about the hardware implementation [13], [14], [17], [21].

In our 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 micro-behavior. At the same time, Puppeteer is not too complex (as demonstrated in Section III-C) to implement in hardware and scales well with the size/complexity of the dataset.

III. Puppeteer Design

A. Puppeteer System Level Overview

In Figure 1, we show the system-level design of an example prefetching system that uses Puppeteer. The prefetchers track memory access patterns and prefetch data from main memory to last-level cache (LLC), from LLC to L2$, and from L2$ to L1$. These prefetchers can sometimes act overly aggressively, 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 of the prefetchers or memory bandwidth utilization to throttle prefetchers in such adverse scenarios [10].

Puppeteer works as a meta-controller and complements these heuristics-based throttling mechanisms. At runtime, Puppeteer periodically updates the PSC, i.e., 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 as inputs to determine the next PSC. Effectively, throttling mechanisms are used in prefetchers to regulate the short-term behavior of the prefetchers, while Puppeteer controls the longer-term system-level behavior using a more effective ML-based approach. With our approach we are capable of coping with changes in the long-term application behavior.

B. Puppeteer Algorithm

Regression vs Classification: To train Puppeteer, we can use a classification approach where we label the PSCs based on the probability that a PSC will give good performance or bad performance. However, sometimes the application phase performance is agnostic of the choice of the PSC, and at other times, the performance is very sensitive to the choice of PSC with as much as 613% change in IPC (as shown in Section V). Simply labeling using a "good performance class" and a "bad performance class" would be insufficient in these cases.

To create the training dataset, prior works [14], [17] use thresholding methods. Here, a trace[3] is run using all PSC options. A PSC is given a label of "1" if the IPC when using

[2]We set the instruction window size to 100,000 instructions.

[3]A trace is a group of instructions that represent a particular 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. We construct one or more unique representative traces from each benchmark [20]. In total, we construct 232 traces.

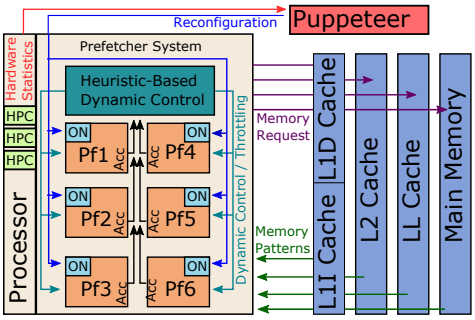


Fig. 1: System-level view of an example prefetching system that uses Puppeteer. Here Pf = prefetcher. Acc = Accuracy of the prefetchers.

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. Otherwise, the PSC receives a label "0". The classification algorithm is then trained for each PSC option and the associated labels. Unfortunately, using such a classification approach can lead to sub-optimal results. As an example, consider the case where we are classifying four traces, and selecting the PSC corresponding to the class of the traces. Let us say we classify three out of the four traces correctly and one incorrectly. So, our trace classification accuracy is 75%, and so three out of those four 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. 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.

To solve this issue, we propose to use regression instead of classification to account 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 will be trained on the IPC values directly, the magnitude information is not lost, and the regression algorithm can learn the magnitude of a good or bad prediction. Here the regression algorithm will be used to predict an IPC value for each one of the PSCs. We then choose the PSC with the highest predicted IPC value.

Suite of RF Regressors vs Single RF Regressor: When using regression algorithms, we have two options: (i) use a single regressor, where all data collected from all the 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 choose the correct PSC for each instruction window that increases the IPC, which indirectly jointly increases the scope and accuracy of the overall prefetching system.

In this work, we use a suite of regressors where we implement each regressor using random forest due to its simple implementation, its robustness to noise in the dataset, its lower overhead (compared to other ML algorithms such as neural networks), and its higher accuracy (compared to other ML algorithms such as decision trees). We have multiple trees per forest and we allow each tree to split at locations that are unique to that PSC. The leaves of each tree in the forest specify the predicted IPC value for the PSC. For each forest, we calculate the average of the predicted IPC values obtained from all the trees in the forest, and then choose the PSC with the highest average predicted IPC. Given that each forest has multiple decision trees, the tolerance of our method increases where even if some of the trees give wrong decisions, other trees can compensate this error.

Training Puppeteer: To train Puppeteer we need to generate a representative dataset. Consider the case where we have a single prefetcher, Pf, at only one memory level. Here $N_{psc} = 2$ with Pf=OFF or Pf=ON as the two PSCs. 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 $N_{trace} * N_{psc}^N$. When N increases, the number of different scenarios will increase exponentially. In order for the ML algorithm to choose the correct PSC for each instruction window during runtime, each unique scenario needs to be represented during training. Because of the exponential explosion of the number of unique scenarios, including each unique scenario during training is not feasible. This means offline training is inherently a difficult problem for prefetcher adaptation.

To handle this problem, we propose to use only the 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) for each PSC. We identify 59 events whose values vary by less than $\pm 10\%$ 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 I shows the 6 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 classes of our ML model, we collect an IPC value per PSC for each instruction window as our ground truth. We form our suite of random forests wherein we train a separate forest for each class (i.e. each PSC) using CART (classification and regression trees) [25] to minimize the IPC prediction error.

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.

C. Puppeteer Hardware Design

Figure 2 shows the hardware design of Puppeteer. We use a single port SRAM array called *Node MEM* to store information about Puppeteer. We load the Puppeteer model

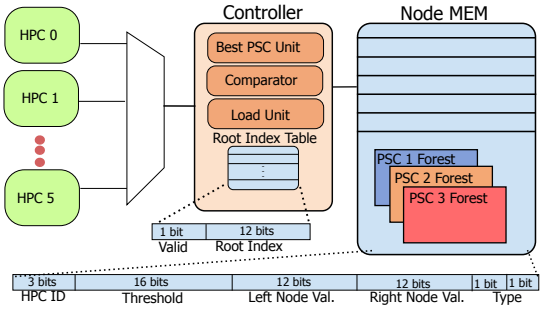


Fig. 2: Puppeteer hardware design.

into the *Node MEM* at startup using firmware. Each entry of *Node MEM* corresponds to one node in one of the random forests and it consists of the following fields: (i) A 3-bit `HPC ID` field that specifies which PSC-invariant event is used by that node to make a decision. The 3-bit encoding enables the node to use one of 6 different PSC-invariant events (see Table I). (ii) 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. (iii) A 12-bit (for 2250 node addresses) `Left Node Value(LNV)` field, and (iv) 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 (v) a 1-bit `Type` field. We use 1 bit each 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 (RIT)* 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*. Based on the *Comparator* output, we choose 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 above steps are repeated for the remaining trees in the forest, and then we calculate the average of the predicted IPC values obtained from all the 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 in Table I and discussed in detail in Section V). If we evaluate all 5 forests in series, where we will require a maximum 250 comparison operations (5 forests $\times$ 5 trees per forest $\times$ 10 comparisons = 250 comparisons), it will take less than 0.3% (assuming each comparison operation takes a clock cycle) of the total time required to execute the 100k instructions in the instruction window. Thus, we end up using the chosen PSC for 99.7% of the instruction window.

We need a total of 2250 nodes to design the trees in Puppeteer, and these nodes require a 12.75 KB-sized *Node MEM* (compared to a typical L1$ of 32 KB). Other than *Node MEM*, we require a $5 * N_{psc}$-entry *RIT* where each entry is 13-bit wide (12 bits for the root node index and 1 valid bit), a 12-bit comparator, a load unit, and a register to store the best PSC information in the *Controller*. For the *Node MEM* operating at $2GHz$, using Cacti 7.0 [4] we find that the area overhead in a $32nm$ process is $\sim 0.035mm^2$ and the power is $\sim 4.5nW$ for 500 *Node MEM* accesses every 100k instructions. The area and power required for the remaining components is negligible.

IV. Evaluation Methodology

For our evaluation, we train Puppeteer using data collected for a 1-core (1C) OoO processor and then evaluate it on a 1-core and 4-core (4C) OoO processor. We use ChampSim [5] for our analysis, where we model 1C and 4C processors to have multiple prefetchers at each level of the cache – private L1I$, L1D$, private L2$ and shared LLC (more details provided in Table I). We train and evaluate Puppeteer using a diverse set of traces generated from SPEC2017 [2], SPEC2006 [1], and Cloud [6] benchmarks. In total we have 232 (traces)$\times$10,000 (instruction windows per trace) = 2,320,000 instructions windows that make up our dataset. To make sure that our training algorithm is not overfitting the data, we severely limit the training set size by dividing the dataset into two disjoint sets - 10% of instruction windows form the training set and the remaining 90% of the instruction windows form the testing set and perform 10-fold cross-validation on the training set. For 4C experiments, we run the same trace on all four cores, hence the number of experiments is still 232.

To generate the 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 as well as the 1st (IPCP [19]), 2nd (Bingo [7]), and 3rd place (MLOP [24]) finalists of the 3rd data prefetching competition (DPC3) [3]; and the 1st (EIP) [22], 2nd (FNL+MMA) [23], and 3rd place (DJOLT) [18] finalists of the 1st instruction prefetching competition (IPC1) [6]. To reduce the hardware overhead of Puppeteer we avoid including

TABLE I: Simulated system parameters, prefetchers used at each cache level, the final PSCs that exist in Puppeteer, and the PSC-invariant events that are used by Puppeteer.

Component	Simulated Parameters
Core	One to four cores, 4GHz, 4-wide, 256-entry ROB
TLBs	64 entries ITLB, 64 entries DTLB, 1536 entry shared L2 TLB
L1I$	32KB, 8-way, 3 cycles, PQ: 8, MSHR: 8, 4 ports
L1D$	48KB, 12-way, 5 cycles, PQ: 8, MSHR: 16, 2 ports
L2$	512KB, 8-way, 10 cycles, PQ: 16, MSHR: 32, 2 ports
LLC	2MB/core, 16-way, 20 cycles, PQ: 32xcores, MSHR: 64xcores
DRAM	4GB 1 channel/1-core, 8GB 2 channels/multi-core, 1600 MT/sec

Final PSCs Used	L1I$	L1D$	L2$	LLC
djolt-bingo-nl-nl	DJOLT [18]	Bingo [7]	Next-line [11]	Next-line
djolt-bingo-no-no	DJOLT	Bingo	No Prefetcher	No Prefetcher
fnl-bingo-spp-nl	FNL+MMA [23]	Bingo	SPP [15]	Next-line
fnl-bingo-spp-no	FNL+MMA	Bingo	SPP	No Prefetcher
no-nl-spp-no	No Prefetcher	Next-line	SPP	No Prefetcher
Overhead	221KB	48.06KB	6KB	0.6KB

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.

multiple PSCs that cover the same traces. To this end, we initially run the traces for 20M instructions with all possible PSCs (5 L1I$ $\times$ 5 L1D$ $\times$ 6 L2$ $\times$ 2 LLC = 300 PSCs). We then choose the PSCs that have the best performance with minimal coverage overlap. Here coverage means the number of traces that the PSC "covers", i.e., the number of traces that have good performance while running with the given PSC. We reduce the number of PSCs by choosing a set of PSCs that provide the maximum coverage with the minimum overlap with each other. These PSCs are independent of the coordinator algorithm. In Table I we show the final 5 PSCs that we selected for our training set. In Table I, for each PSC, we show the prefetcher used at each cache level. We use the default hashed-perceptron for branch predictor and least-recently used (LRU) policy for cache replacement policy provided by ChampSim.

In our evaluation results, we normalize all IPC values to the same state of the art baseline as PPF [8], i.e., SPP (*no-no-spp-no*[4]). We compare Puppeteer against IPCP (*no-ipcp-ipcp-nl*), EIP (*EIP-nl-spp-no*), the final version of the algorithm that uses trial periods to latch onto the PSC [14] (J3), and finally an ML-based algorithm [17] (BT) - where Liao et al. tried several different ML methods (such as decision trees and NN) and conclude that decision trees are the best choice.

V. Evaluation Results

A. Puppeteer applied to 1C Processor

In Figure 3, we show the IPC distribution of various PSCs and prefetcher adapter designs. Broadly, compared to a processor with no prefetchers, Puppeteer provides an average performance gain of 34.6% (peak value of 613%). The true benefit Puppeteer is observed when we look closer into the distribution in Figure 4. We observe that when using BT, 53 traces lose performance and 6 out of those 53 traces lose more than 10% performance. This performance loss would make this solution unviable. Puppeteer has a worst-case loss of only 5% and only 8 traces in total have lower performance than SPP. This clearly shows that Puppeteer provides us with 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 work, we observe that Puppeteer provides 4.65%, 5.8%, and 5.1% average IPC gain over IPCP, EIP, and J3, respectively. One thing to note about J3 is that its average IPC gain is 2.1% lower than BT yet the negative outliers are less in both quantity and magnitude.

[4]PSC=4-tuple: $L1I\$_{prefetcher} - L1D\$_{prefetcher} - L2\$_{prefetcher} - LLC_{prefetcher}$.

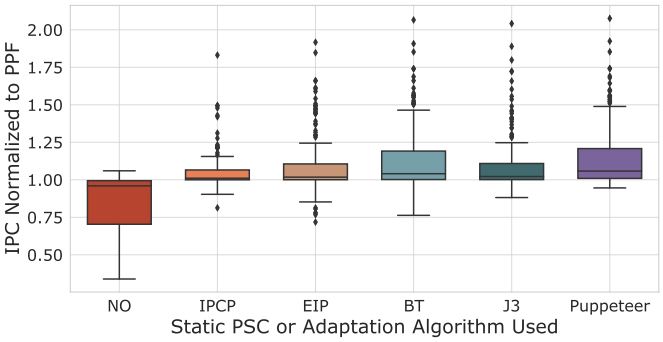


Fig. 3: Box Plot of 1C Experiments - Performance Distribution of Puppeteer and prior work normalized to *SPP*. Here NO = No Prefetching, IPCP = *no-ipcp-ipcp-nl*, EIP = *EIP-nl-spp-no*, BT = Liao et al. [17] and J3 = Jimenez et al. [14].

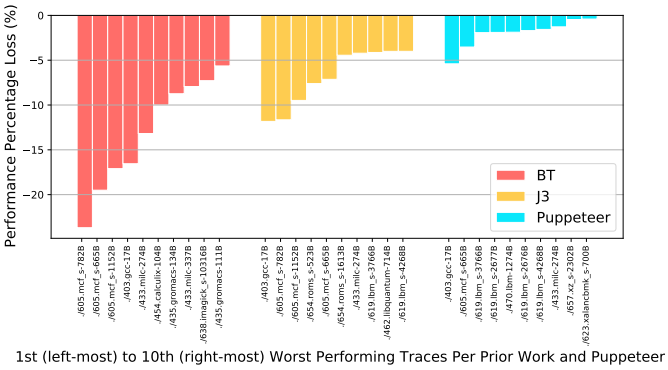


Fig. 4: Bottom Ten Performance Outliers - Performance normalized to *PPF* of 1C data suite experiments. Each group signifies the *Nth*-worst performance for the two adaptation algorithms from prior work we compare to and Puppeteer ordered 10th-worst (left-most) to 1st-worst (right-most). Smaller is better.

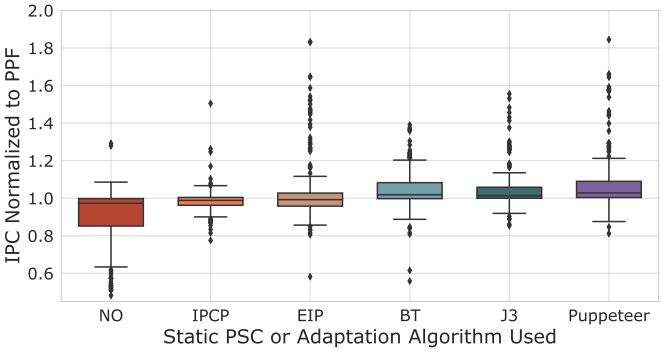


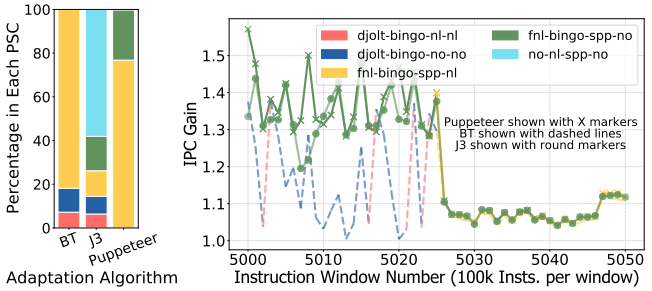
Fig. 5: Box Plot of 4C Experiments - Performance Distribution of Puppeteer and prior work normalized to *SPP*.

B. Puppeteer applied to 4C Processor

In Figure 5, we show the IPC distribution of Puppeteer and the various prior works while using the Puppeteer in a 4C system. We observe that the average IPC gain of Puppeteer over BT has gone up to 4.2%. The worst-case performance loss in BT has gone up to 45%, while the worst-case loss in the case of Puppeteer is at 19%. The number of traces that lose performance is 61 for BT while Puppeteer has just 24 traces that lose performance. Although Puppeteer has only been trained on 1C data, the algorithm is still capable of generalizing to 4C and providing 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 will increase exponentially. The experimental space makes it extremely challenging to train for all cases in 4C and beyond (in our experimental suite the number of unique combinations of traces is 232×232×232×232 = 2.89 billion for 4 core). Therefore, generalization using only 1C data is an important aspect to consider when comparing ML-based algorithms.

C. Temporal Behavior

In Figure 6 we show the temporal behavior of Puppeteer, BT, and J3 while running 429.mcf-217B as an example. Figure 6a shows the percentage of time for which each PSC was used by each adaptation algorithm when executing 1.2B instructions of 429.mcf-217B. In Figure 6b 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 phenomena: (i) Both BT and Puppeteer use *fnl-bingo-spp-no* for the same percentage of time, yet the performance difference between the two is around 20% over the whole trace. This means even a small percentage of the instruction windows may have a large influence on the overall performance. (ii) In the first 25 instruction windows there is a large variation in IPC when using BT and J3, while all three algorithms converge to the same performance and same PSC during the last 25 instruction windows. This shows that Puppeteer does a better job at predicting the PSC in different regions of an application. (iii) J3 uses the same PSC as Puppeteer yet 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 Puppeteer in prior instruction windows allowed Puppeteer to gain more performance in the given instruction windows compared to J3.



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

Fig. 6: (a) Percentage usage of each PSC for the three prefetcher adapters and (b) PSC choices for an example 50 instruction windows when running 429.mcf-217B.

VI. Conclusion and Future Work

In this work, we introduce Puppeteer, a novel ML-based prefetcher adapter 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. Puppeteer improves the performance of applications by 34.6% on average (613% peak value) when compared to a processor with no prefetching. Puppeteer also reduces the number of negative outliers by 89%. As future work, we will train Puppeteer using data collected from multi-core processors and perform a more extensive evaluation using systems with different core counts and mixed trace sets. In addition, we will explore a unified design of a ML-based coordinator that selects from an array of ML-based prefetchers.

References

[1] "SPEC CPU® 2006," https://www.spec.org/cpu2006/.
[2] "SPEC CPU® 2017," https://www.spec.org/cpu2017/.
[3] "DPC3," https://dpc3.compas.cs.stonybrook.edu/, 2019.
[4] "CACTI 7.0," https://github.com/HewlettPackard/cacti, 2020.
[5] "ChampSim," https://github.com/ChampSim/ChampSim, 2020.
[6] "IPC1," https://research.ece.ncsu.edu/ipc/, 2020.
[7] M. Bakhshalipour *et al.*, "Bingo spatial data prefetcher," in *Proc. HPCA*, 2019, pp. 399–411.
[8] E. Bhatia *et al.*, "Perceptron-based prefetch filtering," in *Proc. ISCA*, 2019, pp. 1–13.
[9] A. M. D. Bios., "kernel developer guide (bkdg) for AMD family 10h models 00h-0fh processors," 2010.
[10] E. Ebrahimi *et al.*, "Coordinated control of multiple prefetchers in multi-core systems," in *Proc. MICRO*, 2009, pp. 316–326.
[11] B. Falsafi and T. F. Wenisch, "A primer on hardware prefetching," *Synthesis Lectures on Computer Architecture*, vol. 9, no. 1, pp. 1–67, 2014.
[12] P. . Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, 2011.
[13] J. Hiebel *et al.*, "Machine learning for fine-grained hardware prefetcher control," in *Proc. ICPP*, 2019, p. 3.
[14] V. Jiménez *et al.*, "Making data prefetch smarter: Adaptive prefetching on power7," in *Proc. PACT*, 2012, pp. 137–146.
[15] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.
[16] S. Kondguli and M. Huang, "Division of labor: a more effective approach to prefetching," in *Proc. ISCA*, 2018, pp. 83–95.
[17] S.-w. Liao *et al.*, "Machine learning-based prefetch optimization for data center applications," in *Proc. SC*, 2009, p. 56.
[18] T. Nakamura, T. Koizumi, Y. Degawa, H. Irie, S. Sakai, and R. Shioya, "D-jolt: Distant jolt prefetcher."
[19] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *Proc. ISCA*, 2020, pp. 118–131.
[20] E. Perelman *et al.*, "Picking statistically valid and early simulation points," 2003, p. 244.
[21] S. Rahman *et al.*, "Maximizing hardware prefetch effectiveness with machine learning," in *Proc. HPCC*, 2015, pp. 383–389.
[22] A. Ros and A. Jimborean, "The entangling instruction prefetcher," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 84–87, 2020.
[23] A. Seznec, "The fnl+ mma instruction cache prefetcher," in *IPC-1-First Instruction Prefetching Championship*, 2020.
[24] M. Shakerinava *et al.*, "Multi-lookahead offset prefetching," *The Third Data Prefetching Championship*, 2019.
[25] D. Steinberg, "Cart: classification and regression trees," in *The top ten algorithms in data mining*. Chapman and Hall/CRC, 2009, pp. 193–216.
[26] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM CAN*, vol. 23, no. 1, pp. 20–24, 1995.