Maximizing Hardware Prefetch Effectiveness with Machine Learning

Saami Rahman*, Martin Burtscher†, Ziliang Zong‡, and Apan Qasem§
Department of Computer Science
Texas State University
San Marcos, TX 78666
* saami.rahman@txstate.edu, † burtscher@txstate.edu, ‡ ziliang@txstate.edu, § apan@txstate.edu

***Abstract*—Modern processors are equipped with multiple hardware prefetchers, each of which targets a distinct level in the memory hierarchy and employs a separate prefetching algorithm. However, different programs require different subsets of these prefetchers to maximize their performance. Turning on all available prefetchers rarely yields the best performance and, in some cases, prefetching even hurts performance. This paper studies the effect of hardware prefetching on multithreaded code and presents a machine-learning technique to predict the optimal combination of prefetchers for a given application. This technique is based on program characterization and utilizes hardware performance events in conjunction with a pruning algorithm to obtain a concise and expressive feature set. The resulting feature set is used in three different learning models. All necessary steps are implemented in a framework that reaches, on average, 96% of the best possible prefetcher speedup. The framework is built from open-source tools, making it easy to extend and port to other architectures.**

I. Introduction

A prefetcher monitors and extrapolates streaming access patterns in applications and preemptively brings data into higher levels of cache to hide memory latency. Because of the disparity between CPU and memory speeds, prefetching has always been an important technique for code optimization. The significance of prefetching has increased on current multicore architectures. As an increasing number of cores are being attached to the same memory system, the number of candidate streams for prefetching, often from different threads in the same program, has increased as well. Effective coordination of multiple prefetch streams can not only hide memory latency but also directly affect parallel efficiency.

Although prefetching is a critical transformation on current architectures, determining optimal prefetch parameters poses several challenges. First, a prefetcher must be accurate in predicting memory access patterns. If the prefetcher is incorrect, it can result in increased memory traffic, and more importantly, cause contention for space in the small and valuable cache. Second, a prefetch instruction must be timely. If a prefetch causes data to be present in a higher level of memory earlier than required, it may be evicted to accommodate more urgently required data. These challenges are amplified further in multithreaded programs. Since lower levels of memory such as L2 can be shared across multiple threads, each of which may potentially request data from different parts of memory, it can be difficult to correctly identify memory access patterns.

In this paper, we address these challenges and devise a strategy for effective prefetching based on machine-learning techniques. We study the effect of prefetching on the PARSEC programs [1] as well as on two idealized programs that exhibit sequential and randomized access patterns. We examine the performance when enabling individual hardware prefetchers and combinations thereof on an Intel processor with four built-in prefetchers. We train several machine-learning algorithms to obtain recommendations on which prefetchers should be used for a given program.

A key step to successfully using an machine-learning-based approach is to characterize programs in a quantitative manner that captures their essential differences. We employ hardware performance events for this purpose. Since there are hundreds of such events available, we designed a simple yet effective procedure to prune the number of events needed to characterize a program. We demonstrate the efficacy of our pruning algorithm by testing different feature sets on a decision tree. Finally, we develop a framework that recommends hardware prefetching configurations for unseen programs on the basis of previously seen training programs. On average, this framework can help the user gain up to 96% of the achievable speedup provided by the hardware prefetchers.

There are several advantages of using such a recommendation system to adjust the hardware configuration. First, it enables us to use existing hardware more effectively. Second, it does not require any source-code changes of the program being optimized. Rather, it works directly on the executable. Lastly, the framework relies on open-source technologies, making it is easy to extend and port to other architectures.

II. Related Work

Prefetching is a widely explored area, and the benefits of prefetching are broadly documented and studied. Lee et al. [2] were one of the first researchers to introduce the concept of data prefetching in hardware to hide the memory-access latency. Chen et al. [3], in their early work, studied the effects of data prefetching in both hardware and software. Numerous of work on data prefetching has been done since then.

With the introduction of multicore processors, the effects of prefetching have become more interesting. Prefetching can

The authors acknowledge support from the National Science Foundation through awards CNS-1253292 and CNS-1305302.

hurt performance, and the ill-effects of prefetching have been studied [4], [5]. Puzak et al. [6] demonstrate cases where prefetching hurts and propose a characterization of prefetching based on timeliness, coverage, and accuracy. Lee et al. [7] conducted an in-depth investigation of when and why prefetching works. They performed an extensive analysis of both software and hardware prefetching performance on the SPEC CPU2006 benchmark programs, which are serial workloads.

Jayasena et al. [8] demonstrated the effectiveness of a decision tree to prune performance events used as features for detecting false sharing. Their approach has motivated us to include decision trees in our work. Cavazos et al. [9] developed a machine-learning model to find the best optimization configuration for the SPEC CPU2006 benchmark suite. The goal of their work was to select a set of compiler optimizations that results in good performance improvement. They report high classification accuracies when using performance events as features for a learning algorithm. Milepost GCC [10] is a large project to further crowdsourced optimization learning. It uses static features to characterize programs. Similar work by Demme et al. [11] uses graph clustering based on the data and control flow of programs to characterize program behavior. However, such characterizations are difficult to perform as they involve modifying the compiler or working on an intermediate representation of a program, making them hard to port.

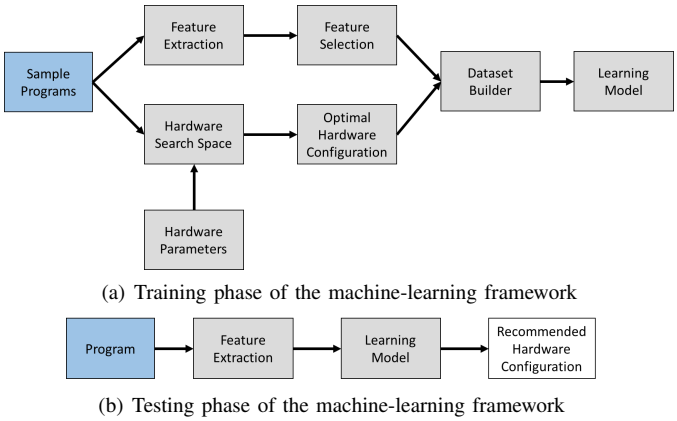
Among the works we have studied, the following two are the most similar to ours. McCurdy et al. [12] characterized the impact of prefetching on scientific applications using performance events. They experimented with a combination of several benchmark programs on AMD processors. Their work primarily focuses on serial workloads, but they also studied the effects of running multiple serial programs simultaneously. Their work hinges on successfully isolating performance events that are expressive enough to capture the effects of prefetching. They hand-picked the performance events for the AMD architectures they worked on and would have to repeat the process of studying all available performance events for any new architecture. Liao et al. [13] presented a machine-learning-based approach to selecting the optimal prefetch configuration in a way similar to ours. However, their work also hinges on identifying architecture-specific performance events, which they did by hand. In addition, their work focuses on serial workloads whereas we are interested in parallel programs.

III. Machine-Learning Framework

Figure 1 shows the major tasks that our machine-learning framework performs. There are two key phases: training and testing. In the training phase, sample programs are run to generate features. Moreover, the programs are run with all possible prefetcher settings to determine the performance and identify the optimal hardware configuration. This information is used to train the learning models. The testing phase is simpler. First, an unseen program is run once to collect its features. Second, these features are fed into the trained model to obtain a recommendation of prefetch configuration.

A. Prefetcher Configurations

Many modern processors implement hardware prefetchers. For example, our Intel Core2 processor is equipped with the following four prefetchers:



(a) Training phase of the machine-learning framework

(b) Testing phase of the machine-learning framework

Fig. 1. Operation of machine-learning framework

1) **Data cache unit (DCU) prefetcher**: this prefetcher attempts to recognize a streaming algorithm and prefetches the next line into the L1 data cache.
2) **Instruction pointer (IP) based stride prefetcher**: this prefetcher tracks individual load instructions and attempts to detect strided accesses. It can detect strides of up to 2kB and prefetches into the L1 cache.
3) **Spatial prefetcher (CL)**: this prefetcher attempts to complete a cache line brought into the L2 by fetching the paired line to form a 128-byte aligned chunk.
4) **Stream prefetcher (HW)**: this prefetcher attempts to detect streaming requests from the L1 cache and brings in anticipated cache lines into the L2 and LLC.

It should be noted that more recent Intel architectures, such as SandyBridge and Haswell, include the same prefetchers. Interestingly, it is not defined which of these four prefetchers are enabled or disabled by default [13]. In this work, we define a *configuration* to be the enabled/disabled state of the prefetchers, expressed by a four-bit bitmask, where each bit represents a prefetcher. The value 1 means the corresponding prefetcher is enabled and a 0 means it is disabled. From most significant bit to least, the mapping of bits to prefetchers is: HW, CL, DCU, and IP. We controlled the prefetcher configuration of the processor using the open-source tool *Likwid* [14].

It may seem that turning on all prefetchers, i.e., configuration `1111`, should result in the best performance. However, other configurations can be superior. In fact, Figure 2 shows that, in some cases, configuration `1111` hurts performance. The reported speedup values are runtime improvements over our baseline configuration, which is `0000`. For each configuration, we carried out three runs and used the best runtime to shield the results somewhat from operating system jitter. Since the run yielding the best performance is not only the least interrupted but also the run that finished the task in the shortest amount of time, we strive to improve upon this value.

Several observations can be made from Figure 2. First, within the same program, changing the prefetcher configuration can have a significant impact on performance. For example, `freqmine` and `streamcluster` show large variances in performance when the configuration is altered. Second, configuration `1111` is not always the best option, as can be seen in the cases of `random`, `stream`, `streamcluster`,

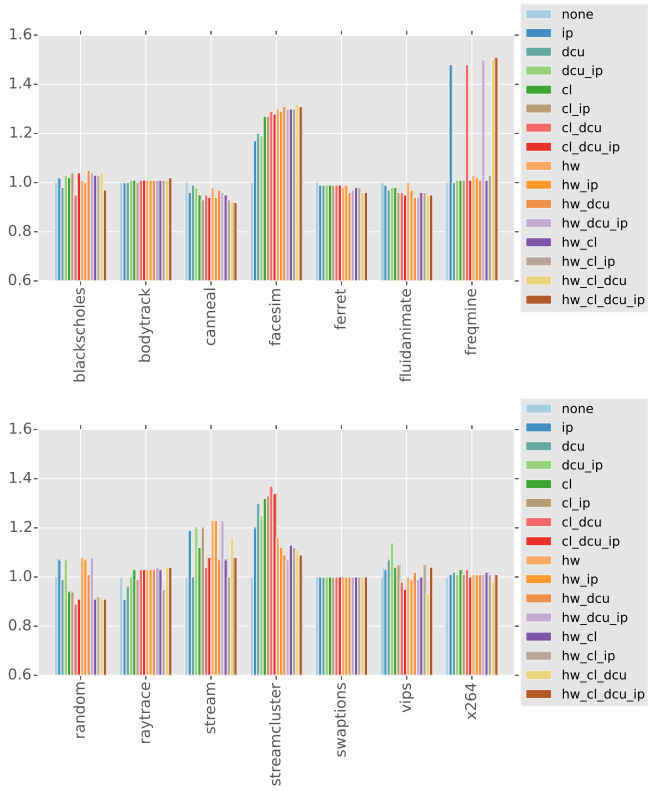


Fig. 2. Performance of all prefetcher configurations relative to no prefetching

and `vips`. Third, configuration `0000` can be the best configuration, that is, using prefetching can hurt performance, as in the case of `canneal`, `fluidanimate`, and in several configurations of `random`. Finally, in several cases, there are multiple "good" configurations, that is, several configurations perform almost as well as the best.

B. Feature Extraction

We measured all performance events supported by our processor and used them as the initial feature set to characterize programs. Counting events such as L2 cache misses and stalled CPU cycles provides quantified insight into a program's behavior and has previously been employed to characterize programs [8], [9], [12], [13]. However, using all available events is problematic as it takes a long time to measure them and the supported events vary across processors. We use the pruning algorithm described below to address these problems.

Once generated, the features are normalized to event counts per million executed instructions. This scaling is necessary to be able to compare different programs and different runs of the same program. For instance, one hundred page faults may not be significant for a program that executes one billion instructions, but it will be significant for a program that only runs ten thousand instructions. For the program with ten thousand instructions, the normalized value will be *higher* compared to the program with one billion instructions, thus correctly capturing the difference in significance.

Figure 3 shows the count for all events for our 14 programs. The x-axis represents the events. The axes are removed as this figure is used only as a visualization. Two key observations can be made from the figure. First, many events have similar values, therefore they do not carry discriminatory information for program behavior. Second, many of the events result in a very high count and cause other events with smaller ranges to disappear. We address this issue with feature scaling.

C. Feature Selection

We used Algorithm 1 to prune the feature set. It consists of two subroutines: `FindEvents` and `EventsUnion`. `FindEvents` takes the event counts from two programs as input. For every distinct event, it checks if the two counts differ by at least ϕ, where ϕ is the relative difference expressed in the interval (0.0, 1.0). If they do, the event is included in a set. At the end of the subroutine, the set contains all events that sufficiently capture the distinctiveness between the two programs. The `Diff` function returns the absolute difference of its arguments divided by their maximum. We experimentally found $\phi = 0.95$ to work well and exclusively use this threshold.

The `EventsUnion` subroutine takes a list of programs as input and, for all possible pairs of these programs, calls the `FindEvents` subroutine. It includes the returned sets in a multiset, which eventually contains the *expressive* events for all program pairs. Next, the `EventsUnion` subroutine employs a map to count the number of times each event occurs in the multiset. Finally, the map is sorted by the counts.

This algorithm is driven by the idea that, if an event has highly distinct counts between two programs, it captures an important aspect in which the two programs differ, and we should consider that event. However, there are many events that differ significantly for *any* two programs, and we end up with a large number of events. This is why we sort the map to prioritize the events by how often they are useful to distinguish between programs. From the sorted map, we can easily select the top most important events to build our ultimate feature set.

We use a final preprocessing step, called feature scaling, before the learning stage. Learning models are known to struggle when there is a large variance in the numeric ranges of the features. For example, if one feature has a range of 1 to 10 and another feature in the same feature set has a range of -10,000 to +10,000, the learning model might struggle to draw accurate decision boundaries. As a remedy, we scale all features in feature vector x using the following formula:

$$x'_i = \frac{x_i - \mu(x)}{max(x) - min(x)} \text{ for } i = 1 \text{ to } len(x)$$

where $\mu(x)$ is the average of the values in feature vector x. After scaling, all features are within the range -1 to 1.

Figure 4 shows the effect of feature scaling. The plots are significantly different from those in Figure 3. Also, the differences in program behavior are now much more apparent. For example, without scaling, the plots for `blackscholes` and `bodytrack` are not as distinguishable as they are with scaling. Additionally, `canneal` now appears substantially different from both `bodytrack` and `blackscholes`. This is because many event counts in `canneal` are not 0, contrary to how it appears in Figure 3. Similarly, the plots for `swaptions`, `vips`, and `x264` look very similar without

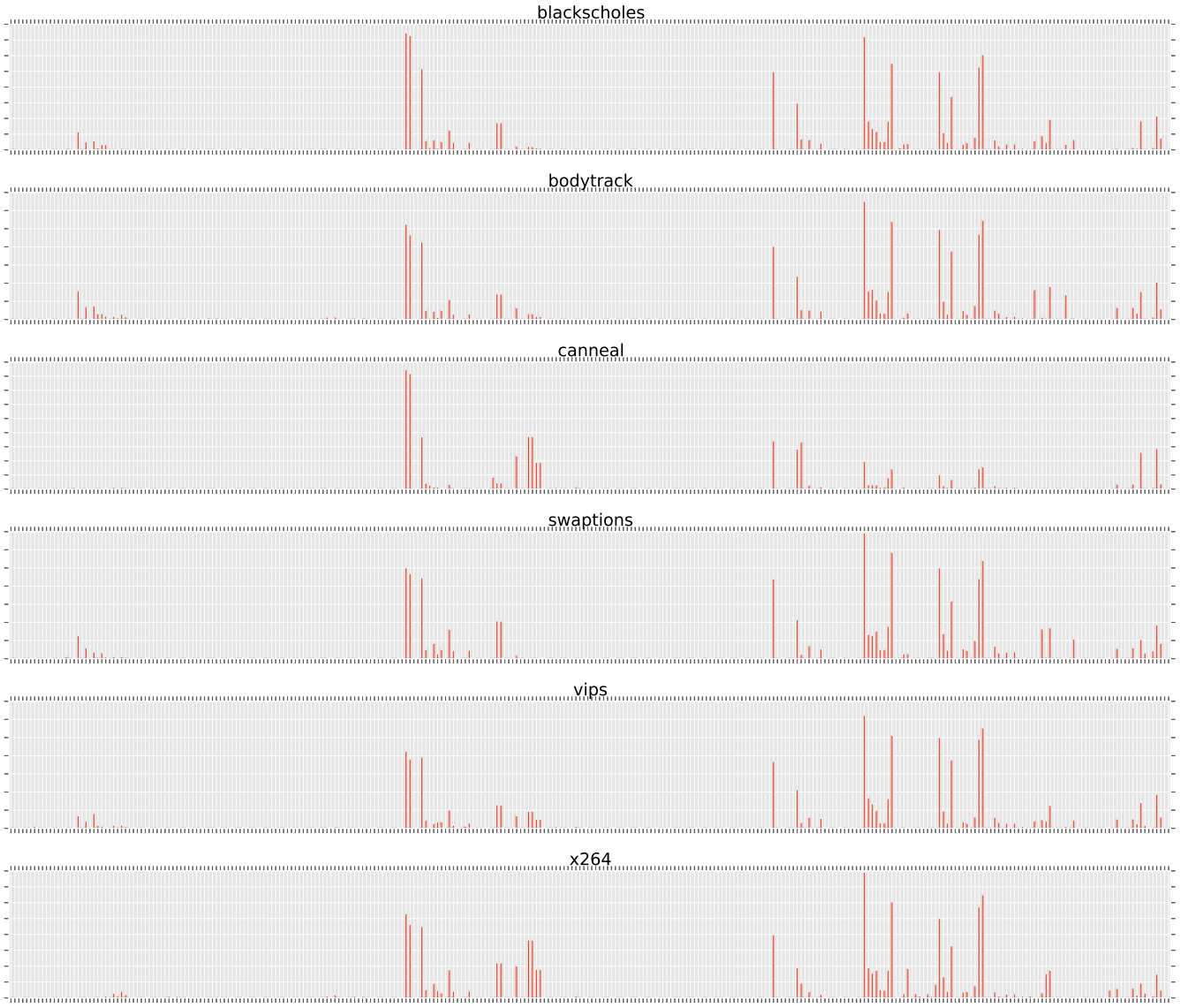


Fig. 3. Event-count visualization before feature scaling

scaling but are quite different once the features are scaled. The same is true for the other programs. The plots of the unscaled features are sparse whereas the plots of the scaled features differ more substantially between programs and are denser.

D. Learning Models

The final component of our framework is a learning model. We evaluated two different traditional learning models, logistic regression and decision trees, and designed a Euclidean-distance-based classifier that is tailored to our needs and captures the available information well.

1) Training Labels: The learning target is formulated in two ways for two different purposes. In the first formulation, we use a binary classifier that specifies whether a program benefits from prefetching or not. If the best prefetching configuration results in a speedup of at least 10% over configuration `0000`, the program is said to benefit from prefetching. According to this metric, programs `ferret`, `swaptions`, `fluidanimate`, and `canneal` do not benefit. We use this formulation to decide how many of the top events found by Algorithm 1 to use (cf. Section IV.C). In the second formulation, we inspect the effect of each prefetcher in isolation. If enabling a specific prefetcher for a given program results in a speedup of at least 2% over configuration `0000`, we classify the prefetcher as useful on that program. This yields four instances of learning models, one per prefetcher. Table I shows the resulting training labels. It also lists the best configuration for each program. Note that learning models should be trained on multiple examples for each class label. This is why we use four instances rather than a single instance that includes all combinations of prefetchers, i.e., 16 distinct class labels, which would require a lot more than our 14 programs to train.

2) Euclidean-Distance-Based Model: We use the Euclidean distance as a similarity metric for predicting good prefetcher configurations for previously unseen programs. The motivation behind this approach is that the four independent

Fig. 4. Event-count visualization after feature scaling

TABLE I. TRAINING LABELS FOR OUR FOUR INDEPENDENT CLASSIFIERS AND ACTUAL BEST CONFIGURATION

Program	Training Labels				Actual best configuration
	hw	dcu	cl	ip	
blackscholes	0	1	0	1	1010
bodytrack	0	0	0	0	1111
facesim	1	1	1	1	1110
ferret	0	0	0	0	0000
freqmine	1	0	0	1	1111
raytrace	1	1	0	0	1011, 1110, 1111
swaptions	0	0	0	0	0000
fluidanimate	0	0	0	0	0000, 1000
vips	0	1	1	1	0011
x264	0	1	1	0	0100, 0110
canneal	0	0	0	0	0000
streamcluster	1	1	1	1	0110
random	1	0	0	1	1000, 1011
stream	1	1	0	1	1000, 1001, 1011

classifiers do not capture the interaction between the prefetchers. Comparing the classifiers to the best configuration in Table I illustrates this problem, i.e., the combination of the four classifiers often is not the configuration that results in the maximum speedup. For example, consider `raytrace`, where the individually recommended prefetchers are `hw` and `dcu`, which corresponds to configuration `1100`. However, the combination of these two prefetchers does not yield the best performance. To counteract this problem, our Euclidean-distance-based model calculates the distance between the unseen program and every known program from the training set. It then uses these distances as weights to compute a score for each of the 16 possible prefetcher configurations. Concretely, the model computes the following 16-element vector:

$$Score = \left[\sum_{i=1}^{m} \frac{P_i[1]}{d_i^2}, \sum_{i=1}^{m} \frac{P_i[2]}{d_i^2}, \ \ldots \ \sum_{i=1}^{m} \frac{P_i[16]}{d_i^2} \right]$$

Algorithm 1 Finding events that differ between two programs

Require: E_a and E_b contain measurements of the same events in the same order

```
function FINDEVENTS(E_a, E_b, φ)
    set ← {}
    for i ← 1 to E_a.size do
        if DIFF(E_a[i].value, E_b[i].value) ≥ φ then        ▷ relative difference
            set.INCLUDE(E_a[i].name)
        end if
    end for
    return set
end function
function EVENTSUNION(programs, φ)
    multiset ← {}
    map ← MAP()                                  ▷ key: event name, value: occurrence
    numProgs ← programs.length
    for i ← 1 to numProgs − 1 do
        for j ← i + 1 to numProgs do
            set ← FINDEVENTS(programs[i], programs[j], φ)
            multiset.INCLUDE(set)
        end for
    end for
    for set in multiset do
        for event in set do
            if map.CONTAINS(event.name) then
                map[event.name] += 1
            else
                map[event.name] = 1
            end if
        end for
    end for
    return SORT(map)                                        ▷ sorts by values
end function
```

where m is the number of programs in the training set. P_i is a 16-element vector associated with program i. Each element in this vector corresponds to a configuration, which is simply the element's index in binary notation. The element's value represents the fraction of the achievable speedup that was reached when using the corresponding configuration on program i. For example, $P_2[3]$ represents the fraction of speedup that was obtained on the 2^{nd} training program using configuration `0011`, since 3_{10} = 0011_2. Finally, d_i denotes the Euclidean distance between the unseen program and the i^{th} training program.

The resulting vector contains a weighted score for each prefetching configuration. The recommended configuration is the binary representation of the index belonging to the maximum element. This approach combines the effect of every prefetching configuration on all training programs and qualifies the scores using distance squared. Thus, the prefetching performance of similar programs carries a greater weight than that of dissimilar programs. Moreover, if an unseen program is close to multiple programs, our approach ensures that the most similar program is not the sole basis for the recommendation.

IV. Results and Analysis

A. Experimental Environment

We performed our measurements on a 2.4 GHz Intel Core2 Quad Q6600 processor with eight 32 kB L1 caches and two 4 MB L2 caches. All programs were compiled using GCC 4.8.2 with optimization level -O2 on an Ubuntu 14.04 operating system. The PARSEC programs were invoked using the `parsecmgmt` script that ships with the suite and run with the `native` input on eight threads. We repeated the experiments with different numbers of threads, but no significant change in prefetching effectiveness was observed.

TABLE II. Effect of varying the number of features on precision, recall, and accuracy

Featureset	Precision	Recall	Accuracy
All events	0.50	0.40	0.64
Top 2	0.25	0.20	0.50
Top 3	0.33	0.20	0.57
Top 4	0.40	0.40	0.57
Top 5	0.40	0.40	0.57
Top 6	0.75	0.60	0.78
Top 7	0.75	0.60	0.78
Top 8	0.80	0.80	0.85
Top 9	0.80	0.80	0.85
Top 10	0.80	0.80	0.85
Top 20	0.60	0.60	0.71
Top 30	0.50	0.60	0.64

B. Evaluation Method

We used (k - 1) cross-validation to asses our learning models. Initially, we focused on the prediction accuracy. However, this does not capture the quality of the recommendation as multiple prefetching configurations may yield near-optimal performance. For instance, the best configuration for `streamcluster` is `0110` giving a speedup of 1.37, but `0111` results in 1.34. Therefore, we found it more useful to look at what fraction of the achievable speedup can be obtained using the recommendations from the learning model.

C. Choosing the Optimal Number of Features

To test the efficacy of the feature-selection procedure in Algorithm 1, we trained and tested a decision tree using the top two events and fed the model with an increasing number of events to identify the saturation point. This point is reached when we use the top eight events, as shown in Table II. Using more than eight events degrades the learning performance. This demonstrates that our pruning algorithm is able to identify events that truly capture the differences between programs. Henceforth, we use the top eight events as our feature set except in the case of the Euclidean-distance-based model, which we found to work best with just the top six events.

D. Recommending a Good Prefetcher Configuration

Figure 5 shows the performance of the three learning models in terms of how close the recommended configuration's speedup is to that of the best possible configuration.

The logistic regression model reaches, on average, 92.4% of the achievable speedup. It performs poorly on `freqmine` because of how the training labels are derived. Applying the `cl` and `dcu` prefetchers separately on this program does not yield a speedup, which is why the models are trained to turn them off. However, good configurations for `freqmine` have both prefetchers turned on. Hence, this is a case where two prefetchers that do not result in significant speedup individually deliver a combined speedup that is greater than their sum.

The decision-tree-based model performs better than the logistic regression model on average, reaching 95.3% of the

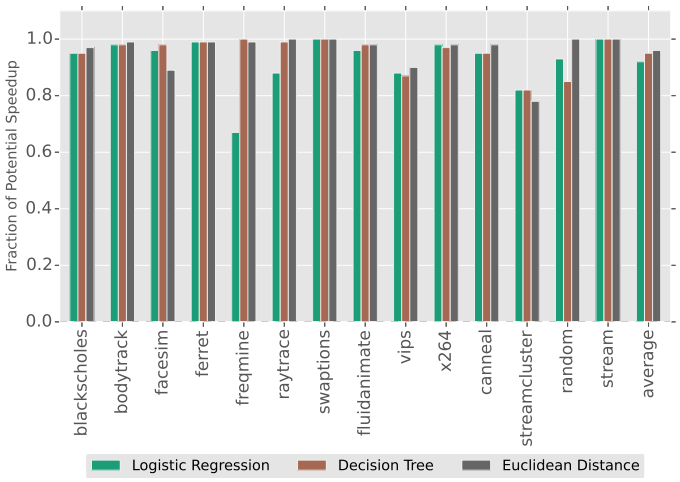


Fig. 5. Fraction of achievable speedup reached by the three learning models

achievable speedup. For `freqmine`, it is able to suggest the best configuration. However, this has likely occurred by chance as information about the interaction between prefetchers is not carried into the independent models.

The Euclidean distance model reaches 96.1% of the achievable speedup on average. It is able to suggest a good configuration for `freqmine` because it takes into account the similarity between the test program and multiple training programs. Moreover, it utilizes information about the performance of every possible prefetcher configuration on all training programs. In contrast, the other classifiers attempt to predict the best configuration based on the impact of the individual prefetchers.

The Euclidean distance model performs substantially worse than the other models on `facesim`. This is because it determines `random` to be the most similar program, i.e., well-performing configurations for `random` greatly influence its decision. When we inspected the model internally, we found the second highest recommendation to be `0110`, which results in 98% of the achievable speedup for `facesim`. Clearly, the similarity metric turned out to be imperfect in this case.

All models perform poorly on `streamcluster`. The logistic regression and decision tree classifiers suffer from the aforementioned problem because the training label is `1111` whereas the actual best configuration is `0110`. The Euclidean distance model suffers because `streamcluster` is significantly different from every other program, i.e., there is no similar program in the training set. It is closest to `stream`, but applying the best configuration from `stream` on `streamcluster` results in a speedup of 1.07, which is only 78% of the best possible speedup of 1.37 using `0110`.

V. Summary

This paper presents a framework to help users maximize the effectiveness of the hardware prefetchers in their systems. The processor on which we conducted our experiments contains four such prefetchers, resulting in 16 possible prefetching configurations as each prefetcher can be enabled or disabled individually. Our framework performs an exhaustive search of these 16 combinations and records the performance of each configuration on a set of training programs. It uses hardware performance events to characterize codes for the purpose of determining similarities between a previously unseen application and the training programs. Since modern processors support hundreds of performance events, our framework employs a pruning algorithm to construct a concise yet expressive feature set. This feature set, combined with the performance of the various prefetcher configurations, is used to train three machine-learning models. One of them is a Euclidean-distance-based model that we designed specifically for recommending prefetcher configurations. On average, its recommendations deliver 96% of the possible prefetching speedup on the PARSEC benchmark suite and two additional programs.

References

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[2] R. L. Lee, P.-C. Yew, and D. H. Lawrie, "Multiprocessor cache design considerations," in *Proceedings of the 14th annual international symposium on Computer architecture*. ACM, 1987, pp. 253–262.

[3] T. Chen and J. Baer, "A performance study of software and hardware data prefetching schemes," in *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on*. IEEE, 1994, pp. 223–232.

[4] H. Kang and J. L. Wong, "To hardware prefetch or not to prefetch?: a virtualized environment study and core binding approach," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 357–368.

[5] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-aware shared resource management for multi-core systems," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3, pp. 141–152, 2011.

[6] T. R. Puzak, A. Hartstein, P. G. Emma, and V. Srinivasan, "When prefetching improves/degrades performance," in *Proceedings of the 2nd conference on Computing frontiers*. ACM, 2005, pp. 342–352.

[7] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn't, and why," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 1, p. 2, 2012.

[8] S. Jayasena, S. Amarasinghe, A. Abeyweera, G. Amarasinghe, H. De Silva, S. Rathnayake, X. Meng, and Y. Liu, "Detection of false sharing using machine learning," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 30.

[9] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters," in *Code Generation and Optimization, 2007. CGO'07. International Symposium on*. IEEE, 2007, pp. 185–197.

[10] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather *et al.*, "Milepost gcc: machine learning based research compiler," in *GCC Summit*, 2008.

[11] J. Demme and S. Sethumadhavan, "Approximate graph clustering for program characterization," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 21, 2012.

[12] C. McCurdy, G. Marin, and J. Vetter, "Characterizing the impact of prefetching on scientific application performance," in *International Workshop on Performance Modeling, Benchmarking and Simulation of HPC Systems (PMBS13)*, 2013.

[13] S. Liao, T.-H. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou, "Machine learning-based prefetch optimization for data center applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 56.

[14] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*. IEEE, 2010, pp. 207–216.