Machine Learning-Based Prefetch Optimization for Data Center Applications

Shih-wei Liao[1,2], Tzu-Han Hung[3], Donald Nguyen[4], Chinyen Chou[2], Chiaheng Tu[2], and Hucheng Zhou[5]

[1]Google Inc., Mountain View, California, USA

[2]National Taiwan University, Taipei, Taiwan

[3]Princeton University, Princeton, New Jersey, USA

[4]University of Texas at Austin, Austin, Texas, USA

[5]Tsinghua University, Beijing, China

Email addresses: sliao@google.com, thhung@cs.princeton.edu, ddn@cs.utexas.edu, r96079@csie.ntu.edu.tw, d94944008@csie.ntu.edu.tw, zhou-hc07@mails.tsinghua.edu.cn

Performance tuning for data centers is essential and complicated. It is important since a data center comprises thousands of machines and thus a single-digit performance improvement can significantly reduce cost and power consumption. Unfortunately, it is extremely difficult as data centers are dynamic environments where applications are frequently released and servers are continually upgraded.

In this paper, we study the effectiveness of different processor prefetch configurations, which can greatly influence the performance of memory system and the overall data center. We observe a wide performance gap when comparing the worst and best configurations, from 1.4% to 75.1%, for 11 important data center applications. We then develop a tuning framework which attempts to predict the optimal configuration based on hardware performance counters. The framework achieves performance within 1% of the best performance of any single configuration for the same set of applications.

1. INTRODUCTION

Data center operators leverage economies of scale to increase performance per watt. The large scale of data center applications encourages operators to be meticulous about application performance. A single-digit improvement can obviate thousands of machines. However, optimizing data center applications is a delicate task. To increase machine utilization, operators host several applications on the same physical machine, but this may reduce aggregate performance by increasing contention on shared hardware resources. Cloud Computing introduces third-party applications into the data center. This adds new dimensions to the configuration space and adds external dependencies into the optimization problem. Consolidation is driven by application scale, the need for high utilization and emerging applications. Consolidation dually multiplies the impact of application optimizations and increases its difficulty.

We propose a system to optimize applications according to data center constraints. We model optimization as a parameter search problem and use machine learning techniques to predict the best parameter values. This framework is designed for seamless integration into existing data center practices and requires minimal operators' intervention.

In this paper, we focus on optimizing memory system performance by configuring memory prefetchers. As hardware moves towards integrated memory controllers and higher bandwidth, configuring prefetchers properly will play an important rule in maximizing system performance. Processor manufacturers or data center operators may design or tune prefetchers for aggressive prefetching. This produces superior performance for applications that are not limited by memory bandwidth, but aggressive prefetching produces pathological behavior for applications that have finely tuned memory behavior. Due to the impact of scale, programmers of data center applications commonly apply memory optimizations that increase the effective utilization of bandwidth like data structure compaction and alignment, and software prefetching. Aggressive prefetching can destroy programmers' careful orchestration of memory.

We have found few papers that address the impact of prefetcher configuration from the perspective of the application programmer or the data center operator, even though proper configuration has a significant performance impact. Instead, most papers address the concerns of hardware designers. Choosing the right configuration, however, has a large impact on performance. On a collection of 11 data center applications, we find a range of improvement between 1.4% to 75.1% comparing the worst to best performing configurations.

We apply machine learning to predict the best configuration using data from hardware performance counters, and we show a methodology that predicts configurations that achieve performance within 1% of the per-application best configuration.

To address the context of dynamic data centers, we consider co-hosted applications, dynamic application mix and large-scale deployments. For instance, when a job is scheduled to be run next, the general tuning framework either predicts or looks up this job's best configuration. In this paper we present the general framework but only report the results on isolated application executions. That is, evaluation on a more dynamic environment is the subject of future papers. To summarize, this paper consists of the following contributions:

1. A system that significantly improves application performance transparently to the application programmer and with minimal overhead to data center operators.

2. A study of the performance of real data center applications under varying hardware prefetcher configurations.

3. An instantiation of machine learning, including selecting the training dataset and problem formulation, that predicts near-optimal prefetch configurations.

The rest of the paper is organized as follows. Section 2 introduces our general framework for parameter space optimization and machine learning-based prediction. Section 3 demonstrates the application of the proposed framework to hardware prefetchers. We present and analyze experimental results in Section 4. Section 5 summarizes the related work in parameter search and optimization. Finally, we conclude this paper in Section 6.

2. PARAMETER SPACE EXPLORATION VIA MACHINE LEARNING

This section describes an application tuning framework for data center operators that uses machine learning for rapid parameter space exploration.

2.1 Framework and Methodology

In the data center, the environment for program optimization is different from traditional compiler optimization. Application programmers give program binaries and execution constraints to a data center operator who, in turn, must allocate sufficient machine resources to execute the application and satisfy the execution constraints while maximizing the effective utilization of the data center as a whole.

To facilitate scheduling, the details of the underlying machine hardware are obscured from the programmer. Even if these details were available, taking advantage of the information would be difficult. In the data center, tasks migrate from machine to machine, fail on one machine and are restarted on another, and are co-hosted with other applications. Additionally, the data center operators are concerned with optimizing the performance of the data center as a whole.

Thus, we propose a simple application tuning framework based on applying machine learning over system-level events like hardware event counters and kernel statistics. This approach is particularly suitable for data center operators because application binaries do not need to be recompiled or modified. Also, the inputs to the machine learning algorithms, hardware and kernel statistics, measure system-level effects that are most relevant to a data center operator. The framework easily extends to more inputs, and since it does not require modifying application binaries, it applies widely to many applications and deployment environments.

The framework is as follows:

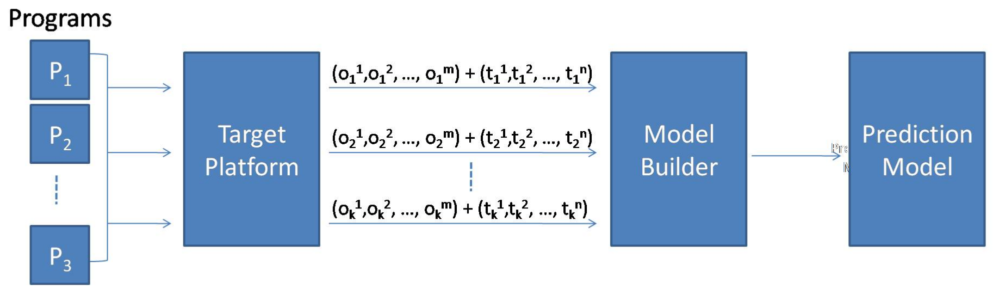
1. *Specify the tunables.* The tunables are the parameters that can be adjusted to improve the performance (throughput, latency, etc.) of a program. The optimization choices can be at the level of source code, library, system, or processor. The setting of the tunables will be the output of the machine learning engine.

2. *Identify the observables.* The observables are events that can be measured by the framework and are dependent on the tunables. For example, cycles per instruction is an appropriate observable if a data center operator wants to improve throughput. If the memory system performance is of interest, the cache miss rate is an appropriate observable. The observables, also called as *features* in the machine learning terminology, will be the input of the machine learning engine.

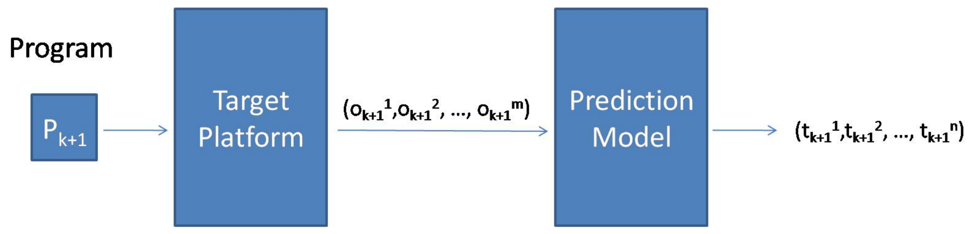
3. *Create a training dataset for machine learning algorithms.* Given a set of applications, the framework runs the applications under varying configurations of tunables and collects the observables specified by the user. The benchmark set should be representative of the applications likely to be seen in the data center. We discuss some issues with selecting application benchmarks in Section 3.4. To produce a training dataset, the framework applies exhaustive search or randomized sampling to find the best or good tunable configurations. Because the space of hardware prefetcher configurations is small, we use exhaustive search in this paper. The observable data and tunable configurations form the training dataset for a machine learning engine. Note that exhaustive search without any statistical techniques is not a general solution for dynamic data centers, but is an acceptable one for creating the training dataset.

4. *Build the prediction models.* After collecting the training dataset, the framework runs machine learning algorithms to construct prediction models. Section 2.2 summarizes the algorithms we use in this experiment. Figure 1(a) illustrates the procedure of model construction. The framework gives each algorithm a set of observables and tunables and asks the algorithm to generate a prediction model that, when given a set of observables, predicts the best set of tunables.

5. *Use the models for prediction.* Now, in the data center, when an application is executed, its execution will be monitored by the framework. Initially, the application will run with a default configuration of tunables, but the framework will collect observable data during execution. After a predetermined period of monitoring, the framework will predict the best tunable configuration by applying the collected observable data to the prediction model. Figure 1(b) illustrates the procedure of predicting the tunables of an incoming application.


Programs
P1
P2
P3
Target Platform
(o1^1,o1^2, ..., o1^m) + (t1^1,t1^2, ..., t1^n)
(o2^1,o2^2, ..., o2^m) + (t2^1,t2^2, ..., t2^n)
(ok^1,ok^2, ..., ok^m) + (tk^1,tk^2, ..., tk^n)
Model Builder
Prediction Model

(a) Workflow of framework construction.



(b) Workflow of framework evaluation.

Figure 1: Framework construction and evaluation. The symbol o denotes an *observable* and the symbol t denotes a *tunable*.

2.2 Machine Learning Algorithms

Many different machine learning algorithms can be used to generate the prediction model. We selected a large set of algorithms and evaluated their performance over our training set. In this section, we summarize the algorithms we considered.

We evaluated several classical machine learning algorithms. The nearest neighbor (NN) [1] algorithm measures the similarity of each data instance using Euclidean distance and classifies a new data instance according to its nearest neighbor in the training dataset. The naïve Bayes (NB) [12] method computes the probability that a given data instance belongs to each class and classifies a new data instance according to the class with the highest probability. A tree-based classifier takes a divide-and-conquer strategy. The C4.5 decision tree (DT) [18] classifier uses a divide-and-conquer strategy. It partitions the training dataset based on selecting pivots that result in the greatest information gain. The partitioning is then recursively applied to each sub-partition. Alternatively, a rule-based classifier uses a separate-and-conquer technique. The Ripper classifier (RP) [4] greedily attempts to construct a rule to classify as many instances as possible within a class. It then separates out the instances that were classified and continues on with those still unclassified.

We evaluated many modern machine learning algorithms as well. Support vector machines (SVM) [17] treat the input data as vectors in an N-dimensional space and find a maximum-margin hyperplane to separate the vectors into two classes, one on each side of the hyperplane. Logistic regression (LR) [19] can predict a categorical type and capture the non-linear relationship between the input features and output predictions. We also evaluated two types of neural networks. One is the typical multi-layer perceptron (MLP) [8] network with backward propagation. The other is the radial basis function (RBF) [2] network which consists of three layers: an input layer, a hidden layer where the inputs are applied to the RBF functions, and an output layer which produces the linear combination of the outputs from the RBF functions.

3. APPLICATION TO HARDWARE PREFETCHERS

We apply the framework introduced previously to find the best configuration of hardware prefetchers for a given application. In this section, we define the tunables and observables used and explain how to build the training dataset.

3.1 Platform

Table 1 lists the major components of our experimental platform. One computing server consists of 2 or 4 sockets. Each socket is an Intel Core 2 Quad CPU. Core 2 Quad comprises 4 CPU cores, equipped with a shared L2 cache for every two cores. Each CPU core has a 32KB L1 data cache and a 32KB L1 instruction cache. Each of the two shared L2 caches is 4MB. All the programs were compiled by GCC 4.3.1 with -O2 flag. We monitor execution and collect hardware events with perfmon2 [16].

3.2 Tunables: Hardware Prefetchers

On an Intel Core 2 CPU, there are four hardware prefetchers [10]:

1. Data Prefetch Logic (DPL) prefetcher: When the processor detects stride access, this prefetcher fetches streams of instructions and data from memory to the unified L2 cache.

2. Adjacent Cache Line (ACL) prefetcher: This prefetcher fetches the matching cache line in a cache line pair (128 bytes) to the L2 unified cache.

3. Data Cache Unit (DCU) prefetcher: This prefetcher fetches the next cache line into the L1 data cache

CPU Model	Intel Core 2 Quad
L1 cache (Data and Instruction)	256KB (4 * (32KB + 32KB))
L2 cache	8MB (2 * 4MB)
Main Memory	16 GB
Operating System	Linux (Kernel version: 2.6.18)
Compiler	GCC 4.3.1 (Flags: -O2)
Performance Monitoring Software	perfmon2

Table 1: Details of the platform.

when it detects multiple loads from the same cache line within a fixed period of time.

4. Instruction Pointer-based (IP) prefetcher: This prefetcher keeps sequential load history information per instruction pointer and uses this data to decide whether to fetch data for an upcoming delinquent load instruction into the L1 data cache.

Each processor core has one DCU prefetcher and one IP prefetcher for its L1 cache and also one DPL prefetcher and one ACL prefetcher for the shared L2 cache. Each hardware prefetcher can be enabled or disabled independently, so there are 16 prefetcher configurations for each core. For the remainder of this paper, we encode the configurations with four bits: bit 0 represents the setting of DPL prefetcher; bit 1, the ACL prefetcher; bit 2, the DCU prefetcher; bit 3, the IP prefetcher. The bit value 0 denotes that the corresponding prefetcher is enabled, and the bit value 1 denotes that the corresponding prefetcher is disabled. For example, the prefetcher configuration `Config-1010` means that the DPL and DCU prefetchers are enabled and the ACL and IP prefetchers are disabled.

In practice, there is no one default configuration for the hardware prefetchers across different models of machines. For example, the default configuration for many of the Intel Q-series Core 2 CPU is `Config-0000` (all prefetchers enabled), but the default configuration for many of the Intel Core 2 E-series is `Config-0011` (the two L1 prefetchers enabled). The configuration can also be changed by BIOS setup. We have observed other preferred configurations such as `Config-0110` (DPL and IP enabled) used as the default in many machines as well. The variety of default configurations combined with strong impact of prefetchers on performance (shown in Section 4) suggest that, at the very least, one should take the prefetcher configuration into consideration when evaluating application performance.

3.3 Observables: Hardware Events

On an Intel Core 2 CPU, there are hundreds of hardware events that can be measured [10]. It is impractical and unnecessary to monitor all the hardware events provided by the Core 2 architecture. Instead, we carefully select potential indicators that help guide the optimization of the prefetcher configuration (see Table 2).

It is difficult to reason about the exact relationship between these indicators and prefetcher performance. For example, if an application runs on a machine with all prefetchers disabled, a high cache miss rate may be a good indicator that some of the prefetchers should be turned on to bring data into cache earlier and reduce the cache miss penalty. However, in an environment where all prefetchers are enabled, a high cache miss rate suggests that the some of the prefetchers should be turned off, because the cache misses are likely caused by a too aggressive prefetcher setting. A high cache miss rate may also just be indicative of poor spatial locality of the application. We only highlight some of the difficulties interpreting cache miss rate. Similar analysis can be done for the rest of the selected hardware events.

The relationship between these observables and performance also depends on the particular processor. Different architectures have different performance models, and relationships derived on one architecture do not necessarily follow on another architecture. The general difficulty establishing a relationship between observed application behavior and optimal prefetcher configuration drives us towards machine learning.

3.4 Data Collection

Before applying our application tuning framework, we must address two issues: what is the training dataset and what is the data format. The ideal training dataset should be as representative and large as possible. Currently, we use profiles (i.e., hardware events) collected from running all programs in the SPEC CPU2006 [22] benchmark, with both *train* and *ref* input data for each program, to build our prediction models. We believe including other representative benchmarks such as SPECweb2009 [23] or TPC-C [24] could further refine the prediction models.

In SPEC CPU2006, CINT2006 and CFP2006 represent two different classes of applications in terms of their performance and other characteristics. We build models with profiles from CINT2006 (`INT`), CFP2006 (`FP`) and the whole CPU2006 suite (`ALL`) to evaluate the sensitivity of the prediction model to the training dataset.

We must also resolve how to formulate the prefetcher configuration problem to the machine learning algorithms. Naïvely, we can consider the problem as selecting the best configuration from the space of 16 possible choices. However, for some programs, the performance difference between two configurations may be negligible. For example, `401.bzip2` with `Config-0010` is only 0.25% slower than with `Config-0011`. Hence, for our experiments, we also try to specify a set of *good* prefetcher configurations that achieve performance within 0.5% to the best configuration, instead of using one single *best* configuration. The task of the machine learning algorithm is to predict good configurations.

Most of the machine learning algorithms mentioned in Section 2.2 can only predict one variable at a time, which dictates that each algorithm chooses one configuration out of 16 possible candidates. But, we also consider decomposing the prediction into choosing the best configuration for each of the four prefetchers individually. We can combine the four prediction results to select the overall prefetcher configuration.

The three problem formulations are summarized below:

1. Single (`S`): specify the single optimal prefetcher configuration.

2. Multiple (`M`): specify a set of near-optimal prefetcher

Event	Description	Relationship with Hardware Prefetching
CPI	Clock per (retired) instruction	Prefetching may improve or worsen CPI
L1D_LD_MISS	L1 data cache miss events due to loads	Prefetching may reduce cache misses (when the prefetching delivers required data to the cache on time) or increase cache misses (when prefetching delivers the wrong data to the cache)
L1D_ALLOC	Cache lines allocated in the L1 data cache	Similar to L1 cache miss
L1D_M_EVICT	Modified cache lines evicted from the L1 data cache	Similar to L1 cache miss
L2_LD_MISS	L2 cache miss events due to loads	Similar to L1 cache miss
L2_LINES_IN	Number of cache lines allocated in the L2 cache (due to prefetching)	Suppressing prefetching might be helpful when this number is too high
L2_LINES_OUT	Number of cache lines evicted from the L2 cache (due to prefetching)	Suppressing prefetching might be helpful when this number is too high
DTLB_MISS	DTLB miss events due to all memory operations	A high DTLB miss suggests to not use prefetching because the program working set is large and unpredictable
DTLB_LD_MISS	DTLB miss events due to loads	Similar to DTLB miss
BLOCKED_LOAD	Loads blocked by a preceding store (with unknown address or data) or by the L1 data cache	Prefetching may not be useful if there are lots of blocked load instructions
BUS_TRAN_BURST	Number of completed burst transactions	Prefetching may deteriorate the use of bus bandwidth
BR_PRED_MISS	Branch mis-prediction events	Prefetching on a wrong path (due to branch misprediction) may unnecessarily pollute the cache

Table 2: Hardware events that are measured in the experiments.

configurations.

3. Independent (`I`): use four instances of the training dataset and independently specify the configuration (enabled or disabled) of each prefetcher.

4. EXPERIMENTAL RESULTS

We use another set of benchmarks, our data center applications (`DCA`), to evaluate the benefit of our prediction models. Table 3 summarizes the applications, whose total codebase is more than one million lines of code. Among many data center applications, we selected 11 of them that were representative of the overall application mix. Since we are concerned with optimizing the total system performance, we weigh each application according to its relative and approximate execution time within a data center.

In this section, we examine the effects of using different training datasets and problem formulations when constructing the prediction models. Then, we analyze in depth the performance improvement of each application in `DCA`. Finally, we carefully examine a decision tree generated by a machine learning algorithm and see if it supports the domain knowledge of an expert.

To report meaningful performance improvement, all the performance numbers are compared with the per-application best results. When calculating the overall performance for `DCA` suite, we use the *weighted geometric mean* for all the applications in `DCA` to reflect the true enhancement observed in our data center environment. The weights of `DCA` applications are listed in Table 3.

4.1 Evaluation of Training Dataset

Figure 2 compares the speedup (i.e., the improvement of wall-clock running time) of our `DCA` testing dataset using prediction models trained on the `INT`, `FP`, and `ALL` training datasets for various machine learning algorithms and using the `M` problem formulation.

From our experiments, `INT` is the best training dataset. The reason is that the programs in `INT` are the most similar to the applications in `DCA`. They share similar characteristics, not only in terms of the high-level tasks (e.g., searching, counting, etc.) but also in terms of the low-level hardware events (e.g., CPI, cache miss rate, branch mis-prediction, etc.) that are observed by the performance monitoring software. The similarity of these programs is a strong indication that models built from the `INT` dataset can best guide the prediction of prefetcher configuration for our data center applications.

On the other hand, models built with the `FP` dataset are consistently worse than those built with the `INT` dataset. From the high-level point of view, the floating-point programs tend to have more regular structure (e.g., control flow and data access patterns), which is not the case for applications in the `DCA` set. These disparities are also visible in the low-level hardware events. For example, the event count of L2_LINE_IN and L2_LINE_OUT are both much higher for `FP`. The regular data access patterns give the hardware prefetchers more opportunities to fetch data into and evict data from the L2 cache. The poor representation of `DCA` by `FP` leads to poor predictive models as well. In many cases, models trained with `FP` predict configurations that do worse than the default prefetcher configuration (see `NN`, `NB`, `MLP`, `RBF`).

App	Category	Weight	Gap	App	Category	Weight	Gap
PageScorer	Searching	∼20%	24.5%	PhraseQueryServer	Searching	∼20%	23.6%
ContentIndexer	Indexing	∼1%	24.5%	Clustering	Indexing	∼1%	11.1%
DocCluster	Learning	∼5%	24.2%	KeywordCluster	Learning	∼5%	51.8%
HugeSort	Sorting	∼1%	50.6%	DataProcessor	MapReduce	∼20%	8.9%
ContentAds	Ads	∼5%	1.4%	StreetImaging	Imaging	∼1%	75.1%
OfflineContent	OCR	∼5%	44.0%				

Table 3: Data center applications with categories. Weights represent approximate contributions of applications or similar application types to the total execution time of a data center. We partition applications into three groups: heavily used (weighted ∼20%), typically used (weighted ∼5%), and rarely used (weighted ∼1%). Naturally the first group represents a wider range of weights than the latter two groups do. Gaps represent the performance difference between the best and worst prefetch configurations.

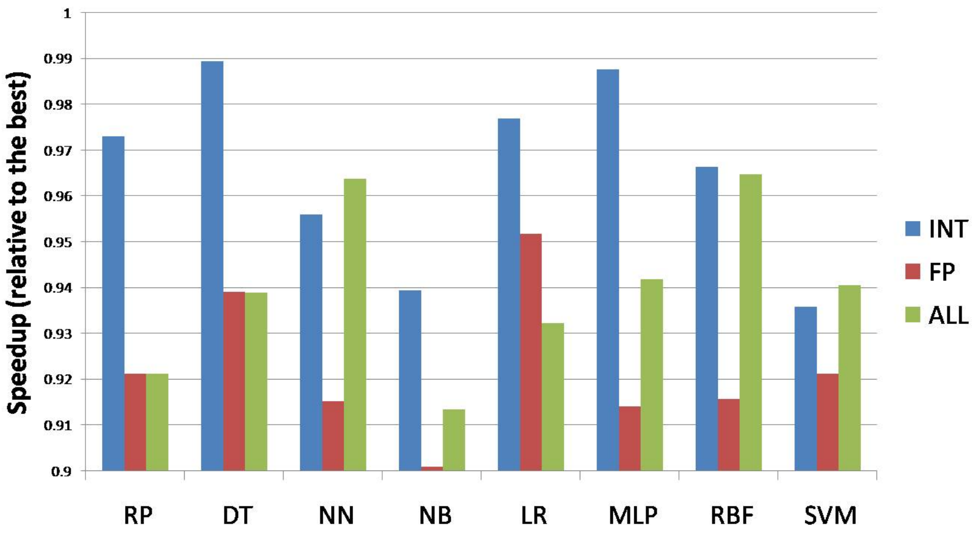


Figure 2: Effect of using different training datasets on machine learning algorithms. Results show the `DCA` performance with predicted configurations, compared against the per-application best configuration. The `M` problem formulation is used.

The quality of the models built with the `ALL` dataset falls between those built with the `INT` and the `FP` datasets. However, for the `NN` model, `ALL` is slightly better than `INT`. This is because there are fewer data instances in `FP` and they are not close enough to the `INT` data instances to confuse the `NN` algorithm. When evaluating the data instances from the `DCA` dataset, the algorithm can still pick up the nearest neighbor from an `INT` data point.

4.2 Evaluation of Problem Formulation

Figure 3 compares the speedup of our `DCA` testing suite using different problem formulations with the `INT` training dataset. Among the three approaches, the `M` formulation is generally the best. This is not surprising because this formulation essentially provides more information to the machine learning algorithms by including all the *good* prefetcher configurations in the training dataset. On the other hand, the `S` formulation only considers the *best* prefetcher configuration, which is a significant information loss, especially because most good configurations have very close performance with the best performance for some programs. However, we do find that if we set the threshold value for defining a *good* configuration too low and thus include too many "good" configurations in the training dataset, the models built are worse. The predictors are confused by too much misleading and redundant information. For example, the performance results of `445.gobmk` for all the 16 prefetcher configurations are within 0.3%, and it is unreasonable to pass all these configurations to the model, so we remove `445.gobmk` from the `INT` dataset for the `M` formulation.

Surprisingly, the `I` formulation is not as good as expected. This approach transforms, not algorithmically, but with domain knowledge, a multi-class classification problem into four independent binary classification problems. This makes the problem conceptually easier to evaluate and thus makes the predictors run faster. The drawback is that this approach ignores the relationships between the four hardware prefetchers, even though some of them are related and their settings should be considered together. For example, the L1 DCU prefetcher and the L2 ACL prefetcher both fetch the next line to the cache. The difference is that the DCU prefetcher keeps a timer and checks if there are multiple loads from the same line within a time period. Thus, if enabling the L1 DCU prefetcher fails to boost the perfor-

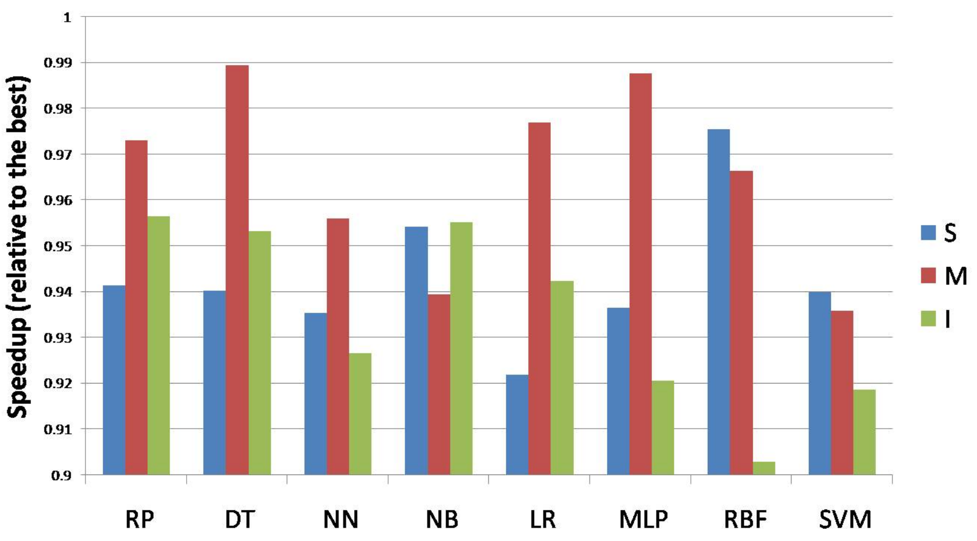


Figure 3: Effect of using different problem formulations. Results show the DCA performance with predicted configurations, compared against the per-application best configuration. The INT training dataset is used.

mance, it is likely that enabling the L2 ACL prefetcher will not help either. With the I formulation, the correlation between these decisions are neglected because the settings of the four prefetchers are determined separately.

4.3 Individual Prediction Results for DCA Applications

In the following experiment, we examine the sensitivity of DCA applications to prefetcher configurations and compare the performance of three selected machine learning models (DT, LR, and MLP) against three commonly used *default* configurations. Table 4 shows the individual speedup for each application and the weighted (geomean) speedup for the whole DCA suite.

For many DCA applications, a good configuration is to have all prefetchers enabled. Four notable exceptions are PageScorer, DocCluster, KeywordCluster, and Clustering. In the case of PageScorer which typically consumes a plurality of cycles, one of the good configurations was Config-1111 which disables all hardware prefetchers. This application already performs many low-level memory optimizations, so we believe that the addition of prefetchers actually hinders performance.

Based on the performance gap of the best and the worst prefetcher configurations, shown in Table 3, the applications in the DCA benchmark can be roughly classified into three categories as shown in Table 4. In the first category, there is about a 10%, or even less, difference between the worst and best configuration. These applications are relatively insensitive to hardware prefetching. Take ContentAds for example. Even the optimal prefetcher configuration only achieves 1.4% performance speedup over the worst prefetcher configuration (see Table 3). Although the room for performance improvement is limited for applications in this group, most predictors still obtain near-optimal performance. The only exception is NB for Clustering. In fact, the performance improvement of using the NB predictor is less stable across programs than other predictors, especially for programs with high performance variations.

The second category has a performance gap between 20%–40%. Although most predictors can obtain near-optimal speedup, some predictors do suffer from a substantial performance degradation. For instance, for DocCluster, RBF loses more than 20% to the best configuration; for PageScorer, NB and SVM both lose more than 10%; for Clustering, RP and RBF both lose around 20%. However, among these programs, PageScorer is one of the most important applications and many predictors get the near-optimal performance improvement. These predictors are two classical algorithms, RP and DT, and two advanced neural networks, MLP and RBF.

The last category includes applications that show a performance gap greater than 40%. The impact of the different prediction models varies dramatically. For OfflineContent, three predictors can only achieve less than 90% of the optimal performance, although the other five predictors can do much better. For StreetImaging, five predictors lose to the optimal performance by 20%, and only three predictors are actually helpful. Overall, for the programs with high performance variations, DT and LR are reasonably stable and good enough to achieve the near-optimal speedup.

It is worth noting that none of the three *default* configurations can deliver the best overall performance. Config-0011 achieves the best performance, although it is still outperformed by the DT, LR, and MLP predictors. The widely used configuration, Config-0000, is the worst among these three, which confirms that the DCA applications and the data center environment are different from traditional optimization contexts.

Overall, most of our data center applications can benefit from a better setting of hardware prefetchers: In addition to OfflineContent, HugeSort, and StreetImaging, all the other eight programs can usually obtain substantial performance improvement with the use of our predicted configuration. Among those prediction models, the DT (decision tree) predictor is usually good enough to have near-optimal predictions, and it is attractive because it produces simple

Benchmark	DT	LR	MLP	Config-0000	Config-0011	Config-0110
ContentAds	0.991	0.993	0.996	0.999	0.996	0.993
LogProcessing	0.985	0.995	0.998	0.999	0.989	0.996
Clustering	1.000	0.980	1.00	0.989	0.978	0.993
PhraseQueryServer	0.987	0.981	0.970	1.000	0.980	0.998
DocCluster	0.937	0.988	0.962	0.972	0.974	0.962
ContentIndexer	0.994	0.956	0.994	1.000	0.867	0.964
PageScorer	1.000	0.961	1.000	0.803	0.989	0.919
OfflineContent	1.000	1.00	0.999	0.999	0.858	0.986
Sorting	0.900	0.812	0.864	1.000	0.923	0.792
KeywordCluster	1.000	0.994	0.995	0.967	0.993	0.659
StreetImaging	1.000	0.991	0.827	1.000	0.784	0.995
Weighted Geomean	0.989	0.978	0.988	0.921	0.976	0.944

Table 4: Performance of applications in DCA with selected prediction models or prefetcher configurations compared to the best prefetcher configuration for each application. The INT training dataset and the M problem formulation are used.

models and tree structures which are easy to interpret.

4.4 A Decision Tree Example

```
DTLB_LD_MISS <= 1.13841 (C1)
|   DTLB_MISS <= 3.161832 (C2)
|   |   BR_PRED_MISS <= 18.97335 (C3)
|   |   |   DTLB_MISS <= 1.062258: 0 (D1)
|   |   |   DTLB_MISS > 1.062258: 15 (D2)
|   |   BR_PRED_MISS > 18.97335 (C4)
|   |   |   L1D_LD_MISS <= 0.612822: 2 (D3)
|   |   |   L1D_LD_MISS > 0.612822 (C5)
|   |   |   |   BR_PRED_MISS <= 22.411147: 6 (D4)
|   |   |   |   BR_PRED_MISS > 22.411147: 4 (D5)
|   DTLB_MISS > 3.161832 (C6)
|   |   BLOCKED_LOAD <= 51.386555: 8 (D6)
|   |   BLOCKED_LOAD < 51.386555: 5 (D7)
DTLB_LD_MISS > 1.13841 (C7)
|   BLOCKED_LOAD <= 50.275612 (C8)
|   |   L1D_LD_MISS <= 26.877868 (C9)
|   |   |   BLOCKED_LOAD <= 46.849349: 2 (D8)
|   |   |   BLOCKED_LOAD > 46.849349: 4 (D9)
|   |   L1D_LD_MISS > 26.877868: 14 (D10)
|   BLOCKED_LOAD > 50.275612 (C10)
|   |   DTLB_MISS <= 15.18429: 1 (D11)
|   |   DTLB_MISS > 15.18429: 10 (D12)
```

Figure 4: A decision tree model for the prefetching optimization problem.

Figure 4 shows a decision tree used in our model. The internal nodes represent conditions and the leaf nodes represent decisions. A condition is composed of a threshold value and a two-way branch. In this example, the threshold values are the numbers of occurrences of the corresponding hardware events in 1000 retired instructions, and the decisions are the 4-bit encodings of the prefetcher configurations. For example, a leaf node labeled with 0 is a decision that all the four hardware prefetchers should be enabled.

This decision tree represents a complex decision-making system that even a domain expert can hardly compete against. It contains 10 conditions and 12 decisions, with a decision depth of five. Although it is possible for an expert in the prefetching optimization problem to select the useful features (i.e., hardware events) to decide the use of hardware prefetchers, it would be difficult to come up with a rule hierarchy that is five levels deep.

Furthermore, the decision tree model greatly confirms the domain knowledge that is essential to the prefetching optimization problem. The features that frequently appear in the decision tree are: DTLB_MISS, L1D_LD_MISS, BLOCKED_LOAD, BR_PRED_MISS.

The event DTLB_MISS measures the DTLB misses due to all memory operations. A high miss rate for DTLB suggests that the program has a large and relatively unpredictable working set and thus the prefetching mechanism is more likely to pollute the cache system by fetching unnecessary data into it. A high DTLB miss rate is a signal for disabling hardware prefetchers. Comparing the sub-trees following conditions C2 and C6, we see that the decisions in the C2 sub-tree are more inclined to turn on DPL, the most effective L2 prefetcher, because the DTLB miss rates are lower.

The event L1D_LD_MISS measures load instructions that cause L1 data cache misses. In our training experiments, the programs are run with all the prefetchers enabled. Therefore, a high L1 miss rate in our training dataset implies that some of the prefetchers should be disabled because the high L1 miss rate is likely due to overly aggressive prefetching. In Figure 4, D3 suggests turning on both L1 prefetchers because L1 miss rate is lower, while D4 and D5 suggest turning on only one L1 prefetcher, the IP prefetcher.

The event BLOCKED_LOAD measures the blocked loads. A load instruction may be blocked if a preceding store instruction has an address or data that is not yet known to the hardware or the number of L1 data misses exceeds the maximum number of outstanding misses that the processor can handle. With more blocked load instructions, the benefits of prefetching tend to be less since prefetching will not be able to hide the miss latency if the blocking cannot be resolved. In Figure 4, with fewer blocked loads, D6 decides to adopt more aggressive prefetching than D7 does.

One interesting contradiction in this decision tree is the use of BR_PRED_MISS (C5). Generally speaking, if the program incurs more branch mis-predictions and thus more wrongly speculated executions, aggressive optimizations should be disabled to preserve performance. However, D4 suggests turning on two prefetchers when mis-predictions are lower while D5 suggests turning on three prefetchers when mis-predictions are higher. This may be a problem due to the limited size of the training dataset or an inherent problem with the model itself. If the size of the training dataset is to blame, expanding the training dataset is a reasonable solution to the problem. If the problem is with the model itself, it is still easier for domain experts to plug-in their knowledge into the model by locally changing the branching structure of the decision tree. Also, one may find the condition C3 to be too strong because the two resulting decisions produce two opposite prefetcher configurations (i.e., all prefetchers on or all prefetchers off). In the same manner, one can put in new conditions manually to smooth the decisions.

Overall, this example decision tree demonstrates the benefits of using machine learning techniques to solve the parameter optimization problem. First, it shows that equipped with a good algorithm, representative datasets, and appropriate attribute sets, it is possible to build an accurate prediction model. Additionally, the model built from the algorithm not only confirms the knowledge from domain experts but also exposes more sophisticated heuristics and threshold values. Finally, it is straightforward for domain experts (from the computer architecture community, not the machine learning one) to fine-tune the model if they discover inconsistencies.

5. RELATED WORK

Data prefetching has long been researched and remains an active research topic since it can highly affect performance of the memory system. Using machine learning to direct program and system optimization has become a popular trend for architecture research communities as it has been rendered effective for solving non-linear problems such as performance prediction and tuning. To the best of our knowledge, our work is the first to use machine leanrning techniques to build a data-center performance optimization framework. We target efficient execution of large-scale severside applications in a data center. This particular paper focuses on finding the best configurations for the processor prefetchers in data center machines.

5.1 Data Prefetching

Data prefetching is essential to the memory system performance. Many researchers have identified and exploited different types of correlation. For example, Wenisch et al. identified the temporal correlation property [25] and Somogyi et al. observed the spatial correlation property [21]. In [20], the authors extended their work and exploited both temporal and spatial correlations to cover the drawbacks of taking advantage of only one type of correlation. Moreover, many mechanisms for detecting and exploiting the correlation have been proposed [9, 3, 13, 6]. Instead of devising a more complicated prefetcher, we rely solely on the relatively simple prefetchers found on modern processors and attempt to obtain the maximal benefit from the existing hardware. Thus, our apporach is more likely to be deployed in a data center today that is filled with commodity CPUs. In addition, we believe that the proposed performance optimization framework can also be tailored to solve other system performance problems.

5.2 Application of Machine Learning

In the computer architecture community, researchers today need to tackle a rapidly growing microarchitectural design space. Detailed simulation can produce the most accurate performance estimation, but the long simulation time usually makes this strategy impractical and prohibitive. One solution to this problem, *temporal sampling* [5, 26], identifies a representative portion of the program execution for detailed simulation and infers the overall performance based on the simulated result of this portion. SimPoint [5], for example, records the frequency of each type of instruction that appears in a basic block during detailed simulation and then uses a k-means clustering algorithm [7] to find the different phases of the program execution. It can recognize and weight each phase to estimate the overall execution. Another approach, *spatial sampling* [11, 15], performs detailed simulation only for some selected points in the design space. For instance, İpek et al. [11] choose only a small number of points for simulation and then use the result (that is, the design parameters and the simulated performance) to train neural networks for predicting the performances of other points in the design space.

Combining both temporal and spatial samplings, Lee et al. [14] target an even harder problem, the design parameters for adaptive microarchitectures. They use regression modeling and genetic algorithms to avoid comprehensive detailed simulations and speed up the performance evaluation process.

Our work specifically addresses the problem of performance tuning from a data center perspective. The problem space differs from traditional compiler optimization in that we consider system-level optimization across the data center. With respect to the related work in computer architecture, we examine a particular class of applications, data center applications. Additionally, we consider the problem of selecting appropriate hardware prefetcher configurations. The right configuration can have a significant impact on performance. In this work, we describe successful strategies for choosing prefetcher configurations.

6. CONCLUSION

In this paper, we first highlight some of the difficulties in optimizing performance within the data center environment. In response we design and implement a framework for optimizing parameter values based on machine learning. Equipped with the framework, we conduct multiple experiments in order to understand the sensitivity of the framework with regard to several influential components: machine learning algorithms, training datasets selection, and modeling or problem formulation methods. For the specific problem of choosing hardware prefetcher configurations, we show that our framework can achieve performance within 1% of the best configuration. We have shown that judicious exploitation of the machine learning-based framework is a promising approach to tackle the parameter value optimization problem.

7. REFERENCES

[1] David W. Aha and Dennis Kibler. Instance-based learning algorithms. *Machine Learning*, 6(1):37–66, 1991.

[2] M. D. Buhmann. *Radial Basis Functions : Theory and Implementations*. Cambridge University Press, Cambridge, UK, 2003.

[3] Y. Chou. Low-cost epoch-based correlation prefetching for commercial applications. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 301–313, Washington, DC, USA, 2007. IEEE Computer Society.

[4] W. W. Cohen. Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.

[5] G. Hamerly, E. Perelman, J. Lau, B. Calder, and T. Sherwood. Using machine learning to guide architecture simulation. *J. Mach. Learn. Res.*, 7:343–378, 2006.

[6] R. A. Hankins, T. Diep, M. Annavaram, B. Hirano, H. Eri, H. Nueckel, and J. P. Shen. Scaling and charact rizing database workloads: Bridging the gap between research and practice. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 151, Washington, DC, USA, 2003. IEEE Computer Society.

[7] J. A. Hartigan and M. A. Wong. A k-means clustering algorithm. *Applied Statistics*, 28(1):100–108, 1979.

[8] S. Haykin. *Neural Networks: A Comprehensive Foundation, 2nd edition*. Prentice Hall, Upper Saddle River, NJ, 1999.

[9] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 397–408, Washington, DC, USA, 2006. IEEE Computer Society.

[10] Intel 64 and IA-32 Architectures Software Developer's Manuals. www.intel.com/products/processor/manuals.

[11] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 195–206, New York, NY, USA, 2006. ACM.

[12] G. H. John and P. Langley. Estimating continuous distributions in bayesian classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345. Morgan Kaufmann, August 1995.

[13] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 252–263, New York, NY, USA, 1997. ACM.

[14] B. C. Lee and D. Brooks. Efficiency trends and limits from comprehensive microarchitectural adaptivity. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 36–47, New York, NY, USA, 2008. ACM.

[15] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 185–194, New York, NY, USA, 2006. ACM.

[16] perfmon2: the hardware-based performance monitoring interface for Linux. perfmon2.sourceforge.net.

[17] J. C. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in kernel methods: support vector learning*, pages 185–208, Cambridge, MA, USA, 1999. MIT Press.

[18] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1993.

[19] S. S. le Cessie and J. van Houwelingen. Ridge estimators in logistic regression. *Applied Statistics*, 41(1):191–201, 1992.

[20] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-temporal memory streaming. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 69–80, New York, NY, USA, 2009. ACM.

[21] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *ISCA '06: Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 252–263, Washington, DC, USA, June 2006. IEEE Computer Society.

[22] SPEC CPU2006. Standard Performance Evaluation Corporation. www.spec.org/cpu2006.

[23] SPECweb2009. Standard Performance Evaluation Corporation. www.spec.org/web2009.

[24] TPC-C. www.tpc.org/tpcc.

[25] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 222–233, Washington, DC, USA, 2005. IEEE Computer Society.

[26] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 84–97. ACM Press, 2003.