AN EXPLORATION INTO THE EFFECTIVENESS OF PREFETCHING ON PROGRAM PERFORMANCE WITH THE HELP OF AN AUTOTUNING MODEL

by

Saami Rahman, B.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
May 2015

Committee Members:

Apan Qasem, Chair

Martin Burtscher

Ziliang Zong

COPYRIGHT

by

Saami Rahman

2015

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

I have been an exceptionally fortunate individual who has received support from the most surprising sources on numerous occasions throughout his life. The Texas State journey would not have been possible without the financial support of several individuals. They are: my aunts Nafisa Noor and Nahidul Akter, my uncle Mohd Reza and my cousin Masud Rahman. They stepped up when my need was great. This thesis is my greatest achievement to date, and I dedicate this thesis to them and every single individual who has helped me reach this stage.

ACKNOWLEDGEMENTS

This thesis would not have been possible without my advisor Apan Qasem. I am always amazed at his patience, compassion and understanding. He has been instrumental in giving shape to the work done in this thesis. I would like to especially thank him for the freedom that he has allowed me in order to pursue my interests in research, and for never discounting any of my ideas.

I would also like to thank Ziliang Zong for his openness of mind and allowing me to use the servers in his lab, and Martin Burtscher for instilling in me the desire to eek out every last possible drip of performance.

Several individuals from the computer science department must be mentioned, as they have been great sources of friendship and comfort during my days here. They are my lab mates from CRL, and Farbod Hesaaraki and Molly O'Neil from ECL.

My undergraduate advisor Ashraful Amin has played a key role in my move to Texas. I am thankful for his faith in me and for introducing me to the fascinating field of machine learning.

Lastly, I want to thank my parents and my sister. They have always fought hard against great odds to give me the gift of education. I am forever indebted to them.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

ABSTRACT

This thesis presents the effects of hardware prefetching on the performance of a collection of programs and how learning algorithms can be used to predict the optimal hardware prefetching algorithms to obtain improved performance. Modern processors are equipped with several hardware prefetchers, each of which implements a different prefetching algorithm. My goal was to select the best combination of these prefetchers, as there is no single combination that results in best performance across various programs. Effective program characterization is necessary when learning models are used to make predictions based on program behavior. This thesis uses hardware performance events in conjunction with a pruning algorithm to create a concise and expressive feature set. The feature set is used in three different learning models. These steps are tied together in the form of an autotuning framework that can, on average, achieve up to 96% of the possible speedup that can be attained by varying the combination of prefetchers in effect. The framework is built using open source tools and frameworks, thereby making the framework easy to use, extend and port to other architectures.

1. INTRODUCTION

Prefetching is a commonly used technique to improve performance. A prefetcher monitors memory accesses and predicts ahead of time the memory address that a running program will need in the near future. Based on this prediction the prefetch unit fetches the data from main memory to fast higher levels of memory such as caches. When the program needs this piece of data, it can find it in a fast high-level memory, and avoid the latency that is incurred on retrieving data from slow lower levels of memory such as main memory. This has been shown to dramatically improve performance [1, 2].

In modern architectures prefetching is more important than before. McCalpin [3] has shown that the difference in CPU speed and DRAM speed differ by almost a factor of 1000. This imposes a huge bottleneck in performance known as the memory wall. The implication of the memory wall is that no matter how much CPU performance is improved, overall performance will be held back by memory. Thus, researchers have continually focused on techniques to hide memory latency, and prefetching is one such technique.

Prefetching can be done in both hardware and software. Both of these techniques have their own advantages and disadvantages. Hardware prefetchers can predict simple memory access patterns well, and adapt to memory accesses taking place in the system at runtime. This is especially useful in case of parallel workloads, as individual threads of the same program as well as threads from other applications running on the same core may exhibit varying memory access patterns. On the other

hand, in case of software prefetching, the compiler analyzes program control and data flow, and inserts machine specific prefetch instructions. This can result in more accurate and effective prefetches, however, it is susceptible to failure due to other programs running in the system [4].

There are several challenges associated with prefetching. Firstly, 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. Secondly, 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 accomodate more urgently required data, since higher levels of memory are smaller. These challenges are amplified further in multi-threaded 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. Also, because multiple threads share the same space-constrained cache, the importance of accurate prefetch becomes more crucial to ensure that other threads are not affected poorly.

In this thesis, I have attempted to understand the effect of prefetching on program performance. I have primarily used the PARSEC benchmark suite [5] and two example programs that exhibit sequential memory access patterns and randomized memory access patterns. I have observed the change in performance by enabling and disabling the available hardware prefetchers on an Intel processor, and that using the prefetchers can have negative effects on performance. On the particular

machine I have conducted my experiments on, 4 hardware prefetchers are available, and 16 different combinations of configurations can be obtained by enabling or disabling them.

Next, I have used machine learning algorithms to obtain recommendations on which configuration should be used for an unseen program. The primary reason behind using learning models is that, programs can vary greatly and exhibit different traits and behaviors. It is difficult to deduce rules of program behavior by hand for a large number of programs that differ from each other. Learning models are well known for their ability to draw decision boundaries over multiple dimensions of data axes, and the problem I am tackling is a good fit for these models.

The first step to successfully use learning models is to be able to characterize programs in a quantitative method that captures the differences between programs. To do so, I have used hardware performance counters. Because there are many hardware performance counters, I have designed a simple, yet effective procedure to prune the number of counters required to characterize a program. I have shown the efficacy of the pruning algorithm by testing different feature sets using a decision tree.

Finally, I have developed an autotuning framework that recommends hardware prefetching configuration for unseen programs on the basis of seen programs. On average, this framework can help the user gain up to 96% of the achievable speedup that is available from the hardware prefetchers.

There are several advantages of using a software based recommendation system to change the hardware configuration. First, it allows to use the existing hardware

effectively by tapping into potential performance improvements. Second, this approach does not require any source level changes for the program being optimized. Lastly, the framework relies on open source technologies, therefore it is easy to extend or port to different architectures.

The thesis is organized from here onwards in the following way: first, chapter 2 presents the reader with material that are helpful in understanding the associated challenges and the problem statement of this thesis as a whole. Chapter 3 is a concise summary of related work done in this field by other researchers. Chapter 4 describes the autotuning framework that has been developed. The framework is divided into separate components, and each of these components give the reader a top-level view of the subproblems in this thesis. Chapter 5 is a description of the results, experimental environment and the outcome from the learning models. This section engages in a thorough examination of the solution to each of the subproblems described in the preceding chapter. The final chapter concludes this thesis with a summary of its contribution and an indication of the possible work that can be conducted from here on.

2. BACKGROUND

2.1 Prefetching

The key idea behind prefetching is to preload data before it is needed by a program. The prefetcher predicts the memory address of the data and loads the data stored at that particular address into high-level memory. If prefetching is done accurately and in a timely manner, it can dramatically improve performance.

2.1.1 What to Prefetch

The first challenge for the prefetcher is to figure out what to prefetch. Incorrect prefetches do not affect the correctness of the program because incorrect prefetch requests are simply unused and dropped by the hardware. However, correct prediction of prefetching is important because prefetching incorrect data can waste resources. Memory bandwidth and memory at higher levels are precious resources. Incorrect prefetch requests can increase memory traffic and also pollute the cache with useless data. A simple measure of prefetch accuracy is to compute the ratio of used prefetches over issued prefetches.

2.1.2 When to Prefetch

Timing of a prefetch request is also very important. A prefetch that arrives after the program needs the data is late and therefore useless. On the other hand, prefetches that are early can be evicted from the cache before they are required, in order to make room for outstanding demand requests.

The timeliness of prefetching can be tuned by adjusting prefetch aggression. Prefetch aggression is the measure of how far ahead the prefetcher issues instructions to load data into cache. This depends on the memory latency that the prefetcher is attempting to hide, and is therefore architecture dependent.

2.1.3 Where to Prefetch

In a modern computer system there are several levels in the memory hierarchy, and the prefetcher can prefetch to several levels. The most common levels to prefetch to are L1 and L2 caches. In the case of hardware prefetchers, to which level the prefetches are done is determined by how much of the memory access pattern is visible to the prefetch unit. For instance, if the prefetcher is prefetching to L2, it sits between the L2 cache and the memory controller. Therefore, it is blind to requests and hits that are made in L1. On the other hand, if the prefetcher is prefetching to L1, it can see all of the memory access pattern, but the L1 cache is small and this opens up a possibility of perfomance penalty due to cache pollution.

It must also be decided where the prefetched data should be placed. The data can be placed in the cache, which is usually the case in most modern systems, or the data can be placed in a separate prefetch buffer, as was done in the ALPHA 21064 processor. The latter results in a more complex memory system, and presents further challenges of keeping the prefetch buffer coherent and also sizing it, but it prevents cache pollution [6].

Another architecture design choice to consider is whether to discriminate between prefetched data and demand data in the cache in terms of replacement

policies. One policy is to consider them alike, while another option is to favor demand fetched blocks. In such a case, if the prefetcher is effective, there is high risk of losing out on performance improvements, since the prefetched blocks would be more likely to be replaced before being used. The replacement policy could also be adaptive in nature, such that if the prefetcher performs well, the policy is adjusted to favor the prefetcher [7]. The Intel Ivy Bridge microarchitecture uses a similar strategy.

2.1.4 How to Prefetch

Prefetching can be done in several ways. The programmer can enter prefetch intrinsics in the source level, in the form of direct prefetch instruction or hints, which the compiler then uses to generate prefetch instructions. The compiler can also enter prefetch instructions based on its internal program analysis. The hardware can issue prefetch instructions during runtime.

A less often used technique is to execute a thread to prefetch data for the main program [8]. This thread can be created by the programmer or the hardware. Doing this by hardware usually requires complex hardware.

2.2 Performance Events in Hardware

Modern processors give us the ability to gain very fine-level quantified insight into program performance through performance event measurements. These events such as stalled CPU cycles, branch mis-predictions, page faults, prefetch requests and many more are measured using special registers available in the CPU called performance counters. These counter values allow us to capture program behavior and

characterize it in terms of program features. For example, given a program, we could profile it during its execution to measure how many times it misses the L2 cache. This number would give us a sense of the program's cache usage. A high miss rate would indicate the program likely has a large working set or perhaps internally uses pointer-based data structures. Thus, features such as these can then be used in various learning models to classify programs into different categories. Examples of such target categories would be "CPU bound programs", "memory bound programs", "large/small working set", etc.

However, characterizing programs based on performance counter values is non-trivial and much research has been invested in solving this problem [9, 10]. This is primarily because there is very little standardization of these events among processor vendors. In fact, there is significant variety in the number of events available and what quantity they measure across processor models of the same vendors [11]. For instance, the Core2 architecture by Intel can measure 387 performance events, and the SandyBridge architecture, a more recent version, also by Intel, can measure 294 performance events. While the nature of the available events in both these architectures are similar, there is little intersection between these two sets. The events are also ill-documented, often leaving the reader to deduce the metric from the name of the event. Understanding these metrics are difficult without input from a seasoned hardware architect.

In addition, measuring performance events often presents a challenge in itself [12]. Vendors such as Intel provide proprietory performance diagnostic tools to

measure these events such as *Intel VTune*. The Linux operating system ships a performance monitoring tool called *perfmon2* as part of its kernel tools package. Open source tools such as *Likwid* also provide a viable alternative. However, measurement of events on real machines is challenging because often times the machine is running other auxiliary programs, daemons, checking over the network for updates, scheduling processes and so forth, and these activities, known as operating system jitter, add noise to the measurement.

2.3 Autotuning

Autotuning is an optimization technique that uses heuristic based search to find the optimal parameters of a build system [13]. High performance computing systems are often under-utilized because their underlying architecture remains largely unexploited. Most domain experts, such as biologists and physicists, are not experts on computer architecture and therefore do not write code that makes use of architectural benefits such as cache locality. Compilers can be passed several compile-time parameter, or hints in the form of macros, for example, loop unroll parameters. This can also result in optimized code. However, many of these concepts remain largely unknown to domain experts. Furthermore, the rapid change in architecture also exacerbates this problem, as newer architectures provide more features. An autotuning framework attempts to explore the search space of these various build system parameters and find the most optimal one. The ideal autotuning framework works across multiple architectures, compilers and types of programs.

Researchers have been investigating various aspects of code that can be exploited using autotuning models in order to gain performance benefits. Qasem et al. [14] have investigated tunable parameters that affect both parallelism and data locality in multi-threaded numerical applications. Hall et al. [15] have propose the use of several 'recipes' that provide a high-level interface to the code transformation and capability of a compiler. Tiwari et al [16] have presented a scalable autotuning framework for compiler optimization that is able to achieve speedup of 1.4 to 3.6 on various ATLAS kernels over the Intel C Compiler. A common theme in most work done in autotuning has been on improving performance of niche programs, such as scientific programs, kernels or programs with a compute intensive loop nest. The work presented in this thesis is different from the cited works in that this work attempts to reach a broader audience by investigating the possible improvements in the PARSEC benchmark suite [5] - a benchmark widely considered to consist of programs across a wide spectrum of applications.

2.4 Machine Learning

As there exists a large number of measurable performance events, many of which are arcane and difficult to understand, it then becomes challenging to understand which set of events should be looked at in order to characterize programs. Often times program characterization is the first step of a larger problem, where the characteristics of a program, such as values of a set of performance events, are fed into a learning model that has a separate defined objective. Learning models are helpful because they

can draw complex decision boundaries across multiple dimensions. This is particularly important because even if a subset of expressive events has been derived, there are usually 10-15 events to look at, and it is difficult to analyze each of these events by hand in order to derive a set of rules that can discern programs based on program behavior.

In this section, the two learning models that have been used, and the reason for choosing them among many other available models are discussed briefly.

2.4.1 Logistic Regression

Logistic regression is a learning model that can classify data points into different classes. There are two types of logistic regression models - binomial and multinomial. Binomial models can classify a data instance into 1 of 2 classes, whereas multinomial models can classify into multiple classes. In this work, the logistic regression models are binomial. The choice of this model was due to its simplicity and success in classification in prior reported work [9].

2.4.2 Decision Trees

Decision trees can be of two types - classifiers and regressors. In this work, classification trees have been used. Decision trees examine values of provided features, and greedily choose the feature that provides the maximal information gain or split in classification. It recursively proceeds to classify until a leaf node is reached. This model was chose because it complements the logistic regression model with a model that internally prunes features. Also, the final tree and its branches can give

helpful insights into how a decision has been reached, rather than in the case of logistic regression, where the classification process cannot be easily understood.

2.5 The PARSEC Benchmark Suite

The PARSEC benchmark suite comprises programs across wide spectrum of applications [5]. It contains programs with varying working sets, locality, data sharing, synchronization and off-chip traffic. It is publicly available and has been developed at Princeton University. PARSEC includes emerging applications in computer vision, data mining, finance as well as simulations. Other benchmarks such as ALPBench [17], BioBench [18], MediaBench [19], and MineBench [20] focus on niche areas and thus limits the applicability of the optimization framework. We have previously worked using the SPEC CPU 2006 [21] benchmark, however, the benchmark only contains single threaded programs. Using a diverse, multi-threaded and relatively newer benchmark such as PARSEC increases the impact of this work significantly.

In addition the programs included in PARSEC, I have written two simple benchmark programs, referred in this thesis as `stream` and `random`. In the `stream` program, several arrays are initialized and repeatedly accessed in streams, thus being an ideal candidate to benefit from prefetching. The `random` program, on the other hand, accesses several arrays in random locations, thus very likely to throw off the prefetchers as they fail to predict which memory locations will be accessed in the future. The two programs were written to complement the real world program

collection of PARSEC, where programs call helper libraries, process user input, read and write from files besides its central objective.

3. RELATED WORK

This section is a summary of existing related work on prefetching, program characterization and use of machine learning in program performance optimization. We briefly describe these works and compare how our work is similar and/or different.

3.1 Prefetching

Prefetching is a widely explored area, and the benefits of prefetching are widely documented and studied. Lee et al. [22] was one of the first researchers to introduce the concept of data prefetching in hardware to hide the memory access time bottleneck. Chen et al. [1, 2], in his early work, studied the effects of data prefetching in both hardware and software. Plenty of work on data prefetching has been done since then. More recently, with the introduction of multicore processors, the effects of prefetching are more interesting and have also been studied in various ways.

Mehta et al. [23] have presented a multi stage co-ordinated prefetch scheme for the Intel SandyBridge and the Intel Xeon Phi architecture. Their compiler framework uses the intermediary representation of source code to estimate the prefetch-distance in loops and inserts prefetch instructions. Prefetching is done in stages, from main memory to L2, and then to L1. They have been able to gain speedup of 1.55x over the hardware prefetchers. However, their solution has several disadvantages. First, their work requires input from an architecture expert, and works for existing architectures only. For new architectures it would be necessary to study the architecture and provide similar fine-grained input. Their solution also involves incorporating their

scheme into an existing compiler, which can be difficult.

Prefetching can hurt performance, and the ill-effects of prefetching have also been studied [24, 25]. Puzak et al. [26] demonstrate the case when prefetching hurts performance and propose the characterization of prefetching based on timeliness, coverage and accuracy. Lee et al. [27] have conducted an in-depth investigation of when and why prefetching works. They have done an extensive analysis of both software and hardware prefetching performance on the SPEC CPU2006 benchmark, however the discussion has focused on serial workloads.

Several studies on prefetching based on available resources and feedback have been performed [28, 29, 30, 31, 32]. While these are effective and novel ways of using prefetching, they present new hardware design considerations. Our approach is to maximize performance based on the hardware and software available currently.

3.2 Program Characterization and Machine Learning

A significant part of this thesis involves successful program characterization and using machine learning models. As such, I have studied several notable works in this regard. Jayasena et al. [33] have demonstrated effective use of a decision tree to prune performance events used as features in order to detect false sharing. This approach has motivated me to include decision trees in this work. Cavazos et al. [9] have developed a machine learning model to find optimal optimization configuration for the SPEC CPU2006 benchmark suite. The goal of this work was to select a set of compiler optimizations that results in performance improvement. They have reported

high classification accuracies in using performance events as features for a learning algorithm. Milepost GCC is [34] a large project to further the attempts at crowdsourced optimization learning. This work, however, uses static program features to characterize programs. A similar work done by Demme et al. [35] uses graph clustering from data and control flow of programs to characterize program behavior. However, these characterizations are difficult, as they involve modifying the compiler, or working on the intermediary representations of the program. Such procedures are difficult to adopt for a wide audience.

Among all the works that I have studied, two are very similar to mine. McCurdy et al. [10] have characterized the impact of prefetching on scientific applications using performance events. They have experimented with a combination of several benchmark programs on AMD processors. Their work is primarily on serial workloads, but they have studied the effects of running multiple serial programs simultaneously. However, their work hinges on successfully isolating the performance events that are expressive enough to capture the effects of prefetching. For the AMD architectures they have worked on, they hand picked the performance events. For new architectures, they would have to repeat the process of studying all available performance events.

Liao et al. [36] have presented a machine learning based approach to selecting the optimal prefetch configuration, in a way very similar to the way we have formulated ours. However, their work, too, hinges on identifying architecture specific performance events, which they have done by hand. In addition, their work involves

serial workloads, whereas I am interested in parallel programs.

4. AUTOTUNING FRAMEWORK

This section discusses the autotuning framework, and explains the various components of the framework and how they are tied together. This section also describes the tools that have been used to build this framework.

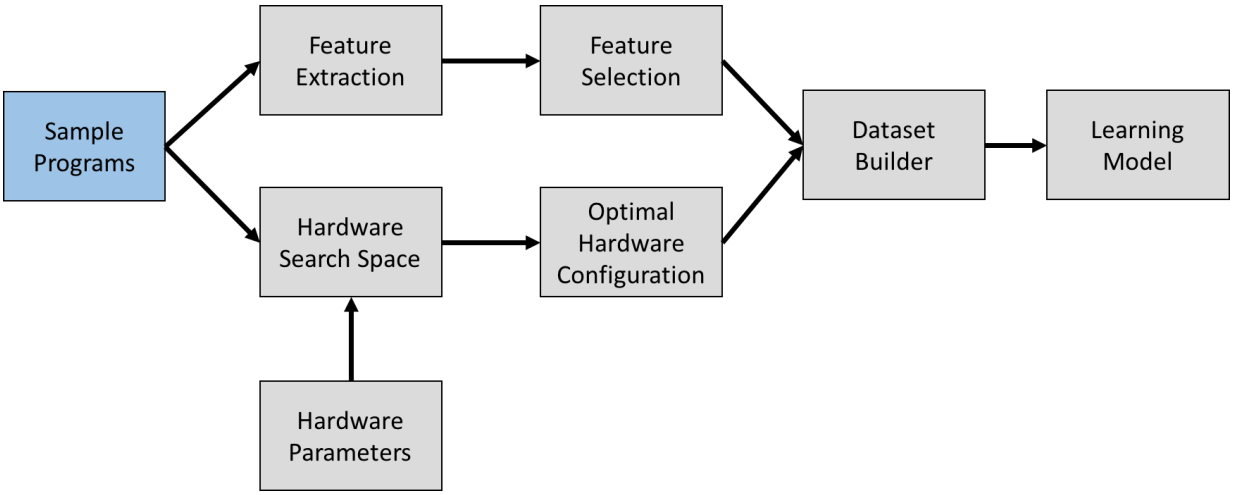
Figure 4.1 shows the major tasks that the autotuning framework performs. There are two phases of the autotuning framework. The first phase is called the training phase. In this phase, sample programs are used to train a learning model. To begin with, programs are run in the hardware search space in order to find the optimal hardware configuration. For this problem statement, the programs are run across all possible prefetching configurations and the ground truth for all the configurations are recorded. The programs are also run in order to generate features, and the feature sets and the found optimal hardware configurations are used to build a dataset. This dataset is used to train a learning model.

The testing phase is relatively simpler. First an unseen program is run in order to collect features. The features are then fed into the trained learning model to obtain prediction or recommendations on the optimal hardware configuration.

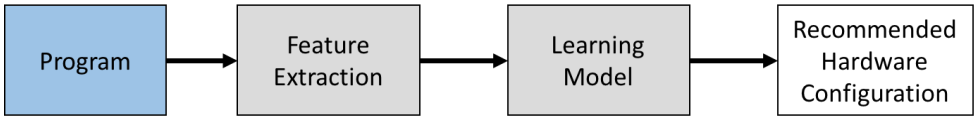
4.1 Prefetch Configurations

Modern processors implement prefetching in hardware. In these experiments an Intel Core2 processor was used, and this architecture is equipped with 4 prefetchers, described below:

1. **Data cache unit (DCU) prefetcher:** this prefetchers attempts to recognize a



(a) Training phase of the autotuning framework



(b) Testing phase of the autotuning framework

Figure 4.1: Autotuning framework

streaming algorithm and prefetches the next line into the L1 cache.

2. **Instruction pointer (IP) based stride prefetcher:** this prefetcher tracks individual load instruction and attempts to detect strided accesses. It can detect strides of up to 2K bytes and prefetches to the L1 cache.

3. **Spatial prefetcher (CL):** this prefetcher attempts to complete every cache line brought to L2 by fetching the pair cache line that completes it to form a 128 byte aligned chunk.

4. **Streamer prefetcher (HW):** This prefetcher attempts to detect streaming requests from the L1 cache, and brings in anticipated cache lines into L2 and LLC.

It should be noted that Intel architectures that have been released after Core2, such as SandyBridge and Haswell also ship with the same prefetchers.

It is not defined what the default configuration in processors is, that is, which of these 4 prefetchers are enabled or disabled [36]. In some machines all the 4 prefetchers are turned on, in some only 2, while none of the prefetchers are turned on in some. In this work, we define a *configuration* to be a combination of enabled or disabled state of all these prefetchers. Configurations can be represented using a bit mask, where each bit represents a prefetcher, and the value 1 represents the prefetcher being on, and 0 represents the prefetcher being off. From most significant bit to least, the mapping of bit to prefetcher is: HW, CL, DCU, IP.

The prefetcher configuration in the processor was changed using an open source tool called *Likwid* [37]. Likwid manipulates bits on the model-specific registers of processors to enable or disable a specified prefetcher. On the Intel Core2 architecture, this can be done at runtime. However, in later architectures, such as SandyBridge, the bit required to toggle the state of the prefetcher is reserved. It is still possible to toggle the prefetcher from the BIOS, but this requires a reboot and can be cumbersome. Additionally, it is up to the BIOS writer to expose this setting to the end user.

It may seem that turning on all the prefetchers, that is configuration `1111`, is a viable policy, however, it was observed that significant performance variation can be seen by changing the configuration, and configuration `1111` is not always the best. As a matter of fact, in some cases configuration `1111` is seen to hurt performance, as can be seen in figure 4.2. The speedup values in this figure are runtime improvements

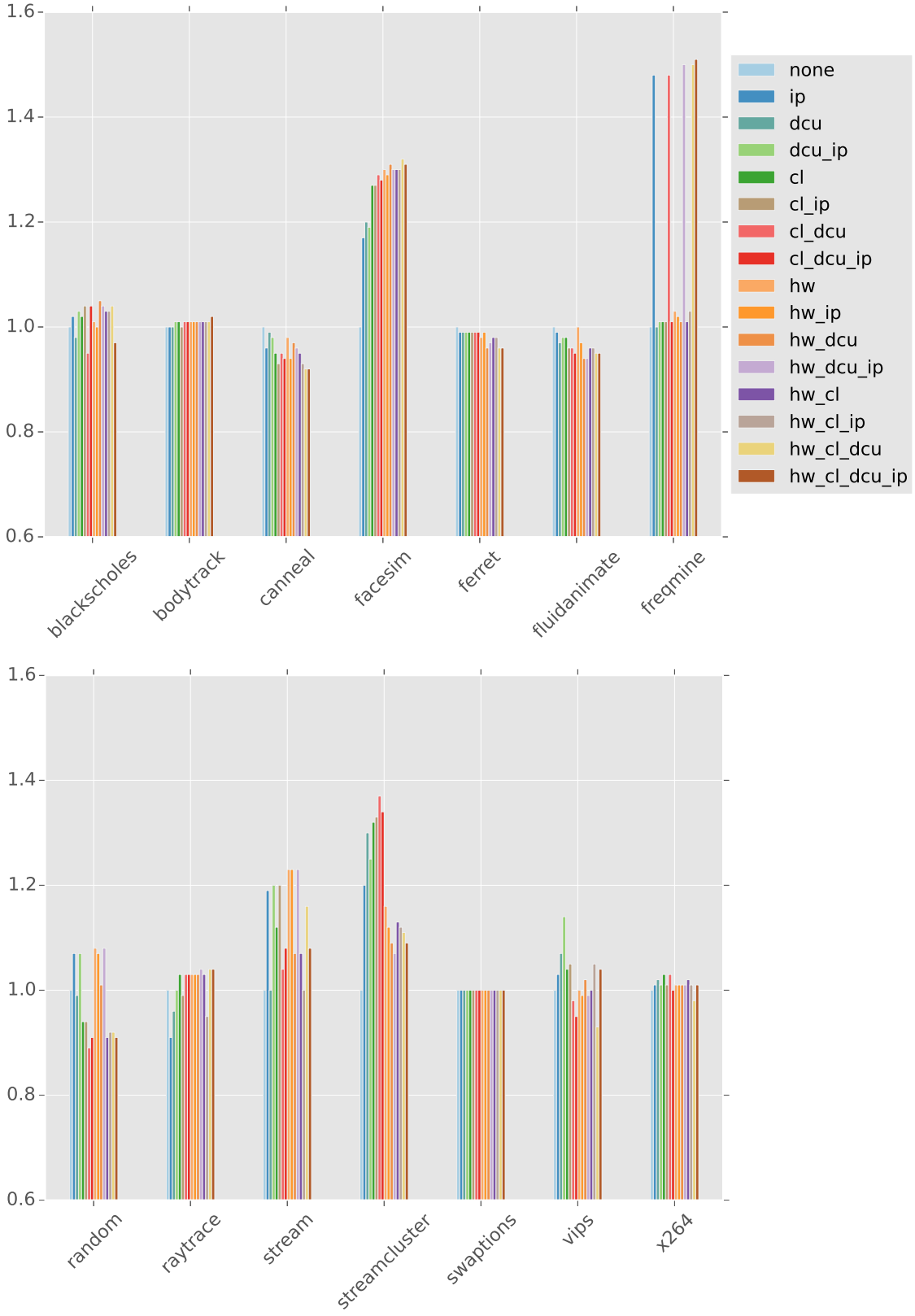


Figure 4.2: Effect of changing configuration on program speedup

reported over the baseline `0000`. For each configuration, 3 runs were carried out, and the best runtime among the 3 was used to calculate the speedup. This helps us shield our experimental results from operating system jitter. The best value was chosen instead of the average of the 3 values because if any of the runs is severely interrupted due to some unknown process, the effect would be propagated through the average calculation. Conversely, the run resulting in the best runtime can not only be seen as the least interrupted, but also as a run that was able to finish the task in the least possible time, therefore, we strive to improve upon this value.

Several observations can be made from this figure. First, within programs, changing the configuration has significant impact on performance. For example, `freqmine` and `streamcluster` show significant variance in performance when the configuration is changed. Second, configuration `1111` is not always the best option, as can be seen in the cases of `random`, `stream`, `streamcluster` 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.

Using these programs, the following sections describe the process of program characterization and the use of learning models to predict optimal prefetching configurations for unseen programs.

4.2 Feature Extraction

This section describes the feature extraction procedure. I collected hardware performance counter values and used these counter values as the feature set describing our programs. I have measured all of the 387 available performance events on the Core2 architecture. I measured all 387 events because I believe these events are able to capture program behavior. Additionally, these events are available on nearly every modern architecture, and can be measured using various tools. The alternate would be to analyze the source code of programs, but no generalized tool that does not require interfacing with the compiler could be found. Also, with such source code analysis I would be limited to static program behavior. The tools that I have used to measure the performance events are easy to install and can be used in many modern architectures.

I wrote a Bash script that uses a Linux tool called *perf* [38]. Perf can measure performance events, and I used it in our framework to find the values of all the events for all the programs. The performance events are measured using hardware registers, and the Core2 architecture has 2 available registers for this purpose. Therefore, it can measure 2 performance events at a time. To measure all the events for any given program, I ran the program 194 times. I could reduce the number of times I have to run a program by sampling the counter values. Perf has a feature that enables the user to specify, for a single run, more events than there are registers available. When this happens, perf internally samples the counter values. For example, I could measure 4 events for a given run, and this would halve the number of runs required to measure all the events to 97. But this would introduce a sampling error, and to avoid this, I

chose to run each program 194 times. Nevertheless, the measurements are not free of errors. If the same performance event is to be measured twice, there will be some difference in the measurements. This is because of several reasons. First, the programs being profiled are multi-threaded and are timing dependent. It is not possible to predict beforehand which thread will finish first or last, and this will cause differences in measurement across multiple runs. Repeating the experiment using different pairs of events on each run can also result in different values. Measurement variations also occur due to operating system jitter, for instance, variances in thread scheduling policies across different runs of the same program. These variances are acceptable and expected on a real system.

Once generated, all the features are normalized over the instruction count of the program. This helps in scaling the programs in relation to each other. For instance, 100 page faults may not be significant for a program with 1 billion instructions, but it will be significant for a program with 10 thousand instructions. For the program with 10 thousand instructions, the normalized value will appear *larger* in relation with the program with 1 billion instructions, which is what I want to capture.

Figures 4.3 and 4.4 demonstrate the value of all the events that have been measured for all 14 programs. The values have been normalized over a million instructions. The figures only provide visualization of all the programs in terms of the performance events. Thus, the axes labels have been removed as it is not realistic to analyze them by hand.

From the figures, it becomes apparent that values for many events are similar for

Figure 4.3: Feature extraction, part 1

almost all programs. These events do not carry any discriminative value. This has driven my pruning algorithm, described in a later section. Also, as I have manually inspected some of the events, I know that several of the events that appear to be 0 in the plots are, in fact, not 0. Because many of the events have a very large range, some of the events with smaller ranges are not visible. I have addressed this issue with a technique called feature scaling.

Figure 4.4: Feature extraction, part 2

4.3 Feature Selection

In this section I discuss the feature selection procedure. As mentioned, I have used performance events to represent our programs, and, there are 387 performance events on the Core2 machine I have run my experiments on. To ensure that the learning models perform well, a concise feature set is required, so that the models are not fed with unnecessary and irrelevant information. Additionally, if all the 387 events are

used, every time the model is tested with a new program, the new program will need to be run 194 times before all its features can be extracted. This is a very high overhead, and another reason why a smaller feature set is needed. I have developed a simple and effective procedure to prune the feature set into a smaller set. The procedure is outlined in algorithm 1 and 2.

Algorithm 1 describes the process of finding events that capture the difference between two programs. For each event that is measured for both the programs, it is checked if the percentage difference, expressed in the interval $(0, 1.00)$ and expressed as ϕ, is beyond a certain threshold. If it is, the event is added to a list called `bucket`. This list contains the set of events that can be used to differentiate between two programs. I have experimentally found ϕ `= 0.95` to work best.

The subroutine `FindEvents` is used in algorithm 2. The `EventUnion` subroutine takes a list of programs, and for all possible unique pairs of programs, it calls the `FindEvents` subroutine. It collects the sets of events that represent each pair of programs and adds it to a 2D list called `buckets`. Therefore, `buckets` contains sets of events from all possible unique pairs of programs.

Next, in lines 11-19, the `EventUnion` subroutine goes through all the sets and for each event in the set, it first checks if the event name exists in a dictionary. If it does not exist, the event name is added and assigned a value of 1. If the event name already exists in the dictionary, the corresponding value of the event name is incremented by 1. Thus, the subroutine counts how many times an event occurs in all the sets.

Finally, the dictionary is sorted based on the count of how many times an event has occured in the sets. We can iterate over the dictionary in descending order to find the most discriminating events first.

The algorithm is driven by the idea that, if an event has largely different values for *any* two programs, it captures an arbitrary aspect in which the two programs differ, and therefore we should inspect that event. However, there are many events that differ significantly for *any* two programs, and we end up with a large set of events to look at. Thus, we sort the dictionary to prioritize the events that differ in value for *more* number of programs than others. Once the dictionary is sorted, we can control how many top events we want to include in our final feature set.

Algorithm 1 Finding events that express difference between two programs, A and B

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

```
1: function FINDEVENTS(E_a, E_b, φ)
2:     bucket ← []
3:     for i ← 1 to E_a.size do
4:         if DIFF(E_a[i].value, E_b[i].value) ≥ φ then        ▷ percentage difference
5:             bucket.INSERT(E_a.name)
6:         end if
7:     end for
8:     return bucket
9: end function
```

A final preprocessing step is performed before using the dataset to train the learning model. This step is called feature scaling. Learning models are known to struggle if there is a large variance in the numeric range of features in a feature set. For example, if a feature has range (1 - 10), while another feature in the same feature set has range (-10000 - 10000), the learning model can struggle to draw accurate

Algorithm 2 Finding union set of all events with their occurence values

```
1:  function EventsUnion(programs)
2:      buckets ← []
3:      dict ← Dictionary()                    ▷ key: event name, value: occurence
4:      numProgs ← programs.length
5:      for i ← 0 to numProgs do
6:          for j ← i to numProgs do
7:              bucket ← FindEvents(programs[i], programs[j])
8:              buckets.insert(bucket)
9:          end for
10:     end for
11:     for bucket in buckets do
12:         for event in bucket do
13:             if dict.contains(event.name) then
14:                 dict.update(event.name, dict[event.name] + 1)
15:             else
16:                 dict.insert(event, 1)
17:             end if
18:         end for
19:     end for
20:     Sort(dict.value)
21:     return dict
22: end function
```

decision boundaries [39]. To address this, features in a feature set are scaled. Generally, to scale a feature vector x, the following formula is used:

$$x_i = \frac{x_i - \mu(x)}{max(x) - min(x)} \text{ for } i = 0 \text{ to } len(x)$$

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

Figures 4.5 and 4.6 show the effect of feature scaling. It can be seen that the plots are now significantly different from those in figures 4.3 and 4.4. Also, the

differences in the behavior of the programs are now much more apparent. For example, when not scaled, the plots for `blackscholes` and `bodytrack` are not as distinguishable as they are when scaled in figure 4.5. Additionally, `canneal` appears significantly different from both `bodytrack` and `blackscholes` in figure 4.5. The plot for `canneal` is indicative that many event values in `canneal` are not 0, contrary to how the plot for `canneal` appears in figure 4.3. Also, in figure 4.4, the plots for `swaptions`, `vips` and `x264` are very similar looking, but once the features are scaled it is clear that there are differences in their performance event values. This holds for all other programs. The plots of unscaled features are visibly sparse and empty, where as the plots for scaled features are much more drastically different across programs, and also very dense.

In addition to providing an effective visualization of the program in term of their programs, from feature scaling it is clear that to characterize programs the use of learning models well justified, as generating rules by hand from such a large feature set is not feasible. This also adds value to the pruning algorithm we have presented. From the plots in figures 4.3 and 4.4 it might appear that one could select performance events by hand based on the spikes in the graphs, but after feature scaling it is not quite as obvious.

4.4 Learning Models

The final component of the autotuning framework is a learning model. We have evaluated two different traditional learning models, logistic regression and decision

Figure 4.5: Feature scaling, part 1

trees, and designed a simple Euclidean distance based classifier that is well tailored to our needs and captures the available information best.

4.4.1 Training Labels

Learning models usually have two phases, training and testing. Both these phases are carried out on a dataset. The dataset is comprised of data instances. Each data instance has a feature set and a target. The learning models that I have used require that the

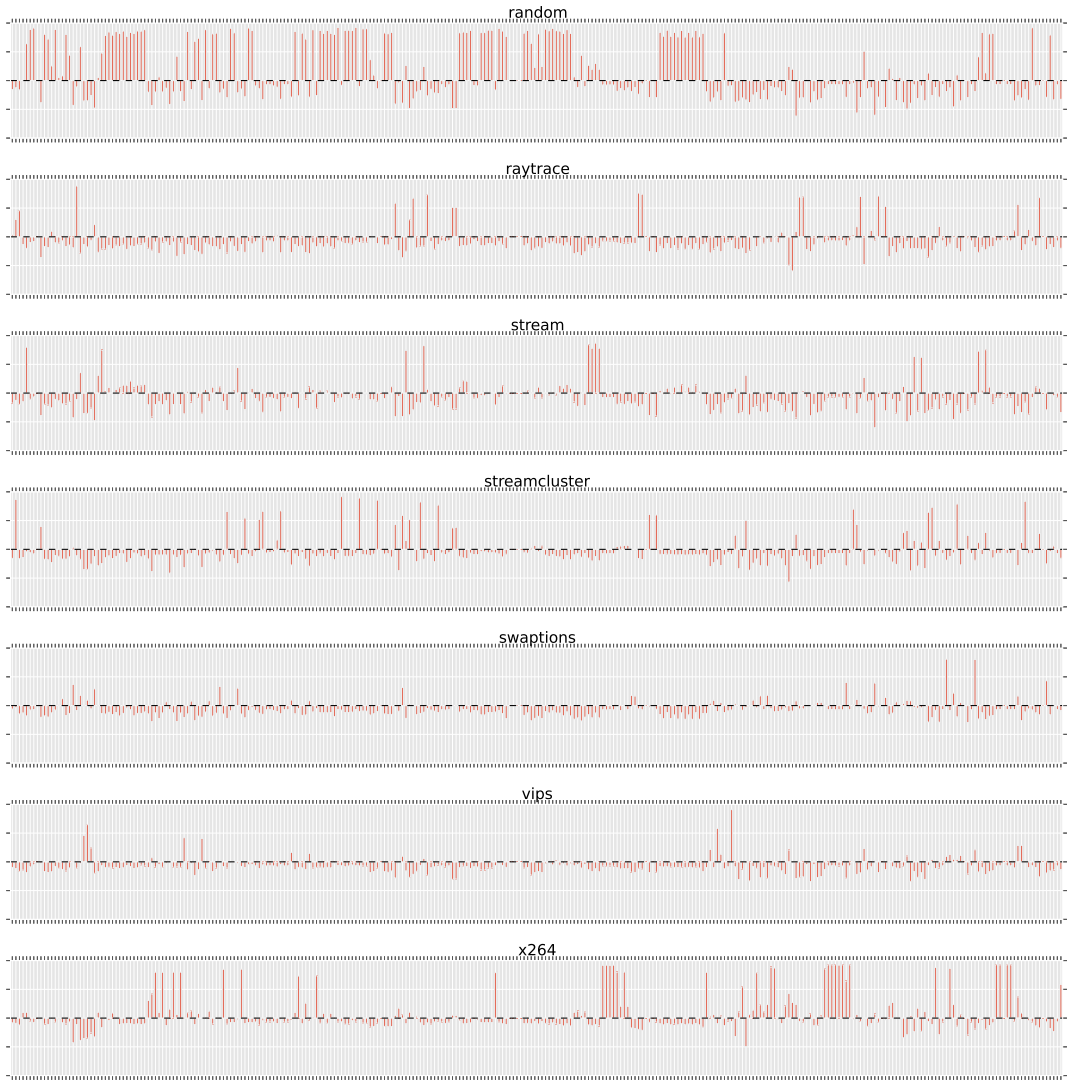


Figure 4.6: Feature scaling, part 2

feature set in all the data instances of a dataset be identical. When the training phase is carried out, the learning model internally uses statistical methods to draw decision boundaries such that each data instance is mapped to its corresponding target. The found decision boundary can subsequently be used to predict unseen data instances.

For our domain, the feature set is a collection of performance events. The target is formulated in two ways. In the first method, program is classified as whether it can benefit from prefetching. If by applying any configuration of prefetching has resulted

Table 4.1: Maximum achievable speedup by varying prefetch configurations

Program	Speedup	Program	Speedup
blackscholes	1.05	fludianimate	1.00
bodytrack	1.02	vips	1.14
facesim	1.32	x264	1.03
ferret	0.99	canneal	1.00
freqmine	1.51	streamcluster	1.37
raytrace	1.04	random	1.08
swaptions	1.00	stream	1.23

in a speedup of at least 10% over configuration `0000` the program is considered to have benefitted from prefetching. Out of the 14 programs, 4 do not benefit from prefetching at all. They are `ferret`, `swaptions`, `fluidanimate` and `canneal`. The other 10 have at least 1.02 speedup. If we classified programs as benefitting if they simply performed better on `1111`, then this would be a highly skewed dataset. Also, the effort for creating an autotuning framework is better justified for speedups above 10%.

We classify each program in this manner to form the dataset. From table 4.1 we can see that 5 data instances can be classified to be as benefitting from prefetching.

This approach, however, fails to distinguish between different prefetch configurations. From figure 4.2, it is evident that configuration `1111` is outperformed by other configurations in several cases. To address this, 4 independent learning models for 4 hardware prefetchers are used. Each classifier is tied to a prefetcher and predicts, for a given program, whether the corresponding prefetcher should be applied. For each program, the effect of applying only that prefetcher on configuration `0000` is considered. If a speedup of more than or equal to 2% is observed, the program is labelled as one that benefits from that prefetcher. I chose 2% here because

Table 4.2: Classifier training labels 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

contributions by the individual prefetchers may add up to a larger speedup.

4.4.2 Euclidean Distance Based Model

The final approach taken to predict configurations uses Euclidean distance as a similarity metric. When presented with an unseen program, this model computes distance between the unseen program and every other known program, and uses these distances as weights to compute the weighted score of the 16 prefetcher configurations across all known programs. Concretely, the model computes the following 16-length vector:

$$ConfigurationScore = \left[\sum_{i=0}^{m} \frac{P_i[0]}{d_i^2}, \sum_{i=0}^{m} \frac{P_i[1]}{d_i^2}, \ \ldots \ \sum_{i=0}^{m} \frac{P_i[15]}{d_i^2} \right]$$

where m is the number of the programs present in the training dataset. P_i is a

vector of length 16, corresponding to program i. Each index in the vector represents a configuration. Conversion of the index in its decimal form to binary results in the configuration. Each value in the vector represents the fraction of achievable speedup that was reached by using the corresponding configuration on program i. For example, $P_2[3]$ represents the fraction of speedup that was obtained on the 2^{nd} program using configuration `0011`, since $3_{10} = 0011_2$. Finally, d_i is the Euclidean distance between the unseen program and the i^{th} program.

When fully computed, the vector contains the weighted score of all the prefetching configurations across all programs. The recommended configuration is the binary representation of the index belonging to the maximum element in the vector. This approach considers the effect of every prefetching configuration on all known programs and qualifies this score using distance squared. Thus, the scores of more similar programs have greater effect than those that are different. Also, in the event that an unseen program is very similar to multiple programs, and not just one, this approach ensures that the most similar program is not the only basis of decision, as the scores from all programs are used.

The motivation for designing this approach stems from the failure of capturing interaction between independent prefetchers when using 4 independent classifiers for predicting prefetch configurations. From table 4.2 it is observed that a fully accurate classifier would not be able to recommend prefetcher configurations that result in the maximum speedup. For example, consider `raytrace`, where the recommended prefetchers are `hw` and `dcu`, which is configuration `1100`. However, the combination

of the two prefetchers does not yield the best performance. Thus, I have designed a Euclidean distance based metric that captures similarity in program behavior sufficiently to recommend prefetch configuration.

5. RESULTS AND ANALYSIS

5.1 Experimental Environment

In this section the machine and compile time configurations used throughout the experiment are described.

5.1.1 Machine Configuration

The machine the experiments were conducted on has a 4 core Intel Core2 Quad CPU. Each core has a clock speed of 2.40GHz. The individual cores have 32KB of L1 cache. Cores 0 and 2 share a 4MB L2 cache, and cores 1 and 3 share a separate 4MB L2 cache, resulting in a total 8MB of L2 cache. The machine also has 4GB of main memory.

5.1.2 Build Configuration

The programs are all built using GCC 4.8.2 using optimization level -O2 on an Ubuntu 14.04 operating system. The PARSEC programs were invoked using the provided `parsecmgmt` script, and all 14 programs were run using 8 threads on the `native` input. We initially experimented with the effect of varying the number of threads across different prefetching configurations, however little change in speedup was observed. 8 threads were chosen even though the machine has 4 cores because it can slightly skew the operating system thread scheduling policy in favor of the experiment.

5.2 Evaluation Method

This section discusses how the recommendations from the learning models have been assessed.

5.2.1 Cross Validation

We have validated our learning models using a well-known validation method called $(k-1)$ validation, also known as leave-one-out validation. In this method, for a dataset containing k data instances, the learning model is trained using $(k-1)$ data instances, and tested using the k^{th} data instance. This is repeated k times such that each data instance is used as a test data instance. We chose this validation approach because of the low number of data instances in our data set.

5.2.2 Metrics

Initially, a simple prediction accuracy to assess the performance of the models was used. However, prediction accuracy in this problem statement does not capture the effectiveness of learning models. For instance, the best configuration for the program `streamcluster` is `0110`, which results in a speedup of 1.37, and the models are trained to predict this configuration. However, if the model predicts `0111`, then this prediction should not be considered to be completely incorrect because `0111` yields a speedup of 1.34. Therefore, when assessing the performance of the learning model, it is more effective to look at how much of the achievable speedup can be found using the recommendation from the learning model. Thus, ideally the goal is for the

learning models to predict configurations that allow the user to reach 100% of the possible speedup.

5.3 Choosing the Optimal Number of Features

I have performed feature selection based on algorithm 1 and 2. However, the efficacy of the pruning algorithm needs to be tested. Also, the optimal number of features for the learning algorithms also need to be determined. Concretely, from the sorted dictionary, obtained form algorithm 2, it is yet to be determined how many of these top performance events one should use to train the models. To do this, increasing number of features are used to train a decision tree based classifier, until the learning model benefits from more features or starts to perform poorly. I chose a decision tree for this purpose because the tree output of a decision tree allows inspection of the decision making process. On the other hand, a logistic regression classifier would function as a black box model and it would be difficult to understand the underlying process. Figure 5.1 shows a sample decision tree that was found after training a model with top 8 performance event values. In this sample tree, the decision tree makes its decision on the basis of two events: `SIMD_UOP_TYPE_EXEC_ARITHMETIC` and `SIMD_UOP_TYPE_EXEC_LOGICAL`. If the scaled value of the first event is greater or equal to -0.0719, then it suggests the user to not prefetch. If the value is less, then it checks the scaled value of the second event. If it is greater than or equal to -0.2926 it suggests prefetching, otherwise not.

The classification task for this purpose is to predict whether given a program, the

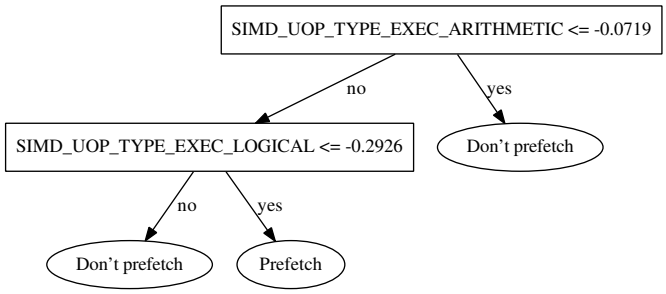


Figure 5.1: Decision tree found from training on top 8 performance events

program is likely to benefit by more than 10% from config `1111`. First, the classifier is trained with values of all the performance events, and then repeatedly trained the classifier using increasing number of top performance events found from the pruning algorithm. The precision, recall and accuracy in prediction for the model are measured. The $(k-1)$ cross validation was used. Precision and recall are defined as:

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

True positive are data instances that belong to positive class and have been correctly classified as such. False positives are those that have been incorrectly classified as positive. True negative and false negative are also defined similarly - true negative are correctly classified negative instances and false negatives are those that were incorrectly classified as false.

Precision tells us out of the classes labelled positive by the classifier, how many were actually positive. This is a stronger metric than simply accuracy. Recall tells us,

Table 5.1: Effect of varying the number of features on learning

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

out of all the positive instances in the data set, how many was the classifier able to identify. In our case, we define a data instance as positive if we should apply prefetching on it.

From table 5.1 it can be seen that the best classification occurs with top 8, 9 and 10 features. It also results in the best precision and recall, too. The decision trees found from these feature sets are identical, in that they make their decisions on the same features and same values. Adding more features causes a decline in the classifier's prediction. For the rest of the experiments top 8 performance events are used as the feature set for the logistic regression and decision tree models. For the Euclidean distance based model, top 6 features were used. During the experiments, it was discovered that using 6 features for this model improved the quality of recommendation. The 8 features and their accompanying definition from Intel manuals are provided in table 5.2.

Table 5.2: Top 8 performance events used as features

Feature/Event	Definition
FP_ASSIST	This event counts the number of floating point operations executed that required micro-code assist intervention.
SIMD_COMP_INST_ RETIRED_SCALAR_ DOUBLE	This event counts the number of computational SSE2 scalar-double instructions retired. Computational instructions perform arithmetic computations (for example: add, multiply and divide).
SIMD_INST_RETIRED_ SCALAR_SINGLE	This event counts the number of SSE scalar-single instructions retired.
SIMD_UOP_TYPE_EXEC_ LOGICAL	This event counts the number of SIMD packed logical micro operations executed.
SIMD_COMP_INST_ RETIRED_SCALAR_ SINGLE	This event counts the number of computational SSE scalar- single instructions retired. Computational instructions perform arithmetic computations (for example: add, multiply and divide).
SIMD_UOP_TYPE_EXEC_ ARITHMETIC	This event counts the number of SIMD packed arithmetic micro-ops executed.
SIMD_INST_RETIRED_ SCALAR_DOUBLE	This event counts the number of SSE2 scalar-double instructions retired.
SIMD_INST_RETIRED_ VECTOR	This event counts the number of SSE2 vector integer instructions retired.

5.4 Recommending Optimal Prefetching Configuration

In this section I present the outcome of using learning models to predict the optimal prefetch configuration. I used 4 independent learning models for each of the prefetchers. To evaluate the models I measured how much of the achievable speedup can be obtained from using these models.

The performance of the logistic regression model is shown in table 5.3. On average this model reaches 92.42% of the achievable speedup. It suffers in predicting a configuration for `freqmine`. This is because the training data is formed on the basis of whether applying a prefetcher results in a speedup of more than 2% over

Table 5.3: Fraction of achievable speedup reached by the three different models

Program	Logistic Regression	Decision Tree	Euclidean Distance
blackscholes	0.95	0.95	0.97
bodytrack	0.98	0.98	0.99
facesim	0.96	0.98	0.89
ferret	0.99	0.99	0.99
freqmine	0.67	1.00	0.99
raytrace	0.88	0.99	1.00
swaptions	1.00	1.00	1.00
fluidanimate	0.96	0.98	0.98
vips	0.88	0.87	0.90
x264	0.98	0.97	0.98
canneal	0.95	0.95	0.98
streamcluster	0.82	0.82	0.78
random	0.93	0.85	1.00
stream	1.00	1.00	1.00

`0000`. Applying the `cl` and `dcu` prefetcher separately as config `0100` and `0010` does not yield speedup, therefore the models are trained to leave them turned off. However, the better combinations have both these prefetchers turned on. In this case, two prefetchers that did not result in significant speedup individually, resulted in speedup greater than their sums combined.

The decision tree based model performs better than the logistic regression model on average, reaching 95.26% of the achievable speedup. For `freqmine` it is able to suggest the best configuration. This has likely occured by chance because the information about the interaction of the prefetchers is not carried into the independent models.

The Euclidean distance reaches 96.11% of achievable speedup on average. Like the decision tree, it is able to suggest a good configuration for `freqmine`, however this is not a case where the model simply got lucky. This is because the configurations

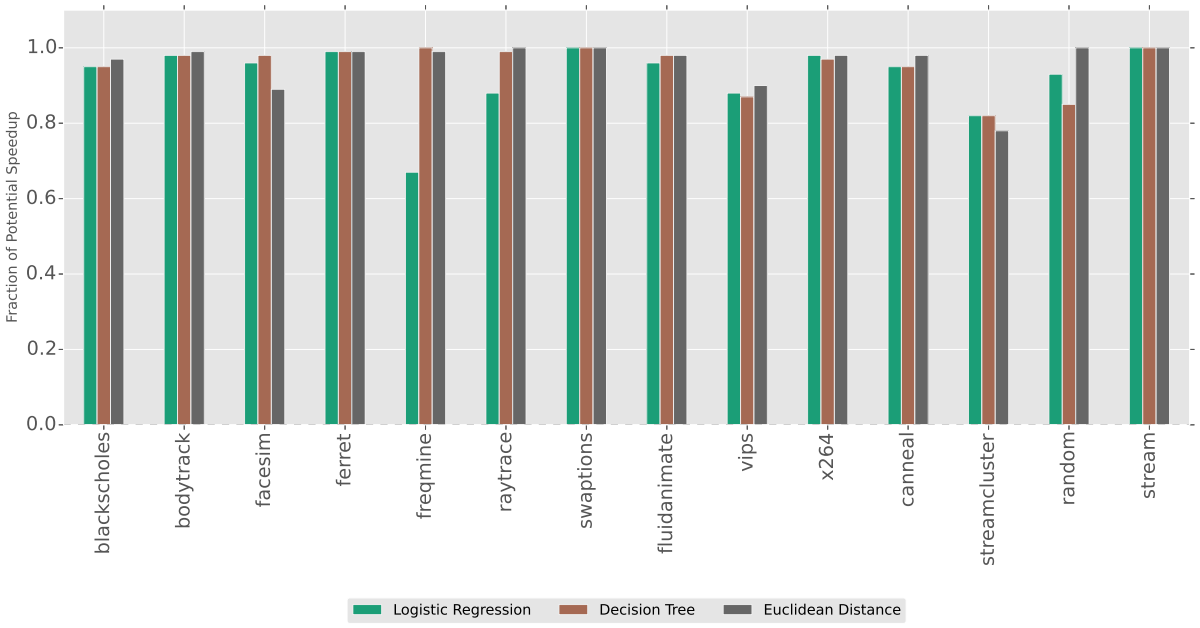


Figure 5.2: Fraction of achievable speedup reached by the three different models

are recommended based on the most similar program in the training data. Thus, this model makes predictions based on program behavior, instead of the effect of individual prefetchers.

The Euclidean distance based classifier is a good fit because it makes full use of the information found from our exhaustive search of the hardware parameter space for the optimal configuration. The other classifiers attempt to make predictions for the *best* configuration based on the impact of the individual prefetchers, discarding the information that is available about mixed configurations where multiple prefetchers are turned on. This model makes use of that information by associating an unseen program with its closest neighbor.

The only instance where this model is outperformed by the other happens in the case of `stream`. We inspected the distance between `stream` and every other

program and found that it is closest `raytrace`. The best configuration for `raytrace` is `1111` which results in a speedup of 1.04 in `stream`, but the maximum possible speedup in `stream` is 1.23. Also, `stream` achieves its best speedup on configuration `1000`, which has only one prefetcher enabled. Since the other models predict single prefetchers, they perform better in this case.

All the models also suffer in predicting configurations for `streamcluster`. This can be explained by inspecting the Euclidean distance model. I found out that `streamcluster` is significantly far from every other program. It is closest to `stream`, and using the recommendation for `stream` on `streamcluster`, speedup of 1.16 is gained. However, because it is possible to get speedup up to 1.37 by using configuration `0110`, the bar for `streamcluster` in figure 5.2 is at 0.84. Nonetheless, even though the learning models have not been able to recommend the optimal configuration, the suggested configuration still results in significant speedup.

6. CONCLUSION

This section summarizes the contribution and the work done in this thesis, and indicates the direction this work can take in the future.

6.1 Contribution

This thesis presents the effects of hardware prefetching on the PARSEC benchmark suite and 2 sample programs. On Intel processors, 4 hardware prefetchers are available, and 16 different prefetching configurations can be formed by enabling and disabling each and/or a combination of the prefetchers. The effects of applying all these prefetching configurations on the 14 programs were recorded. The programs were also characterized using performance events. However, there are too many measurable performance events available on modern machines, making it very difficult to understand each and every one of them. All the events can not be measured in a single execution of a program that needs to be characterized. On the Intel Core2 machine that was used, each program was run 194 times to collect the value of all the performance events. This is a huge overhead, and to eliminate most of it, I used an effective pruning algorithm to identify a concise set of performance events that should be used to characterize programs. After pruning, it was possible to come down to 8 performance events. These events were used to characterize the programs.

Once it was possible to characterize programs, 3 learning models were used to to predict optimal prefetching configurations. The models are logistic regression, Gini decision tree and a Euclidean distance based similarity model. I used $(k-1)$

validation to assess the performance of our models. For this problem statement simple prediction accuracy was not a good fit, as multiple prefetch configurations result in significant speedup. Therefore, it was observed how much of the achievable speedup could be extracted by using the learning models. The best performing model was the Euclidean distance based model. On average, it allowed extraction of 96.11% of the achievable speedup across all programs. Liao et al. [36] formulated their problem in a similar way to how was done in this thesis, by observing how much of the possible speedup can be achieved using learning models. They reported achieving up to 96.50% performance using a support vector machine on a collection of data center applications. This measure, however, is a weighted geometric mean, where the weights are reflections of how often an application is used in a data center.

6.2 Future Work

In this work I presented a framework that predicts optimal hardware prefetching configurations. There are several ways I would like to extend this work. First, I would like to assess the effectiveness of the framework on more programs. I have currently used the PARSEC benchmark suite because it is diverse in the nature of the applications it contains, as well as it is one of the newer benchmarks. I would now like to target other benchmark suites and assess how well our learning models perform. In addition, it would be interesting to see how well the models perform when there is a large number of programs.

Next, I would like to apply this framework on a different architecture, and also a

different memory hierarchy. Finally, I would like to use the framework with a different search space. I have currently used it to study the effects of prefetching. It would be useful to validate whether the general process of optimization holds for other areas such as working set size and thread migration strategies.

LITERATURE CITED

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

[2] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *Computers, IEEE Transactions on*, vol. 44, no. 5, pp. 609–623, 1995.

[3] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," tech. rep., University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[4] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 316–326, ACM, 2009.

[5] 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*, pp. 72–81, ACM, 2008.

[6] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pp. 364–373, IEEE, 1990.

[7] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely Jr, and J. Emer, "Pacman: prefetch-aware cache management for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 442–453, ACM, 2011.

[8] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Inter-core prefetching for multicore processors using migrating helper threads," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 393–404, 2011.

[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*, pp. 185–197, IEEE, 2007.

[10] 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.

[11] Intel, *Intel 64 and IA-32 Architectures Software Developers Manual Volume 3 (3A, 3B & 3C): System Programming Guide*, February 2014.

[12] J. Demme and S. Sethumadhavan, "Rapid identification of architectural bottlenecks via precise event counting," in *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 353–364, ACM, 2011.

[13] K. Naono, K. Teranishi, J. Cavazos, and R. Suda, *Software automatic tuning: from concepts to state-of-the-art results*. Springer Science & Business Media, 2010.

[14] A. Qasem, J. Guo, F. Rahman, and Q. Yi, "Exposing tunable parameters in multi-threaded numerical code," in *Network and Parallel Computing*, pp. 46–60, Springer, 2010.

[15] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan, "Loop transformation recipes for code generation and auto-tuning," in *Languages and Compilers for Parallel Computing*, pp. 50–64, Springer, 2010.

[16] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–12, IEEE, 2009.

[17] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes, "The alpbench benchmark suite for complex multimedia applications," in *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, pp. 34–45, IEEE, 2005.

[18] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "Biobench: A benchmark suite of bioinformatics applications," in *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pp. 2–9, IEEE, 2005.

[19] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pp. 330–335, IEEE Computer Society, 1997.

[20] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "Minebench: A benchmark suite for data mining workloads," in *Workload Characterization, 2006 IEEE International Symposium on*, pp. 182–188, IEEE, 2006.

[21] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[22] 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*, pp. 253–262, ACM, 1987.

[23] S. Mehta, Z. Fang, A. Zhai, and P.-C. Yew, "Multi-stage coordinated prefetching for present-day processors," in *Proceedings of the 28th ACM international conference on Supercomputing*, pp. 73–82, ACM, 2014.

[24] 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, pp. 357–368, ACM, 2013.

[25] 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.

[26] 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*, pp. 342–352, ACM, 2005.

[27] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesnt, and why," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 1, p. 2, 2012.

[28] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 63–74, IEEE, 2007.

[29] S. W. Son, M. Kandemir, M. Karakoy, and D. Chakrabarti, "A compiler-directed data prefetching scheme for chip multiprocessors," in *ACM Sigplan Notices*, vol. 44, pp. 209–218, ACM, 2009.

[30] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Computing Surveys (CSUR)*, vol. 32, no. 2, pp. 174–199, 2000.

[31] M. Kandemir, Y. Zhang, and O. Ozturk, "Adaptive prefetching for shared cache based chip multiprocessors," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 773–778, European Design and Automation Association, 2009.

[32] I. Ganusov and M. Burtscher, "On the importance of optimizing the configuration of stream prefetchers," in *Proceedings of the 2005 workshop on Memory system performance*, pp. 54–61, ACM, 2005.

[33] 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*, p. 30, ACM, 2013.

[34] 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.

[35] 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.

[36] 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*, p. 56, ACM, 2009.

[37] 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*, pp. 207–216, IEEE, 2010.

[38] R. A. Vitillo, "Performance tools developments." Accessed: 2015-04-01, 2011.

[39] A. Ng, "Feature scaling." 2014.