Adaptive Cache Prefetching using Machine Learning and Monitoring Hardware Performance Counters

A Thesis

SUBMITTED TO THE FACULTY OF

UNIVERSITY OF MINNESOTA

BY

Pranita Maldikar

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

Adviser

David Lilja

June, 2014

© Pranita Maldikar 2014

ALL RIGHTS RESERVED

Acknowledgements

I want to thank my adviser Prof. David Lilja for his support and immense faith he had in me to allow me to do 11 month internship. I would like to thank Dr. Kingsum Chow and Robert Scott in Intel Corporation for offering me the summer internship opportunity in their group. I got to work on a very diverse and exciting project. I am sincerely thankful for their insight, help and guidance throughout my internship duration. Due to the nature of my project I got to stay with Intel Corporation for 11 months. During my internship I got to meet many other talented interns. I will take this opportunity to thank my fellow intern Manjunath Shevgoor for the constructive feedback. I would also like to thank people from our team Khun Ban and Huijun Yan, at Intel Corporation.

Along with the people from my team I would like to extend my gratitude to people from Intel Labs Chris Wilkerson, Dr. Zeshan Christi and Dr. Shei-lein Lu for the stimulating discussions and their wonderful feedback.

I would like to thank my thesis committee: Prof. John Sartori and Prof. Pen-Chung Yew.

I thank my friends Harshal Chaudhari, Parth Chaudhari and Amogha Gundavaram for the sleepless nights they spent talking to me and giving me moral support.

And last but not the least I would like to thank my family: my parents Prakash Maldikar and Gitika Maldikar for giving me this opportunity to study at such a wonderful university and to follow my dreams; and to my siblings Sonali Maldikar and Akshaj Maldikar for being there for me always.

Dedication

I would like to dedicate my thesis to my parents Prakash and Gitika.
It’s because of them I got this opportunity.

Abstract

Many improvements have been made in developing better prefetchers. Improvements in prefetching usually starts by coming up with a new heuristic. The static threshold values for prefetching modules might become obsolete in near future. Given the huge amount of hardware performance counters we can examine, we would like to find out if it is possible to derive a heuristic by applying machine learning to the data we routinely monitor. We propose an adaptive solution that can be implemented by monitoring the performance of system at run-time.

Machine learning makes system smarter by enabling it with ability to make decisions. So for future complex problem instead of running lot of experiments to figure out optimal heuristic for a hardware prefetcher we can have the data speak for itself, and the machine will choose a heuristic that is good for it. We will train the system to create predictive models that will predict prefetch options at run-time.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

The first single chip microprocessor was invented in 1970s. Since then there has been tremendous growth in field of microprocessors. With every generation, the processor performance is improving. Each new processor generation tries to tackle and overcome different bottleneck components. In the earlier decades the processor performance was underwent rapid growth compared to memory performance. This resulted in a huge performance gap between processor and memory. Fig 1-1 describes the trend in performance boost over years for processor and memory.

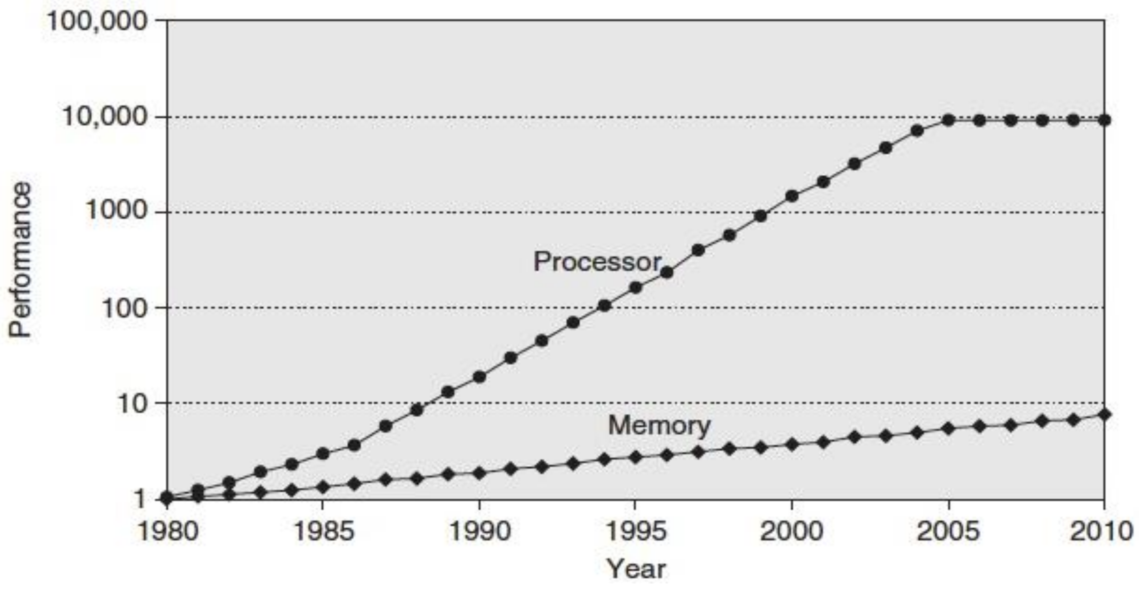


Figure 1-1 Performance gap between process and memory over years [1]

Current generation processors take only couple to cycles to execute an operation but the memory access takes tens of cycles; thereby degrading the performance. Thus the system speed will still be limited because of memory. For years, research has being conducted to match memory speed to processor speed. There are many techniques working in the background to minimize this memory latency. Prefetcher is one such technique that is a very promising alternative to bridge the gap between processor and memory performance.

Prefetching works in the background to ensure that memory does not become a bottleneck parameter. A prefetcher preloads the data accurately and timely before the data is requested by the program. This reduces or completely eliminates the memory latency for demand requests. Prefetching does not require the program to have seen the data, so it can eliminate the compulsory cache misses. Thus reducing miss rate and miss latency. Most of the prefetchers initiate a prefetch request when they see a demand miss in cache. Along with the address requested, the prefetched data is also brought into the cache. These prefetch requests are brought into the cache with a lower priority, so that preference is given to demand requests. Over years many prefetching approaches and heuristics have been proposed and implemented. The present lot of prefetching algorithms track or profile the access pattern of the program to make best possible decision for prefetching data/instruction. The main objective of prefetching is to correctly predict the access stream of the program and bring the data closer to CPU before it is requested by the program. As we know that the phase of the workload keeps on changing we cannot use the same prefetch option throughout the course of the workload. With access pattern profiling or by establishing the miss pattern we can predict the access stream of workload and issue prefetchers accordingly.

This thesis presents a new approach for prefetching data/instruction into cache. Primary contribution of this thesis is to eliminate the need to have fixed heuristics for prefetchers. It uses machine learning techniques to decide heuristics for prefetchers on interval basis. Based on the hardware counters it decides the heuristics for prefetchers. These counters are usually collected on interval basis. Hence the prefetcher heuristics can be different during different intervals. Machine learning techniques are very effective in making the system smarter and they enable the system with decision making capability. My efforts have been along the same line. With all the performance counters collected at every interval, we train the system using these counters to make better prefetch decision on its own.

Chapter 2 Related Work

As described in previous chapter prefetching is an important concept which helps bridge the gap between processor and memory. This chapter covers basics of prefetching technique, conventional prefetchers, and some current prefetching algorithms proposed.

A good prefetcher needs to take into account few things as described in paper [2]. Prefetcher module should predict the access pattern of the program correctly otherwise it will bring in lot of useless data which the program might not use. This will result in wastage of expensive resources like memory bandwidth, cache or prefetch buffer, energy consumption etc. This phenomenon is called cache pollution. Hence the prefetcher module needs to accurately predict the data that the program might request in the near future.

Along with accurate prefetching, the initiated prefetch requests should be timely. If the data is prefetched too early then it might not be used before it is evicted from the cache. Also if the prefetch request is issued late then it might not hide the entire memory latency. The data needs to be present in the cache before the processor issues the memory request. The timeliness of the prefetcher can be improved by making the prefetcher aggressive i.e. prefetching data far ahead in the program's access stream. Prefetching at software level employs machine learning techniques to figure out when to start prefetching and from where.

The prefetch requests could be stored in cache or in separate prefetch buffer. If it is stored in cache then due to cache pollution we may lose some demand data. If prefetch buffer is used then there would be some design considerations like size of buffer, placement of buffer in memory hierarchy and the coherency in the buffer which needs to be taken care of. Thus implementation of it will be difficult.

The destination where the prefetched data is stored also makes a difference. If data is brought directly into is L1 cache from memory, cache pollution will have a greater impact on performance due to its smaller capacity. At L1-L2 cache boundary we can have a highly accurate conservative prefetcher and an aggressive prefetcher on last level cache.

The state of the prefetched data is also important, i.e. whether to treat prefetch blocks as demand blocks or not? With LRU policy the prefetched data will be placed at the MRU position. It can also be placed at the LRU position if we have some idea about the accuracy of prefetcher. For higher accuracy prefetches we can place them in MRU position and for less accurate prefetches we can put them in LRU position. Prefetched data placed at LRU position can minimize the effect of cache pollution. The paper [10] explains some of the concepts of LRU and MRU policy

2.1 Different types of prefetching technique

Prefetching can be done either at software level or hardware level. Software prefetches are inserted in the code by programmer or compiler. Hardware prefetches are issued when you see a demand miss in cache and data is prefetched based on the heuristic of the hardware prefetcher.

2.1.1 Software Prefetching

Instruction Set Architecture provides some prefetching instructions. Software prefetching utilizes this feature. In software prefetching the programmer or the compiler inserts these prefetch instructions. It is programmer's responsibility to add these prefetch instructions to improve the performance of program. Paper [3] talks about software controlled data prefetching. Software prefetching works well for programs that exhibit regular access pattern. But it takes up some execution bandwidth of the system. With software prefetching it is difficult to predict timeliness of prefetch request. The compiler does not have feedback about the latency of memory accesses. So where to insert the prefetch instruction becomes an important decision. The prefetch instructions can be inserted for every load access but that will increase the memory and execution bandwidth consumption. With software prefetching we can have profiler to profile the code and determine the loads which are likely to miss. But this profiled input data set might not be representative. It is difficult to do software based prefetching on pointer based data structure, but with compiler optimizations and for regular access patterns software prefetching is very beneficial.

2.1.2. Execution based prefetcher

This type of prefetching could be implemented in software or hardware. This approach uses a separate thread to prefetch data for the main program. This thread can be dynamically generated by software or hardware. Run-ahead execution can be thought of execution based prefetcher [9]. A very simple example of this type of prefetcher can be a main program which has a helper thread or prefetch thread inserted in between the code. This thread comprises of instructions which are necessary to predict the address required for prefetcher. When the main program reaches this instruction of launching the helper thread or prefetch thread, it will execute the helper thread and helper thread will quickly compute the address required for prefetch before the main program reaches the point where it is going to generate cache miss for that data.

2.1.3 Hardware Prefetching

Hardware prefetcher operates in the background without programmer's intervention. It is triggered when cache miss occurs. Based on the heuristics provided to the hardware prefetcher it will prefetch cache lines. Some of the hardware prefetchers monitors the subsequent cache misses and prefetch data based on the stride pattern exhibited by the program. It is micro-architect's responsibility to design a prefetching algorithm which will accurately and timely generate the prefetch requests. Some of the conventional hardware prefetchers and heuristics are mentioned below.

2.1.3.1 Next 'N' Line prefetchers:

It is the simplest form of hardware prefetching. It always prefetches next 'N' lines after demand misses. It assumes that the program exhibits spatial and temporal locality. So if the program is making a request for address 'A' then in future it will request for address closer to A. The value of N will determine the aggressiveness of the prefetcher. The term aggressiveness is described in the section 2.2.2. For workloads exhibiting irregular patterns this prefetching scheme might not work and result in performance degradation rather than improvement [4].

2.1.3.2. Stream Prefetchers:

A stream prefetcher unlike next line prefetcher is a confirmation-based prefetcher. A miss at address A will not result in prefetch request. This address will be recorded. The stream prefetcher will now begin to monitor if the program is making a request for address at some offset from A. Let us say that the program then sees a memory request A+1. It will still not issue prefetch request as there is not enough evidence that this is the true stream. Only after *A+2* is seen, the stream is fully confirmed and prefetching will start. For regular access pattern the stream prefetcher works better but for shorter streams it does not work well due to the wait required to confirm the stream. Also for irregular access patterns like indirect array accesses, linked data structures, multiple regular stride, random patterns correlation based prefetchers, content directed prefetcher, pre computation or execution based prefetchers [5, 6, 7] works well instead of stream prefetcher.

2.2 Performance metrics that can be used to evaluate goodness of a prefetcher

Prefetcher impacts performance of the system. A prefetcher can improve the performance by huge margin or it can even degrade the performance. The metrics listed below helps us understand if a prefetcher is going to improve the performance or not.

2.2.1 Prefetch Accuracy

It is a measure of how accurately the prefetcher can predict the access stream of the program. Highly accurate prefetchers completely capture the access pattern of the program. Access pattern of the program can be predicted by looking at the previous access stream. It is defined as out of the total prefetches issued, number of prefetches used by the program. This ratio should be close to 1.

2.2.2 Aggressiveness of Prefetcher

Aggressiveness of the prefetcher means how far ahead your prefetcher can issue prefetches. The prefetch distance will determine the aggressiveness of the prefetcher. Aggressive prefetcher will prefetch lines which are far ahead from the current demand miss. This can result in lower accuracy due to two reasons. First, it is difficult to predict accesses far ahead in the program. Second, for an aggressive prefetcher the data might be brought into cache too early and it might get evicted before it is used. So we need a trade-off between aggressiveness of prefetcher and its accuracy. If the prefetcher is not aggressive enough then the prefetch requests might be on its way to cache. Hence we might not be able to hide the entire memory latency.

2.2.3 Coverage

Coverage of a prefetcher means out of the total number of compulsory or demand cache misses, number of cache misses eliminated by the prefetcher. For higher coverage you need an accurate prefetcher which will predict the addresses correctly also you need an aggressive prefetcher so that the data is already present in the cache.

2.2.4 Timeliness of Prefetcher

Timeliness of Prefetcher is defined as, out of the total used prefetched lines number of prefetches present in the cache before the compulsory miss occurred. For a timely prefetcher you need an aggressive prefetcher so that it will always stay ahead in the processor's access stream.

2.2.5 Cache Pollution

To accommodate the prefetched data in cache the eviction policy knocks out some of the data from the cache. If these prefetches are inaccurate we are knocking out data from cache to make space for this useless data. It might be evicting out some useful stuff to accommodate this data that might not be referenced. This is called as cache pollution. Aggressive prefetchers bring data to cache by looking far ahead in the access stream but due to early prefetch it causes cache pollution.

2.3 Current Prefetching Algorithms

Recent hardware prefetching algorithms try to monitor access pattern of the program. They also try to monitor the effect of it on actual hardware. If the prefetching scheme is hampering the performance of the system then it will either detach the prefetching model or it will alter the prefetching scheme to tune it to benefit the access pattern. Listed below are the two modern prefetching algorithms considered. Feedback directed prefetching [8] and Sandbox prefetching [23].

2.3.1 Feedback Directed Prefetching [8]

Aggressive prefetchers look far ahead in the program access stream. If we choose an aggressive prefetcher we have to compromise on accuracy. We need to fine tune the prefetching aggressiveness based on the access stream and performance of that prefetcher. Figure 2-1 shows the effect of aggressive prefetcher on SPECcpu workloads. Ammp and applu workload's performance degrades with aggressive prefetchers whereas rest of the workloads show increase in IPC value [8]. Hence based on the premise of the paper [8] we need to tune the aggressiveness of the prefetcher based on its effect on performance.

Feedback directed prefetching [8] reduces the negative performance and bandwidth impact of aggressive prefetching while preserving the large performance benefits provided by aggressive prefetching. It maintains performance counters like prefetch accuracy, prefetch latency and cache pollution which tracks the impact of different prefetch options. There are five levels of aggressiveness described in the paper [8]

1. Very conservative prefetching - Prefetch distance 4 and prefetch degree 1
2. Conservative prefetching - Prefetch distance 8 and prefetch degree 1
3. Mid-level prefetching - Prefetch distance 16 and prefetch degree 2
4. Aggressive Prefetching - Prefetch degree 32 and prefetch degree 4
5. Very aggressive prefetching - Prefetching Distance 64 and prefetch degree 4

At the end of an interval these counters are computed. Using static threshold values and these performance counter values, aggressiveness of the prefetcher is calculated. Figure 2-2 below shows the performance of FDP prefetcher.

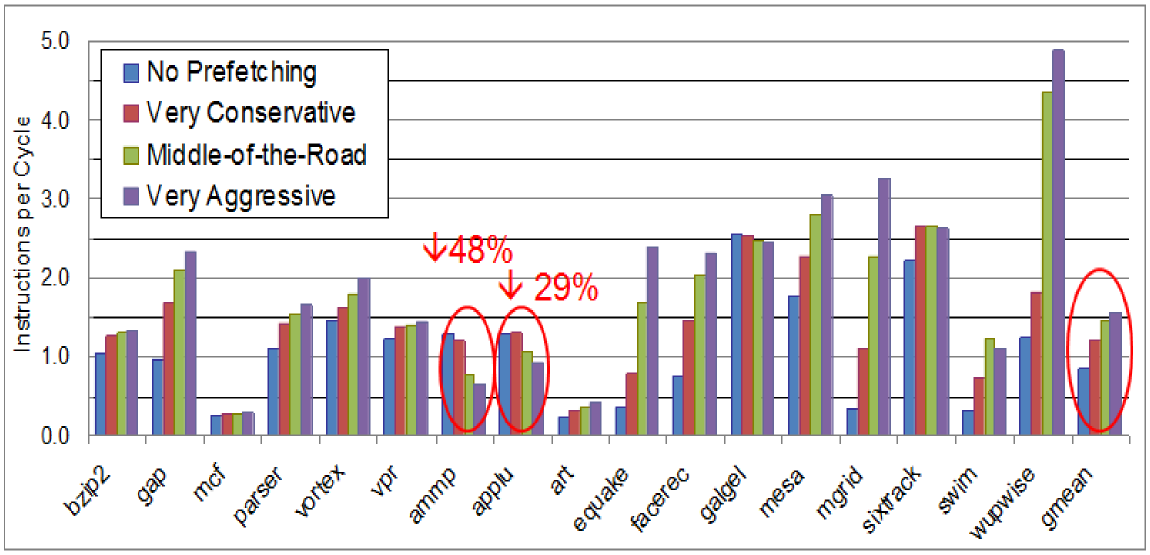


Figure 2-1 Performance of various workloads with different prefetching aggressiveness [8]

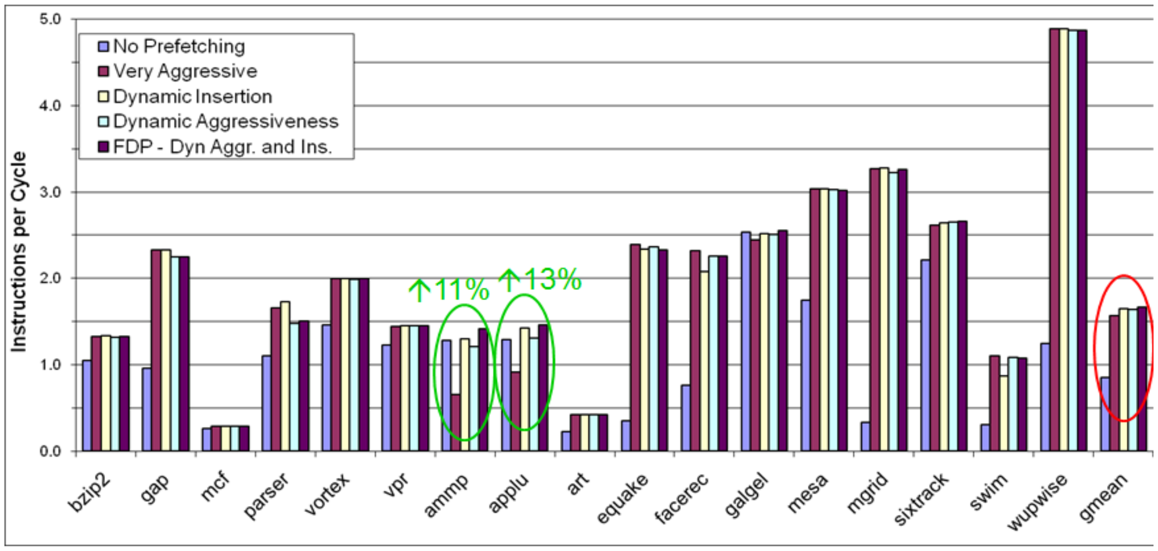


Figure 2-2 Performance of various workloads with various prefetching mechanism [8]

2.3.2 Sandbox Prefetching Algorithm:

Sandbox prefetcher [23] evaluates various prefetching options without affecting the actual cache contents. It has a storage structure which tracks the access stream of the workload. If a prefetching option has potential to benefit the performance then that particular prefetch option is turned on. It uses threshold approach to select the potentially accurate prefetch option. Sandbox prefetcher improves performance by 47.6% compared to no prefetch and 18.7% compared to Feedback Directed Prefetch [8] and 1.4% compared to Access Map Pattern Matching Prefetcher [13].

Sandbox Prefetcher does extremely well in computing the accuracy of the prefetching options and detecting streams within the workload. But the only limitation is the static threshold. In the paper [23] the author has used a score of 512 to turn on 2 highest scoring prefetchers, score of 768 to turn on three highest scoring prefetchers and a cut-off score of 256. The performance is sensitive to these score values.

I have replicated sandbox prefetching algorithm in which the score value used is the prefetch accuracy. The detailed implementation of sandbox prefetcher is in chapter 5. The graph shown below fig 2-3 is the performance (CPI) of various workloads with different threshold values (cut off scores).

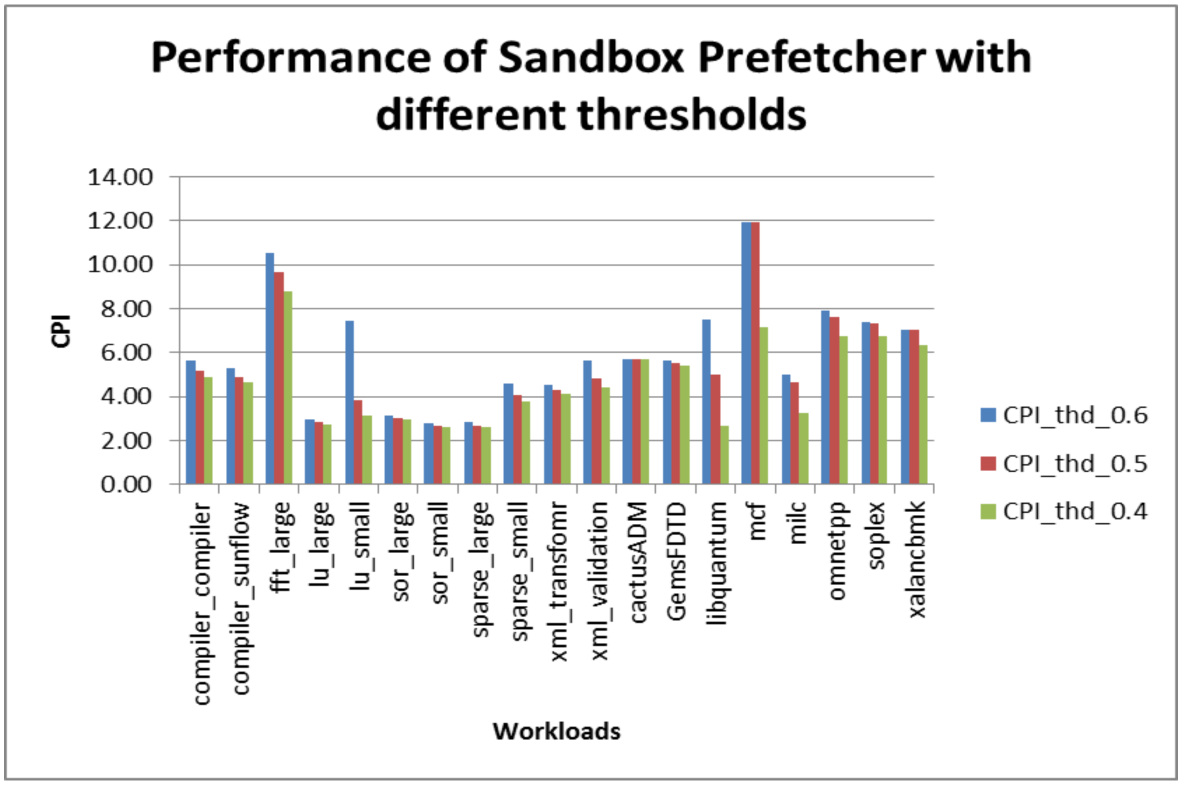


Figure 2-3 Performance of different workloads when subjected to different threshold value

Figure 2-3 comprises of different workloads from desktop to server space. From the graph you can see that cactusADM and GemsFDTD do not have a lot of performance improvement by tuning the threshold value. Whereas libquantum and mcf are greatly affected by threshold value selected. CactusADM has an MPKI of 14.25 and GemsFDTD has MPKI of 27.54. They have potential to benefit from prefetching but the threshold values might be too high from them. Libquantum on the other hand benefits from lower threshold value. Mcf has almost same performance for threshold of 0.6 and 0.5 but as you lower the threshold to 0.4 you see performance improvement. Some threshold value might benefit some workload but it might not benefit other workloads.

2.4 Conclusion

There are various other prefetching algorithms that we saw above which track the access stream of workload and choose prefetch option. But with every improvement the architect will be faced with the same problem of choosing the heuristic. This thesis presents an approach which eliminates the need to fix heuristics. It lets the system decide for itself which prefetching options to choose.

Chapter 3 Experimental Setup

In the previous chapter we discussed few of the latest prefetching algorithms. It is difficult to prove the performance impact of these prefetchers on actual hardware. We require sophisticated software tools to do performance prediction for these prefetching algorithms. There are various simulators like Rsim, Simics, Simple scalar, Asim etcetera which could be considered for performance prediction. I am using Simics: a full system simulator [11] for performance prediction. Simics runs unmodified firmware, operating system kernels, and device drivers.

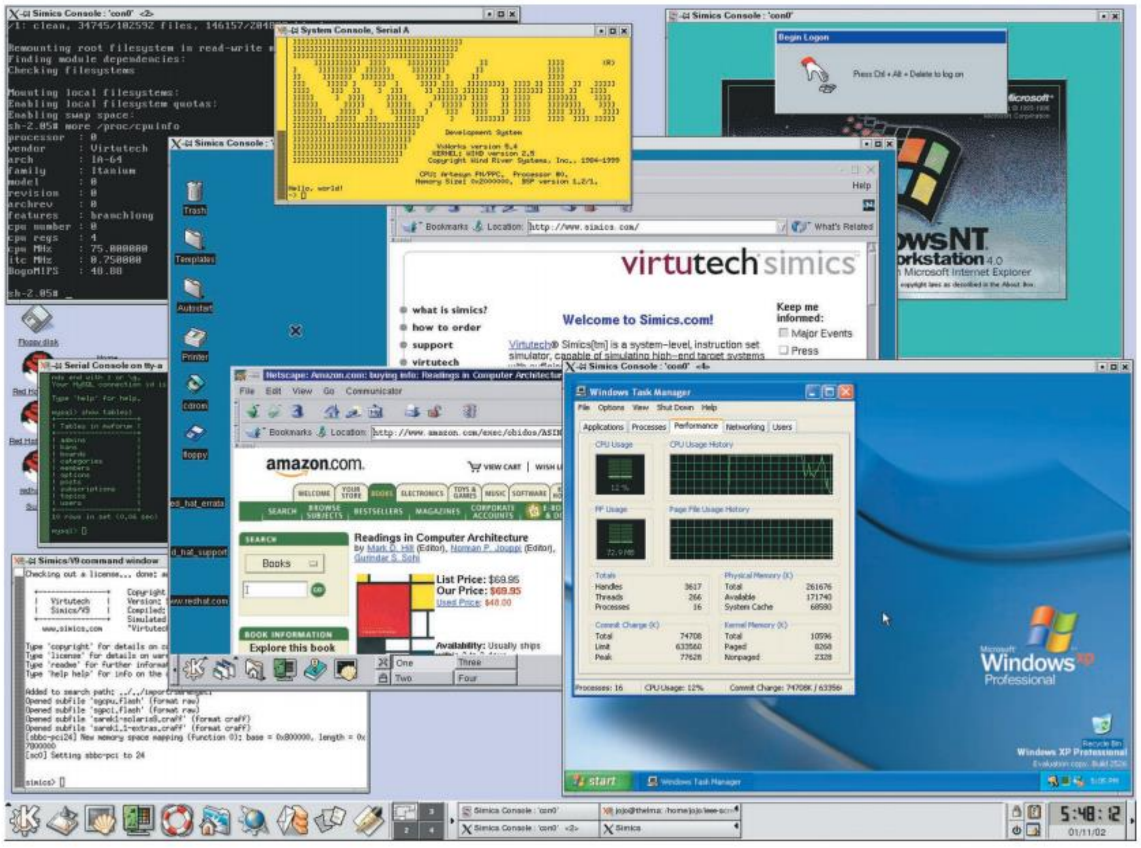
3.1 Simics: A full system simulation Platform [11]

Simics tries to strike balance between accuracy and performance. It is designed to run unmodified operating systems like linux, solarsis, and windows XP etcetera. With this feature we can bring in workloads inside the Simics environment and observe the performance of these workloads on the underlying processor model integrated inside Simics. Simics simulates processor at instruction-set level, including the full supervisor state [11]. Currently Simics supports processor models for UltraSparc, Alpha, x86, x86-64(hammer), PowerPC, IPF, MIPS and ARM [11]. Simics has no impact on the target software but the user has opportunity to modify the underlying hardware modules to test the performance of target systems on them. Figure 3-1 shows how it simulates various target systems based on different processor architectures. The architecture considered for the thesis is Nehalam x-86 IA processor model.

One of the most important features of Simics is its determinism and repeatability. This deterministic behavior is achieved by the use of checkpoints which is explained in section 3.4.1. Simics supports various APIs which makes Simics favorable amongst various computer architects. Some of the features supported by Simics are listed below.

- Fast CPU models: Memory stalling, Cache analysis, instruction trace output for post processing. Simics is modeled at instruction set level [12]. Each instruction is atomic and takes one cycle for execution.

- Simulation Infrastructure: Simics supports breakpoints and checkpoints, scripting, trace etcetera [12].



```
/1: clean, 34745/102592 files, 146157/2048

Remounting root filesystem in read-write m
Finding module dependencies:
Checking filesystems

Mounting local filesystems:
Enabling local filesystem quotas:
Enabling swap space:
sh-2.05# more /proc/cpuinfo
processor  : 0
vendor     : Virtutech
arch       : IA-64
family     : Itanium
model      : 0
revision   : 0
archrev    : 0
features   : branchlong
cpu number : 0
cpu regs   : 4
cpu MHz    : 75.000000
itc MHz    : 0.750000
BogoMIPS   : 40.00

sh-2.05# _
```

```
Checking out a license... done:

  +--------------+     Copyright
  |  Virtutech   |     Version:
  |  Simics/V9   |     Compiled:
  +--------------+     Simulated
   www.simics.com      "Virtutec

Type 'copyright' for details on c
Type 'license' for details on war
Type 'readme' for further informa
Type 'help help' for info on the

Added to search path: ../../import
Opened subfile 'sgcpu.flash' (format raw)
Opened subfile 'sgpci.flash' (format raw)
Opened subfile 'sarek1-solaris8.craff' (format craff)
Opened subfile 'sarek1.1-extras.craff' (format craff)
[sbbc-pci24] New memory space mapping (function 0): base = 0x800000, length = 0x
7800000
[sc0] Setting sbbc-pci to 24


simics> []
```

Figure 3-1 Simics simulation of target systems based on several processors architecture [11]

3.2 Cache behavior modelling using Simics

To evaluate the performance of prefetchers we need cache module to simulate the prefetching algorithm. Default workspace generated by Simics does not provide cache module. It uses its own memory system to obtain high speed simulation. We need to install cache modules into our workspace to model cache behavior. Simics comes with cache profiling and cache timing feature. It supports g-cache which is the standard cache model [14]. It handles one transaction at a time and all operations are performed in an in-order fashion. The cache returns the sum of all the stall times reported for each cache level.

Simics 4.6 version was used as the simulation platform. It supports in order execution. Once this cache module is integrated all the memory requests issued by the processor will go through g-cache. The cache descriptions files also known as the configuration files for cache help us to define each cache object. These files specify some of the important cache parameters. Default cache description file is listed in the figure 3-2 as shown below.

```
cache = pre_conf_object('cache', 'g-cache',
                        config_line_number = 256,
                        config_line_size = 32,
                        config_assoc = 1,
                        config_virtual_index = 0,
                        config_virtual_tag = 0,
                        config_replacement_policy = 'random',
                        penalty_read = 0,
                        penalty_write = 0,
                        penalty_read_next = 0,
                        penalty_write_next = 0,
                        cpus = cpu)
```

Figure 3-2 Cache object description [14]

Using these cache objects we can instantiate entire cache hierarchy. Figure 3-3 describes the cache hierarchy. It specifies sizes of each cache along with the latency values associated with each cache level. Simics module does not support memory module. Inside trans-staller we can build DRAM structure. For my thesis I have used trans-staller to service memory request instead of DRAM. Due to trans-staller all the memory requests will have same latency value. Advantage of this implementation is memory design gets simplified with a disadvantage that we cannot model memory behavior.

The functionality of Simics can be extended by user-written modules. We can introduce models like replacement policy, prefetcher etcetera in the cache module. After implementing the user-written modules we can integrate them here with cache. I have introduced prefetching feature in g-cache.

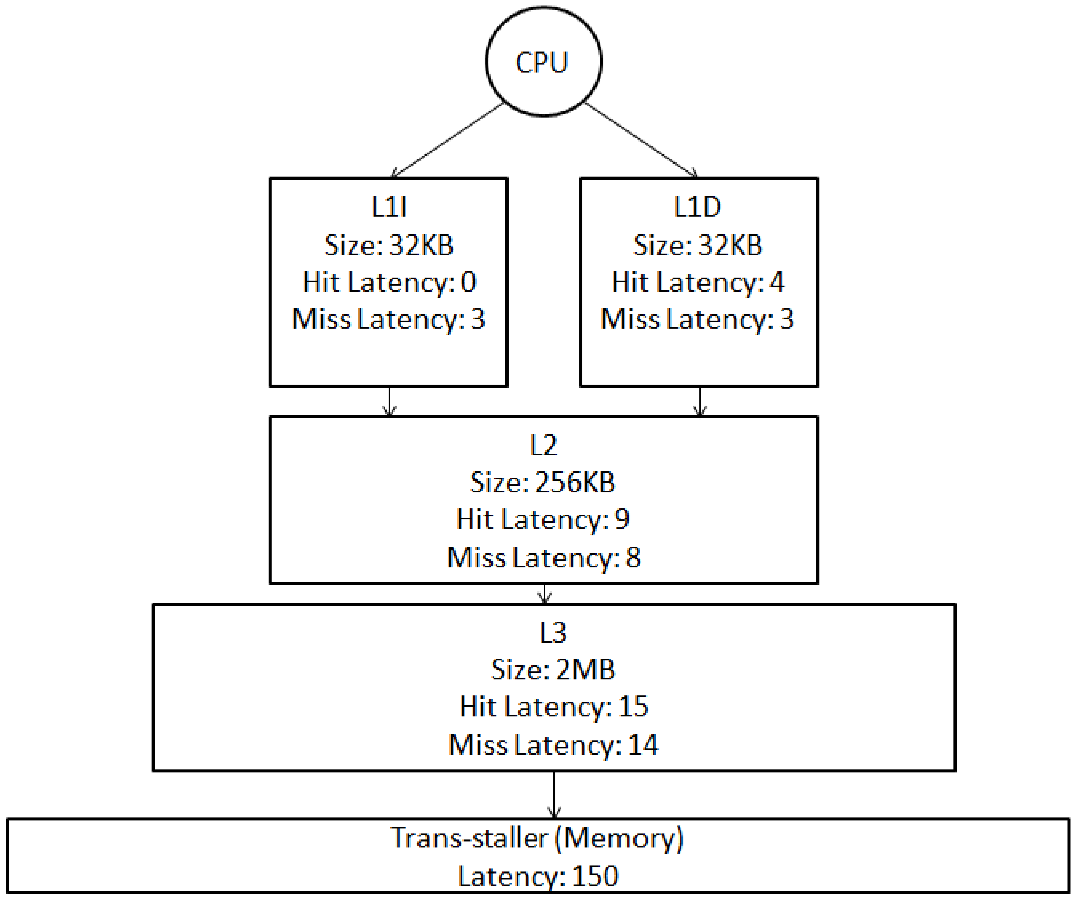


Figure 3-3 Cache hierarchy simulated in Simics

3.3 Prefetching Feature

The prefetching module can be attached to any cache level. A stream prefetcher is usually implemented on mid-level cache so that it can track the miss pattern. Aggressive prefetchers are usually attached to last level cache due to their inaccurate behavior. Last level cache have high capacity and hence inaccuracy due to aggressive prefetchers can be tolerated. The prefetching module is attached to mid-level (L2) cache. It is not compulsory to attach prefetcher module on L2. Prefetch activity is triggered by L2 cache miss. The number of lines prefetched into L2 and the offset of the prefetched line depends on the heuristics of the prefetch algorithm. The implementation of prefetch algorithm is described in Chapter 4.

3.4 Moving workloads to Simics

To characterize the performance of underlying hardware we need to test it on various workloads. With Simics we can simulate the target system (unmodified binaries of operating system) and create checkpoints. The target system will be running the workload. A checkpoint has set of files which save the complete state of simulation. Thus for the target system at time ‘A’, if we create a checkpoint then, when the checkpoint is loaded the target system will start from the same time instant A. Thus to test the performance of the underlying processor model we need checkpoints of various workloads. To create these checkpoints of various workloads we first load the target system using Simics. Next we move workloads into the target system. After that we run the workloads and bring them to steady state. Once the workload reaches steady state we can create checkpoints.

3.4.1 Creating Images and Checkpoints of target system

The first step is to create the target system or load the target system. The target system could be an ubuntu system or red hat system or any other operating system. To move workloads into target system we first need to boot up the target system using Simics. Simics supports machine scripts which loads and configures the target system. In these scripts we can specify the location of the disk image which can be an iso image or craff image. We can also specify machine configuration like the number of processors, memory capacity etcetera. Second step is to connect the target system to the host computer. The host computer already has the benchmark downloaded. Once the connection between them is established transfer the benchmark files. On the target system we can now run the workload. Once the steady state is achieved we can create checkpoint point. After checkpoint is saved we can use this checkpoint to test the processor model.

3.5 Workloads

Checkpoints used for performance analysis belong to SPECcpu2006, SPECjvm2008 and SPECjbb2005 benchmarks. I will be analyzing the performance impact of this new prefetching algorithm on these workloads.

3.5.1 SPEC CPU2006

CPU2006 is CPU intensive benchmark suite developed by SPEC. The workloads in SPECcpu2006 suite are CPU-intensive, stressing on processor's memory subsystem and compiler [15]. The table below lists the workloads that I have used for my analysis.

Table 3-1 Workloads run under SPEC CPU2006 [17]

Workload	**SPECint/ SPECfp**	**Workload Description**
Mcf	SPECint	Combinatorial Optimization
Libquantum	SPECint	Physics: Quantum Computing
Omnetpp	SPECint	Discrete Event Simulation
Xalancbmk	SPECint	XML processing
Milc	SPECfp	Physics: Quantum Chromodynamics
Zeusmp	SPECfp	Physics / CFD
cactusADM	SPECfp	Physics / General Relativity
Soplex	SPECfp	Linear Programming Optimization
GemsFDTD	SPECfp	Computational Electromagnetics
Lbm	SPECfp	Fluid Dynamics
Sphinx3	SPECfp	Speech Recognition

3.5.2 SPEC JVM2008

SPECjvm2008 benchmark suite was designed to measure the performance of JRE (Java Runtime Environment. The workloads execute single application which focuses on measuring performance of hardware processor and memory subsystem [16]. The table below lists the workloads considered and the real workload applications they try to mimic.

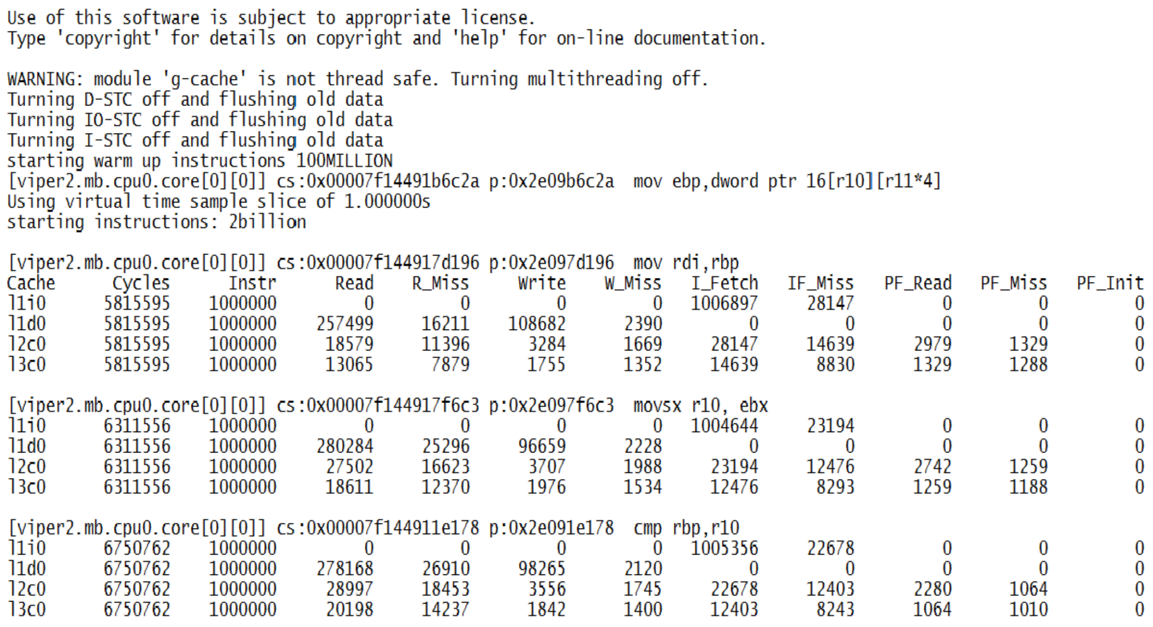
Table 3-2 Workloads run under SPEC JVM2008 [18]

Workload	Workload Description
Compiler	Use openJDK front end compiler to compile a set of .java files
Compress	Data compression using Lempel-Ziv method
Crypto-aes	Encrypt and decrypt using AES and DES protocols
Crypto-rsa	Encrypt and decrypt using RSA protocol
Cyrpto-signverify	Sign and verify using different protocols
Derby	BigDecimal computation
MPEGaudio	It is floating point heavy and good test for mp3 decoding
Scimark	It has various subtests (fft, lu, sor, sparse) with two versions large and small
Serial	Serializes and deserializes primitives and objects using data from JBOSS
Sunflow	Tests graphics visualization
XML-transform	Exercises JRE's implement of javax.xml.transform and associated APIs by applying style sheets to XML documents
XML-validation	Excercises JRE's implementation of javax.xml.validation and associated APIs by validating XML instance documents

3.6 Working with Simics for data collection.

Once we have everything set up: prefetching algorithm integrated inside g-cache, cache description files, and checkpoints for different workloads we can move towards data collection. The output from the simulator looks like figure 3-4.

Along with prefetcher there are other performance counters integrated in the cache module. Figure 3-4 lists all the performance counters. Read, write and I_Fetch counters count the total number of read request, write request and the instruction fetch request seen by the particular cache level. Cycles and Instr reports the total number of cycles taken for an interval and instr reports the total number of instructions executed during the interval. These counters are collected on interval basis. The interval selected is 1 Million instructions. Each workload runs for total of 2 Billion instructions



```
Use of this software is subject to appropriate license.
Type 'copyright' for details on copyright and 'help' for on-line documentation.

WARNING: module 'g-cache' is not thread safe. Turning multithreading off.
Turning D-STC off and flushing old data
Turning IO-STC off and flushing old data
Turning I-STC off and flushing old data
starting warm up instructions 100MILLION
[viper2.mb.cpu0.core[0][0]] cs:0x00007f14491b6c2a p:0x2e09b6c2a  mov ebp,dword ptr 16[r10][r11*4]
Using virtual time sample slice of 1.000000s
starting instructions: 2billion

[viper2.mb.cpu0.core[0][0]] cs:0x00007f144917d196 p:0x2e097d196  mov rdi,rbp
Cache       Cycles      Instr      Read    R_Miss     Write    W_Miss   I_Fetch   IF_Miss   PF_Read   PF_Miss   PF_Init
l1i0       5815595    1000000         0         0         0         0   1006897     28147         0         0         0
l1d0       5815595    1000000    257499     16211    108682      2390         0         0         0         0         0
l2c0       5815595    1000000     18579     11396      3284      1669     28147     14639      2979      1329         0
l3c0       5815595    1000000     13065      7879      1755      1352     14639      8830      1329      1288         0

[viper2.mb.cpu0.core[0][0]] cs:0x00007f144917f6c3 p:0x2e097f6c3  movsx r10, ebx
l1i0       6311556    1000000         0         0         0         0   1004644     23194         0         0         0
l1d0       6311556    1000000    280284     25296     96659      2228         0         0         0         0         0
l2c0       6311556    1000000     27502     16623      3707      1988     23194     12476      2742      1259         0
l3c0       6311556    1000000     18611     12370      1976      1534     12476      8293      1259      1188         0

[viper2.mb.cpu0.core[0][0]] cs:0x00007f144911e178 p:0x2e091e178  cmp rbp,r10
l1i0       6750762    1000000         0         0         0         0   1005356     22678         0         0         0
l1d0       6750762    1000000    278168     26910     98265      2120         0         0         0         0         0
l2c0       6750762    1000000     28997     18453      3556      1745     22678     12403      2280      1064         0
l3c0       6750762    1000000     20198     14237      1842      1400     12403      8243      1064      1010         0
```

Figure 3-4 Output from simulator

3.7 Conclusion

With checkpoints of different workloads, prefetcher algorithm implemented in g-cache and the machine script, data is collected for different workloads. The next chapter will talk about how we can play with the data and build prediction models which will then help us achieve the adaptive feature in prefetching algorithm.

Chapter 4 Proposed Approach

We need a dynamic prefetching technique to select the heuristic for a prefetcher without the use of static threshold. With every step that we take forward the prefetchers are going to be more and more complex. Also the workloads will keep on changing and evolving. Use of static thresholds might limit the performance improvement in long run. Hence we need dynamically adjusting values to tune the prefetching schemes. There are various ways in which we can achieve this dynamic behavior. One way to do it is, to train the system to make decision about prefetching options. Machine learning has proved to be effective in many fields. With machine learning once the system is trained, it can make decisions for itself. Cache performance is very susceptible to the prefetching options selected; hence we can use this correlation to predict the prefetching options.

Machine learning is a statistical method which makes the system smarter and gives it capability to make decision. The smart engine that makes decision is called model. There are two approaches that can be taken to make the system smarter.

1) Throw in bunch of data and the system will group data with similar behavior. This is known as unsupervised learning.
2) We can teach the system by feeding it with labelled dataset and telling it that for x, y, z attributes the response seen is ‘A’. This is known as supervised learning.

For our problem we need to teach the system to pick up correct prefetching options by looking at cache performance data. Hence the approach used is supervised learning. With large enough dataset we can randomly divide it into training and test. With the training dataset we can teach the system to pick up correct response. The test dataset is then used to test the system’s ability to make correct decision.

There are various supervised learning algorithm which will work great, but choosing the algorithm is a tough decision as you need to take into account the dataset. The response that needs to be predicted is prefetch option that takes either ‘0’ or ‘1’ value. That is why the algorithms considered are classification algorithms.

4.1 Logistic Regression

Logistic regression assumes nonlinear relationship between inputs and output. Logistic regression estimates the probability of the response variable.

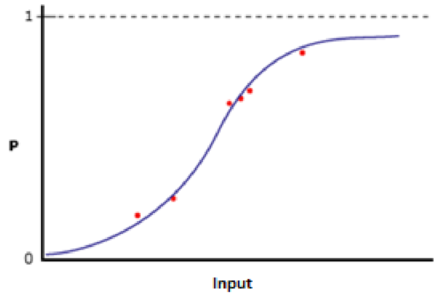


Figure 4-1 Graph specifying relation between input variable and output [19]

Logistic regression uses the maximum likelihood approach to estimate model parameters. With maximum likelihood approach the algorithm will give us the probability of the observed zeros and ones in the data set. Logistic regression just like linear regression supports multiple attributes and we can use interaction between the attributes to improve the prediction accuracy of the model.

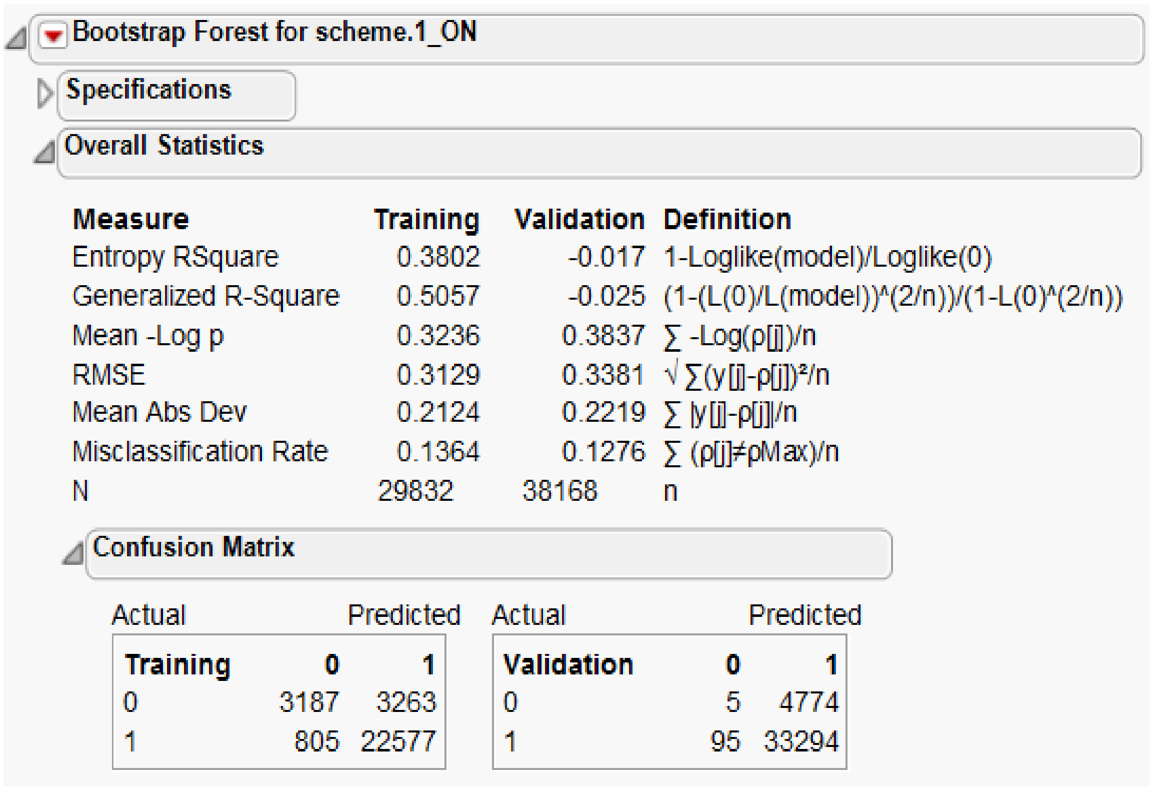
Logistic regression uses one model to predict the response. With cross validation we can increase our confidence in the model. Cross validation is nothing but creating new splits in the dataset to select multiple training, validation and test set so that we are more confident about our model. Logistic regression works very well for some datasets. But sometime we need more accurate models. Cache statistics are very susceptible to prefetch option and change in one prefetching option can modify the performance drastically which might look like a random behavior to the model. Hence building just one model to predict the prefetching option might not be an optimal solution.

4.2 Bootstrap Forest

This classification approach uses multiple models to predict the response. The word ‘forest’ in Bootstrap Forest algorithm suggests that it has multiple models. This algorithm creates multiple trees forming a forest. These trees are decision trees. Each decision tree is a model which makes a prediction. This algorithm creates multiple models (decision trees) that learn from the subset of the learning dataset. Each model is given a smaller portion of the learning dataset. Finally when all the models learn from the datasets given to them we can then test the working of this forest. The validation data set is used to test the ability of the algorithm to make correct decision. With validation dataset each tree will take one sample from it and make a prediction. Once all the trees are done making prediction for that sample the algorithms takes a vote or averages the prediction make by all the trees. Thus in this algorithm with multiple models making the decision we get a pretty good overall model.

4.3 Bootstrap Forest vs. Logistic Regression

Prediction models need to accurate enough so that they can predict the prefetch options. For building these models we need to consider distribution of dataset and the algorithm implemented by the prediction model. For some datasets simple models work whereas for others we need complex models to get accurate results. Logistic regression as described in section 4.1 uses a single model to make prediction but Bootstrap model as described in section 4.2 uses multiple models to make prediction. To test the accuracy of model we can use confusion matrix. Confusion matrix gives us a prediction table that tells how the model classifies the samples. It gives us the misclassification ratio. Misclassification ratio will tell us the percentage of samples those were wrongly classified. Misclassification ratio should be low for accurate models. Figure 4.3 and figure 4.4 shows the confusion matrix for Logistic regression and Bootstrap forest algorithm. Based on the misclassification ratio I decided to go with Bootstrap forest approach.


Bootstrap Forest for scheme.1_ON

Specifications

Overall Statistics

Measure	Training	Validation	Definition
Entropy RSquare	0.3802	-0.017	1-Loglike(model)/Loglike(0)
Generalized R-Square	0.5057	-0.025	(1-(L(0)/L(model))^(2/n))/(1-L(0)^(2/n))
Mean -Log p	0.3236	0.3837	∑ -Log(ρ[j])/n
RMSE	0.3129	0.3381	√ ∑(y[j]-ρ[j])²/n
Mean Abs Dev	0.2124	0.2219	∑ |y[j]-ρ[j]|/n
Misclassification Rate	0.1364	0.1276	∑ (ρ[j]≠ρMax)/n
N	29832	38168	n

Confusion Matrix

Actual / Predicted

Training	0	1
0	3187	3263
1	805	22577

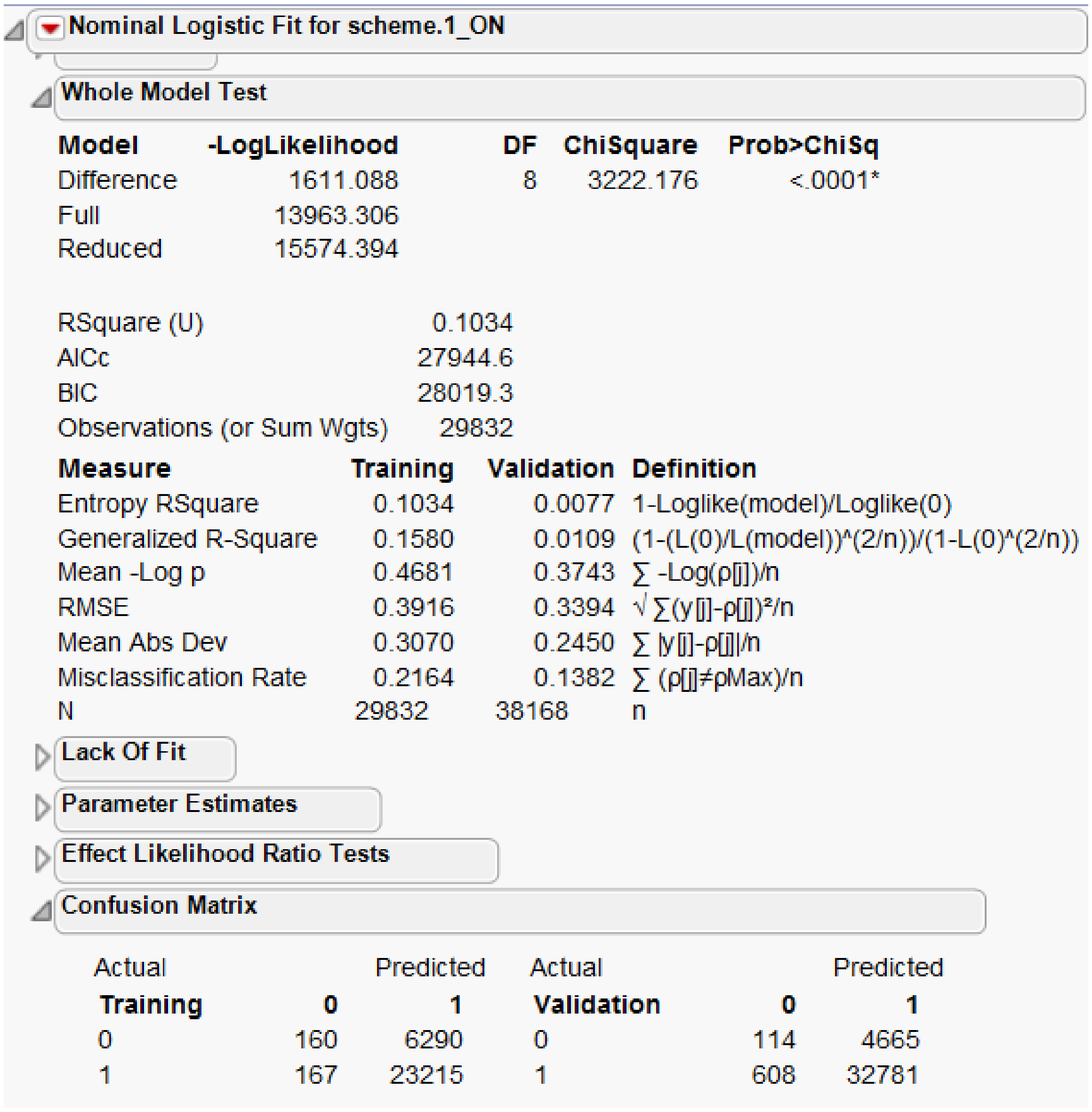
Actual / Predicted

Validation	0	1
0	5	4774
1	95	33294

Figure 4-2 Model specification for Bootstrap forest

4.4 Intuitively why will machine learning approach work

With machine learning we will teach the system to decide which prefetch option to choose. The dataset used to train the model is generalized. Generalized dataset comprises samples from all possible workloads. If the dataset has seen some samples of all different kinds of workload then our dataset is generalized. The dataset collected tries to look at as many workloads as possible. Each sample contains cache statistics along with prefetch options selected. With this generalized dataset we can train our model which can then make prefetch prediction. Once we train our model with this dataset it will have seen some samples of different workloads. Thus at run-time this model will monitor the cache performance counters on interval basis and by looking at the cache statistics it can predict prefetch options. This approach thus eliminates the need for static approach and gives us adaptive tuning of the prefetching options.


Nominal Logistic Fit for scheme.1_ON

Whole Model Test

Model	-LogLikelihood	DF	ChiSquare	Prob>ChiSq
Difference	1611.088	8	3222.176	<.0001*
Full	13963.306			
Reduced	15574.394			

RSquare (U)	0.1034
AICc	27944.6
BIC	28019.3
Observations (or Sum Wgts)	29832

Measure	Training	Validation	Definition
Entropy RSquare	0.1034	0.0077	1-Loglike(model)/Loglike(0)
Generalized R-Square	0.1580	0.0109	(1-(L(0)/L(model))^(2/n))/(1-L(0)^(2/n))
Mean -Log p	0.4681	0.3743	∑ -Log(ρ[j])/n
RMSE	0.3916	0.3394	√ ∑(y[j]-ρ[j])²/n
Mean Abs Dev	0.3070	0.2450	∑ |y[j]-ρ[j]|/n
Misclassification Rate	0.2164	0.1382	∑ (ρ[j]≠ρMax)/n
N	29832	38168	n

Lack Of Fit

Parameter Estimates

Effect Likelihood Ratio Tests

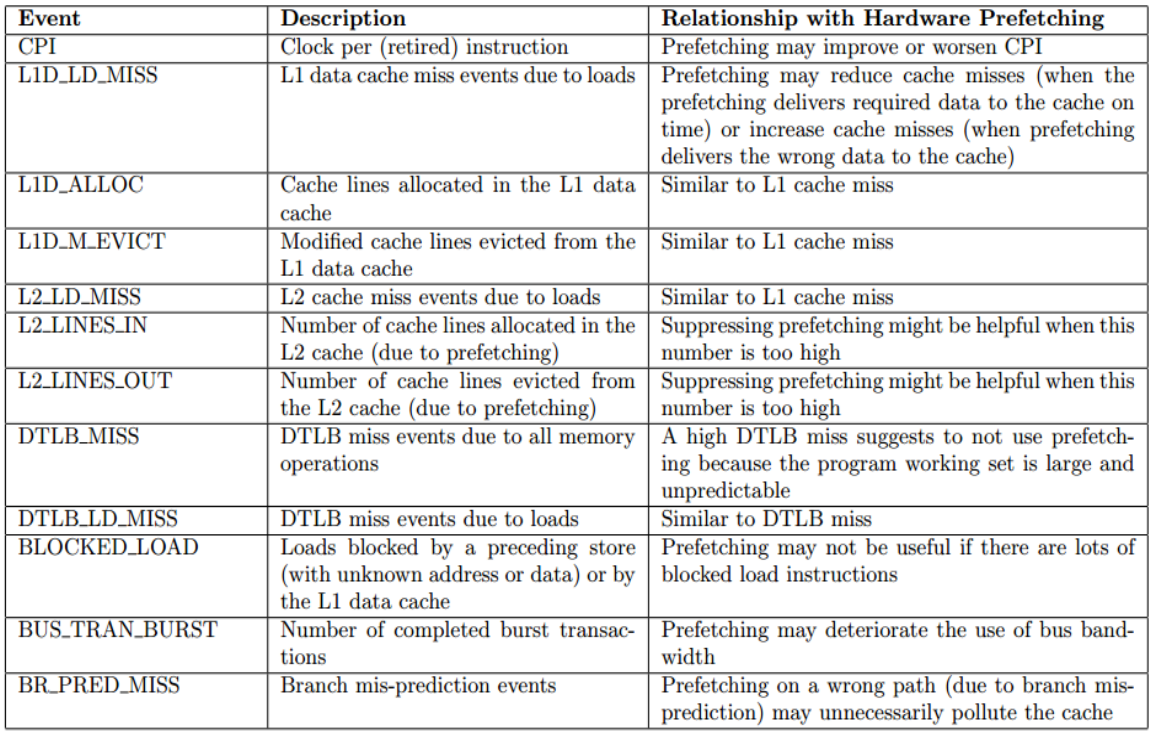
Confusion Matrix

Actual	Predicted		Actual	Predicted	
Training	0	1	Validation	0	1
0	160	6290	0	114	4665
1	167	23215	1	608	32781

Figure 4-3 Model specification for Logistic Regression

4.4.1 Previous work done in this field

Paper [20] talks about how to use machine learning to predict near optimal prefetch configuration out of the four BIOS prefetch option. This paper focuses on selecting one of the best prefetch configurations out of the available prefetchers: Data Prefetch Logic, Adjacent Cache Line, Data Cache Unit, and Instruction Pointer based prefetcher. These prefetchers have effect on various performance counters of cache which is specified in table 2. This paper uses various machine learning techniques to map these counters to the prefetchers selected. This framework achieves performance improvement within 1% of the best configuration [20].



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

Figure 4-4 Counters affected by prefetcher

4.5 Conclusion

This chapter talks about how machine learning can be used to predict prefetch options. Next two chapters will talk about the implementation of this adaptive approach and the results.

Chapter 5 Construction of Dataset

In the previous chapter we discussed different machine learning algorithms that can be used for prediction. Bootstrap forest algorithm was selected as the machine learning algorithm due to its low misclassification ratio. This chapter will cover data collection, data processing after collecting data from Simics and building prediction models using the dataset.

5.1 Data collection using Simics

We want to predict prefetch options by looking at cache counters. For that we need to inject different prefetch options and monitor the cache performance counters. The most challenging task here is injecting the prefetch options. Before injecting the prefetch options we need to decide the prefetching options.

5.1.1 Prefetch options

Prefetch option is a combination of prefetch degree and prefetch stride. We need to look at various prefetch options. Table 5-1 shows different prefetch options those are being evaluated.

Prefetch degree: It is number of lines the prefetching scheme is going to fetch. Prefetch degree of 1 means it will prefetch single cache line. Prefetch degree of 4 means it will prefetch 4 cache lines.

Prefetch stride: It is offset which gets added to the address which results in demand miss. So if the program wants to bring cache line A due to demand miss then prefetch stride of 1 will prefetch line A+1. A prefetch stride of 2 will prefetch address A+2

Table 5-1 Different Prefetch options

Sr. No	Pf degree	Pf stride
1	1	1
2	1	2
3	1	4
4	1	8
5	1	-1
6	1	-2
7	1	-4
8	1	-8
9	2	1

10	4	1
11	8	1
12	2	-1
13	4	-1
14	8	-1
15	64	1
16	64	2
17	64	4
18	64	8
19	64	-1
20	64	-2
21	64	-4
22	64	-8

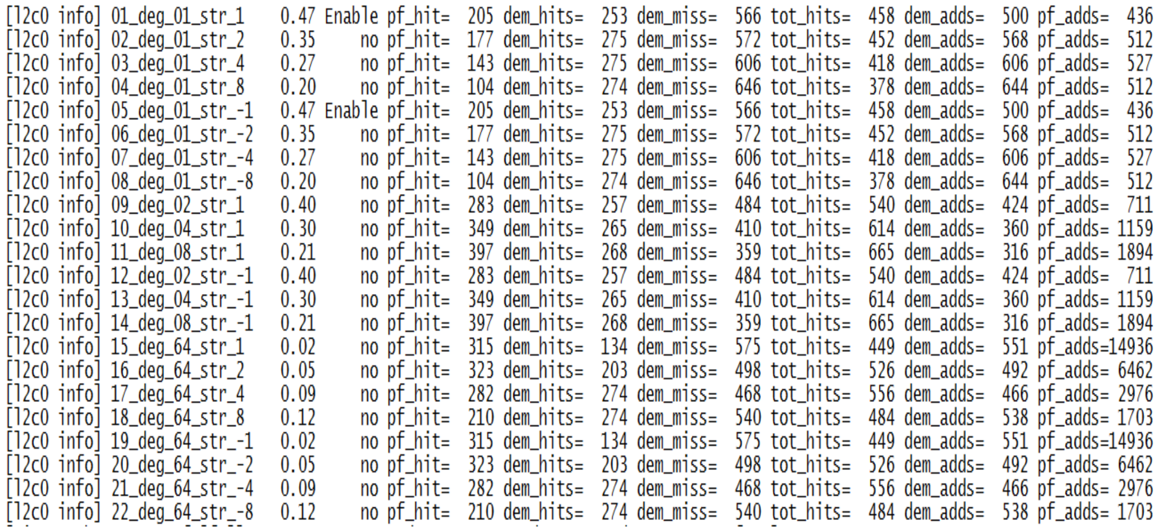
We can inject prefetch options on interval basis. For each interval we inject prefetch options let it run for 1 Million instruction and then collect the cache statistics. But deciding which prefetch options to apply is a difficult task. Unless all the possible combination of prefetch options are tested we cannot say which prefetch options will benefit the workload. 22 different prefetch options are considered. Therefore considering all combination 22^{22} is not feasible. One of the smart ways to inject prefetch options is by using Sandbox prefetcher [23]. Sandbox prefetcher evaluates different prefetch options integrated within it and selects the prefetcher with good enough accuracy.

The selection criteria for choosing prefetching options depend on the score value. By changing this score value we can control the selection of prefetch options. The next section will talk about how we use sandbox to inject prefetch options.

5.1.2 Sandbox Prefetcher

Sandbox prefetcher evaluates all the schemes integrated within it safely without polluting the contents of cache. Sandbox module keeps track of all the memory references made to cache level on which it is attached. Inside sandbox we deal with addresses and not the actual data content. Sandbox module has an array structure which records all the addresses requested by the program as well as the addresses prefetched based on the evaluation of prefetching schemes within sandbox. Sandbox keeps evaluating these prefetching schemes in the background, while the actual prefetching happening. Evaluation window is given to sandbox and after the evaluation window is finished sandbox gives us the performance of all the prefetching schemes integrated within it which helps us make decision at run-time. This evaluation window is kept smaller so that we can capture the changes in the workload behavior. For every interval sandbox will be evaluating all the prefetching schemes. After evaluation is done, in the next interval we will turn on the prefetching schemes which Sandbox module has selected. For the next interval sandbox module will be cleared and reset so that it will again keep tracking the memory accesses and the performance of the prefetching. Thus sandbox module evaluates the prefetching schemes by looking at the access pattern of the workload to make decision at run-time. The assumption made here is, evaluation window is small enough to capture changes in access pattern of the workload and the access pattern will not change drastically. Table 5-2 shows the evaluation statistics. The score value is nothing but the prefetch accuracy. This score value is shown immediately next to the prefetching scheme. Based on the threshold value (0.4 in this case) the appropriate prefetch options are turned on (in this case scheme 1 and scheme 5).

Table 5-2 Evaluation statistics



```
[l2c0 info] 01_deg_01_str_1    0.47 Enable pf_hit=  205 dem_hits=  253 dem_miss=  566 tot_hits=  458 dem_adds=  500 pf_adds=  436
[l2c0 info] 02_deg_01_str_2    0.35     no pf_hit=  177 dem_hits=  275 dem_miss=  572 tot_hits=  452 dem_adds=  568 pf_adds=  512
[l2c0 info] 03_deg_01_str_4    0.27     no pf_hit=  143 dem_hits=  275 dem_miss=  606 tot_hits=  418 dem_adds=  606 pf_adds=  527
[l2c0 info] 04_deg_01_str_8    0.20     no pf_hit=  104 dem_hits=  274 dem_miss=  646 tot_hits=  378 dem_adds=  644 pf_adds=  512
[l2c0 info] 05_deg_01_str_-1   0.47 Enable pf_hit=  205 dem_hits=  253 dem_miss=  566 tot_hits=  458 dem_adds=  500 pf_adds=  436
[l2c0 info] 06_deg_01_str_-2   0.35     no pf_hit=  177 dem_hits=  275 dem_miss=  572 tot_hits=  452 dem_adds=  568 pf_adds=  512
[l2c0 info] 07_deg_01_str_-4   0.27     no pf_hit=  143 dem_hits=  275 dem_miss=  606 tot_hits=  418 dem_adds=  606 pf_adds=  527
[l2c0 info] 08_deg_01_str_-8   0.20     no pf_hit=  104 dem_hits=  274 dem_miss=  646 tot_hits=  378 dem_adds=  644 pf_adds=  512
[l2c0 info] 09_deg_02_str_1    0.40     no pf_hit=  283 dem_hits=  257 dem_miss=  484 tot_hits=  540 dem_adds=  424 pf_adds=  711
[l2c0 info] 10_deg_04_str_1    0.30     no pf_hit=  349 dem_hits=  265 dem_miss=  410 tot_hits=  614 dem_adds=  360 pf_adds= 1159
[l2c0 info] 11_deg_08_str_1    0.21     no pf_hit=  397 dem_hits=  268 dem_miss=  359 tot_hits=  665 dem_adds=  316 pf_adds= 1894
[l2c0 info] 12_deg_02_str_-1   0.40     no pf_hit=  283 dem_hits=  257 dem_miss=  484 tot_hits=  540 dem_adds=  424 pf_adds=  711
[l2c0 info] 13_deg_04_str_-1   0.30     no pf_hit=  349 dem_hits=  265 dem_miss=  410 tot_hits=  614 dem_adds=  360 pf_adds= 1159
[l2c0 info] 14_deg_08_str_-1   0.21     no pf_hit=  397 dem_hits=  268 dem_miss=  359 tot_hits=  665 dem_adds=  316 pf_adds= 1894
[l2c0 info] 15_deg_64_str_1    0.02     no pf_hit=  315 dem_hits=  134 dem_miss=  575 tot_hits=  449 dem_adds=  551 pf_adds=14936
[l2c0 info] 16_deg_64_str_2    0.05     no pf_hit=  323 dem_hits=  203 dem_miss=  498 tot_hits=  526 dem_adds=  492 pf_adds= 6462
[l2c0 info] 17_deg_64_str_4    0.09     no pf_hit=  282 dem_hits=  274 dem_miss=  468 tot_hits=  556 dem_adds=  466 pf_adds= 2976
[l2c0 info] 18_deg_64_str_8    0.12     no pf_hit=  210 dem_hits=  274 dem_miss=  540 tot_hits=  484 dem_adds=  538 pf_adds= 1703
[l2c0 info] 19_deg_64_str_-1   0.02     no pf_hit=  315 dem_hits=  134 dem_miss=  575 tot_hits=  449 dem_adds=  551 pf_adds=14936
[l2c0 info] 20_deg_64_str_-2   0.05     no pf_hit=  323 dem_hits=  203 dem_miss=  498 tot_hits=  526 dem_adds=  492 pf_adds= 6462
[l2c0 info] 21_deg_64_str_-4   0.09     no pf_hit=  282 dem_hits=  274 dem_miss=  468 tot_hits=  556 dem_adds=  466 pf_adds= 2976
[l2c0 info] 22_deg_64_str_-8   0.12     no pf_hit=  210 dem_hits=  274 dem_miss=  540 tot_hits=  484 dem_adds=  538 pf_adds= 1703
```

5.1.3 Working of Sandbox module in Simics

Sandbox prefetching module is attached to L2 cache level. It will track all the memory requests made to L2. Whenever a program makes a request it is first processed by L1 cache. If the address is not in L1 the request is forwarded to L2 cache. Before the request is serviced by L2 cache that address is put in the sandbox module. Inside sandbox module, that address is recorded in an array. The prefetch/demand field for that address is marked as demand field i.e. 0 as it was a demand request. For that address depending on the prefetching scheme under evaluation, prefetched address is calculated and stored. For the prefetched address we mark the field as prefetch field i.e. 1. We do this for all prefetching schemes until we finish the evaluation duration. Along with this prefetch/demand field we have bunch of other counters shown in the table 5-2. Each counter is described below.

1. Prefetch add: This counter keeps records of how many addresses were put in the sandbox module as a result of prefetching.
2. Prefetch hits: This counter keeps track of number of addresses used out of the addresses put in the sandbox because of prefetch.
3. Demand hits: This counter keeps track of number of addresses used out of the addresses put in sandbox because of demand request.
4. Demand add: It records number of addresses put in the sandbox module as a result of demand request.

Once the evaluation is completed score/prefetch accuracy for each prefetch option is calculated. Once this evaluation phase is over we need to select prefetch option. As we cannot evaluate all the 2^{22} combinations we randomly inject prefetching options

5.1.4 Injecting Prefetching Options

A threshold value can be used for turning on prefetching options at random. For a higher threshold value we will be turning on conservative prefetching options, but for a lower threshold value we will be turning on aggressive prefetching options. Hence random selection of threshold will give us more coverage. This random selection of threshold is done on interval basis. Therefore for every interval we select a new threshold value randomly. As each workload is run for total of 2 Billion instructions per workload we get around 2000 samples. Total we are considering 31 different workloads therefore we have a total of 62000 samples. This gives us a reasonable coverage.

With this dataset we can now start moving towards building prediction models. The first step towards building the prediction model is to bring the data in correct format so that we can apply machine learning techniques. The next section will talk about how to bring data in correct format.

5.2 Bring the data in correct format for data analysis

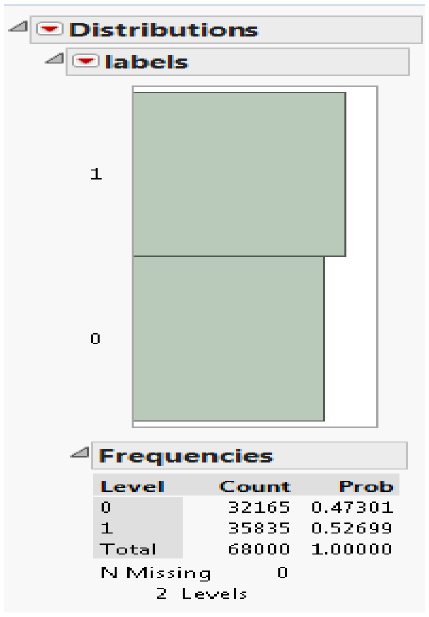
The data collected from Simics comprise of cache objects reporting its statistics after every 1M instructions. The workload is let to run for a total of 2 billion instructions. Based on Hadley Wickham's [21] paper on tidy data we need to rearrange the data collected from Simics. In Simics the output is reported on interval basis. After 1M instructions each cache objects reports its statistics. So for every interval we have 4 rows of data. But based on the tidy data concept each row should represent one observation. Hence instead of having 4 rows with 11 attributes for one observation we need to have one row with 44 attributes for one observation. The first 11 attributes of a row correspond to attributes of L1-I cache 12 to 22 attributes correspond to L1-D cache 23 to 33 correspond to L2 cache and lastly 34 to 44 correspond to L3 cache. But along with cache statistics we need prefetch options also. So after the 44 attributes we concatenate all the 22 prefetch options. These 22 prefetch options are represented in a bit vector format. Each bit corresponds to a prefetch option. A bit value of 1 means that particular prefetch option will be turned on and 0 means it will be turned off. The prediction models built are for each bit. So we have a total of 22 different prediction models predicting their individual prefetch bit. Thus each row comprises of 44 cache attributes and 22 prefetch options.

Once we have data arranged in the format that can be used in machine learning algorithm we have to move on towards labelling the dataset. As we are using supervised learning algorithm this is an important step and our results will be sensitive to labelling the dataset.

5.3 Labelling data samples

Each sample in the dataset comprises of performance counters collected on interval basis. For every sample we choose the threshold value randomly between 0.0 and 0.9. The next step is building the prediction model. For that we first need to label the data. Here the tricky part is we need to predict the prefetch option so our response is the prefetch option for each prefetching scheme. But not all the samples from the dataset have positive impact on the performance. We need to build a model based on the positive samples so that the model will predict prefetching options having positive impact. For building the model if negative samples are used then the model might map some of the cache statistics to prefetching options which might degrade the performance.

Samples are labelled into positive and negative based on the performance impact. To decide if the prefetching option is beneficial or not we use the no prefetch cycle count as the baseline. If the current sample shows 25% improvement over the no prefetch option we have some confidence that the prefetching options selected were beneficial. It is unrealistic to collect no prefetch cycle count for 2000 samples for 2Billion instructions. And then again collect data by randomly selecting threshold. We assume that the workload will not change the phase for 2 billion instructions which is like 5ms real time. Hence we turn off prefetching for the first sample and use that cycle count as the baseline to label samples as positive or negative. After we have labelled data we can use the positive sample to create the prediction model i.e. use it as the training data set and the rest of the samples will be in test set. From the figure 5-1 we can see that 52% of the samples are used for creating the model. These samples are the positive samples. The rest 48% samples are the negative samples which comprise of the test set.



Level	Count	Prob
0	32165	0.47301
1	35835	0.52699
Total	68000	1.00000

```
N Missing      0
    2  Levels
```

Figure 5-1 Distribution of positive and negative samples with threhsold value of 0.25

5.4 Prefetchers used for comparison

To verify how good this adaptive prefetching technique is I have used Sandbox prefetcher [23]. This prefetcher was published in HPCA 2014 and was amongst the best prefetcher paper. I have also compared this approach to a hypothetical prefetcher which eliminates all memory requests. I have called it zero memory latency concept. This hypothetical prefetcher will service all the demand access by fitting the entire data into cache so that we do not need to go to main memory. Comparison with this ideal scenario will let us know how much margin of improvement is still available for the prefetching algorithm.

The next chapter will talk about how to use the dataset to build prediction model. It will walk through the results obtained by building models in different ways.

Chapter 6 Prediction Models and Results

The last chapter provided a brief introduction to building prediction models. Based on the labelling criteria used we have the training and test set. The next step is to select the attributes that will affect the response. There is lot of data dependency among different performance counters. Before building the models we need to take care of dependency issues. Performance counters considered are the read and instruction fetch hit and miss counters at all cache level. A miss at L1 level is propagated to L2 level which then results to either hit or miss in L2 level. The figure 6-1 below explains this phenomenon.

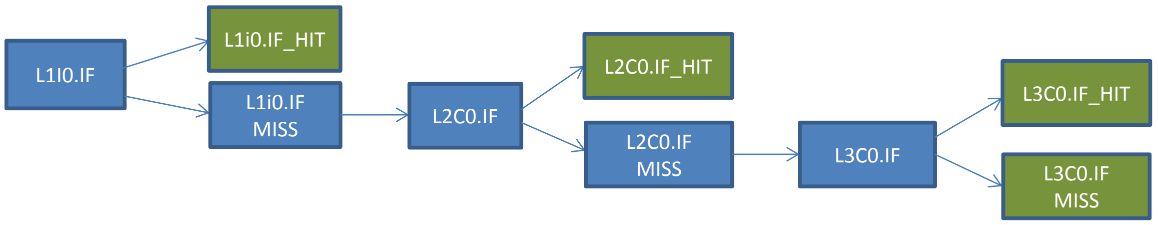


Figure 6-1 Propagation of instruction request in cache

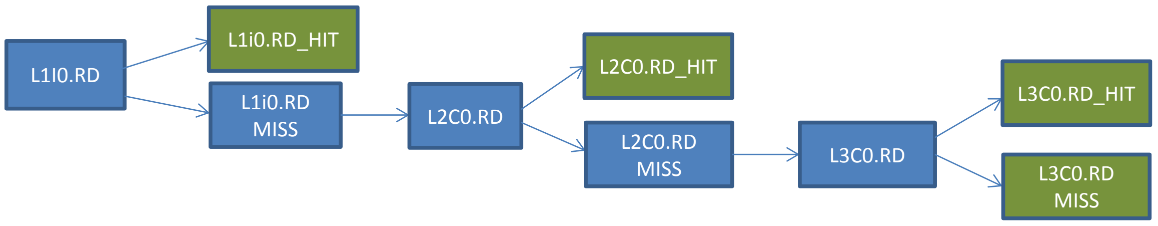


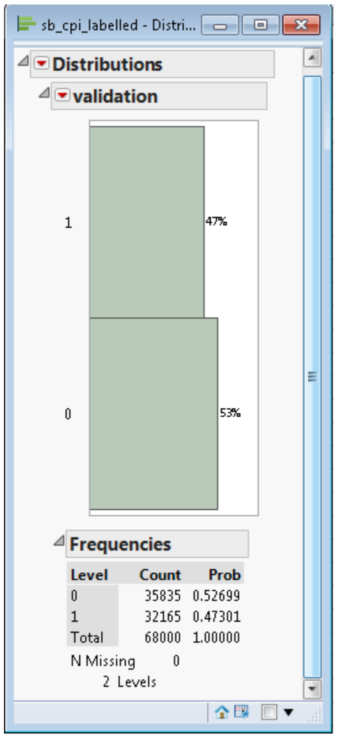
Figure 6-2 Propagation of data request in cache

So for the attributes if we choose only the leaves we will eliminate the singularity and the data dependence problem.

6.1 Building Model-I

This model builds one prediction model per prefetch option. Thus for model-I we will have total of 22 prediction models. The simulation cycle count value is used to label samples as positive or negative. For a sample to be positive its performance for 1M instructions should be 25% better than the baseline. Baseline cycle count is obtained by turning off prefetching at the start of workload. Assumption made in that is the phase of workload will not change by a huge margin and the first cycle will be somewhat representative. Using this baseline cycle and the cycles computed at run-time for every interval the samples are labelled as positive or negative. The leaves of the trees shown in figure 6-1 and 6-2 are used as attributes to create the model. Figure 6-3 shows the distribution and division of training and test set. The samples falling in level 0 used in training. They are the positive samples. From the figure 6-3 53% of the samples are used to build the model. The rest 47% of samples fall in level 1 which is used as test data set. There is no validation set as only 53% of samples are positive.

Table 6-1 talks about the performance of all the prediction models on training data set. The classification algorithm used is the bootstrap method. The misclassification ratio should be low. Based on the values of misclassification rate we have pretty reasonable models.



Level	Count	Prob
0	35835	0.52699
1	32165	0.47301
Total	68000	1.00000

N Missing 0
2 Levels

Figure 6-3 Distribution of training and validation dataset

Using JMP when the models are built, they will make predictions on all the samples. Once the predictions for all the samples are made, these predicted prefetch options are copied back in Simics environment. Ideal scenario would be to implement these prediction models in Simics. But due to complex nature of the model JMP is used to create these prediction models. As Simics is deterministic in nature we can map the predicted prefetch options to its corresponding interval.

Table 6-1 Prediction for Model - I

Training set : 35835 samples			
Prefetch Option	Prefetch Degree	Prefetch Stride	Misclassification Ratio
1	1	1	0.1272
2	1	2	0.2428
3	1	4	0.2140
4	1	8	0.2088
5	1	-1	0.1720
6	1	-2	0.1680
7	1	-4	0.1345
8	1	-8	0.1164
9	2	1	0.2699
10	4	1	0.2177
11	8	1	0.1764
12	-2	1	0.1552
13	-4	1	0.1300
14	-8	1	0.1017
15	64	1	0.0974
16	64	2	0.0972
17	64	4	0.0985
18	64	8	0.1028

19	64	-1	0.0350
20	64	-2	0.0357
21	64	-4	0.0349
22	64	-8	0.0345

6.1.1 Performance of Model-I

Once the predicted prefetch options are obtained from JMP, they are integrated in Simics. The workloads are run with these predicted options and the overall performance of the workload is observed. Figure 6-4 shows the performance of adaptive prefetching approach over sandbox prefetcher [23]. The performance comparison is done using normalized CPI. It is normalized to no prefetch. Most of the time sandbox prefetcher works better than this adaptive prefetching technique. But the performance difference between the two is very less. Also the adaptive prefetching is always better than no prefetch. The performance of machine learning technique is within 5% of sandbox prefetcher.

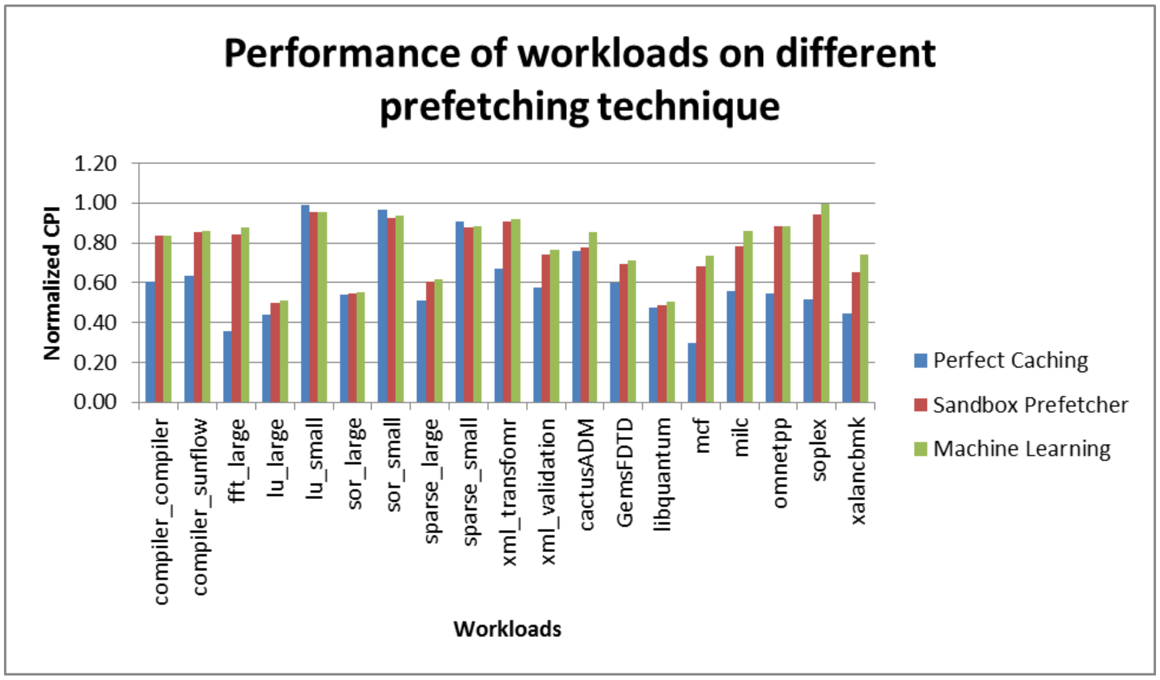


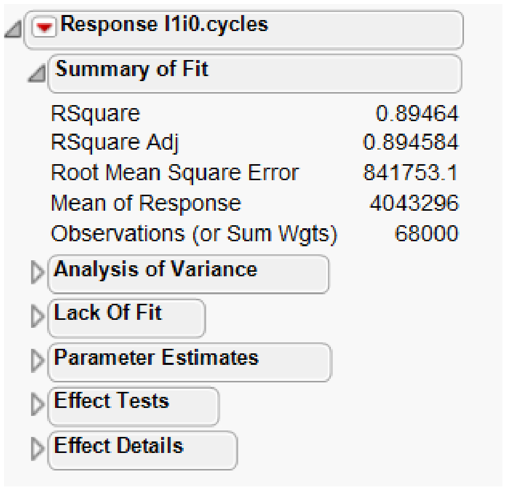
Figure 6-4 Comparing performance of adaptive prefetching (Machine Learning) with other prefetching techniques

6.1.2 Limitation of this approach

The labeling of samples into positive negative depends on the value of baseline which is computed once at the start of the workload. What if this value is faulty? Also if the phase of the workload changes this baseline value might be too high or too low for that interval. Also what if the cycles computed by simulator is incorrect? Let us tackle each question independently.

Most of the workload goes through various phases. Hence we need to collect baseline samples more frequently. For that we have another data set in which the prefetching option is triggered using random threshold but after every 100M instructions i.e. after every 100 samples we are turning off prefetching and that becomes our baseline for next 100 samples. Section 6.2 will talk about the model built with this dataset.

For simulation cycle, based on paper [22] we can do a quick check if the values are faulty or noisy. A simple linear regression for response as simulation cycle and factors as read and instruction fetch misses for each cache level gives a reasonable rsquare. As it is a simulator we would expect an rsquare close to 1. But due to some long latency instructions we have noise in the simulation cycle. Here in our case we got an rsquare of approximately 0.9 which is reasonable. There is some noise in the dataset but we can work with it and continue with using this metric for labelling samples.



RSquare	0.89464
RSquare Adj	0.894584
Root Mean Square Error	841753.1
Mean of Response	4043296
Observations (or Sum Wgts)	68000

Figure 6-5: Linear model for simulation cycle

6.2 Building Model-II

In the previous model the baseline samples for labelling the dataset was collected once per workload. But this one sample might not be representative for all the samples as the phase of the workload might change. For this model we frequently collect the baseline samples for labelling data. After every 100M instruction we turn off prefetching and collect these baseline samples. Rest of the things are same as compared to the previous model in section 6.1.

Table shows the misclassification ratio for all the prediction models built for model-II. These models will now make performance prediction for all the samples.

Table 6-2 Prediction for Model - II

Training set : 35835 samples			
Prefetch Option	Prefetch Degree	Prefetch Stride	Misclassification Ratio
1	1	1	0.2748
2	1	2	0.2542
3	1	4	0.2289
4	1	8	0.2250
5	1	-1	0.2760
6	1	-2	0.2525
7	1	-4	0.2289
8	1	-8	0.2247
9	2	1	0.2943
10	4	1	0.2797
11	8	1	0.2462
12	-2	1	0.2938
13	-4	1	0.2797
14	-8	1	0.2463
15	64	1	0.2190
16	64	2	0.2149
17	64	4	0.2133

18	64	8	0.2205
19	64	-1	0.2187
20	64	-2	0.2145
21	64	-4	0.2131
22	64	-8	0.2194

6.2.1 Performance of Model-II

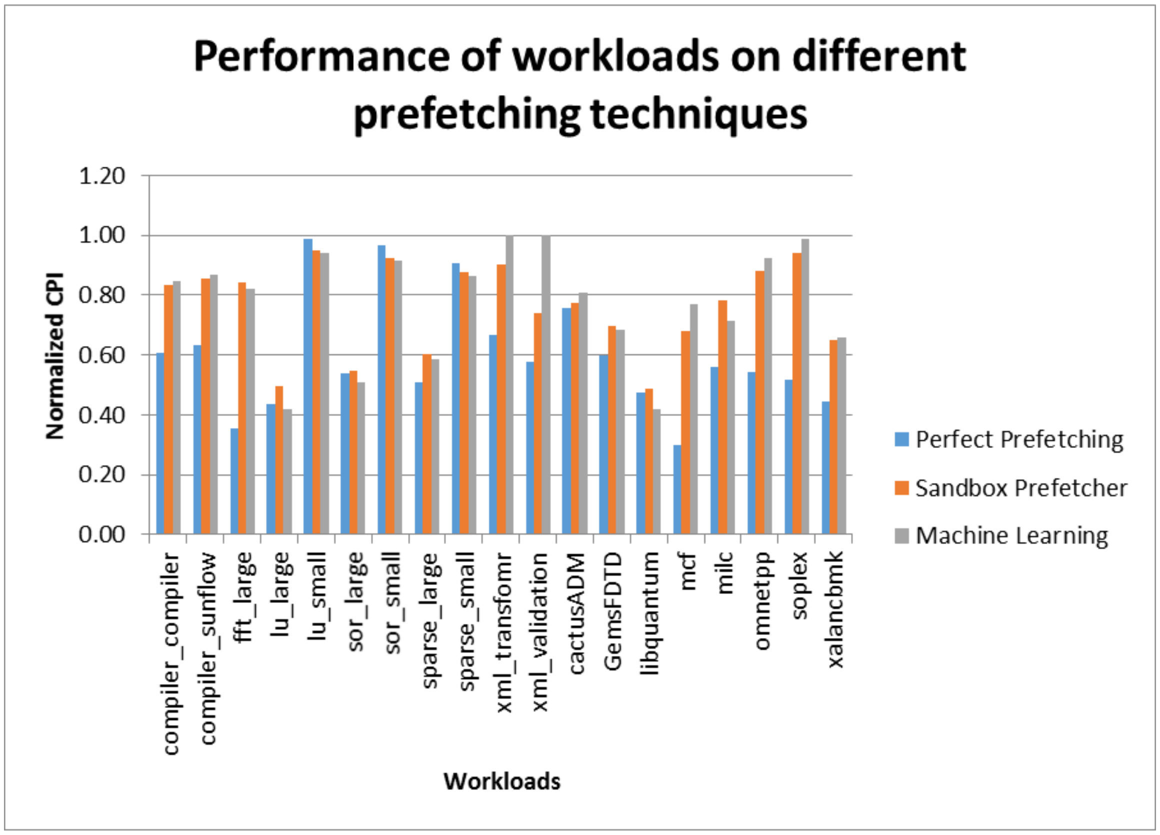


Figure 6-6 Comparing performance of adaptive prefetching (Machine Learning) with other prefetching techniques

This model does 2% better than sandbox for SPECjvm workloads and 1% better than sandbox for SPECcpu workloads. The reason why it is able to beat sandbox is, now the labelling of the samples is more meaningful. Earlier we had only one sample per workload. But this model looks at phase of workload every 100M instruction and then labels the samples into positive negative buckets.

6.2.2 Comparing Model-I and Model-II

The graph below compares of both the models. It is quite evident from the graph that model-2 works best. Just by labelling the data in a more representative way we can get good prediction.

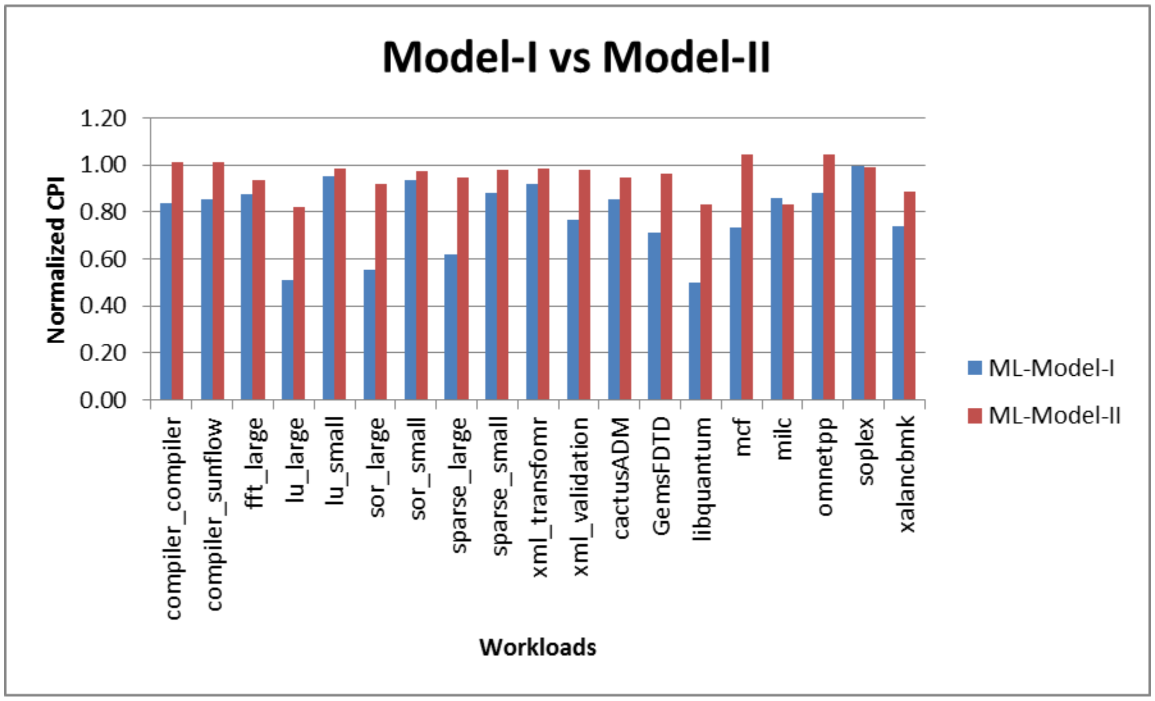


Figure 6-7 Comparing both the models

With this approach we can dynamically tune the prefetch options without any predefined heuristics. All that is needed is a good training dataset which has reasonable coverage of different workloads possible and various different prefetch options which benefit the workload.

Chapter 7 Limitation and Future Work

In the last chapter we discussed various models built and the performance of those models. It works well with both SPECcpu and SPECjvm workloads. This chapter will cover the limitation and future work of this adaptive approach.

7.1 Limitation

1. The prefetch options predicted by the models are very sensitive to the way the models are labelled. Thus the performance of this prefetcher highly depends on this step of labelling samples into positive negative bucket
2. The prediction models are built outside the simulator. Ideally speaking they should be built inside the simulator so that they can adaptively look at the hardware performance counters to predict the prefetch options. As we have models built outside the simulator the results obtained by this approach will be little different than what would be obtained when this model is implemented in hardware.
3. The machine learning algorithm used is very complex and difficult to implement on actual hardware. It will be very expensive to build it. Hence we need a good alternative algorithm that can be cheaply implemented on hardware or simulator.
4. The performance counters which are being monitored by the model to make prediction are very limited and biased. We need more counters which gives us some information about the stride and memory performance.

In spite of having so many limitations this approach still gives us a reasonable performance improvement. This may not be the best prefetching technique but with the results so far we have a good confidence in the direction we are heading towards. This technique will eliminate the need of selecting heuristics for prefetcher.

7.2 Future Work

If we go along this route the final product will look like figure 7-1. Figure 7-1 shows that every socket will have a predictive model that will decide the prefetch options at run-time. On interval basis it will monitor the hardware performance counters like EMON and decide the prefetch option. Once the model is built it doesn’t have to keep track of the access history. The model is built before the SKU is released.

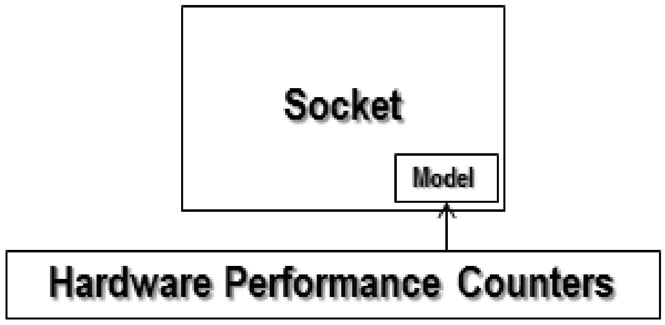


Figure 7-1 Predictive Model in the socket

7.3 Alternative uses of this approach

This approach need not be limited to only prefetching. It can be used in places where we need to have adaptive behavior without having fixed heuristics. For DRAM access the energy consumption is higher. We can use this kind of a smart engine or model to decide if we need to propagate the prefetch request to the main memory. It can be used in hybrid architecture to schedule tasks on different cores. We can use this technique wherever we need to have a dynamic behavior in the system

Finally to conclude this project is all about integrating machine learning with hardware prefetching. With machine learning we can eliminate the need to fix heuristics for prefetching.

Chapter 8 Bibliography

[1] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, San Fransico: Morgan Kaufmann Publishers Inc, 2011.

[2] S. Byna and X.-H. Sun, "Taxonomy of Data Prefetching for Multicore Processors," *JOURNAL OF COMPUTER SCIENCE,* p. 405{417, May 2009.

[3] T. C. Mowry, M. S. Lam and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *ASPLOS V Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, New York, 1992.

[4] R. Hegde, "Optimizing Application Performance on Intel® Core™ Microarchitecture Using Hardware-Implemented Prefetchers".

[5] E. Ebrahimi, O. Mutlu and Y. Patt, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *IEEE 15th International Symposium on High Performance Computer Architecture*, Feb. 2009.

[6] R. Chappell, J. Stark, S. Kim, S. Reinhardt and Y. Patt, "Simultaneous subordinate microthreading (SSMT)," in *Proceedings of the 26th International Symposium on Computer Architecture*, 1999.

[7] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *ISCA '01 Proceedings of the 28th annual international symposium on Computer architecture*, New York, May 2001 .

[8] S. Srinath, O. Mutlu, H. Kim and Y. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," in *IEEE 13th International Symposium on High Performance Computer Architecture*, Feb. 2007.

[9] O. Mutlu, J. Stark, C. Wilkerson and Y. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003,* Feb. 2003.

[10] X. Gu and C. Ding, "On the theory and potential of LRU-MRU collaborative cache management," in *Proceedings of the international symposium on Memory management*, November 2011 .

[11] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt and B. Werner, "Simics: A full system simulation platform," in *Computer , vol.35, no.2, pp.50,58,,* Feb 2002.

[12] "WIND RIVER," [Online]. Available: http://www.windriver.com/products/product-notes/8033_Simics_PN_0612.pdf.

[13] Y. Ishii, M. Inaba and K. Hiraki, "Access map pattern matching for data cache prefetch," in *Proceedings of the 23rd international conference on Supercomputing,*

2009.

[14] "Wind River Simics Full System Simulator," [Online]. Available: http://www.windriver.com/products/simics/.

[15] "Standard Performance Evaluation Corporation CPU2006 Benchmark Suite," [Online]. Available: http://www.spec.org/cpu2006/..

[16] "Standard Performance Evaluation Corporation JVM 2008 Benchmark Suite," [Online]. Available: https://www.spec.org/jvm2008/.

[17] "Standard Performance Evaluation Corporation CPU 2006," [Online]. Available: https://www.spec.org/cpu2006/Docs/.

[18] "Standard Performance Evaluation Corporation JVM 2008 Benchmark Suite," [Online]. Available: https://www.spec.org/jvm2008/docs/benchmarks/index.html.

[19] "Microsoft Logistic Regression Algorithm Technical Reference," [Online]. Available: http://technet.microsoft.com/en-us/library/cc645904.aspx.

[20] S.-W. 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*, Nov. 2009.

[21] H. Wickham, "Tidy Data," *Journal of Statistical Software,* vol. VV, no. II.

[22] K. Chow, P. Maldikar, R. Scott, P.-f. Chuang and K. Ban, "Experiments and Analytics for Software-Hardware Optimization," in *Workshop on Reproducible Research Methodologies at HPCA 2014*, Orlando, February 2014.

[23] S. H. Pugsley, d. R. Balasubramonian, Z. Chishti, C. Wilkerson, S.-L. Lu, P.-f. Chuang, R. L. Scott and K. Chow, "Sandbox Prefetching: Safe Run-Time Evaluation of Aggressive Prefetchers," in *20th IEEE International Symposium On High Performance Computer Architecture*, Orlando, 2014.