การทำนายข้อมูลใช้ซ้ำเพื่อเลี่ยงการบันทึกแคช

นางสาววริษา ศรีไตรรัตนรักษ์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรดุษฎีบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2558

DATA REUSABILITY PREDICTION FOR CACHE BYPASSING

Miss Warisa Sritriratanarak

A Dissertation Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy Program in Computer Engineering

Department of Computer Engineering

Faculty of Engineering

Chulalongkorn University

Academic Year 2015

Thesis Title	DATA REUSABILITY PREDICTION FOR CACHE BYPASSING
By	MissWarisa Sritriratanarak
Field of Study	Computer Engineering
Thesis Advisor	Professor Prabhas Chongstitvatana, Ph.D.
Thesis Co-advisor	Assistant Professor Mongkol Ekpanyapong, Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Doctoral Degree

..... Dean of the Faculty of Engineering
(Professor Bundhit Eua-arporn, Ph.D.)

THESIS COMMITTEE

..... Chairman
(Assistant Professor Sukree Sinthupinyo, Ph.D.)

..... Thesis Advisor
(Professor Prabhas Chongstitvatana, Ph.D.)

..... Thesis Co-advisor
(Assistant Professor Mongkol Ekpanyapong, Ph.D.)

..... Examiner
(Assistant Professor Krerk Piromsopa, Ph.D.)

..... External Examiner
(Assistant Professor Kemathat Vibhatavanij, Ph.D.)

วริษา ศรีไตรรัตนรักษ์ : การทำนายข้อมูลใช้ซ้ำเพื่อเลี่ยงการบันทึกแคช (Data Reusability Prediction for Cache Bypassing) อ. ที่ปรึกษาวิทยานิพนธ์หลัก : ศ.ดร. ประภาส จงสถิตย์วัฒนา, อ. ที่ปรึกษาวิทยานิพนธ์ร่วม : ผศ.ดร.มงคล เอกปัญญาพงศ์, 72 หน้า.

การปรับปรุงหน่วยความจำแคชด้วยวิธีบายพาสเป็นวิธีเพิ่มสมรรถนะสำหรับหน่วยความจำแคชระดับสุดท้ายในหน่วยประมวลผลหลายแกนซึ่งข้อมูลส่วนใหญ่ไม่เคยถูกนำกลับมาใช้ใหม่ อย่างไรก็ตามเทคนิคการสร้างทางลัดเกือบทั้งหมดพึ่งพาแต่วิธีอย่างใดอย่างหนึ่งโดยเฉพาะ เช่น การใช้ตัวนับและตาราง ทำให้ไม่สามารถแก้ปัญหาการมีงานที่สลับซับซ้อนของตัวประมวลผลหลายแกนได้ วิทยานิพนธ์ฉบับนี้เสนอวิธีทางเลือกโดยการทำนายผลของการทำบายพาสโดยใช้ซัพพอร์ตเวคเตอร์แมชชีน โดยใช้การเก็บข้อมูลที่ถูกเรียกใช้จากหน่วยประมวลผลโดยโปรแกรมจำลองการทำงานของหน่วยประมวลผลหลายแกน ซึ่งข้อมูลที่ได้จะทำมาฝึกการเรียนรู้ด้วยเครื่องจักรที่เรียกว่าซัพพอร์ตเวคเตอร์แมชชีน ผลลัพธ์ที่ได้คือตัวทำนายการบายพาสที่เหมาะสมสำหรับตัวประมวลผลหลายแกน ซัพพอร์ตเวคเตอร์แมชชีนจะตัดสินว่าข้อมูลแต่ละตัวจะถูกเก็บลงในหน่วยความจำแคชหรือไม่ แล้วทำการวัดผลด้วยโปรแกรมจำลองการทำงานของหน่วยความจำแคชเพื่อวัดประสิทธิภาพของหน่วยความจำแคชว่าดีขึ้นเพียงใด ผลของการทดลองแสดงให้เห็นว่าเมื่อตั้งพารามิเตอร์ที่เหมาะสมให้กับซัพพอร์ตเวคเตอร์แมชชีน จะสามารถสร้างแบบจำลองเพื่อตัดสินใจเก็บข้อมูลหรือปล่อยผ่านข้อมูลได้เป็นอย่างดี และสามารถเพิ่มอัตราการเข้าถึงข้อมูลแล้วพบได้ 5.34% ภายใต้โปรแกรมวัดค่าสแปลชทู

ภาควิชา : วิศวกรรมคอมพิวเตอร์ ลายมือชื่อนิสิต.....

สาขาวิชา: วิศวกรรมคอมพิวเตอร์ ลายมือชื่อ อ.ที่ปรึกษาวิทยานิพนธ์หลัก.....

ปีการศึกษา 2558 ลายมือชื่อ อ.ที่ปรึกษาวิทยานิพนธ์ร่วม.....

5371814921: MAJOR COMPUTER ENGINEERING
KEYWORDS: CACHE MEMORY / CACHE BYPASSING/ SUPPORT VECTOR MACHINE

WARISA SRITRIRATANARAK : DATA REUSABILITY PREDICTION FOR CACHE BYPASSING. ADVISOR : PROF. PRABHAS CHONGSTITVATANA, Ph.D., CO-ADVISOR : ASST. PROF. MONGKOL EKPANYAPONG, Ph.D., 72 pp.

Cache bypassing emerged as a performance improvement method for shared Last-Level Caches (LLC) in multicore processors where large portions of data are never reused. However, most bypass techniques have relied on ad hoc methods such as counters and tables which can not tackle the complexity of multicore workloads. In this dissertation, we propose an alternative method to predict cache bypassing using Support Vector Machine (SVM) models. Based on access traces obtained from representative benchmarks running on the Multi2Sim simulator, supervised SVM training was performed in order to obtain a bypass prediction model suitable for LLC in multi-core processors. The SVM outputs bypassing classifiers which are integrated on the simulator to quantify LLC performance improvements. Results show that, with appropriate parameters and kernel functions, SVM is capable of generating bypassing models which improve LLC performance on multicore processors, achieving an average 5.34% hit rate improvement across SPLASH2 benchmark combinations.

Department : Computer Engineering	Student's Signature .....
Field of Study : Computer Engineering	Advisor's Signature .....
Academic Year : 2015	Co-advisor's Signature.....

Acknowledgements

A PhD dissertation is a hard proof of patience and perseverance of a person dedicated with their life. This work could never be accomplished without a kind guidance from my adviser, Prof. Prabhas Chongstitvatana. My deepest gratitude is always expressed towards him. Also, I am sincerely grateful to my co-supervisor, Asst. Prof. Mongkol Ekpanyapong, for his informative guidance and support throughout my Ph.D. candidature. My dissertation would not be completed without his insightful and knowledgeable suggestions. *I am truly proud that I was under their supervision.*

I greatly appreciate Asst. Prof. Sukree Sinthupinyol, Asst. Prof. Krerk Piromsopa, and Asst. Prof. Kemathat Vibhatavanij, for being in my dissertation committee and giving many useful comments and suggestions to improve my dissertation.

Many thanks are due to all Intelligent System Laboratory's members and all Ph.D. fellows for their discussions and contributions to research works. And to all my dear friends who understand a Phd's life and always so supportive.

I would like to thank Department of Computer Engineering for the opportunity and meaningful experience. During my time at this department, I have learned an important lesson: *Patience and effort are the most important things during a hard time.*

Last but not least, this long journey could never be completed without the support from my lovely family whom enjoyed my not-so-schedule work hours and exploited them extensively. Also my life partner, Paulo, who dedicated himself to enlighten me during those depressive hours and let me know the limit of his patience is endless. *This dissertation is dedicated to them.*

Contents

List of Tables

Table Page

List of Figures

CHAPTER I

INTRODUCTION

Computer system performance nowadays relies heavily on the efficiency of the memory. Although the performance of the processor has drastically improved in the past decades, the access time to the main memory has sped up more slowly, therefore leaving a huge performance gap between processor and the memory, e.g., the processor performance is improving approximately 75% annum while the DRAM speed steadily increasing by 7% each year (McKee, 2004). In addition, there are efforts to increase the processor performance such as pipelining, branch prediction, out-of-order execution, which eventually leads to a memory bottleneck. The reason is that memory size is inversely proportional to memory performance, and size requirements increased more quickly than memory technology evolved. Main memory has improved, from SDRAM technology to DRAM and then to DDR-RAM technologies, which allows higher frequencies and bandwidths (HPm, 2010). However, these improvements still can not cope with the bandwidth required by processors for high utilization, maintaining a performance gap between performance and memory.

The concept of memory hierarchy is introduced to ameliorate the problem. The use of a small but fast associative memory called look-aside memory is suggested by Lee in 1963 to improve the overall performance of the computer. The term cache was first applied when it was implemented in the IBM System/360 Model 85 as a buffer between processor and main storage (Liptay, 1968). It was announced as a successful memory hierarchy implementation that reduced waiting time from storage from microseconds to nanoseconds. The aim of having the cache is to reduced the long waiting time to acquire data from the main memory. As the processor-memory performance gap increased, more levels of cache were used. The level closest to the processor is called the highest level or the level 1 cache (L1). The levels below are closer to the main memory and called L2, L3,..., last-level cache. For example, the Intel Pentium processor has level 1 on-chip cache and level 2 off-chip cache implemented (Horton, 1995); the L2 cache moved up to be on-chip in the Intel Pentium Pro (Pen, 1996). With the advent of multicore processors, several private and shared levels of cache are used, i.e., the Intel Core Xeon X5550 has four cores and three layers of cache: each core has private L1 and L2 cache; the Last-

level cache (LLC) is shared among four cores. With 2.66GHz processor speed, the size of the cache are 32KB and 256KB for L1 and L2 cache on each core, and the last-level cache size is 8MB. The access time to L1, L2, and L3, is approximately 4, 10 and 40 cycles, respectively, while the access time to the main memory is approximately 60ns or 120cycles (Levinthal, 2010). Sharing the LLC can reduce the number of duplicate copies when many cores are running identical applications. Also, when each core has different working set sizes, shared LLC has better utilization than private LLC. Throughout this dissertation, the term last-level cache and LLC will be used interchangeably.

An implementation of multilevel cache requires a definition of the relationship between each level. *Inclusion* defines a relationship in which data have copies in every lower levels of cache. When a datum is propagated from main memory to cache, a copy of it is allocated to all levels of cache. When a datum is modified, all copies of the data also have to be modified transparently. This is done to simplify the cache management and make the higher level cache work like an optimized lookup for the lower level cache. Whenever the upper level cache is full, the data block can be easily discarded because there is always another copy on the lower level cache; but if the datum on the lower level is discarded, all the other copies on the higher levels have to be discarded too. Also, since the last-level cache contains all the data from all upper levels, the looking up can be done by looking in the LLC only; this allows the L1 to work continuously without being disturbed. On the other hand, *exclusion* means that the data can have only one copy in all levels of cache, avoiding data duplicates. To allocate data on cache, all levels have to be checked to make sure that no other copies exist elsewhere. It reduces the space wasted from duplication but looking up a datum requires probing all levels of cache and managing zero duplication in all levels in all cores can be very complicated. The combination of the two is called *non-inclusion*, which does not force either inclusion or exclusion on cache. A datum can have multiple copies like an inclusive cache but when a datum is discarded from the lower level cache, the copies in the upper levels are not required to be discarded too. When the datum moves up from lower level to higher level, the exclusive cache will force deletion of the copy in lower level while the non-inclusive cache will allow leaving the copy in lower level, avoid wasting any blocks. In this dissertation, we focus on the non-inclusive cache only. Non-inclusive memory hierarchies are the simplest to implement, since each cache can manage its data independently of upper and lower levels. Although this complicates cache coherency mechanisms, it has been proven that non-inclusive cache achieve better

performance than inclusive cache (Zahran, 2007). The extensive detail on the advantage of non-inclusive cache can be found in Zahran (2007).

Cache is efficient because it holds the data that are being used by the processors. The key idea that makes cache useful is the concept of locality of reference (Denning, 1972). The locality of reference is based on the notion that the memory access tends to be localized in time and space. It stems from the behavior of the programmers that tend to use data sequentially and looping. The tendency that next required data would be at the address adjacent to the currently used data means *spatial locality* while the tendency that same data will be used again soon means *temporal locality*. As the complexity of the programs increased, the memory-access behavior becomes more complicated and could exhibit larger access patterns. Algorithmic locality defines the access pattern that repeated, either in time or space, but in a manner larger than normal spatial or temporal locality. It would depend on the algorithm being used in the application. For instance, a computer simulation that repeatedly accesses a very large data set stored in dynamic data structures. The pattern could be captured by observing program behavior and adapting to it via run-time mechanism. With the benefit of localities, we can exploit the predictability and improve the memory system performance by accurately predicting which data should be allocated in cache and reduce the number of processor stall cycles.

Cache terms used to describe the efficiency of the cache are *cache hit*, which is when the processor requested data is found in cache, and *cache miss*, when the data is not found. The *hit rate* is the number of hits to the number of data accesses while the *miss rate* is the ratio of misses instead. Data in the cache are usually stored in *blocks* or *lines* whose sizes differ between architectures. The goal of using the memory hierarchy is to have a memory system that hides latency as much as possible, i.e., maximizes the hit rate and minimizes the access time, while maintaining the system cost-effective. The goal could be achieved by improving performance of cache by finding algorithms that accurately predict what to put in cache and how to replace it when it is full. For this reason, most researchers are focusing their works on techniques for bringing data to the cache which will be required in the near future, through some form of prediction (prefetching) and techniques for adequately managing which data to replace on cache when it is full (cache replacement policy).

Nonetheless, with the use of multicore and multithreaded modern processors, memory usage has drastically changed to become more complex and the algorithms to manage cache need to be improved. In multicore, the last-level cache is usually shared and contention from many cores could cause memory *thrashing*, i.e., the working set is larger than the cache size and cause the requests to be constant misses. The complexity of the multicore workloads and the concern for energy consumption make writing all data to all levels of cache seem lavish. Many workloads today are multimedia applications which load millions of blocks of data and use them only once. Some data are placed in cache and never reused again until they are evicted from cache. This is especially true in the LLC, since exploitation of temporal locality in high level cache means an inversion of temporal locality on LLC; in other words, data that is accessed frequently will always hit on level 1 caches, thus remain unused on levels 2 and 3. Although the data is sometimes requested again from LLC (since L1 is smaller, it might eventually be evicted while it is still in LLC), the time between first and subsequent requests does not justify its presence on the LLC if it replaces more valuable data; hence, the justification for bypass. On levels closer to processor, the high temporal locality makes this approach less feasible. Study shows that numerous data blocks are allocated to the last-level cache and never reused or accessed again(Li et al., 2012). Fig 1.1 shows the percentage of distant reuse blocks in the 2MB LRU LLC over a memory-intensive benchmarks from SPEC CPU2006, 81.2% of blocks are not reused before eviction and 25.6% of blocks are never accessed again. Those blocks that are placed in the LLC and never accessed should never be allocated on cache to waste precious cache space. The method of not allocating some data to cache is called *Cache Bypassing*

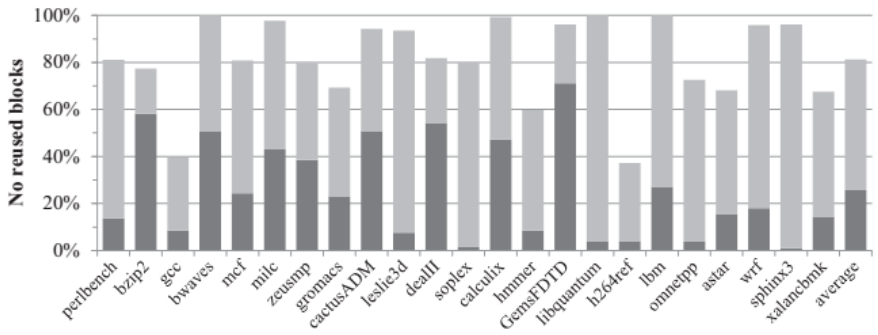

Figure 1.1: Percentage of distant reuse blocks in the 2MB LRU LLC over a memory-intensive benchmarks from SPEC CPU2006, 81.2% of blocks are not reused before eviction and 25.6% of blocks are never accessed again.

Cache bypassing is a term describing a method that bypasses data from one or more levels of memory hierarchies. Bypassing could reduce costs from writing unnecessary data to cache and reduce memory bandwidth in transferring data. In fact, when the cache size is smaller than workload requirements, writing data to cache wastes more than the cost of writing only to memory. It also evicts existing valuable data from cache which could lead to more cache misses and increase processor stall cycles. For this reason, avoiding allocating never-accessed data is a good solution to improve cache performance. The use of the non-inclusive cache allows the data to be bypassed from the LLC without changing any existing architecture. Previous methods for classifying which data should be bypassed involved using predictors and counters. For example, applying the same approaches to bypass as the ones applied for branch prediction e.g., assuming a status until error and then reversing the status when it fails (Tyson et al., 1995; Jalminger and Stenström, 2003), or assuming one out of every n number should be bypassed (Kharbutli and Solihin, 2008; Kharbutli et al., 2013). This information was then stored on on-cache tables to direct bypassing logic. While these approaches sufficed for very simple workloads, these ad-hoc prediction methods can not be easily applied to complex, multicore workloads, driving the need for new bypass mechanisms. More sophisticated methods should be applied to the problem. Since we can simplify the problem into classification problem, data should be bypassed or not, it is known that the perfect method that could intelligently classify data into two groups is Machine Learning. The two-class classification is called *binary classification*; a field where Support Vector Machines have been known to excel. Hence, this dissertation explores the use of SVM for bypass prediction in shared Last Level Caches, in order to determine its suitability for the bypass problem.

1.1 Objectives of Research

- To develop an efficient cache architecture that allows bypassing based on the result of prediction.
- To develop a new technique to predict cache reusability based on machine learning algorithms.

1.2 Scopes of Study

The scope of this dissertation is limited to the following:

- This work would focus on improving cache performance in multicore processors, i.e. , quad-core or more using shared last level cache.
- The benchmarks that will be used to test the performance is SPLASH2.
- Performance measurement can be either instruction per cycle or the number of misses in accessing cache.

1.3 Summary of Contributions

In this dissertations, we provide the following contributions:

- We provide a proof of concept that machine learning is capable of predicting data reusability. It can decide whether to bypass data from cache.
- Support Vector Machine is an appropriate tool for classifying which data should be bypassed from the shared last-level cache.
- We provide a list of important attributes that have effects on making bypass decisions.

1.4 Dissertation Organization

The rest of the dissertation is organized as follows. The next chapter introduces background on cache and machine learning. Chapter 3 provides literature reviews in relevant fields. Chapter 4 describes the method to prove the theory we proposed. Then we show the results from the experiment in Chapter 5 and state the result discussions. Finally, the last chapter provides the summary of the dissertation, limitations, and future works.

CHAPTER II

BACKGROUND

In this chapter, we provide the background on cache and describe basic cache terms used in this dissertation.

2.1 Cache

Cache is the name given to the temporary storage that buffers between the processors and and the main memory. It can also mean any buffer that temporarily stores data to speed up the accesses to other storage devices. In this dissertation, we only focus on the cache that buffers data between the processors and the main memory. The goal of having a cache is to improve the performance of the processors by having data ready for the processors to use once needed, while maintaining the cost-efficiency balance. In general, the performance of the cache is defined by the following terms:

- *Cache hit* is a the ratio of the number of hits to the number of total cache accesses. Here, we represent them in percentage.

$$CacheHitPercentage = \frac{numberofhits}{numberofaccesses} \times 100 \tag{2.1}$$

- *Cache miss* is the ratio of the number of misses to the number of total cache accesses. Here, we represent them in percentage.

$$CacheMissPercentage = \frac{numberofmisses}{numberofaccesses} \times 100 \tag{2.2}$$

- *Latency* is the amount of time that the processor has to wait for a datum, starting from request until reception of such datum.

To understand cache, further details on cache terminologies need to be mentioned.

2.1.1 Cache Organization

Cache stores data in blocks or lines whose sizes vary between architectures. In modern processors, the block size is usually 32-byte or 64-byte. Each block has an address

to identify itself. There are three basic organizations to arrange cache: *fully associative*, *direct-mapped*, and *set associative*. They are illustrated in Fig. 2.1

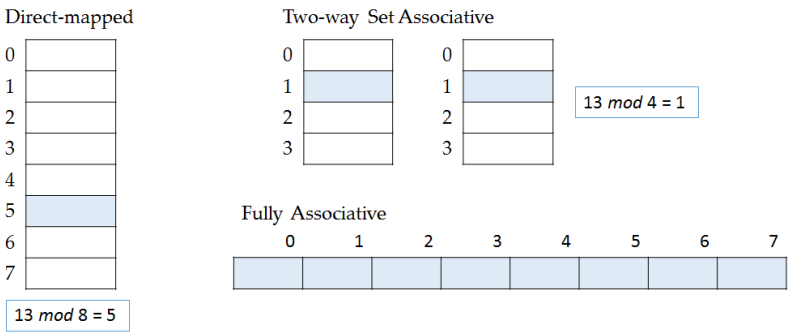

Figure 2.1: Three types of cache organizations. This is an example of the cache where block address 13 can be allocated in the gray area. In direct-mapped, 13 mod 8 = 5; the block should be located at location 5. In two-set associative cache, 13 mod 4 = 1; the block can be allocated at one of the two blocks at location 1. In fully-associative cache, a block can be allocated anywhere.

In direct-mapped cache, a block can be allocated in one specific place. In general, the specific place is where the block address *mod* with the number of blocks in the cache. For example, if the address is 13 and the number of blocks in the cache is 8, the place to put the data is the block number 5. This makes the look up very fast for direct-mapped cache but it is possible that many data may require the same spot. The competition for the same block space is called *cache contention* and could degrade the performance. When it is fully associative cache, a block can be allocated anywhere in the cache so the cache contention for a block space does not occur. All the tags in all cache are looked up to find the match concurrently so it is very fast to find the requested data but the implementation requires more hardware cost to compare tags at all blocks at once. The combination of two organizations is a set-associative cache. It is when the blocks are grouped into *sets*, the block can be allocated where the block address *mod* with the number of sets in cache. For example, if the address is 13 and the number of sets is 4 and the size of each set is 2, the place to put the block can be one of the two blocks in set 1. The set-associative is a fast look-up time and cost-friendly organization. When there are n block in each set, the organization is called *n-way set associative*. A 32-bit address for a 32-byte block is shown in Fig. 2.2

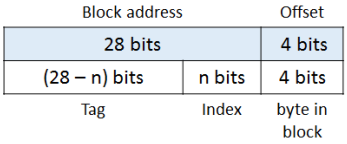

Figure 2.2: An example view of 32-bit address for 32-byte block. The block address is used to identify the block, the offset is used to identify which byte in the block. In case of n-way set associative, the tag is used for checking which block in the set, and the index is used to select which set. In fully associative the n in the index field is zero.

2.1.2 Cache Replacement Policy

Placing a data block on cache should be easy if the size of the cache is large enough to store all the data. In reality, the cache size is small to be efficient, fast and conserve energy. If the cache organization is direct-mapped, there is only one place to put the data block, no decision needs to be make as to where to allocate a new block into cache. However, when the cache is full and a miss occurs in the set-associative or fully-associative cache, the incoming block has to replace one of the data in cache. The block chosen to be replaced is referred to as a *victim block*. There are different methods to choose the victim block but a few basic cache replacement policies should be mentioned. *Random* replacement policy randomly selects the victim block to be replaced. The benefit is the simplicity in the implementation. In *First In First Out (FIFO)*, the first block that goes into cache will be replaced first. It is also simple to implemented with simple hardware required. The most common policy is the *least-recently used (LRU)*. All accesses to block have to be timely recorded and the block that has no reference (read or write) for the longest period is replaced first. Since the implementation of LRU is complicated and expensive, there is a simpler version of LRU called *Pseudo-LRU* which tries to approximate the LRU policy with less cost. The tree Pseudo-LRU is the most popular method widely used in modern processor caches. Its implementation is a binary search tree which recognizes which side of the leaf has been traversed. On a similar principle with LRU, the *least frequently used (LFU)* policy chooses the cache block that were referenced least often to be the victim block. With much less hardware overhead, *not recently used (NRU)* has one status bit to recognized if the block referenced recently. Besides from the basic cache replacement policies, Belady (1966) proposed a method to find the best performance possible from the cache replacement policy. The key idea is simply to find the block that will be reused furthest in the future to be a victim block. It requires knowledge of future accesses which

is impossible to implement but provides the best performance possible, typically as a baseline to compare with other methods. There are other recent replacement policies suggested by researchers to improve the cache performance; we will discuss them in the next chapter.

2.1.3 Cache Writes

Most instructions read data and do not write to memory. However, there are two types of writes that need to be mentioned. When processors are writing to cache, the writing is to both cache and main memory. This policy is called *Write through*. When processors only write to cache and the modified cache block will be written to main memory later, it is called *Write back* policy. To identify which data is modified, there is a status bit attached to each cache block. The status bit is also called *the dirty bit*, its duty is to tell whether the data is modified. The write back has the advantage of saving the bandwidth between the cache and main memory. While the write through policy is easier to implement.

2.1.4 Multilevel cache

In modern processors, the cache usually has one to three levels, namely, L1, L2, L3. The first level cache (L1) is the fastest cache but also smallest because of its price and the power dissipation. When there are many cores together, all levels of cache can be either *private* or *shared* cache. The term private means that the cache space belongs to one core and reads and writes are exclusive to that core only. On the other hand, shared cache means that any core can request to read and write the data on the cache. To use the limited space efficiently, L1 is commonly two split caches that separate instructions and data. The purpose is to allow the core to access both instruction and data simultaneously via two different ports. The second level cache (L2) is a larger, medium latency cache, mostly private to each core and implemented as unified cache, combining both instructions and data. The last-level cache (L3 or LLC) is normally unified cache which is shared among cores. The main benefit is more flexible and dynamic allocation of resources, for example, when some cores are idle, the active core can utilize the whole LLC space. Since shared last-level cache has become pervasive, there are many researchers focusing on improving the performance on this cache.

2.1.5 Cache Consistency

When many copies of the same datum exist, some policy must be employed to ensure that the processor receives the correct version of the datum requested. In case of writes, data in other places require transparent update to guarantee correctness. Cache coherence verifies that the data is updated in a timely fashion and memory consistency ensures that processors see exactly the same sequence of changes of all values. There are two cache coherence protocols used in modern processors, directory-based and snoop-based protocols. In directory-based protocol, the directory is a table storing state of each block such as ownership and its sharing status. It is scalable because directories can be distributed across memories. When there is a write to each block, the status in the directory has to be updated and send point-to-point updates to processors. On a snooping system, each cache monitors every bus, comparing the address with the ones it has stored. Should there be a match, the cache either updates its own copy or marks it as dirty. Although snooping is rather simple to implement in a small system, it does not scale well; in a many-core system with multiple caches, each would have to have access to every bus, leading to area and bandwidth issues. Thus, when the size of the cache is large, it is more popular to implement the directory-based protocol.

2.2 Machine Learning

Machine learning involves techniques and methods which give the machine an ability to learn from examples and provide patterns or predictions as results. It is designed to solve some tasks that could not be defined well by human or some complex problems which could not be solved by human. Each learning algorithm has different assumptions and creates different functions and is hence preferred for different kinds of problems, e.g., data mining, pattern recognition, classification, knowledge extraction, etc. They could be classified by types of learning to generate output function into two major groups, unsupervised learning and supervised learning.

Unsupervised learning refers to a method to uncover structure in unlabeled data. There is no reward or any signal that could evaluate the solution. Example problem that required unsupervised learning is data clustering, where objects are grouped into clusters based on similarities, e.g., document clustering which groups similar documents together. Another example is knowledge extraction in which we learn rules that are

underlined in data. In supervised learning, the input data are labeled with desired outputs as examples to be observed. An algorithm will try to generate a function that maps labeled input to desired outputs based on training data. The result function will be capable of distinguishing between different examples (or patterns) and predict answers for unseen data. If the desired output is discrete, the problem is called classification problem, and the function is known as a classifier. The classifier should accurately predict correct classes for any valid input data. In our problem, we could classify output as two groups of data, to be reused and never reused. This kind of problem is called binary or two-class classification where output could be generalized to simply answer with yes or no only. As it should be supervised learning, we need to label input data or patterns to give samples to the machine learning algorithms and selectively describe our input by features or the characteristic of data to solve our problem. This problem of binary classification could be solved by several machine learning methods, for example, decision tree, artificial neural network, support vector machines, etc.

Decision tree is a flowchart-like tree structure tool for supervised learning where *internal nodes* indicate a test on an input data, *branches* indicate the outcome of the test, and *leaves* or *terminal nodes* hold a class label. The process of testing starts at the topmost node or the *root* and repeats until the leaf node is reached and received the output. For its simplicity, decision tree is easy to understand and fast learning algorithm. It is used in large and realistic classification problems, such as medical diagnosis, credit risk analysis. However, decision tree is extremely sensitive to small perturbations in the data, i.e., a slight change can result in a drastically different tree, and our prediction is based on memory address of accessing data which could easily change over time and hence the tree needs to be reconstructed very often. Moreover, since our training data size needs to grow if the prediction accuracy is not good enough, training could be difficult because the tree needs to restart from the beginning if the training data has changed. Accordingly, the tree could not learn online and would not fit workloads that change continuously.

Support vector machine (SVM) is learning algorithm introduced by Cortes and Vapnik (1995), it is a prediction tool for classification and regression using machine learning theory to maximize predictive accuracy while avoiding over-fitting to data. In binary classification, its operation is to search for optimal hyperplane for linearly separable patterns by learning from sample data. SVM is used extensively in broad subjects to solve

different complex problems, e.g., web spam detection, facial recognition, medical data extraction, etc. Support vectors are the data points that lie closest to the decision surface where classification is most difficult. If the data is non-linearly separable, the kernel function will be applied to map sample data from original space to higher dimensional feature space to create a linearly separable pattern. The distance from the hyperplane to the instances closest to it on either side is called the *margin*. SVM will attempt to extend maximum margin between two class data samples while maintaining least errors.

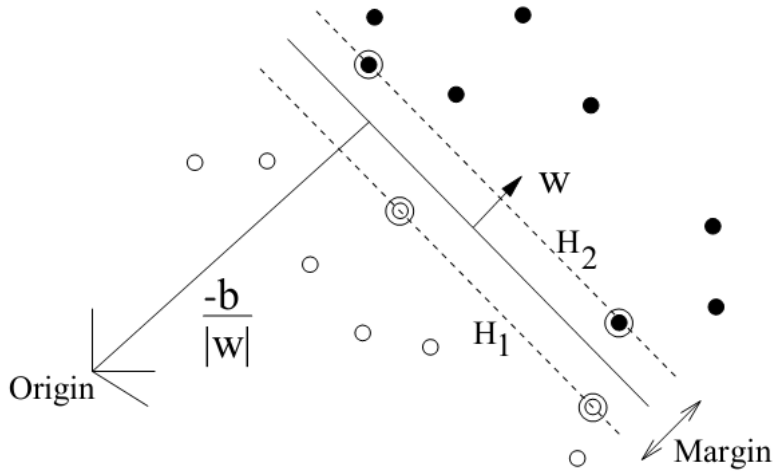

Figure 2.3: linear separating hyperplane for the separable case. The support vectors are circled. (Burges, 1998)

To be specific, sample data are tagged with label -1 and $+1$ for the two classes. The training samples are $\{\mathbf{x_i}, y_i\}$, where $i = \{1, .., l\}$ and $y_i \in \{-1, 1\}, \mathbf{x_i} \in R^d$. The points $\mathbf{x_i}$ which lie on the hyperplane satisfy $\mathbf{w} \cdot \mathbf{x} + b = 0$, where $\mathbf{w}$ is normal to the hyperplane, $\frac{|b|}{\|\mathbf{w}\|}$ is the perpendicular distance from the hyperplane to the origin, and $\|\mathbf{w}\|$ is the Euclidean norm of $\mathbf{w}$. The equation could be written as

$$\mathbf{w} \cdot \mathbf{x_i} \geq +1 \text{ for } y_i = +1$$

$$\mathbf{w} \cdot \mathbf{x_i} \leq -1 \text{ for } y_i = -1$$

which could be combine to

$$y_i(\mathbf{w} \cdot \mathbf{x_i} + b) \geq +1 \ \forall i \tag{2.3}$$

Any tuples $\mathbf{x_i}$ that falls on the $H1 : \mathbf{w} \cdot \mathbf{x_i} + b = 1$ or $H2 : \mathbf{w} \cdot \mathbf{x_i} + b = -1$ hyperplanes are called support vectors and hence, the perpendicular distance from origin to the hyperplane $H1$ and $H2$ is $\frac{|1-b|}{\|\mathbf{w}\|}$ and $\frac{|-1-b|}{\|\mathbf{w}\|}$ respectively. Therefore the distance between two hyperplanes or the *margin* is $\frac{2}{\|\mathbf{w}\|}$. In order to predict patterns or classification of test data $\mathbf{x^t}$, the equation could re-written based on the Lagrangian formulation to find the maximum margin hyperplane as

$$d(\mathbf{x^t}) = \sum_{i=1}^{l} y_i \alpha_i \mathbf{x_i} \mathbf{x^t} + b_0 \tag{2.4}$$

where y_i is the class label of support vector $\mathbf{x_i}$, $\mathbf{x^t}$ is a test tuple and α_i and b_0 are numeric parameters that were determined automatically by the optimization or SVM algorithm above; and l is the number of support vectors. We could test the tuple $\mathbf{x^t}$ by plugging in equation 2.4 to see if the $\mathbf{x^t}$ lie on which side of the hyperplane. Accordingly, if the result is positive, it means that $\mathbf{x^t}$ lies on or above hyperplane $H1$ and it belongs to class +1. Otherwise, if the result is negative then $\mathbf{x^t}$ lies on or below hyperplane $H2$ that it belongs to class -1. In case of non-linearly separable data, a kernel function could be used to maps the original data into a higher dimensional space. The most common kernel functions used are listed in Table 2.1

Table 2.1: The most common kernel functions

Kernel	Kernel Function
Linear	$K(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T\mathbf{y}$
Polynomial	$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T\mathbf{y} + 1)^q$
Radial Basis Function	$K(\mathbf{x}, \mathbf{y}) = exp[-\frac{\|\mathbf{x} - \mathbf{y}\|^2}{2s^2}]$
Sigmoid	$K(\mathbf{x}, \mathbf{y}) = tanh(\delta \mathbf{x}\mathbf{y} - \delta)$

CHAPTER III

LITERATURE REVIEWS

Many works have been focused on improving the performance of the cache. Various techniques can be divided into different groups; cache partitioning, cache replacement policy, cache prefetching, and cache bypassing.

3.1 Cache Partitioning

By exploiting benefits of locality, early works focus on improving the small direct-mapped cache by adding an extra buffer. To start with, Jouppi (1990) introduced an approach to improve the hit ratio of a direct-mapped cache by adding a small fully-associative buffer called victim cache. The victim cache is used to hold the data evicted from the main cache and give them a second chance to be hit. It shows temporal locality of data that evicted data could be reused again. The improved hit rate of Jouppi's cache could be as good as two-way set associative cache but requires less circuitry. Similarly, a small buffer called assist cache is added to the main cache in HP-7200 (Hay et al., 1996) but it appears to function like a primary cache which stores incoming data and only forwards data that shows temporal locality to the main cache and bypass data that shows spatial locality back to the memory. Another work by Rivers and Davidson (1996) also add a fully-associative buffer called NT buffer to support the main cache. An NT bit is added to each line in caches to identify the non-temporal data and decide whether the data should be put in the NT buffer or the main cache. By managing data based on temporal locality, the cache is not polluted by non-temporal data so the conflicts in the main cache are reduced and the hit ratio is increased.

Adding identifying bits can help recognize lines that has temporal locality. McFarling (1992) also uses sticky-bit and hit-last bit to note that the line has reused and should be kept in the cache. His dynamic exclusion gives priority to temporal data and preserved them in cache longer result in better cache performance. Difference approach in managing two different locality is proposed by González et al. (1995). González et al. proposed a cache that separated into two partitions based on data locality, temporal and spatial cache. When the predictor predicts that data has no locality, it will be bypassed from both caches.

3.2 Cache Replacement Policy

The main purpose of cache replacement policies is to predict which block in the cache should be the last one to be re-referenced and that last one should be replaced by a new block. As mentioned in Chapter 2, LRU is the basic cache replacement policy widely used in modern processors. LRU policy keeps track of the time when each blocks are being referenced (read or write). It is similar to a linked list that has the most recently used block, the latest block referenced, at the head of the list and the least recently used block at the tail. The victim block that will be replaced is always the one at the tail.

With LRU as basic replacement policy, some modifications to improve the LRU performance are proposed by changing the status of the block inserted, called the *insertion policy*. Qureshi et al. (2007) proposed LRU Insertion Policy (LIP) to insert the new incoming block at the end of the LRU chain instead. Unless promoted by being referenced again, the block will be replaced next. This gives priority to the block that is being re-referenced more than once. Another method proposed is Bimodol Insertion policy (BIP) which randomly selects where to insert the block, between the MRU and LRU, with more priority at the latter. The actual run, called Dynamic Insertion Policy (DIP), is implemented as set-dualing that switches between BIP and the traditional LRU policy depending on which policy is giving better hit rate at the time. A little modification of DIP is to make the algorithm thread aware. Thread-aware dynamic insertion policy (TADIP) has different counters for each different thread. The method claimed to achieve throughput near the traditional LRU with double cache size.

Besides from LRU, a number of researchers have proposed many methods to select the victim block. The challenge is to keep track of the time each block was referenced or to keep the list in order. The counter used to store the time or frequency could take up precious cache space. With the expense of implementing true LRU, Jaleel et al. (2010) proposed a Re-Reference Interval Prediction (RRIP) chain to replace the LRU chain by roughly dividing data into four groups. The RRIP represents the order that the blocks are predicted to be re-referenced from soonest to furthest. When cache is empty, the new incoming block is inserted in a position second from the furthest group. It will be promoted to the sooner group when it is referenced. When the cache is full, the block in the furthest group will be replaced first. The authors' method ensures that the data

repeatedly re-referenced stay longer in cache and the data that used only once (refer to as *scan* by author) are kept in the cache very shortly while requiring less hardware and outperforming traditional LRU policy. An improvement of the RRIP is proposed in Wu et al. (2011), Prefetch aware cache management (PACman) is a version of RRIP with the awareness of which blocks are prefetched or demand fetched. It gives the priority to the blocks that are demand-fetched to stay longer in the cache. As a result, PACman improved overall workloads more than 20% from traditional LRU replacement policy.

Khan et al. (2010) observed that the blocks in LLC are dead 86% of the time in memory-intensive benchmarks, i.e., the blocks are placed in cache and never reused again. To solve the problem, the authors create a predictor that samples program counter (PC) to determine if the block is dead and should be removed from LLC. The predictor can reduce the LLC miss rate from traditional LRU by 23% on multicore workloads while the sampling requires less overhead than conventional predictors with states storage.

3.3 Cache Prefetching

Prefetching is a method that moves a block of data up in the memory hierarchies before it is actually needed by the processor (Vanderwiel and Lilja, 2000). It could be done by either hardware or software or both. *Software prefetch* generally required sophisticated compilers to modify code in order to insert fetch data instructions. It would require extra processor cycles since it added extra lines to activate the prefetch in the code. As a result, it incurs in significant overhead and could degrade performance in some benchmarks as shown in Bernstein et al. (1995); Santhanam et al. (1997); Lipasti et al. (1995). Since software prefetch tends to specific within applications, we would focus on hardware prefetch here.

Hardware prefetch adds prefetching to the system with no need to modify the code. Smith (1982) notes in his survey that three important things to concern in prefetch are: 1)when the prefetch should be initiated 2) which data should be prefetched and 3)what status should be given to the prefetched data. First, the time to activate prefetch can be either prefetch-on-hit or prefetch-on-miss. Difference is prefetch activates when the data required by the processor is found in a cache or not. Sometimes it could be prefetch always, i.e., every time data in cache is accessed, but it would highly cost memory bandwidth. As the prefetched data displaces existing data in cache, if the prefetched data is fetched too

soon that it idles a long time in cache or it is removed before its use then that data will become *cache pollution*. Cache pollution could cause not only reduced cache performance but also wasted memory bandwidth and energy.

Second, choosing which data to prefetch is extremely important. Sequential prefetch exploits the spatial locality by prefetch data that adjacent to the currently accessed data. The simplest one is fetching one block of data that adjacent to the block currently accessed Smith (1978) called one block look-ahead (OBL). For example, if the address that is currently accessing is `a` then the prefetch data is `a+1`. The differences between current address `a` and the next address `a+1` can be more than one and called *stride prefetching*. Jouppi (1990) introduced extra stream buffers to store prefetched next-address data separately from the main cache. Later, Baer and Chen (1991) record load/store instructions in the reference prediction table and predict future accesses that could be non-unit strides. It is called *stride prefetching*. Other works alternated stride prefetching by applied it to different memories Fu et al. (1992), Ibáñez et al. (1998), Kim and Veidenbaum (1997), Sklenar (1992).

Palacharla and Kessler (1994) proposed equal-sized stream buffers as a secondary cache. It detected strides and captures stream data behavior without the need for program counter which is very useful to memory other than on-chip cache. However, constant stride prefetching is only efficient on large programs with array or data accesses that exhibit spatial locality. For data access patterns that are different, Baer and Chen (1995) proposed aggressive prefetcher which predicts the size of stride dynamically. They also compare the method with Mowry et al. (1992) in Chen and Baer (1994) and concluded that the performance improvement depends on the program characteristics and there is no prefetcher with consistently better performance.

Other than stride-based prefetchers, Joseph and Grunwald (1997) interestingly apply Markov model to learn the miss addresses and predict addresses to prefetch. The Markov-like model is one kind of correlation-based prefetchers which use one address as a *key* and prefetch the next address follows that key. The disadvantage is that the prefetcher must see the miss reference repeats before it can predict future miss correctly. Sair et al. (2002) suggested that load instruction behavior can be classified into four groups as follow: next-stride, stride, same-object, and pointer-based, then concluded that the prefetch

should be able to classify the behavior and prefetch the correct stream of data. The same-object refers to prefetching a large object that loaded into different blocks. Zhang and Torrellas (1995) recognized if the objects are used together and prefetch all blocks. Pointer-based application is more difficult to predict since the addresses are not pattern. Many works (Collins et al., 2002; Cooksey et al., 2002; Ramos et al., 2000; Wang et al., 2003) are targeted to solve the problem. Sair et al. concluded that Markov predictor is suitable for the type. They improved and proposed predictor-directed stream buffer based on Sherwood et al. (2000) similar to Kandiraju and Sivasubramaniam (2002); Oly and Reed (2002) that also modified the Markov predictor and achieved impressive performance.

Besides from predicting method, there are software and hardware based lookahead method. Software means spending one thread to execute the program in advance only to fetch the correct prefetched data to another working thread Luk (2001); Moshovos et al. (2001); Roth et al. (1998). While hardware (Mutlu et al., 2003) use runahead technique to learn what to prefetch by allowing the processor to pretend to execute program during the processor stall cycles but not actually commit them. Similarly, Ganusov and Burtscher (2005); Zhou (2005), use advantages of multicore processor and let one core executes program ahead to prefetch correct data. These techniques are not quite popular because all the programs must execute twice only to reduce stall time, it would be a trade off between energy and time which may not worth spending.

For specific application like multimedia applications, Lee et al. (2003) examined data access patterns of multimedia applications and grouped them into three types, fixed-stride, 2way, and 2d streams and add three prefetchers to prefetch data to separated stream cache. As a result, the misses in multimedia benchmarks are drastically reduced. This work has showed that data access pattern in specific application could be learned and guided to achieve impressive performance. Somogyi et al. (2009) use the synergy between spatial and temporal locality to enhance prefetching performance. Their approach, Spatio-Temporal Memory Streaming (STeMS), looks at miss history to drive the prefetcher and prevent further misses through out the memory region.

3.4 Cache Bypassing

The idea of predicting what data to fetch is similar to predicting which data not to cache. Tyson et al. (1995) proposed a method to manage data cache by selectively allow data allocation. It stems from observation that only small number of instructions are responsible for a large number of cache misses occurred in data cache, i.e., data allocation from some instructions are the main causes of cache misses. Consequently, those instructions should be prohibited from allocating data in cache in order to reduce the miss ratio. Instructions are marked with C/NA (Cacheable/Non Allocatable) if they are predicted to create a lot of misses. The prediction is made by a version of two-bit branch predictor by Yeh and Patt (1991). The important result is not the improved hit ratio but it is the substantially reduced memory bandwidth.

From the very first work, Johnson et al. (1999) proposed another cache bypassing which based the predictions on the memory access addresses instead of load instructions. They observed the memory access distribution and traversed through address trace to analyze the optimal bypassing decisions and the upper bound hit ratio that it could achieve. The prediction is predicted by partitioning data into Macroblocks and store access history in the Memory Access Table. It is reported that the result often achieved hit ratios close to the upper bound which is quite impressive improvement.

Later, Jalminger and Stenström (2003) applied the two-level branch predictor similar to Tyson's but the prediction is based on the memory access addresses like Johnson's. The aim is to predict whether a cache block will be reused within a limit distance or not. If the data will be reused then it would be allocated in the cache otherwise it would be placed in the bypass buffer for a short period. There are two types of table, one is for record each cache line access history called Reuse History Table and the other is for the two-bit predictor for each pattern per each cache line called Reuse Prediction Table. The two-bit predictor has four states which update every time a block is accessed based on the reuse history. Though Jalminger and Stenström's propose achieve high prediction accuracy (66%-94%) and reduce miss rate for up to 32%, it require a large amount of memory to record these tables and enormous bandwidth to update two tables every time the cache is accessed.

Piquet et al. (2007) observed that some data blocks are bought into cache and never

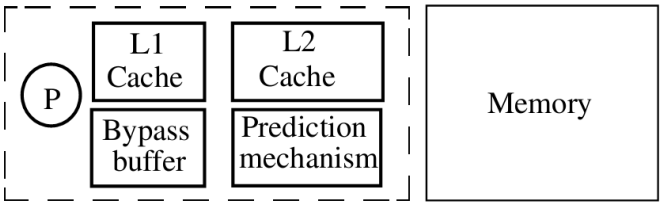

Figure 3.1: Organization of Jalminger and Stenström (2003) novel cache approach

accessed again before being evicted. Those blocks are named *single-usage* blocks (SU) and storing single-usage blocks on the cache creates *single-usage pollution*. The experiment shows that single usage pollution occurs only 6% of the time in L1 data cache but 33% in the L2 (last-level cache), and hence, data should be bypassed from the L2. The authors propose a method to decide which data should be bypassed based on the instruction who requested the data. Each line in the L2 is tagged with the Instruction Address (IA) and an SU bit that set to one when the data is accessed after being allocated in cache. The IA are stored in the BUP table accompanied with a counter. When a block in the L2 is evicted, the counter associated with the IA is updated; if the SU bit is one the counter is reset to zero, and if the SU bit is zero then the counter is incremented. When there is an L2 allocation, the IA is looked up in the BUP table. If the counter associated with the IA tag is saturated, the data block will be bypassed from the L2. The organization is shown in Fig 3.2

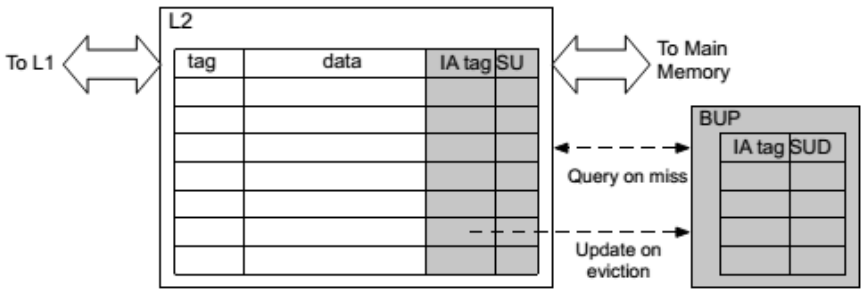

Figure 3.2: Organization of Piquet et al. (2007) bypassing method.

Another bypassing approach called Counter-based replacement and bypassing algorithm is introduced in Kharbutli and Solihin (2008). Kharbutli and Solihin observed that data that are brought to L2 cache often idle for a long period in cache after its last use because of the Least Recently Used (LRU) replacement policy that hold data a

long time before evicted. For some lines, they are never re-accessed again after they are brought to L2 cache. Therefore, the authors relaxed cache inclusion between L1 and L2 and suggested a method that detect never-reaccessed lines, bypass those lines from L2 cache, and load them to L1 only. Based on counter that count specific access to memory address, the authors proposed method to identify expired lines and replace them, also, it could identify bursty temporal data and put them in L1 cache only. As a result, the algorithm gain speed up in 10 out of 21 SPEC2000 benchmarks while degraded less than one percent in the rest.

Furthermore, Xiang et al. (2009) suggest that if the working set is larger than the cache size, remove only lines that will never reused at all is not enough to preserve the temporal data, more blocks should also bypass from cache. In addition, those blocks that will be reused only a few times should be removed by using Less Reused Filter (LRF). LRF consists of Reuse Frequency Predictor for predicting how many times each block will be reused, A filter buffer for storing less reused lines separated from the main cache, and the shadow tags for storing only tags of the evicted cache lines from the filter buffer. With the reuse count prediction, LRF could determine when the data should be placed. When most of the working sets are retained in cache the higher hit rate can be achieved. An improvement to LRF is proposed by Qiao et al. (2011) by adjusting LRF to be shared cache in multicore processor. It received better IPC compared to the uniprocessor and reduced the cache miss rate. The disadvantage of both LRF is the expensive hardware cost from keeping many counter values in each cache line and the large size of shadow tags.

Feng et al. predict data reusability based on previous reuse information and create a new replacement policy in Feng et al. (2011). The method adaptively switches between two predictors, reuse information based replacement policy and typical LRU replacement policy. Similarly, PDP (Duong et al., 2012) use reuse prediction to protect potentially reuse blocks in cache from being replace, if all the blocks are protected then the incoming block is bypassed. Optimal Bypass Monitor (OBM) (Li et al., 2012) tried to implement the optimal replacement policy and adding bypassing feature by comparing whether the incoming or the victim block will be reuse first and keep the nearest reuse data. The algorithm keeps track of the reuse information in the form of *incoming block* and *victim block* pair, and also requires another table to store a counter use to predict the reuse

distance of each incoming block. Storing and updating the two tables on every access cause enormous traffic amount. Another simple approach that claims to be more efficient than OBM is SCIP (Kharbutli et al., 2013). SCIP basically counts the number of time each block requested to L2. If the two-bit counter incremented to three then the LLC would allocate the block, bypasses around 75% of the data. As in their result, SCIP bypasses at least 82% of the incoming block and achieve an average of 18% speed up proving that the LLC require to allocate only some portion of data.

Combining partitioning and bypass, most recent work(Khan et al., 2014) proposed a cache management method that distinguishes between read and write requests and gives more priority to the read to reduce stalls from write. The author adapted the reuse prediction method in (Piquet et al., 2007) but only to predict data that will be read reuse only and bypass predicted writes and not reuse data from cache. The method speedup an overall average on SPEC2006 benchmarks 5% and 14% on caches-sensitive benchmarks compare to baseline LRU replacement policy.

CHAPTER IV

METHODOLOGY

Shared cache behavior on multicore cannot be predicted by analyzing each core's workloads separately. The dynamics of multicore operation result in shared LLC effects, e.g., cache thrashing, that must be analyzed at shared LLC level. This is in contrast with private caches behavior where traces can be obtained directly from processor cores, and bypass algorithm could be directly deduced from processors behavior, e.g., by analyzing the program counter; this approach is not feasible for shared last-level caches where multicore interactions must be tackled by more sophisticated methods based on LLC access traces. In this chapter, we provide details on how support vector machine can learn from training examples and generate model classifier to predict which data should be bypass from the shared last-level cache. An overview of the system is shown in Fig. 4.1.

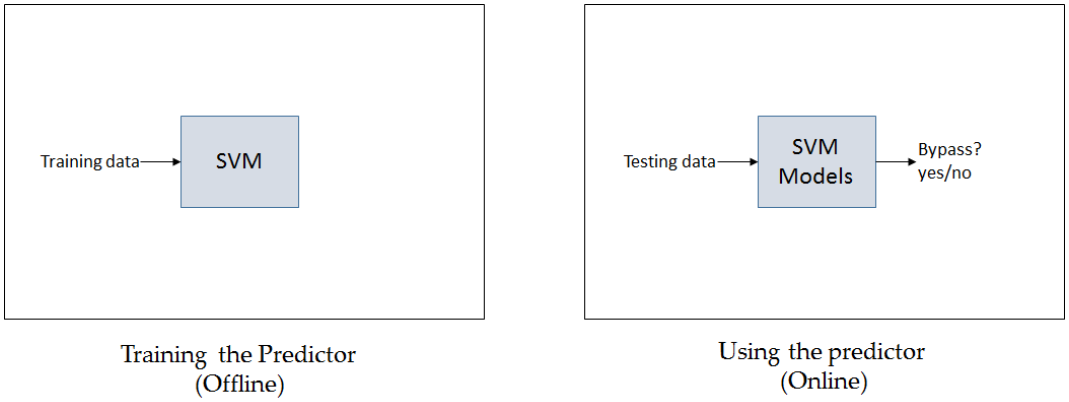

Figure 4.1: An overview of the system

The system is separated into two parts, the offline process and the online process. The offline process is the SVM learning process to generate models to classify data into bypass and not-bypass. It starts from deciding what are the data, getting those data from the simulator, and preparing the training data for the SVM. The online process is the cache simulation with the use of the bypass classifier to decide whether to allocate data on the last-level cache. The offline process detail will be described first and the online process detail will be described in the latter section.

4.1 The offline process

4.1.1 Preparing data

First, SVM requires labeled training data to learn. The definition of the training data is a set of known inputs with a known response to the input data. What we consider that should be the inputs of the SVM are the state of the parameters involving the memory and the core. The output is the class of the data, bypass or not-bypass. To create input data, we create a trace of addresses that the cores requested to use, both instruction and data, including all the features with the following motivations: 1) previous works (Piquet et al., 2007), (Khan et al., 2014) show that it is important to know which instruction is requesting the data, thus Program Counter (PC) has impact on making bypass decisions. 2) The locality of data provides the assumption that the data requested are related to the data previously requested, either in space or time. Therefore, the address of the previously requested data from the core and the previous miss from the higher level cache (L1) should have impact on bypass decision. 3) The core number that requested data indicates whose request generated from. It can help distinguish which application that is running and decide which application should be bypassed. 4) It is likely that data should be bypassed than instructions, so it is important to distinguish between the data and the instructions. 5) The temporal locality suggested that the time of access should be the key feature to decide bypassing. In conclusion, the trace includes the following information: address requested, program counter, core number that requested data, access type (Instruction or Data), previous address requested from the core, previous address requested to the LLC, and the time of access.

4.1.2 The simulator

We use Multi2Sim simulator (Ubal et al., 2012) to model a quad-core processor with X86 ISA. All caches are non-inclusive, write-through and write-allocate with true LRU replacement policy. The cache parameters are depicted in Table 4.1.

Table 4.1: Parameters of the simulated cache

L1 Instruction Cache	8KB, 32B-line size, 2-way
L1 Data Cache	8KB, 32B-line size, 2-way
L2 Shared Last-level Cache	64KB, 32B-line size, 4-way

Cache size is small in relation to benchmark memory usage (working set) in order to force contention in the shared last-level cache. Since the purpose is to prove the SVM ability to selectively bypass the data, two-level cache is sufficient to demonstrate the feasibility and exploiting the relationship between each core's private cache and the shared LLC.

4.1.3 Benchmarks

We use SPLASH2 benchmarks because of its multithreaded support and its purpose of studying parallel machine. The application domain of each benchmark are summarized in Table 4.2. We create 7 benchmark combinations by randomly selecting 4 out of 11 SPLASH2 benchmarks to run simultaneously, four threads per benchmark. Each combination is listed in Table 4.3 and simulated for 200 million committed instructions after fast forwarding the first 100 million instructions.

Table 4.2: Benchmark application domain

Program	Application Domain
Barnes	High-Performance Computing
Cholesky	High-Performance Computing
FFT	Signal Processing
FMM	High-Performance Computing
LU	High-Performance Computing
Ocean	High-Performance Computing
Radiosity	Graphics
Radix	General
Raytrace	Graphics
Water-nsquared	High-Performance Computing
Water-spatial	High-Performance Computing

Table 4.3: Benchmark combination

com 1	Raytrace, Radiosity, Water-nsquared, Water-spatial
com 2	Radiosity, Lu, Ocean, FFT
com 3	Cholesky, Fmm, Water-nsquared, Radix
com 4	Barnes, Fmm, FFT, Radix
com 5	Ocean, Lu, Barnes, Water-spatial
com 6	Fmm, Cholesky, Lu, Raytrace
com 7	FFT, Barnes, Radiosity, Water-nsquared

4.1.4 The training data

The traces are separated into two parts, the training set and the testing set. We select the first 1 million accesses to be the training data or the input for the SVM. The rest are reserved for testing the classifier generated by the SVM. On the training data, each datum must be identified, as belonging to "bypass" or "not-bypass" classes; the data is marked so it can be used for SVM training (sample data). We manually divide the data using our knowledge of future behavior, similar to the optimal lookahead in Li et al. (2012). The difference is the lookahead is limited to the window of size n to check for future reuse. The marking method is illustrated in Fig. 4.2.

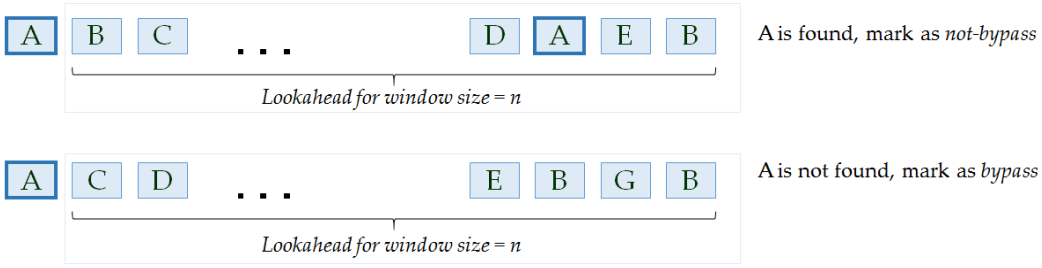

Figure 4.2: The marking method to create training data

For example, when address A is accessed, LLC trace is checked for n addresses, determining whether or not address A will be reused. If so, then A will be marked as *not-bypass* and will be allocated in LLC normally. If not, then A is marked as *bypass* and will be bypassed from LLC. Experimental results suggest the most efficient window size for our experiment is *n=5000*; varying window size between 1000 and 10000. We show the result of our marking method with 3 different n sizes in Table 4.4.

Table 4.4: The hit rate of the training data with different window sizes, approximately first 100k accesses

Name	Benchmarks	baseline	n=2k	n=5k	n=10k
Com1	Raytrace, Radiosity, Water-nsquared, Water-Spatial	69.10	74.56	77.12	76.63
Com2	Radiosity, Lu, Ocean, FFT	44.52	49.36	50.27	50.58
Com3	Cholesky, FMM, Water-nsquared, Radix	76.93	77.73	80.02	80.25
Com4	Barnes, FMM, FFT, Radix	64.91	69.49	72.01	71.21
Com5	Ocean, Lu, Barnes, Water-Spatial	53.37	62.66	65.48	61.84
Com6	FMM, Lu, Cholesky, Raytrace	86.86	85.37	87.04	88.02
Com7	FFT, Barnes, Radiosity, Water-nsquared	59.33	65.14	67.18	67.57

4.1.5 Features and Kernel Functions

We use the four most common kernel functions as described in Chapter 2 to find the most appropriate kernel function for each benchmark combination: Linear, Polynomial, Radial basis function, and Sigmoid. The SVM that we used is the HR-SVM (Vateekul et al., 2014), an SVM-based technique specifically tailored for hierarchical multi-label classification. Multi-label classification consists of multiple binary classification i.e. when there are ten classes to classify, first the SVM divide the data into class 1 and the rest 9 classes first, then divide the rest 9 classes into class 2 and the rest 8 classes and so on. Thus, the HR-SVM highly suitable for binary classification with the imbalanced class issue; our training data have majority in *not-bypass* more than *bypass*. For feasible training, the data is scaled to -1, 1 and sampling was performed to obtain adequate training data. Empirical results showed that sampling every fifth address for a total of 100,000 data yielded adequate SVM training results. Table 4.4 display the number of data that are marked as bypass in each training data. The parameters of the SVM are varied as in Table 4.6

Table 4.5: The number of data in bypass class in 100k training data

Name	Benchmarks	n=2k	n=5k	n=10k
Com1	Raytrace, Radiosity, Water-nsquared, Water-Spatial	43989	23272	15709
Com2	Radiosity, Lu, Ocean, FFT	54302	47701	45812
Com3	Cholesky, FMM, Water-nsquared, Radix	34335	20203	15838
Com4	Barnes, FMM, FFT, Radix	44551	19473	14460
Com5	Ocean, Lu, Barnes, Water-Spatial	53551	35977	29547
Com6	FMM, Lu, Cholesky, Raytrace	33396	23629	20576
Com7	FFT, Barnes, Radiosity, Water-nsquared	43749	27370	18884

Table 4.6: SVM parameters

C	γ
0.5	0.000488
2	0.0025
4	0.025
8	0.007813
16	0.25
32	0.25

We use the 21 training data we created in the previous section for the HRSVM to learn. With 4 kernels and 6 parameters each, we then have 24 classifiers for each training

data. We use the trace part that reserved for testing (after 1 million accesses) to test the efficiency of the classifier. The SVM reads the input trace and then labeled each data to one of the two classes. The input plus the output from the SVM testing will be referred to as the *predicted trace* and will be used in the next section.

4.2 The online process

To measure the efficiency of the classifier, the cache simulator is modified to be augmented with bypassing capability. The predicted trace will be evaluated through re-simulation as if the classifier is making the bypass decision on the fly. The system structure is shown in Fig. 4.3. First, the trace is read as addresses request from the cores and the L1 cache operates as a normal cache with no modification. When there is a request to LLC, the LLC first check if there is a hit in the LLC. If it is a miss, the simulator look up for the data block in the main memory and check the trace which class the data belongs to, bypass or not-bypass. If the data marked not-bypass, then it is allocated to LLC. If the data marked as bypass, then the data read will be sent to L1 without allocating to LLC.

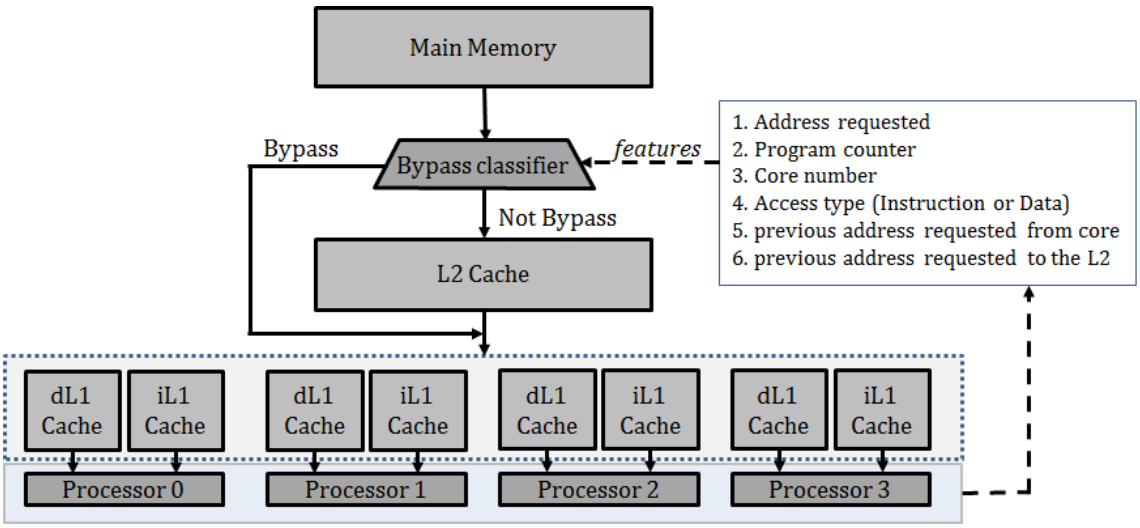

Figure 4.3: System Structure. The dotted arrows represent the information required by the classifier. The solid arrows represent data flows when cores request data from main memory (L2 missed).

For simplicity, we treat read requests and write requests equally assuming the write-through policy with allocate on miss. The replacement policy is true LRU because the

hardware overhead from the implementation can be ignored.

4.3 The implementation considerations

The actual implementation details would require the cores to simultaneously send the features information to the bypass classifier. The hardware implementation of the classifier can be done by reducing the models we achieved through HRSVM into an SVM mathematic equations. In the real-time calculation, it is likely that sometime some SVM classifiers perform better than the other. It is possible to implement multiple classifiers with a multiplexor to select the best classifier at the time. One of the methods that is possible to implement is the *set dueling* (Qureshi et al., 2007) which has small dedicated sets for each replacement policy and the remainder are the followers that would choose the policy following the set that provide the better performance. We could adapt the set dueling to implement some dedicated sets with some classifiers and the rest can follow the classifier that provide the best performance at the time. Other than that, in some specific applications, for example, the video encoding processor; it is possible to learn the application characteristics beforehand and implement only the specific classifier for that specific type of workload.

CHAPTER V

RESULTS AND DISCUSSION

5.1 Results

In this chapter, we show the experimental results of the SVM bypassing. First, we show the information of each combination in Table 5.1

Table 5.1: Memory and cycles usage for each benchmark combinations

Name	Benchmarks	Memory Usage (Bytes)	Number of cycles
Com1	Raytrace, Radiosity, Water-nsquared, Water-Spatial	187,219,968	152,902,582
Com2	Radiosity, Lu, Ocean, FFT	164,872,192	183,185,868
Com3	Cholesky, FMM, Water-nsquared, Radix	126,083,072	257,923,134
Com4	Barnes, FMM, FFT, Radix	140,763,136	225,048,277
Com5	Ocean, Lu, Barnes, Water-Spatial	150,843,392	253,674,883
Com6	FMM, Lu, Cholesky, Raytrace	159,490,048	167,651,903
Com7	FFT, Barnes, Radiosity, Water-nsquared	164,716,544	136,200,572

Since we focus on the accesses to the last-level cache which is L2, the number of LLC accesses and the hit rate are shown in Table 5.2

Table 5.2: Baseline LLC hit rate for each benchmark combinations

Name	Benchmarks	LLC accesses	Baseline Hit Rate
Com1	Raytrace, Radiosity, Water-nsquared, Water-Spatial	2,794,208	72.64%
Com2	Radiosity, Lu, Ocean, FFT	1,807,811	43.36%
Com3	Cholesky, FMM, Water-nsquared, Radix	3,262,796	41.59%
Com4	Barnes, FMM, FFT, Radix	2,635,956	43.56%
Com5	Ocean, Lu, Barnes, Water-Spatial	3,086,273	49.44%
Com6	FMM, Lu, Cholesky, Raytrace	2,998,467	70.48%
Com7	FFT, Barnes, Radiosity, Water-nsquared	1,715,070	66.21%

With all the possible combinations from all the parameters, we try to find the impact of having different number of features. We compare the results from using all 7 features and leave out some features and found the best performance from having 6 features and leave out the time of access. Also, we compare the results from using training data with

different window sizes and found the best results from the window size $n = 5000$. Using the same training data, the experimental results for the 6 features and 7 features are shown in Table 5.3.

Table 5.3: Best hit rate achieved using 6 features and 7 features using training data with n=5000 window size

	Baseline	6 features	%improved	7 features	%improved
Com1	72.64	74.39	2.42%	73.52	1.21%
Com2	43.36	44.09	1.68%	43.62	0.60%
Com3	41.59	44.75	7.60%	44.94	8.05%
Com4	43.56	50.96	16.99%	49.66	14.00%
Com5	49.44	52.27	5.72%	51.52	4.21%
Com6	70.48	69.48	1.42%	71.17	0.98%
Com7	66.21	69.12	4.40%	69.06	4.30%
		average	5.34%	average	4.77%

The best lookahead window size for the training data is *n=5000*. The best hit rate results from each kernel function are shown in Table 5.4 and Table 5.5 and illustrated in Fig 5.1 and Fig 5.2.

Table 5.4: Best hit rate achieved from the classifier prediction with 6 features and using training data with n=5000 window size

	Baseline	Linear	Polynomial	RBF	Sigmoid	Best	diff
Com1	72.64	74.05	74.37	74.39	74.32	74.39	+1.75
Com2	43.36	43.50	43.96	44.09	43.51	44.09	+0.73
Com3	41.59	44.75	42.68	42.03	40.79	44.75	+3.16
Com4	43.56	49.18	49.89	49.89	50.96	50.96	+7.40
Com5	49.44	51.75	52.27	51.76	52.01	52.27	+2.83
Com6	70.48	67.09	69.46	69.48	68.30	69.48	-1.00
Com7	66.21	68.50	68.59	68.94	69.12	69.12	+2.91

The detailed experimental results are listed at the end of this chapter.

5.2 Discussions

Our experimental results conclude that using proper parameters and kernel function, SVM can classify which data should be bypassed from the LLC. Evaluating results yields several conclusions: (1) From the results in Table 5.4 and Table 5.5, the trace with 6 features yield better average hit rate percentage increase. However, having 7 features provides a hit rate percentage increase in all benchmark combinations. It can be assumed that the time of access feature can help classify data in time-sensitive benchmarks and

Table 5.5: Best hit rate achieved from the classifier prediction with 7 features and using training data with n=5000 window size

	Baseline	Linear	Polynomial	RBF	Sigmoid	Best	diff
Com1	72.64	74.31	73.52	74.31	74.32	73.52	+0.88
Com2	43.36	43.53	43.46	43.62	43.53	43.62	+0.26
Com3	41.59	44.94	41.6	41.59	41.43	44.94	+3.35
Com4	43.56	49.10	48.67	49.59	49.66	49.66	+6.1
Com5	49.44	51.42	50.20	51.52	51.49	51.52	+2.08
Com6	70.48	70.81	71.17	70.71	70.71	71.17	+0.69
Com7	66.21	68.15	69.06	68.87	68.87	69.06	+2.85

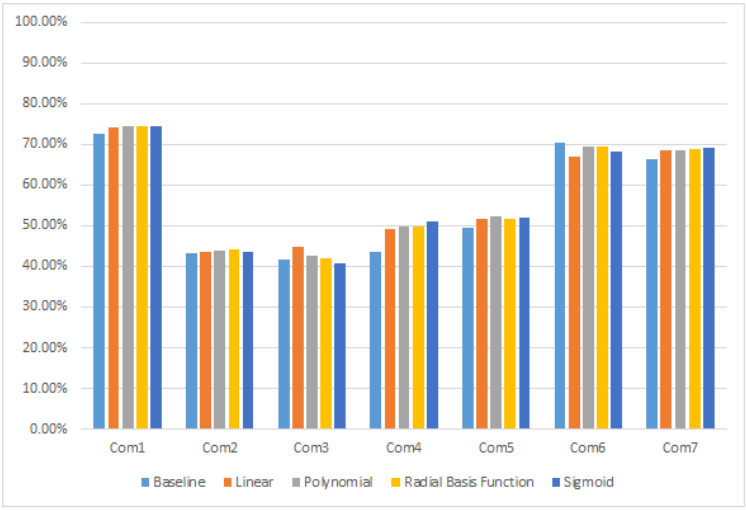

Figure 5.1: Hit rate achieved from classifier prediction with 6 features and using training data with n=5000 window size

that using 7 features seems more promising to improve the hit rate in all combinations.

(2) The kernel that gives the best improvement is not the same throughout combinations. It is observed that there is no one perfect kernel function for all benchmarks. The kernel function suitable for each benchmark depends on the workload and should be fine tuned for each benchmark to achieve the best result. However, we can imply from the results in Table 5.6 that for the 6 features training data, the Polynomial kernel function gives the best average improvement for all benchmark combinations. While with 7 features, the best average improvement for all benchmarks is from the Linear kernel function, the results are shown in Table 5.7.

(3) An observation on the analysis of the results shows that the hit rate improvement

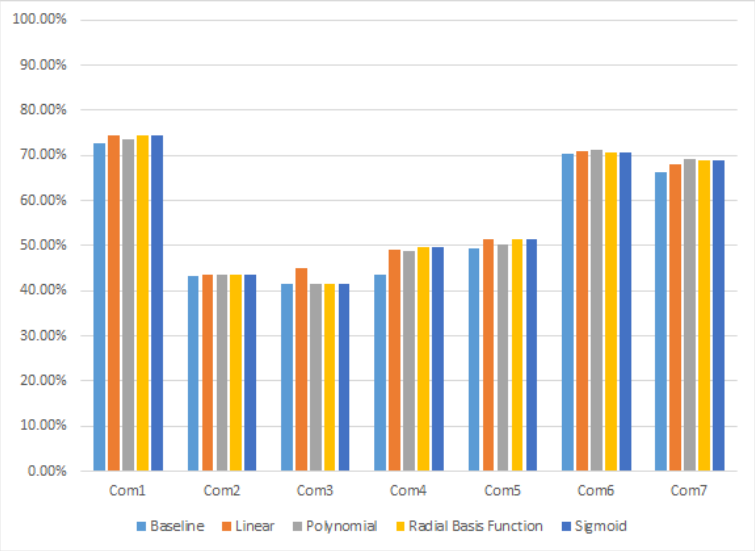

Figure 5.2: Hit rate achieved from classifier prediction with 7 features and using training data with n=5000 window size

Table 5.6: The percentage improved from different kernel functions using 6 features and training data with n=5000 window size.

	Com1	Com2	Com3	Com4	Com5	Com6	Com7	Average
Linear	1.94%	0.32%	7.60%	12.90%	4.67%	-4.81%	3.46%	3.73%
Polynomial	2.38%	1.38%	2.62%	14.53%	5.72%	-1.45%	3.59%	4.11%
RBF	2.41%	1.68%	1.06%	14.53%	4.69%	-1.42%	4.12%	3.87%
Sigmoid	2.31%	0.35%	-1.92%	16.99%	5.20%	-3.11%	4.40%	3.46%

by the SVM classifier does not seem to depend on the baseline hit rate nor the number of LLC accesses. Instead, it appears to depend on the ratio between the two quantities. Fig. 5.3 shows hit rate improvement in ascending order, the hit rate and the number of LLC accesses (scaled down for better visualization). The graph visualized that the hit rate increases when the ratio between the two quantities decreases.

In conclusion, using the SVM classifier may not be useful for (a) the high number of LLC accesses with poor hit rate. This could cause by the workload that shows no locality of reference or a large amount of data are never reused. (b) the small number of LLC accesses with high hit rate. This type of workload already shows strong locality of reference which traditional replacement policy is already sufficed to ensure high performance. Consequently, our SVM classifier is useful for the trace with the characteristic between (a) and (b), where reusable data normally evicted due to never-reuse or less-reuse data

Table 5.7: The percentage improved from different kernel functions using 7 features and training data with n=5000 window size.

	Com1	Com2	Com3	Com4	Com5	Com6	Com7	Average
Linear	2.30%	0.39%	8.05%	12.72%	4.00%	0.47%	2.93%	4.41%
Polynomial	1.21%	0.23%	0.02%	11.73%	1.54%	0.98%	4.30%	2.86%
RBF	2.30%	0.60%	0.00%	13.84%	4.21%	0.33%	4.02%	3.61%
Sigmoid	2.31%	0.39%	-0.38%	14.00%	4.15%	0.33%	4.02%	3.54%

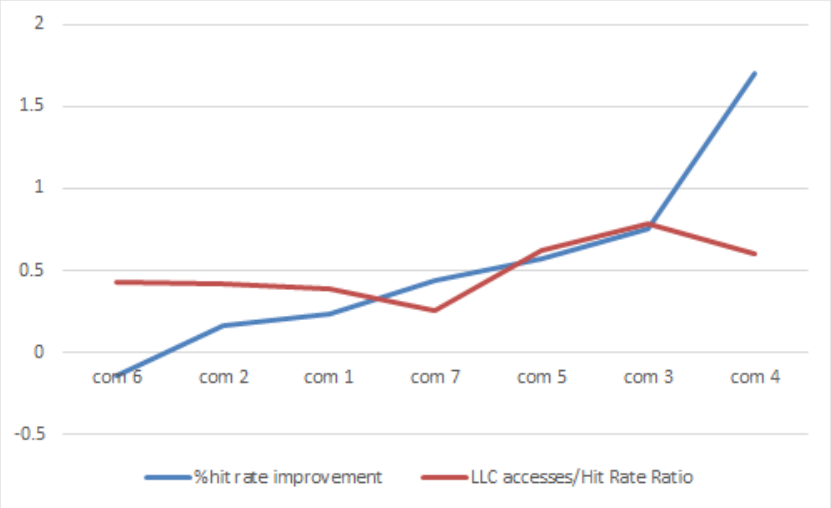

Figure 5.3: Best percentage improved compared with the ratio of the LLC accesses and the baseline hit rate (both scaled down for visualization)

in the workload or the working set size is frequently larger than the cache size and cause cache thrashing most of the time.

In comparison to previous methods, cache bypassing methods at the beginning are focused on the private cache on single core. Jalminger and Stenström (2003) present an approach to determine bypass by augmenting each cache entry with reuse history bits, and using a counter to determine whether or not to bypass data from L1. Their approach is only experimented on L1 (private) cache for single-core systems, reducing the miss rate by an average of 12%.

Other bypassing methods also use counters as a method of prediction, some based the prediction on the Instruction that requesting the datum (Tyson et al., 1995; Piquet et al., 2007; Li et al., 2012; Khan et al., 2014), the others based on the address of the data itself (Johnson et al., 1999; Jalminger and Stenström, 2003; Kharbutli and Solihin,

2008; Kharbutli et al., 2013). Our approach utilizes both PC and address and also other parameters to help the predictor both instruction-aware and thread-aware.

More complete implementations are presented in Piquet et al. (2007); Kharbutli and Solihin (2008); Li et al. (2012); Kharbutli et al. (2013); Khan et al. (2014) where bypassing occurs in the last-level cache. Most methods based on counters to predict cache bypass but using different approach in updating the counter. Piquet et al. (2007) update the counter every time the block is evicted from LLC. If the block was never accessed after allocated in the LLC, the counter is incremented. If it has been accessed while in the cache before eviction, the counter resets to zero. When a block should be allocated on cache, the counter is consulted and if the counter is saturated, the block will be bypassed from cache. The Piquet et al.counters method is applied to predict bypass for read requests in Khan et al. (2014) by bypassing all write requests without prediction.

The counter by Kharbutli and Solihin increment by the number of times the block is re-accessed after it was placed in the cache. If the counter is zero, it means that the block was never reused. Each block's counter is then stored to be looked up in the future, assuming that it would always have the same behavior. The bypass decision is made when the same block is going to be allocated on the cache again, if the history counter is zero then the block will be bypassed from cache. Kharbutli and Solihin (2008) achieve a miss rate reduction of 19%, using a form of per-cache line counters, which is also the approach taken in Kharbutli et al. (2013). In SCIP (Kharbutli et al., 2013), the counter has changed to bypass more data from the LLC, as a result of the temporal locality inverted in last-level cache. SCIP bypassed approximately 3 out of 4 data which is around 75% of data and achieved 18% speed up.

Li et al. (2012) update the counter when the incoming block matches the IB-VB pair in the table. The counter is used to remember whether the incoming block or the victim block was used earlier and bypass the one that will be used later. Similar to all of the aforementioned methods, it requires two tables stored on-chip cache and update the tables every cache accesses. In contrast, our approach is based on machine learning, and we achieve a miss rate reduction in average of 5.34%. Although the miss rate decrease is smaller, it suffices to show the suitability of SVM for bypass prediction. In addition, the bypass mechanism generated by the SVM (the predictor) is of a different nature than

the related work, which applies a mechanism on each cache line. When the cache size is larger, the conventional methods would have to expand and scale according to the cache size. It is different from our approach that our predictor would not grow along with the size of the cache as only one predictor per cache is required. It can also be modified to replace any counters on any methods to perform as a predictor only.

The only disadvantage from our predictor is that it required information from the core which may consume higher bandwidth from the bus but the implementation detail can be explored later. To summarize, our work demonstrated the feasibility of SVM prediction (kernel function separation of bypass and not-bypass data), leading the way to research on hardware implementation and optimization of SVM generated predictors.

Table 5.8: Result of Com1 using 6 features from window size n=2000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	1794208	1336207	1235183	**68.84**	437049	24.36
Linear	2	0.0025	1794208	1343544	1231823	**68.66**	441681	24.62
Linear	4	0.025	1794208	1348151	1231375	**68.63**	442167	24.64
Linear	8	0.007813	1794208	1335470	1235187	**68.84**	436948	24.35
Linear	16	0.25	1794208	1341456	1232665	**68.70**	440494	24.55
Linear	32	0.25	1794208	1339481	1233852	**68.77**	438983	24.47
Polynomial	0.5	0.000488	1794208	1598797	1103678	**61.51**	599808	33.43
Polynomial	2	0.0025	1794208	1654846	1124181	**62.66**	578538	32.24
Polynomial	4	0.025	1794208	1289799	1236457	**68.91**	434633	24.22
Polynomial	8	0.007813	1794208	1639829	1138527	**63.46**	560739	31.25
Polynomial	16	0.25	1794208	1457667	1181915	**65.87**	507235	28.27
Polynomial	32	0.25	1794208	1508545	1153093	**64.27**	542719	30.25
RBF	0.5	0.000488	1794208	1299668	1239522	**69.08**	426677	23.78
RBF	2	0.0025	1794208	1289591	1249590	**69.65**	419790	23.40
RBF	4	0.025	1794208	1302647	1221181	**68.06**	441986	24.63
RBF	8	0.007813	1794208	1285608	1259538	**70.20**	395627	22.05
RBF	16	0.25	1794208	1255928	1254377	**69.91**	411492	22.93
RBF	32	0.25	1794208	1382578	1243677	**69.32**	426892	23.79
Sigmoid	0.5	0.000488	1794208	1453961	1218431	**67.91**	460863	25.69
Sigmoid	2	0.0025	1794208	1359177	1220990	**68.05**	455227	25.37
Sigmoid	4	0.025	1794208	1759620	1117579	**62.29**	588417	32.80
Sigmoid	8	0.007813	1794208	1340989	1207430	**67.30**	469259	26.15
Sigmoid	16	0.25	1794208	1759284	1139415	**63.51**	564651	31.47
Sigmoid	32	0.25	1794208	1759160	1139799	**63.53**	564206	31.45

Table 5.9: Result of Com1 using 6 features from window size n=5000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	1794208	560091	1326889	**73.95**	229093	12.77
Linear	2	0.0025	1794208	523709	1328696	**74.05**	220118	12.27
Linear	4	0.025	1794208	523809	1328710	**74.06**	220127	12.27
Linear	8	0.007813	1794208	544973	1327625	**74.00**	224690	12.52
Linear	16	0.25	1794208	524446	1328675	**74.05**	220224	12.27
Linear	32	0.25	1794208	523588	1328694	**74.05**	220094	12.27
Polynomial	0.5	0.000488	1794208	1264729	1081365	**60.27**	609089	33.95
Polynomial	2	0.0025	1794208	523656	1334311	**74.37**	204305	11.39
Polynomial	4	0.025	1794208	615590	1322028	**73.68**	240738	13.42
Polynomial	8	0.007813	1794208	627784	1322807	**73.73**	248969	13.88
Polynomial	16	0.25	1794208	425599	1329917	**74.12**	183567	10.23
Polynomial	32	0.25	1794208	419747	1329745	**74.11**	182090	10.15
RBF	0.5	0.000488	1794208	529710	1333501	**74.32**	212697	11.85
RBF	2	0.0025	1794208	496887	1334394	**74.37**	209325	11.67
RBF	4	0.025	1794208	581503	1321231	**73.64**	230440	12.84
RBF	8	0.007813	1794208	583277	1326990	**73.96**	235983	13.15
RBF	16	0.25	1794208	503474	1334157	**74.36**	207446	11.56
RBF	32	0.25	1794208	505235	1334624	**74.39**	208866	11.64
Sigmoid	0.5	0.000488	1794208	552022	1328609	**74.05**	222078	12.38
Sigmoid	2	0.0025	1794208	527387	1328709	**74.06**	219272	12.22
Sigmoid	4	0.025	1794208	801957	1288501	**71.81**	292013	16.28
Sigmoid	8	0.007813	1794208	521294	1333519	**74.32**	212806	11.86
Sigmoid	16	0.25	1794208	1520475	1189219	**66.28**	479716	26.74
Sigmoid	32	0.25	1794208	1520213	1189290	**66.28**	479591	26.73

Table 5.10: Result of Com1 using 6 features from window size n=10000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	1794208	470279	1330661	**74.16**	187022	10.42
Linear	2	0.0025	1794208	327539	1322906	**73.73**	132767	7.40
Linear	4	0.025	1794208	1662376	1095954	**61.08**	590262	32.90
Linear	8	0.007813	1794208	406062	1332195	**74.25**	167163	9.32
Linear	16	0.25	1794208	470106	1332796	**74.28**	197946	11.03
Linear	32	0.25	1794208	413219	1332056	**74.24**	166465	9.28
Polynomial	0.5	0.000488	1794208	596735	1218447	**67.91**	335910	18.72
Polynomial	2	0.0025	1794208	208681	1318371	**73.48**	82814	4.62
Polynomial	4	0.025	1794208	336783	1326573	**73.94**	133110	7.42
Polynomial	8	0.007813	1794208	340757	1326991	**73.96**	134536	7.50
Polynomial	16	0.25	1794208	340971	1326201	**73.92**	136740	7.62
Polynomial	32	0.25	1794208	341574	1326149	**73.91**	136797	7.62
RBF	0.5	0.000488	1794208	397691	1331246	**74.20**	162397	9.05
RBF	2	0.0025	1794208	438142	1329870	**74.12**	183448	10.22
RBF	4	0.025	1794208	352584	1326071	**73.91**	138918	7.74
RBF	8	0.007813	1794208	391917	1328568	**74.05**	158189	8.82
RBF	16	0.25	1794208	363639	1332206	**74.25**	143732	8.01
RBF	32	0.25	1794208	369324	1332501	**74.27**	147434	8.22
Sigmoid	0.5	0.000488	1794208	447917	1330197	**74.14**	179530	10.01
Sigmoid	2	0.0025	1794208	362260	1330229	**74.14**	158175	8.82
Sigmoid	4	0.025	1794208	368629	1327746	**74.00**	146205	8.15
Sigmoid	8	0.007813	1794208	485970	1334577	**74.38**	187752	10.46
Sigmoid	16	0.25	1794208	886107	1320531	**73.60**	228683	12.75
Sigmoid	32	0.25	1794208	886219	1320538	**73.60**	228713	12.75

Table 5.11: Result of Com1 using 7 features from window size n=5000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	1794208	485599	1333237	**74.31**	202859	11.31
Linear	2	0.0025	1794208	499682	1332198	**74.25**	208408	11.62
Linear	4	0.025	1794208	485041	1333241	**74.31**	202672	11.30
Linear	8	0.007813	1794208	504180	1331684	**74.22**	210212	11.72
Linear	16	0.25	1794208	509847	1331108	**74.19**	212131	11.82
Linear	32	0.25	1794208	494227	1332816	**74.28**	206080	11.49
Polynomial	0.5	0.000488	1794208	1364456	1234945	**68.83**	417737	23.28
Polynomial	2	0.0025	1794208	757130	1312890	**73.17**	228085	12.71
Polynomial	4	0.025	1794208	147873	1311772	**73.11**	56184	3.13
Polynomial	8	0.007813	1794208	266087	1317928	**73.45**	105719	5.89
Polynomial	16	0.25	1794208	230351	1319034	**73.52**	93288	5.20
Polynomial	32	0.25	1794208	221871	1318117	**73.47**	90725	5.06
RBF	0.5	0.000488	1794208	509419	1333306	**74.31**	208265	11.61
RBF	2	0.0025	1794208	296646	1322743	**73.72**	128012	7.13
RBF	4	0.025	1794208	48820	1306046	**72.79**	18835	1.05
RBF	8	0.007813	1794208	135679	1310694	**73.05**	52406	2.92
RBF	16	0.25	1794208	1794207	1077955	**60.08**	632557	35.26
RBF	32	0.25	1794208	1794208	1077955	**60.08**	632558	35.26
Sigmoid	0.5	0.000488	1794208	518285	1333533	**74.32**	210122	11.71
Sigmoid	2	0.0025	1794208	497886	1333002	**74.29**	206828	11.53
Sigmoid	4	0.025	1794208	1557465	1066919	**59.46**	627350	34.97
Sigmoid	8	0.007813	1794208	974252	1189000	**66.27**	449304	25.04
Sigmoid	16	0.25	1794208	1113018	1083038	**60.36**	559511	31.18
Sigmoid	32	0.25	1794208	1218151	1083029	**60.36**	576518	32.13

Table 5.12: Result of Com2 using 6 features from window size n=2000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	807811	571844	346772	**42.93**	334007	41.35
Linear	2	0.0025	807811	590632	346095	**42.84**	336073	41.60
Linear	4	0.025	807811	594928	345908	**42.82**	336803	41.69
Linear	8	0.007813	807811	580063	346515	**42.90**	334796	41.44
Linear	16	0.25	807811	575609	346626	**42.91**	334397	41.40
Linear	32	0.25	807811	586719	346261	**42.86**	335543	41.54
Polynomial	0.5	0.000488	807811	580055	346171	**42.85**	343719	42.55
Polynomial	2	0.0025	807811	582640	346143	**42.85**	344507	42.65
Polynomial	4	0.025	807811	455901	356137	**44.09**	299769	37.11
Polynomial	8	0.007813	807811	541708	348765	**43.17**	332753	41.19
Polynomial	16	0.25	807811	497636	354409	**43.87**	325323	40.27
Polynomial	32	0.25	807811	506672	353161	**43.72**	330502	40.91
RBF	0.5	0.000488	807811	541224	348015	**43.08**	331304	41.01
RBF	2	0.0025	807811	548532	348497	**43.14**	334118	41.36
RBF	4	0.025	807811	421501	351004	**43.45**	275301	34.08
RBF	8	0.007813	807811	415452	346764	**42.93**	265233	32.83
RBF	16	0.25	807811	503147	355323	**43.99**	326857	40.46
RBF	32	0.25	807811	486483	353504	**43.76**	314300	38.91
Sigmoid	0.5	0.000488	807811	543545	347834	**43.06**	331684	41.06
Sigmoid	2	0.0025	807811	583993	346439	**42.89**	335051	41.48
Sigmoid	4	0.025	807811	641705	343398	**42.51**	335178	41.49
Sigmoid	8	0.007813	807811	604802	345618	**42.78**	337999	41.84
Sigmoid	16	0.25	807811	706670	340600	**42.16**	340954	42.21
Sigmoid	32	0.25	807811	706218	340598	**42.16**	340762	42.18

Table 5.13: Result of Com2 using 6 features from window size n=5000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	807811	494803	351318	**43.49**	323413	40.04
Linear	2	0.0025	807811	503946	350648	**43.41**	324631	40.19
Linear	4	0.025	807811	496124	351246	**43.48**	323548	40.05
Linear	8	0.007813	807811	503889	350684	**43.41**	324603	40.18
Linear	16	0.25	807811	494924	351398	**43.50**	323313	40.02
Linear	32	0.25	807811	497966	351188	**43.47**	323613	40.06
Polynomial	0.5	0.000488	807811	540699	347948	**43.07**	332250	41.13
Polynomial	2	0.0025	807811	538513	348121	**43.09**	331748	41.07
Polynomial	4	0.025	807811	492434	355100	**43.96**	324630	40.19
Polynomial	8	0.007813	807811	489429	351739	**43.54**	324761	40.20
Polynomial	16	0.25	807811	498323	354776	**43.92**	328310	40.64
Polynomial	32	0.25	807811	498433	354771	**43.92**	328359	40.65
RBF	0.5	0.000488	807811	580068	345152	**42.73**	342315	42.38
RBF	2	0.0025	807811	518703	350453	**43.38**	325759	40.33
RBF	4	0.025	807811	433447	351167	**43.47**	283185	35.06
RBF	8	0.007813	807811	440451	349283	**43.24**	287235	35.56
RBF	16	0.25	807811	498333	356126	**44.09**	323985	40.11
RBF	32	0.25	807811	495230	355757	**44.04**	324076	40.12
Sigmoid	0.5	0.000488	807811	529822	348360	**43.12**	329289	40.76
Sigmoid	2	0.0025	807811	496616	351121	**43.47**	323832	40.09
Sigmoid	4	0.025	807811	574411	349430	**43.26**	315973	39.11
Sigmoid	8	0.007813	807811	490925	351502	**43.51**	322462	39.92
Sigmoid	16	0.25	807811	703191	334525	**41.41**	325820	40.33
Sigmoid	32	0.25	807811	704582	334388	**41.39**	326684	40.44

Table 5.14: Result of Com2 using 6 features from window size n=10000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	807811	491527	351757	**43.54**	322550	39.93
Linear	2	0.0025	807811	492697	351714	**43.54**	322658	39.94
Linear	4	0.025	807811	491906	351759	**43.54**	322570	39.93
Linear	8	0.007813	807811	488244	352007	**43.58**	321934	39.85
Linear	16	0.25	807811	491990	351777	**43.55**	322505	39.92
Linear	32	0.25	807811	490809	351876	**43.56**	322244	39.89
Polynomial	0.5	0.000488	807811	550745	347450	**43.01**	335130	41.49
Polynomial	2	0.0025	807811	544392	348047	**43.09**	334027	41.35
Polynomial	4	0.025	807811	496512	354986	**43.94**	326905	40.47
Polynomial	8	0.007813	807811	495215	350892	**43.44**	326725	40.45
Polynomial	16	0.25	807811	497787	354764	**43.92**	327999	40.60
Polynomial	32	0.25	807811	497711	354746	**43.91**	327991	40.60
RBF	0.5	0.000488	807811	495399	350539	**43.39**	325610	40.31
RBF	2	0.0025	807811	478897	352828	**43.68**	319259	39.52
RBF	4	0.025	807811	437496	351390	**43.50**	285277	35.31
RBF	8	0.007813	807811	450378	349814	**43.30**	294308	36.43
RBF	16	0.25	807811	497722	356899	**44.18**	322186	39.88
RBF	32	0.25	807811	501449	356604	**44.14**	323590	40.06
Sigmoid	0.5	0.000488	807811	491732	350781	**43.42**	324925	40.22
Sigmoid	2	0.0025	807811	491821	351599	**43.52**	322796	39.96
Sigmoid	4	0.025	807811	581517	348326	**43.12**	317604	39.32
Sigmoid	8	0.007813	807811	476356	352355	**43.62**	318362	39.41
Sigmoid	16	0.25	807811	695409	336691	**41.68**	324178	40.13
Sigmoid	32	0.25	807811	699187	336658	**41.68**	326074	40.37

Table 5.15: Result of Com2 using 7 features from window size n=5000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	807811	115057	351664	**43.53**	89372	11.06
Linear	2	0.0025	807811	119512	351618	**43.53**	92368	11.43
Linear	4	0.025	807811	122100	351571	**43.52**	93904	11.62
Linear	8	0.007813	807811	121129	351597	**43.52**	93318	11.55
Linear	16	0.25	807811	121806	351577	**43.52**	93721	11.60
Linear	32	0.25	807811	117688	351614	**43.53**	91202	11.29
Polynomial	0.5	0.000488	807811	557992	347082	**42.97**	336924	41.71
Polynomial	2	0.0025	807811	551556	347432	**43.01**	335024	41.47
Polynomial	4	0.025	807811	25864	351091	**43.46**	18433	2.28
Polynomial	8	0.007813	807811	387215	350595	**43.40**	259927	32.18
Polynomial	16	0.25	807811	0	350310	**43.37**	0	0.00
Polynomial	32	0.25	807811	0	350310	**43.37**	0	0.00
RBF	0.5	0.000488	807811	176966	351297	**43.49**	131572	16.29
RBF	2	0.0025	807811	24790	351292	**43.49**	18329	2.27
RBF	4	0.025	807811	73767	352367	**43.62**	51399	6.36
RBF	8	0.007813	807811	647002	343769	**42.56**	333653	41.30
RBF	16	0.25	807811	554566	321882	**39.85**	293676	36.35
RBF	32	0.25	807811	503695	320257	**39.65**	261739	32.40
Sigmoid	0.5	0.000488	807811	248407	349975	**43.32**	175322	21.70
Sigmoid	2	0.0025	807811	126227	351614	**43.53**	97184	12.03
Sigmoid	4	0.025	807811	229857	345140	**42.73**	79760	9.87
Sigmoid	8	0.007813	807811	151524	351447	**43.51**	113466	14.05
Sigmoid	16	0.25	807811	90346	344259	**42.62**	37564	4.65
Sigmoid	32	0.25	807811	88509	344329	**42.62**	36784	4.55

Table 5.16: Result of Com3 using 6 features from window size n=2000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	2262796	1826452	720668	**31.85**	1516852	67.03
Linear	2	0.0025	2262796	1811101	732088	**32.35**	1503419	66.44
Linear	4	0.025	2262796	1808120	735180	**32.49**	1499834	66.28
Linear	8	0.007813	2262796	1785248	738708	**32.65**	1495139	66.07
Linear	16	0.25	2262796	1801807	742545	**32.82**	1491536	65.92
Linear	32	0.25	2262796	1829935	748683	**33.09**	1485928	65.67
Polynomial	0.5	0.000488	2262796	2258835	587257	**25.95**	1675291	74.04
Polynomial	2	0.0025	2262796	2015709	734422	**32.46**	1502602	66.40
Polynomial	4	0.025	2262796	1870846	747094	**33.02**	1489613	65.83
Polynomial	8	0.007813	2262796	2002444	736916	**32.57**	1504305	66.48
Polynomial	16	0.25	2262796	2018310	727938	**32.17**	1520440	67.19
Polynomial	32	0.25	2262796	2010306	727409	**32.15**	1519677	67.16
RBF	0.5	0.000488	2262796	1899820	721447	**31.88**	1522768	67.30
RBF	2	0.0025	2262796	1830684	724791	**32.03**	1515126	66.96
RBF	4	0.025	2262796	1810467	754286	**33.33**	1474333	65.16
RBF	8	0.007813	2262796	1774634	747173	**33.02**	1486577	65.70
RBF	16	0.25	2262796	2007781	739462	**32.68**	1506013	66.56
RBF	32	0.25	2262796	2000909	754909	**33.36**	1486752	65.70
Sigmoid	0.5	0.000488	2262796	1923727	715621	**31.63**	1529242	67.58
Sigmoid	2	0.0025	2262796	1835499	721187	**31.87**	1517267	67.05
Sigmoid	4	0.025	2262796	1972823	786623	**34.76**	1436477	63.48
Sigmoid	8	0.007813	2262796	1698323	739445	**32.68**	1483955	65.58
Sigmoid	16	0.25	2262796	1935616	370107	**16.36**	1651712	72.99
Sigmoid	32	0.25	2262796	1935865	370038	**16.35**	1651950	73.00

Table 5.17: Result of Com3 using 6 features from window size n=5000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	2262796	1115428	1011265	**44.69**	786283	34.75
Linear	2	0.0025	2262796	1156567	1010718	**44.67**	831447	36.74
Linear	4	0.025	2262796	1172465	1006274	**44.47**	857151	37.88
Linear	8	0.007813	2262796	1280010	1002441	**44.30**	911099	40.26
Linear	16	0.25	2262796	1080414	1012593	**44.75**	789065	34.87
Linear	32	0.25	2262796	1292191	1001341	**44.25**	917708	40.56
Polynomial	0.5	0.000488	2262796	1513100	801098	**35.40**	1204867	53.25
Polynomial	2	0.0025	2262796	740737	965857	**42.68**	584985	25.85
Polynomial	4	0.025	2262796	1106039	821639	**36.31**	911501	40.28
Polynomial	8	0.007813	2262796	1081922	821745	**36.32**	892380	39.44
Polynomial	16	0.25	2262796	1669973	811563	**35.87**	1191576	52.66
Polynomial	32	0.25	2262796	1676284	808662	**35.74**	1198214	52.95
RBF	0.5	0.000488	2262796	564625	925382	**40.90**	416966	18.43
RBF	2	0.0025	2262796	1124457	933394	**41.25**	847926	37.47
RBF	4	0.025	2262796	712894	897677	**39.67**	518560	22.92
RBF	8	0.007813	2262796	1031968	951139	**42.03**	762077	33.68
RBF	16	0.25	2262796	1518041	850252	**37.58**	1271854	56.21
RBF	32	0.25	2262796	1499731	845487	**37.36**	1270135	56.13
Sigmoid	0.5	0.000488	2262796	582334	923060	**40.79**	531464	23.49
Sigmoid	2	0.0025	2262796	2262796	537783	**23.77**	1725013	76.23
Sigmoid	4	0.025	2262796	1984366	330883	**14.62**	1717161	75.89
Sigmoid	8	0.007813	2262796	1436035	856238	**37.84**	1042916	46.09
Sigmoid	16	0.25	2262796	2139012	345388	**15.26**	1821690	80.51
Sigmoid	32	0.25	2262796	1842901	787308	**34.79**	1402516	61.98

Table 5.18: Result of Com3 using 6 features from window size n=10000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	2262796	1044462	913762	**40.38**	627440	27.73
Linear	2	0.0025	2262796	1443400	965422	**42.67**	894462	39.53
Linear	4	0.025	2262796	1080271	943251	**41.69**	648951	28.68
Linear	8	0.007813	2262796	1650441	905489	**40.02**	1001699	44.27
Linear	16	0.25	2262796	1125427	1004381	**44.39**	851997	37.65
Linear	32	0.25	2262796	1883284	821645	**36.31**	1232829	54.48
Polynomial	0.5	0.000488	2262796	1934319	698341	**30.86**	1536599	67.91
Polynomial	2	0.0025	2262796	1506422	700145	**30.94**	1157197	51.14
Polynomial	4	0.025	2262796	653701	887210	**39.21**	516598	22.83
Polynomial	8	0.007813	2262796	638076	887529	**39.22**	509101	22.50
Polynomial	16	0.25	2262796	1477429	857703	**37.90**	1153374	50.97
Polynomial	32	0.25	2262796	1489704	860913	**38.05**	1154835	51.04
RBF	0.5	0.000488	2262796	220832	935250	**41.33**	174983	7.73
RBF	2	0.0025	2262796	506601	922685	**40.78**	391255	17.29
RBF	4	0.025	2262796	403957	925647	**40.91**	332300	14.69
RBF	8	0.007813	2262796	444566	929025	**41.06**	351876	15.55
RBF	16	0.25	2262796	1415282	857056	**37.88**	1221207	53.97
RBF	32	0.25	2262796	1412706	857920	**37.91**	1223208	54.06
Sigmoid	0.5	0.000488	2262796	2099105	384601	**17.00**	1788147	79.02
Sigmoid	2	0.0025	2262796	1755462	812226	**35.89**	1220717	53.95
Sigmoid	4	0.025	2262796	2154002	337838	**14.93**	1841335	81.37
Sigmoid	8	0.007813	2262796	2145782	334817	**14.80**	1836665	81.17
Sigmoid	16	0.25	2262796	2142102	346305	**15.30**	1823016	80.56
Sigmoid	32	0.25	2262796	2130258	339451	**15.00**	1820307	80.45

Table 5.19: Result of Com3 using 7 features from window size n=5000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	2262796	773948	1015828	**44.89**	611594	27.03
Linear	2	0.0025	2262796	798737	1016452	**44.92**	630974	27.88
Linear	4	0.025	2262796	813070	1016483	**44.92**	642318	28.39
Linear	8	0.007813	2262796	820430	1016614	**44.93**	647686	28.62
Linear	16	0.25	2262796	819343	1016570	**44.93**	646759	28.58
Linear	32	0.25	2262796	820597	1016817	**44.94**	647959	28.64
Polynomial	0.5	0.000488	2262796	0	941117	**41.59**	0	0.00
Polynomial	2	0.0025	2262796	237	941107	**41.59**	225	0.01
Polynomial	4	0.025	2262796	9835	940977	**41.58**	7804	0.34
Polynomial	8	0.007813	2262796	3772	941425	**41.60**	3337	0.15
Polynomial	16	0.25	2262796	2066910	669581	**29.59**	1585844	70.08
Polynomial	32	0.25	2262796	2096034	684627	**30.26**	1571476	69.45
RBF	0.5	0.000488	2262796	5658	941057	**41.59**	4502	0.20
RBF	2	0.0025	2262796	0	941117	**41.59**	0	0.00
RBF	4	0.025	2262796	1187594	448766	**19.83**	1072945	47.42
RBF	8	0.007813	2262796	189271	916446	**40.50**	91769	4.06
RBF	16	0.25	2262796	2260401	554095	**24.49**	1708594	75.51
RBF	32	0.25	2262796	2259617	552281	**24.41**	1710401	75.59
Sigmoid	0.5	0.000488	2262796	1297456	874795	**38.66**	1168147	51.62
Sigmoid	2	0.0025	2262796	2208012	535267	**23.66**	1713387	75.72
Sigmoid	4	0.025	2262796	2262796	537783	**23.77**	1725013	76.23
Sigmoid	8	0.007813	2262796	2224144	457214	**20.21**	1795756	79.36
Sigmoid	16	0.25	2262796	2262796	537783	**23.77**	1725013	76.23
Sigmoid	32	0.25	2262796	205344	937511	**41.43**	123677	5.47

Table 5.20: Result of Com4 using 6 features from window size n=2000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	1635956	635621	806594	**49.30**	614839	37.58
Linear	2	0.0025	1635956	637398	806787	**49.32**	616172	37.66
Linear	4	0.025	1635956	635094	806716	**49.31**	614207	37.54
Linear	8	0.007813	1635956	636516	806545	**49.30**	615591	37.63
Linear	16	0.25	1635956	635993	806596	**49.30**	615105	37.60
Linear	32	0.25	1635956	636307	806547	**49.30**	615408	37.62
Polynomial	0.5	0.000488	1635956	670783	809807	**49.50**	619188	37.85
Polynomial	2	0.0025	1635956	664968	809823	**49.50**	619191	37.85
Polynomial	4	0.025	1635956	510943	817258	**49.96**	448858	27.44
Polynomial	8	0.007813	1635956	658575	809187	**49.46**	616970	37.71
Polynomial	16	0.25	1635956	218243	726680	**44.42**	140628	8.60
Polynomial	32	0.25	1635956	203685	726471	**44.41**	132814	8.12
RBF	0.5	0.000488	1635956	674637	810959	**49.57**	626302	38.28
RBF	2	0.0025	1635956	633747	806676	**49.31**	611904	37.40
RBF	4	0.025	1635956	421651	783089	**47.87**	349177	21.34
RBF	8	0.007813	1635956	397795	779945	**47.68**	330227	20.19
RBF	16	0.25	1635956	901366	832716	**50.90**	660454	40.37
RBF	32	0.25	1635956	896875	831303	**50.81**	660908	40.40
Sigmoid	0.5	0.000488	1635956	851254	818774	**50.05**	657606	40.20
Sigmoid	2	0.0025	1635956	638457	807061	**49.33**	617101	37.72
Sigmoid	4	0.025	1635956	556933	784824	**47.97**	518268	31.68
Sigmoid	8	0.007813	1635956	640314	806893	**49.32**	618833	37.83
Sigmoid	16	0.25	1635956	479824	810798	**49.56**	415062	25.37
Sigmoid	32	0.25	1635956	484620	811665	**49.61**	418117	25.56

Table 5.21: Result of Com4 using 6 features from window size n=5000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	1635956	625469	804487	**49.18**	603822	36.91
Linear	2	0.0025	1635956	625593	804489	**49.18**	603862	36.91
Linear	4	0.025	1635956	625590	804489	**49.18**	603862	36.91
Linear	8	0.007813	1635956	625529	804482	**49.18**	603838	36.91
Linear	16	0.25	1635956	625472	804469	**49.17**	603817	36.91
Linear	32	0.25	1635956	625418	804458	**49.17**	603792	36.91
Polynomial	0.5	0.000488	1635956	691650	810632	**49.55**	620965	37.96
Polynomial	2	0.0025	1635956	686582	810145	**49.52**	619256	37.85
Polynomial	4	0.025	1635956	616171	799985	**48.90**	589740	36.05
Polynomial	8	0.007813	1635956	756378	816188	**49.89**	637344	38.96
Polynomial	16	0.25	1635956	620867	800958	**48.96**	604700	36.96
Polynomial	32	0.25	1635956	621873	800878	**48.95**	605920	37.04
RBF	0.5	0.000488	1635956	708589	811051	**49.58**	626827	38.32
RBF	2	0.0025	1635956	832655	816051	**49.88**	646191	39.50
RBF	4	0.025	1635956	808285	813375	**49.72**	644748	39.41
RBF	8	0.007813	1635956	838902	816231	**49.89**	646978	39.55
RBF	16	0.25	1635956	86877	727072	**44.44**	72334	4.42
RBF	32	0.25	1635956	121311	731255	**44.70**	95584	5.84
Sigmoid	0.5	0.000488	1635956	747713	812054	**49.64**	633601	38.73
Sigmoid	2	0.0025	1635956	627381	804547	**49.18**	604550	36.95
Sigmoid	4	0.025	1635956	549387	833628	**50.96**	478444	29.25
Sigmoid	8	0.007813	1635956	621421	803902	**49.14**	602282	36.82
Sigmoid	16	0.25	1635956	52825	721113	**44.08**	38190	2.33
Sigmoid	32	0.25	1635956	38812	720346	**44.03**	29356	1.79

Table 5.22: Result of Com4 using 6 features from window size n=10000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	1635956	960349	783015	**47.86**	738197	45.12
Linear	2	0.0025	1635956	949786	784013	**47.92**	732322	44.76
Linear	4	0.025	1635956	965855	782889	**47.86**	741204	45.31
Linear	8	0.007813	1635956	950917	783460	**47.89**	733578	44.84
Linear	16	0.25	1635956	942547	792234	**48.43**	721434	44.10
Linear	32	0.25	1635956	949662	785635	**48.02**	730619	44.66
Polynomial	0.5	0.000488	1635956	917075	822289	**50.26**	665378	40.67
Polynomial	2	0.0025	1635956	931074	822164	**50.26**	671342	41.04
Polynomial	4	0.025	1635956	776940	798405	**48.80**	686023	41.93
Polynomial	8	0.007813	1635956	1000272	789433	**48.26**	745249	45.55
Polynomial	16	0.25	1635956	680202	806633	**49.31**	634167	38.76
Polynomial	32	0.25	1635956	683297	806632	**49.31**	635047	38.82
RBF	0.5	0.000488	1635956	958087	797168	**48.73**	717860	43.88
RBF	2	0.0025	1635956	970124	782771	**47.85**	739603	45.21
RBF	4	0.025	1635956	963754	797407	**48.74**	720563	44.05
RBF	8	0.007813	1635956	958474	795723	**48.64**	719598	43.99
RBF	16	0.25	1635956	253957	742467	**45.38**	171571	10.49
RBF	32	0.25	1635956	278641	745845	**45.59**	185744	11.35
Sigmoid	0.5	0.000488	1635956	961184	797522	**48.75**	717616	43.87
Sigmoid	2	0.0025	1635956	965294	783006	**47.86**	739676	45.21
Sigmoid	4	0.025	1635956	753889	822362	**50.27**	640284	39.14
Sigmoid	8	0.007813	1635956	916771	798343	**48.80**	709383	43.36
Sigmoid	16	0.25	1635956	1003915	795681	**48.64**	720815	44.06
Sigmoid	32	0.25	1635956	469131	791730	**48.40**	328276	20.07

Table 5.23: Result of Com4 using 7 features from window size n=5000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	1635956	621144	803311	**49.10**	602168	36.81
Linear	2	0.0025	1635956	621127	803309	**49.10**	602161	36.81
Linear	4	0.025	1635956	621159	803308	**49.10**	602178	36.81
Linear	8	0.007813	1635956	621162	803312	**49.10**	602182	36.81
Linear	16	0.25	1635956	621137	803306	**49.10**	602169	36.81
Linear	32	0.25	1635956	621154	803309	**49.10**	602175	36.81
Polynomial	0.5	0.000488	1635956	1011997	796138	**48.67**	728734	44.54
Polynomial	2	0.0025	1635956	1148872	755803	**46.20**	809262	49.47
Polynomial	4	0.025	1635956	602038	795558	**48.63**	586600	35.86
Polynomial	8	0.007813	1635956	1430177	633633	**38.73**	974376	59.56
Polynomial	16	0.25	1635956	77553	727379	**44.46**	59762	3.65
Polynomial	32	0.25	1635956	82409	727790	**44.49**	63577	3.89
RBF	0.5	0.000488	1635956	726022	811198	**49.59**	630493	38.54
RBF	2	0.0025	1635956	1241622	717045	**43.83**	840438	51.37
RBF	4	0.025	1635956	1485158	428440	**26.19**	1152328	70.44
RBF	8	0.007813	1635956	1201108	749202	**45.80**	765448	46.79
RBF	16	0.25	1635956	1632820	627117	**38.33**	1008567	61.65
RBF	32	0.25	1635956	1635581	552629	**33.78**	1083278	66.22
Sigmoid	0.5	0.000488	1635956	762082	812357	**49.66**	636501	38.91
Sigmoid	2	0.0025	1635956	621961	803470	**49.11**	602436	36.82
Sigmoid	4	0.025	1635956	988239	709330	**43.36**	576464	35.24
Sigmoid	8	0.007813	1635956	724357	809076	**49.46**	636216	38.89
Sigmoid	16	0.25	1635956	850126	688828	**42.11**	493713	30.18
Sigmoid	32	0.25	1635956	830509	687236	**42.01**	485335	29.67

Table 5.24: Result of Com5 using 6 features from window size n=2000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	2086273	1846063	1056563	**50.64**	772420	41.84
Linear	2	0.0025	2086273	1846073	1056564	**50.64**	772416	37.02
Linear	4	0.025	2086273	1845813	1056545	**50.64**	772358	37.02
Linear	8	0.007813	2086273	1846022	1056558	**50.64**	772407	37.02
Linear	16	0.25	2086273	1847405	1056617	**50.65**	772706	37.04
Linear	32	0.25	2086273	1851031	1056678	**50.65**	773488	37.08
Polynomial	0.5	0.000488	2086273	1998485	1052976	**50.47**	801531	38.42
Polynomial	2	0.0025	2086273	2006823	1053028	**50.47**	802159	38.45
Polynomial	4	0.025	2086273	1955147	1055516	**50.59**	795483	38.13
Polynomial	8	0.007813	2086273	1993856	1053696	**50.51**	801332	38.41
Polynomial	16	0.25	2086273	1726515	1085608	**52.04**	732734	35.12
Polynomial	32	0.25	2086273	1725504	1085941	**52.05**	732510	35.11
RBF	0.5	0.000488	2086273	1904486	1055558	**50.60**	789012	37.82
RBF	2	0.0025	2086273	1860572	1057540	**50.69**	776568	37.22
RBF	4	0.025	2086273	1951792	1059657	**50.79**	785641	37.66
RBF	8	0.007813	2086273	1930812	1055467	**50.59**	782831	37.52
RBF	16	0.25	2086273	1693858	1056354	**50.63**	701936	33.65
RBF	32	0.25	2086273	1611202	1062925	**50.95**	665991	31.92
Sigmoid	0.5	0.000488	2086273	1925789	1053327	**50.49**	793720	38.04
Sigmoid	2	0.0025	2086273	1856254	1057200	**50.67**	775037	37.15
Sigmoid	4	0.025	2086273	2086273	1046482	**50.16**	817404	39.18
Sigmoid	8	0.007813	2086273	1832497	1058638	**50.74**	766840	36.76
Sigmoid	16	0.25	2086273	2079011	1046987	**50.18**	815520	39.09
Sigmoid	32	0.25	2086273	2078714	1046972	**50.18**	815494	39.09

Table 5.25: Result of Com5 using 6 features from window size n=5000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	2086273	1378727	1079649	**51.75**	676316	32.42
Linear	2	0.0025	2086273	1378703	1079643	**51.75**	676303	32.42
Linear	4	0.025	2086273	1378613	1079644	**51.75**	676300	32.42
Linear	8	0.007813	2086273	1378908	1079626	**51.75**	676321	32.42
Linear	16	0.25	2086273	1378534	1079564	**51.75**	676357	32.42
Linear	32	0.25	2086273	1400501	1079436	**51.74**	678469	32.52
Polynomial	0.5	0.000488	2086273	1213501	1062003	**50.90**	584808	28.03
Polynomial	2	0.0025	2086273	1481144	1073972	**51.48**	662110	31.74
Polynomial	4	0.025	2086273	1598113	1081238	**51.83**	708363	33.95
Polynomial	8	0.007813	2086273	1479830	1074222	**51.49**	662015	31.73
Polynomial	16	0.25	2086273	1578136	1090556	**52.27**	686361	32.90
Polynomial	32	0.25	2086273	1577710	1090378	**52.26**	685470	32.86
RBF	0.5	0.000488	2086273	1438065	1076104	**51.58**	667483	31.99
RBF	2	0.0025	2086273	1524883	1081335	**51.83**	696680	33.39
RBF	4	0.025	2086273	1336542	1026449	**49.20**	558021	26.75
RBF	8	0.007813	2086273	1527960	1079831	**51.76**	692805	33.21
RBF	16	0.25	2086273	1455338	1056507	**50.64**	611987	29.33
RBF	32	0.25	2086273	1416930	1055605	**50.60**	600030	28.76
Sigmoid	0.5	0.000488	2086273	1429710	1074352	**51.50**	661259	31.70
Sigmoid	2	0.0025	2086273	1434500	1078975	**51.72**	681250	32.65
Sigmoid	4	0.025	2086273	1368349	1079211	**51.73**	646476	30.99
Sigmoid	8	0.007813	2086273	1343866	1077925	**51.67**	656455	31.47
Sigmoid	16	0.25	2086273	1639068	1085146	**52.01**	650529	31.18
Sigmoid	32	0.25	2086273	1641176	1085082	**52.01**	651099	31.21

Table 5.26: Result of Com5 using 6 features from window size n=10000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	2086273	1326023	1075139	**51.53**	652859	31.29
Linear	2	0.0025	2086273	1340902	1075833	**51.57**	656846	31.48
Linear	4	0.025	2086273	1285468	1070871	**51.33**	637526	30.56
Linear	8	0.007813	2086273	1337560	1075732	**51.56**	655775	31.43
Linear	16	0.25	2086273	1338898	1075726	**51.56**	656280	31.46
Linear	32	0.25	2086273	1290957	1071366	**51.35**	639782	30.67
Polynomial	0.5	0.000488	2086273	1246667	1069893	**51.28**	589942	28.28
Polynomial	2	0.0025	2086273	1325679	1065637	**51.08**	617635	29.60
Polynomial	4	0.025	2086273	1255823	1064978	**51.05**	594579	28.50
Polynomial	8	0.007813	2086273	1229568	1072854	**51.42**	583844	27.99
Polynomial	16	0.25	2086273	1464357	1085557	**52.03**	660501	31.66
Polynomial	32	0.25	2086273	1462917	1085189	**52.02**	658869	31.58
RBF	0.5	0.000488	2086273	1475926	1078508	**51.70**	669187	32.08
RBF	2	0.0025	2086273	1236873	1071022	**51.34**	617132	29.58
RBF	4	0.025	2086273	1051736	1034321	**49.58**	447173	21.43
RBF	8	0.007813	2086273	1051396	1062989	**50.95**	523447	25.09
RBF	16	0.25	2086273	1085963	1069031	**51.24**	451866	21.66
RBF	32	0.25	2086273	1209515	1040997	**49.90**	511806	24.53
Sigmoid	0.5	0.000488	2086273	1438140	1075346	**51.54**	657984	31.54
Sigmoid	2	0.0025	2086273	1328443	1075562	**51.55**	651377	31.22
Sigmoid	4	0.025	2086273	1197002	1075321	**51.54**	584568	28.02
Sigmoid	8	0.007813	2086273	1352131	1076352	**51.59**	659538	31.61
Sigmoid	16	0.25	2086273	691814	1095228	**52.50**	340451	16.32
Sigmoid	32	0.25	2086273	914410	1080965	**51.81**	469075	22.48

Table 5.27: Result of Com5 using 7 features from window size n=5000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	2086273	1271532	1072661	**51.42**	633224	30.35
Linear	2	0.0025	2086273	1270388	1072626	**51.41**	632856	30.33
Linear	4	0.025	2086273	1270559	1072665	**51.42**	632899	30.34
Linear	8	0.007813	2086273	1272169	1072699	**51.42**	633365	30.36
Linear	16	0.25	2086273	1272010	1072671	**51.42**	633354	30.36
Linear	32	0.25	2086273	1271654	1072686	**51.42**	633228	30.35
Polynomial	0.5	0.000488	2086273	1818485	1047256	**50.20**	765322	36.68
Polynomial	2	0.0025	2086273	2078092	1046403	**50.16**	815593	39.09
Polynomial	4	0.025	2086273	2086207	1046492	**50.16**	817313	39.18
Polynomial	8	0.007813	2086273	2084350	1046241	**50.15**	816256	39.13
Polynomial	16	0.25	2086273	2080968	1045877	**50.13**	816285	39.13
Polynomial	32	0.25	2086273	2082253	1045965	**50.14**	816413	39.13
RBF	0.5	0.000488	2086273	1390927	1074847	**51.52**	654191	31.36
RBF	2	0.0025	2086273	2085949	1046421	**50.16**	817259	39.17
RBF	4	0.025	2086273	1489935	1050864	**50.37**	562822	26.98
RBF	8	0.007813	2086273	2058355	1040749	**49.89**	811313	38.89
RBF	16	0.25	2086273	739968	1052880	**50.47**	264712	12.69
RBF	32	0.25	2086273	744255	1054203	**50.53**	264672	12.69
Sigmoid	0.5	0.000488	2086273	1427979	1074133	**51.49**	660907	31.68
Sigmoid	2	0.0025	2086273	1267485	1072356	**51.40**	631294	30.26
Sigmoid	4	0.025	2086273	1392103	1073146	**51.44**	647103	31.02
Sigmoid	8	0.007813	2086273	1087881	1069622	**51.27**	569919	27.32
Sigmoid	16	0.25	2086273	420611	1024404	**49.10**	154594	7.41
Sigmoid	32	0.25	2086273	420345	1024414	**49.10**	154494	7.41

Table 5.28: Result of Com6 using 6 features from window size n=2000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	1998467	628227	1395053	**69.81**	280490	14.04
Linear	2	0.0025	1998467	624724	1395017	**69.80**	279370	13.98
Linear	4	0.025	1998467	635493	1395053	**69.81**	282811	14.15
Linear	8	0.007813	1998467	643052	1395069	**69.81**	285209	14.27
Linear	16	0.25	1998467	618874	1395145	**69.81**	277370	13.88
Linear	32	0.25	1998467	645600	1395016	**69.80**	285993	14.31
Polynomial	0.5	0.000488	1998467	703156	1394348	**69.77**	298471	14.93
Polynomial	2	0.0025	1998467	686395	1394674	**69.79**	295115	14.77
Polynomial	4	0.025	1998467	634389	1395160	**69.81**	278358	13.93
Polynomial	8	0.007813	1998467	682996	1394769	**69.79**	294398	14.73
Polynomial	16	0.25	1998467	612330	1398889	**70.00**	267344	13.38
Polynomial	32	0.25	1998467	609205	1398900	**70.00**	266675	13.34
RBF	0.5	0.000488	1998467	671910	1394341	**69.77**	291513	14.59
RBF	2	0.0025	1998467	697386	1392752	**69.69**	297601	14.89
RBF	4	0.025	1998467	612619	1398495	**69.98**	269025	13.46
RBF	8	0.007813	1998467	677290	1394567	**69.78**	291603	14.59
RBF	16	0.25	1998467	633893	1399102	**70.01**	275694	13.80
RBF	32	0.25	1998467	623538	1399629	**70.04**	271410	13.58
Sigmoid	0.5	0.000488	1998467	723378	1391612	**69.63**	305706	15.30
Sigmoid	2	0.0025	1998467	649932	1394904	**69.80**	286835	14.35
Sigmoid	4	0.025	1998467	643423	1403502	**70.23**	269779	13.50
Sigmoid	8	0.007813	1998467	623341	1395113	**69.81**	278141	13.92
Sigmoid	16	0.25	1998467	1003427	1395483	**69.83**	328825	16.45
Sigmoid	32	0.25	1998467	1047840	1395087	**69.81**	345417	17.28

Table 5.29: Result of Com6 using 6 features from window size n=5000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	1998467	990294	1340528	**67.08**	524041	26.22
Linear	2	0.0025	1998467	990538	1340541	**67.08**	524035	26.22
Linear	4	0.025	1998467	989780	1340483	**67.08**	526990	26.37
Linear	8	0.007813	1998467	990085	1340489	**67.08**	523976	26.22
Linear	16	0.25	1998467	989235	1340665	**67.08**	523593	26.20
Linear	32	0.25	1998467	988711	1340821	**67.09**	523185	26.18
Polynomial	0.5	0.000488	1998467	1012689	1341681	**67.14**	523098	26.17
Polynomial	2	0.0025	1998467	1022408	1340961	**67.10**	525460	26.29
Polynomial	4	0.025	1998467	918720	1345090	**67.31**	484290	24.23
Polynomial	8	0.007813	1998467	1025958	1340913	**67.10**	525832	26.31
Polynomial	16	0.25	1998467	812600	1388184	**69.46**	407060	20.37
Polynomial	32	0.25	1998467	813702	1388170	**69.46**	407144	20.37
RBF	0.5	0.000488	1998467	1004724	1341164	**67.11**	524810	26.26
RBF	2	0.0025	1998467	985181	1344686	**67.29**	517753	25.91
RBF	4	0.025	1998467	703901	1350581	**67.58**	357042	17.87
RBF	8	0.007813	1998467	783050	1360611	**68.08**	394947	19.76
RBF	16	0.25	1998467	785530	1388526	**69.48**	411118	20.57
RBF	32	0.25	1998467	809265	1375308	**68.82**	429069	21.47
Sigmoid	0.5	0.000488	1998467	1012709	1341234	**67.11**	525789	26.31
Sigmoid	2	0.0025	1998467	989819	1340518	**67.08**	523998	26.22
Sigmoid	4	0.025	1998467	1105672	1340715	**67.09**	533231	26.68
Sigmoid	8	0.007813	1998467	991726	1340601	**67.08**	524230	26.23
Sigmoid	16	0.25	1998467	716880	1364827	**68.29**	259583	12.99
Sigmoid	32	0.25	1998467	716877	1364813	**68.29**	259602	12.99

Table 5.30: Result of Com6 using 6 features from window size n=10000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	1998467	608163	1396183	**69.86**	270119	13.52
Linear	2	0.0025	1998467	606733	1396292	**69.87**	269525	13.49
Linear	4	0.025	1998467	609475	1396113	**69.86**	270497	13.54
Linear	8	0.007813	1998467	608184	1396430	**69.88**	269714	13.50
Linear	16	0.25	1998467	605568	1396959	**69.90**	268299	13.43
Linear	32	0.25	1998467	608163	1396981	**69.90**	268996	13.46
Polynomial	0.5	0.000488	1998467	532098	1403347	**70.22**	243277	12.17
Polynomial	2	0.0025	1998467	622370	1400687	**70.09**	267458	13.38
Polynomial	4	0.025	1998467	585018	1399622	**70.03**	258508	12.94
Polynomial	8	0.007813	1998467	641840	1396270	**69.87**	279496	13.99
Polynomial	16	0.25	1998467	443787	1403711	**70.24**	219801	11.00
Polynomial	32	0.25	1998467	442388	1403731	**70.24**	219204	10.97
RBF	0.5	0.000488	1998467	643935	1395361	**69.82**	282598	14.14
RBF	2	0.0025	1998467	604420	1395711	**69.84**	270149	13.52
RBF	4	0.025	1998467	552157	1401388	**70.12**	248294	12.42
RBF	8	0.007813	1998467	554712	1400444	**70.08**	250268	12.52
RBF	16	0.25	1998467	547863	1417577	**70.93**	228494	11.43
RBF	32	0.25	1998467	511247	1417942	**70.95**	217710	10.89
Sigmoid	0.5	0.000488	1998467	657854	1395657	**69.84**	285706	14.30
Sigmoid	2	0.0025	1998467	614277	1395586	**69.83**	272915	13.66
Sigmoid	4	0.025	1998467	817678	1392347	**69.67**	323538	16.19
Sigmoid	8	0.007813	1998467	612471	1396046	**69.86**	271485	13.58
Sigmoid	16	0.25	1998467	711113	1383183	**69.21**	263036	13.16
Sigmoid	32	0.25	1998467	707383	1383412	**69.22**	261574	13.09

Table 5.31: Result of Com6 using 7 features from window size n=5000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	1998467	208185	1415184	**70.81**	121860	6.10
Linear	2	0.0025	1998467	211061	1415116	**70.81**	123294	6.17
Linear	4	0.025	1998467	213654	1414897	**70.80**	124627	6.24
Linear	8	0.007813	1998467	207679	1415103	**70.81**	121532	6.08
Linear	16	0.25	1998467	211739	1415030	**70.81**	123568	6.18
Linear	32	0.25	1998467	216890	1415198	**70.81**	126214	6.32
Polynomial	0.5	0.000488	1998467	161709	1422346	**71.17**	91632	4.59
Polynomial	2	0.0025	1998467	177724	1421808	**71.14**	99942	5.00
Polynomial	4	0.025	1998467	16402	1412939	**70.70**	15680	0.78
Polynomial	8	0.007813	1998467	186443	1421455	**71.13**	103659	5.19
Polynomial	16	0.25	1998467	77909	1419107	**71.01**	54086	2.71
Polynomial	32	0.25	1998467	77488	1418999	**71.00**	53721	2.69
RBF	0.5	0.000488	1998467	392727	1401976	**70.15**	191114	9.56
RBF	2	0.0025	1998467	17315	1413192	**70.71**	16981	0.85
RBF	4	0.025	1998467	13659	1412329	**70.67**	13529	0.68
RBF	8	0.007813	1998467	1023	1408419	**70.47**	918	0.05
RBF	16	0.25	1998467	1454336	1291694	**64.63**	590938	29.57
RBF	32	0.25	1998467	1441669	1282608	**64.18**	595158	29.78
Sigmoid	0.5	0.000488	1998467	509902	1398260	**69.97**	229327	11.48
Sigmoid	2	0.0025	1998467	240756	1405732	**70.71**	138167	6.91
Sigmoid	4	0.025	1998467	879370	1383844	**69.25**	365386	18.28
Sigmoid	8	0.007813	1998467	484584	1405267	**70.32**	231490	11.58
Sigmoid	16	0.25	1998467	161315	1398324	**69.97**	35410	1.77
Sigmoid	32	0.25	1998467	164393	1398006	**69.95**	36491	1.83

Table 5.32: Result of Com7 using 6 features from window size n=2000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	715070	488474	493274	**68.98**	166556	23.29
Linear	2	0.0025	715070	477479	493485	**69.01**	165590	23.16
Linear	4	0.025	715070	486858	493280	**68.98**	166450	23.28
Linear	8	0.007813	715070	479205	493440	**69.01**	165753	23.18
Linear	16	0.25	715070	478166	493474	**69.01**	165659	23.17
Linear	32	0.25	715070	478246	493479	**69.01**	165661	23.17
Polynomial	0.5	0.000488	715070	501995	493436	**69.01**	164966	23.07
Polynomial	2	0.0025	715070	501669	493343	**68.99**	164528	23.01
Polynomial	4	0.025	715070	487588	493114	**68.96**	167289	23.39
Polynomial	8	0.007813	715070	516509	493832	**69.06**	164781	23.04
Polynomial	16	0.25	715070	455916	494895	**69.21**	163010	22.80
Polynomial	32	0.25	715070	461397	494758	**69.19**	163438	22.86
RBF	0.5	0.000488	715070	481968	493543	**69.02**	166371	23.27
RBF	2	0.0025	715070	449202	494677	**69.18**	161785	22.63
RBF	4	0.025	715070	454228	494911	**69.21**	159759	22.34
RBF	8	0.007813	715070	450260	494948	**69.22**	159535	22.31
RBF	16	0.25	715070	461424	495351	**69.27**	161628	22.60
RBF	32	0.25	715070	458795	495901	**69.35**	160804	22.49
Sigmoid	0.5	0.000488	715070	553510	492068	**68.81**	169174	23.66
Sigmoid	2	0.0025	715070	476830	493373	**69.00**	165766	23.18
Sigmoid	4	0.025	715070	507955	493394	**69.00**	158552	22.17
Sigmoid	8	0.007813	715070	477501	493584	**69.03**	165455	23.14
Sigmoid	16	0.25	715070	620059	490365	**68.58**	167256	23.39
Sigmoid	32	0.25	715070	610445	490493	**68.59**	166240	23.25

Table 5.33: Result of Com7 using 6 features from window size n=5000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	715070	402645	489811	**68.50**	154696	21.63
Linear	2	0.0025	715070	400868	489831	**68.50**	154597	21.62
Linear	4	0.025	715070	393226	489688	**68.48**	153539	21.47
Linear	8	0.007813	715070	384798	489141	**68.40**	152110	21.27
Linear	16	0.25	715070	403564	489784	**68.49**	154776	21.64
Linear	32	0.25	715070	399427	489837	**68.50**	154414	21.59
Polynomial	0.5	0.000488	715070	559175	450814	**63.04**	207567	29.03
Polynomial	2	0.0025	715070	480384	448322	**62.70**	204745	28.63
Polynomial	4	0.025	715070	332874	490476	**68.59**	138111	19.31
Polynomial	8	0.007813	715070	335744	490284	**68.56**	138430	19.36
Polynomial	16	0.25	715070	178492	478852	**66.97**	79637	11.14
Polynomial	32	0.25	715070	196600	476851	**66.69**	88138	12.33
RBF	0.5	0.000488	715070	357488	488750	**68.35**	144428	20.20
RBF	2	0.0025	715070	338916	488578	**68.33**	144318	20.18
RBF	4	0.025	715070	292930	492955	**68.94**	125904	17.61
RBF	8	0.007813	715070	311651	491061	**68.67**	133566	18.68
RBF	16	0.25	715070	203873	490718	**68.63**	95577	13.37
RBF	32	0.25	715070	191762	487665	**68.20**	90248	12.62
Sigmoid	0.5	0.000488	715070	362831	488822	**68.36**	145320	20.32
Sigmoid	2	0.0025	715070	391753	489348	**68.43**	152794	21.37
Sigmoid	4	0.025	715070	4222227	494280	**69.12**	139472	19.50
Sigmoid	8	0.007813	715070	405648	490024	**68.53**	155472	21.74
Sigmoid	16	0.25	715070	693888	476601	**66.65**	190332	26.62
Sigmoid	32	0.25	715070	692373	475951	**66.56**	189602	26.52

Table 5.34: Result of Com7 using 6 features from window size n=10000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	715070	176268	493964	**69.08**	89719	12.55
Linear	2	0.0025	715070	354884	444743	**62.20**	180092	25.19
Linear	4	0.025	715070	193014	495038	**69.23**	89720	12.55
Linear	8	0.007813	715070	684876	450181	**62.96**	218008	30.49
Linear	16	0.25	715070	177266	495522	**69.30**	90177	12.61
Linear	32	0.25	715070	161389	493483	**69.01**	86245	12.06
Polynomial	0.5	0.000488	715070	335672	445462	**62.30**	175855	24.59
Polynomial	2	0.0025	715070	349094	444851	**62.21**	180064	25.18
Polynomial	4	0.025	715070	203460	495925	**69.35**	95502	13.36
Polynomial	8	0.007813	715070	192217	495401	**69.28**	89753	12.55
Polynomial	16	0.25	715070	157880	486371	**68.02**	71058	9.94
Polynomial	32	0.25	715070	156241	486264	**68.00**	70409	9.85
RBF	0.5	0.000488	715070	208038	495149	**69.24**	92470	12.93
RBF	2	0.0025	715070	211937	496050	**69.37**	96566	13.50
RBF	4	0.025	715070	182246	495642	**69.31**	93469	13.07
RBF	8	0.007813	715070	184233	495029	**69.23**	91408	12.78
RBF	16	0.25	715070	159161	492587	**68.89**	74478	10.42
RBF	32	0.25	715070	152084	488303	**68.29**	69221	9.68
Sigmoid	0.5	0.000488	715070	208235	495884	**69.35**	99511	13.92
Sigmoid	2	0.0025	715070	175139	492009	**68.81**	88112	12.32
Sigmoid	4	0.025	715070	427866	493625	**69.03**	138602	19.38
Sigmoid	8	0.007813	715070	189195	494826	**69.20**	93982	13.14
Sigmoid	16	0.25	715070	264746	483384	**67.60**	90175	12.61
Sigmoid	32	0.25	715070	257195	483154	**67.57**	87886	12.29

Table 5.35: Result of Com7 using 7 features from window size n=5000 training data

Kernel	C	gamma	#accesses	#predicted bypass	#Hits	Hitrate%	Bypassed	Bypassed %
Linear	0.5	0.000488	715070	108307	487293	**68.15**	64409	9.01
Linear	2	0.0025	715070	107091	487118	**68.12**	63793	8.92
Linear	4	0.025	715070	106836	487088	**68.12**	63659	8.90
Linear	8	0.007813	715070	107077	487117	**68.12**	63787	8.92
Linear	16	0.25	715070	107413	487165	**68.13**	63962	8.94
Linear	32	0.25	715070	108535	487319	**68.15**	64522	9.02
Polynomial	0.5	0.000488	715070	176685	493862	**69.06**	88109	12.32
Polynomial	2	0.0025	715070	487679	492693	**68.90**	162139	22.67
Polynomial	4	0.025	715070	266349	484769	**67.79**	106882	14.95
Polynomial	8	0.007813	715070	353059	489674	**68.48**	135391	18.93
Polynomial	16	0.25	715070	272675	487166	**68.13**	125964	17.62
Polynomial	32	0.25	715070	271043	487258	**68.14**	125457	17.54
RBF	0.5	0.000488	715070	142154	492488	**68.87**	79392	11.10
RBF	2	0.0025	715070	181427	481177	**67.29**	93535	13.08
RBF	4	0.025	715070	555616	492432	**68.86**	166693	23.31
RBF	8	0.007813	715070	689548	447477	**62.58**	215469	30.13
RBF	16	0.25	715070	715071	451366	**63.12**	220960	30.90
RBF	32	0.25	715070	715071	451366	**63.12**	220960	30.90
Sigmoid	0.5	0.000488	715070	141995	492448	**68.87**	79312	11.09
Sigmoid	2	0.0025	715070	109407	487436	**68.17**	64971	9.09
Sigmoid	4	0.025	715070	145634	477056	**66.71**	63739	8.91
Sigmoid	8	0.007813	715070	102288	486897	**68.09**	61484	8.60
Sigmoid	16	0.25	715070	352053	474650	**66.38**	124284	17.38
Sigmoid	32	0.25	715070	352608	474650	**66.38**	124525	17.41

CHAPTER VI

CONCLUSION

6.1 Dissertation summary

In this dissertation, we studed the possibility of improving the performance of the shared last-level cache by using the cache bypassing method. We proposed a novel method to predict cache reusability by using a machine learning tool, specifically the Support Vector Machine. A number of experiments have been conducted to find appropriate features, parameters and kernel functions, that affect the decision to bypass the data. In summary, we proved that the SVM is capable to learn the data classification and classify which data should be bypassed from the LLC. To the best of our knowledge, this is the first proof-of-concept of applying machine learning technique to caches or memory systems. Moreover, we propose a list of important attributes or features that affect the bypass decision and, also, the appropriate kernel function to be used. Lastly, the bypass method is a complementary technique that could be employed in conjunction with the aforementioned high performance cache replacement policies or cache prefetching to deliver a higher cache performance.

6.2 Limitations and future work

Our work has shown a preliminary result of the possibility to have machine learning enhanced the cache bypassing prediction. However, the SVM bypass prediction method we proposed and the benchmarks we tested have a few limitations that can be extended as follow:

- Firstly, our hardware resources limit a larger benchmark combinations, e.g. SPEC2006. Many memory intensive benchmarks can be tested to find other suitable kernel functions to implement bypass prediction for large workloads. It can be combined with multimedia benchmarks to test the possibility to detect larger mixed workloads and bypass appropriate data to improve the LLC performance.
- Secondly, fine tuning the SVM requires a lot of resources to implement. More experi-

ments with different parameters for each kernel function could achieve understanding about more suitable parameters specific to each benchmark.

- Lastly, hardware implementation of the classifier is an area that could be in the future work. The implementation of the classifier models into hardware classifier is another field of research that could be explored.

References

2010Memory technology evolution: an overview of system memory technologies Available from: http://h10032.www1.hp.com/ctg/Manual/c00256987.pdf [].

1996Pentium® pro processor with 1 mb l2 cache at 200 mhz Available from: http://www.intel.com/design/archives/processors/pro/docs/24357001.pdf [].

Baer, J.-L. and Chen, T.-F. 1991. An effective on-chip preloading scheme to reduce data access penalty. In Supercomputing, 1991. Supercomputing '91. Proceedings of the 1991 ACM/IEEE Conference on , pp. 176 –186.

Baer, J.-L. and Chen, T.-F. 1995. Effective hardware-based data prefetching for high-performance processors. IEEE Trans. Comput. 44.5 (May 1995): 609–623.

Belady, L. A. 1966. A study of replacement algorithms for a virtual-storage computer. IBM Syst. J. 5.2 (June 1966): 78–101.

Bernstein, D., Cohen, D., and Freund, A. 1995. Compiler techniques for data prefetching on the powerpc. In Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques, pp. 19–26. Manchester, UK, UK: IFIP Working Group on Algol.

Burges, C. J. C. 1998. A tutorial on support vector machines for pattern recognition. Data Min. Knowl. Discov. 2.2 (June 1998): 121–167.

Chen, T.-F. and Baer, J.-L. 1994. A performance study of software and hardware data prefetching schemes. SIGARCH Comput. Archit. News 22.2 (April 1994): 223–232.

Collins, J., Sair, S., Calder, B., and Tullsen, D. M. 2002. Pointer cache assisted prefetching. In Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture, pp. 62–73. Los Alamitos, CA, USA: IEEE Computer Society Press.

Cooksey, R., Jourdan, S., and Grunwald, D. 2002. A stateless, content-directed data prefetching mechanism. SIGARCH Comput. Archit. News 30.5 (October 2002): 279–290.

Cortes, C. and Vapnik, V. 1995. Support-vector networks. Machine Learning 20 (1995): 273–297.

Denning, P. J. 1972. On modeling program behavior. In Proceedings of the May 16-18, 1972, Spring Joint Computer Conference, pp. 937–944. New York, NY, USA: ACM.

Duong, N., Zhao, D., Kim, T., Cammarota, R., Valero, M., and Veidenbaum, A. V. 2012. Improving cache management policies using dynamic reuse distances. In Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 389–400. Washington, DC, USA: IEEE Computer Society.

Feng, M., Tian, C., Lin, C., and Gupta, R. 2011. Dynamic access distance driven cache replacement. ACM Trans. Archit. Code Optim. 8.3 (October 2011): 14:1–14:30.

Fu, J. W. C., Patel, J. H., and Janssens, B. L. 1992. Stride directed prefetching in scalar processors. SIGMICRO Newsl. 23.1-2 (December 1992): 102–110.

Ganusov, I. and Burtscher, M. 2005. Future execution: a hardware prefetching technique for chip multiprocessors. In Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on , pp. 350 – 360.

González, A., Aliagas, C., and Valero, M. 1995. A data cache with multiple caching strategies tuned to different types of locality. In International Conference on Supercomputing, pp. 338–347.

Hay, C. C., Schumacher, F. X., Kurpanek, G. P., Zheng, J., Keller, J. R., and Chan, K. K. 1996. Design of the HP PA 7200 CPU. (1996):

Horton, T. 1995. Selecting the right cache architecture for high performance pcs. In Electro/95 International. Professional Program Proceedings., pp. 111–122.

Ibáñez, P., Viñals, V., Briz, J. L., and Garzarán, M. J. 1998. Characterization and improvement of load/store cache-based prefetching. In Proceedings of the 12th international conference on Supercomputing, pp. 369–376. New York, NY, USA: ACM.

Jaleel, A., Theobald, K. B., Steely, S. C., Jr.., and Emer, J. 2010. High performance cache replacement using re-reference interval prediction (rrip). In Proceedings of

the 37th annual international symposium on Computer architecture, pp. 60–71. New York, NY, USA: ACM.

Jalminger, J. and Stenström, P. 2003. A Novel Approach to Cache Block Reuse Predictions. In International Conference on Parallel Processing, pp. 294–302.

Johnson, T. L., Connors, D. A., Merten, M. C., and mei W. Hwu, W. 1999. Run-Time Cache Bypassing. IEEE Transactions on Computers 48 (1999): 1338–1354.

Joseph, D. and Grunwald, D. 1997. Prefetching using markov predictors. SIGARCH Comput. Archit. News 25.2 (May 1997): 252–263.

Jouppi, N. P. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In Proceedings of the 17th annual international symposium on Computer Architecture, pp. 364–373. New York, NY, USA: ACM.

Kandiraju, G. B. and Sivasubramaniam, A. 2002. Going the distance for tlb prefetching: an application-driven study. SIGARCH Comput. Archit. News 30.2 (May 2002): 195–206.

Khan, S., Alameldeen, A., Wilkerson, C., Mutluy, O., and Jimenezz, D. 2014. Improving cache performance using read-write partitioning. In High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on, pp. 452–463.

Khan, S. M., Tian, Y., and Jimenez, D. A. 2010. Sampling dead block prediction for last-level caches. In Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 175–186. Washington, DC, USA: IEEE Computer Society.

Kharbutli, M., Jarrah, M., and Jararweh, Y. 2013. Scip: Selective cache insertion and bypassing to improve the performance of last-level caches. In Applied Electrical Engineering and Computing Technologies (AEECT), 2013 IEEE Jordan Conference on, pp. 1–6.

Kharbutli, M. and Solihin, Y. 2008. Counter-based cache replacement and bypassing algorithms. IEEE Trans. Comput. 57.4 (April 2008): 433–447.

Kim, S. and Veidenbaum, A. V. 1997. Stride-directed prefetching for secondary caches. In Proceedings of the international Conference on Parallel Processing, pp. 314–. Washington, DC, USA: IEEE Computer Society.

Lee, F. F. 1969. Study of "look-aside" memory. IEEE Trans. Comput. 18.11 (November 1969): 1062–1064.

Lee, J., Park, C., and Ha, S. 2003. Memory access pattern analysis and stream cache design for multimedia applications. In Proceedings of the 2003 Asia and South Pacific Design Automation Conference, pp. 22–27. New York, NY, USA: ACM.

Levinthal, D. 2010. Performance analysis guide for intel® core™ i7 processor and intel® xeon™ 5500 processors Available from: https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf [].

Li, L., Tong, D., Xie, Z., Lu, J., and Cheng, X. 2012. Optimal bypass monitor for high performance last-level caches. In Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, pp. 315–324. New York, NY, USA: ACM.

Lipasti, M. H., Schmidt, W. J., Kunkel, S. R., and Roediger, R. R. 1995. Spaid: software prefetching in pointer- and call-intensive environments. In Proceedings of the 28th annual international symposium on Microarchitecture, pp. 231–236. Los Alamitos, CA, USA: IEEE Computer Society Press.

Liptay, J. S. 1968. Structural aspects of the system/360 model 85: Ii the cache. IBM Syst. J. 7.1 (March 1968): 15–21.

Luk, C.-K. 2001. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. SIGARCH Comput. Archit. News 29.2 (May 2001): 40–51.

McFarling, S. 1992. Cache replacement with dynamic exclusion. In Proceedings of the 19th annual international symposium on Computer architecture, pp. 191–200. New York, NY, USA: ACM.

McKee, S. A. 2004. Reflections on the memory wall. In Proceedings of the 1st Conference on Computing Frontiers, pp. 162–. New York, NY, USA: ACM.

Moshovos, A., Pnevmatikatos, D. N., and Baniasadi, A. 2001. Slice-processors: an implementation of operation-based prediction. In Proceedings of the 15th international conference on Supercomputing, pp. 321–334. New York, NY, USA: ACM.

Mowry, T. C., Lam, M. S., and Gupta, A. 1992. Design and evaluation of a compiler algorithm for prefetching. SIGPLAN Not. 27.9 (September 1992): 62–73.

Mutlu, O., Stark, J., Wilkerson, C., and Patt, Y. N. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In Proceedings of the 9th International Symposium on High-Performance Computer Architecture, pp. 129–. Washington, DC, USA: IEEE Computer Society.

Oly, J. and Reed, D. A. 2002. Markov model prediction of i/o requests for scientific applications. In Proceedings of the 16th international conference on Supercomputing, pp. 147–155. New York, NY, USA: ACM.

Palacharla, S. and Kessler, R. E. 1994. Evaluating stream buffers as a secondary cache replacement. In Proceedings of the 21st annual international symposium on Computer architecture, pp. 24–33. Los Alamitos, CA, USA: IEEE Computer Society Press.

Piquet, T., Rochecouste, O., and Seznec, A. 2007. Exploiting single-usage for effective memory management. In Proceedings of the 12th Asia-Pacific Conference on Advances in Computer Systems Architecture, pp. 90–101. Berlin, Heidelberg: Springer-Verlag.

Qiao, F., Yu, B., Ma, J., Chen, T., and Hu, T. 2011. Slrf: A high-efficiency shared less reused filter in chip multiprocessors. In Proceedings of the 2011 Fourth International Conference on Intelligent Computation Technology and Automation - Volume 02, pp. 1191–1197. Washington, DC, USA: IEEE Computer Society.

Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C., and Emer, J. 2007. Adaptive insertion policies for high performance caching. SIGARCH Comput. Archit. News 35.2 (June 2007): 381–391.

Ramos, L., Ibanez, P., Vinals, V., and Llaberia, J. M. 2000. Modeling load address behaviour through recurrences. In Proceedings of the 2000 IEEE International

Symposium on Performance Analysis of Systems and Software, pp. 101–108. Washington, DC, USA: IEEE Computer Society.

Rivers, J. A. and Davidson, E. S. 1996. Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design. In International Conference on Parallel Processing 1, pp. 154–163.

Roth, A., Moshovos, A., and Sohi, G. S. 1998. Dependence based prefetching for linked data structures. SIGPLAN Not. 33.11 (October 1998): 115–126.

Sair, S., Sherwood, T., and Calder, B. 2002. Quantifying load stream behavior. In Proceedings of the 8th International Symposium on High-Performance Computer Architecture, pp. 197–. Washington, DC, USA: IEEE Computer Society.

Santhanam, V., Gornish, E. H., and Hsu, W.-C. 1997. Data prefetching on the hp pa-8000. In Proceedings of the 24th annual international symposium on Computer architecture, pp. 264–273. New York, NY, USA: ACM.

Sherwood, T., Sair, S., and Calder, B. 2000. Predictor-directed stream buffers. In Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture, pp. 42–53. New York, NY, USA: ACM.

Sklenar, I. 1992. Prefetch unit for vector operations on scalar computers (abstract). In Proceedings of the 19th annual international symposium on Computer architecture, pp. 430–. New York, NY, USA: ACM.

Smith, A. J. 1978. Sequential program prefetching in memory hierarchies. Computer 11.12 (December 1978): 7–21.

Smith, A. J. 1982. Cache Memories. ACM Computing Surveys 14 (1982): 473–530.

Somogyi, S., Wenisch, T. F., Ailamaki, A., and Falsafi, B. 2009. Spatio-temporal memory streaming. In Proceedings of the 36th Annual International Symposium on Computer Architecture, pp. 69–80. New York, NY, USA: ACM.

Tyson, G. S., Farrens, M. K., Matthews, J., and Pleszkun, A. R. 1995. A modified approach to data cache management. In International Symposium on Microarchitecture, pp. 93–103.

Ubal, R., Jang, B., Mistry, P., Schaa, D., and Kaeli, D. 2012. Multi2Sim: A Simulation Framework for CPU-GPU Computing . In Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques.

Vanderwiel, S. P. and Lilja, D. J. 2000. Data prefetch mechanisms. ACM Comput. Surv. 32.2 (June 2000): 174–199.

Vateekul, P., Kubat, M., and Sarinnapakorn, K. 2014. Hierarchical multi-label classification with svms: A case study in gene function prediction. Intelligent Data Analysis 18.4 (2014):

Wang, Z., Burger, D., McKinley, K. S., Reinhardt, S. K., and Weems, C. C. 2003. Guided region prefetching: a cooperative hardware/software approach. SIGARCH Comput. Archit. News 31.2 (May 2003): 388–398.

Wu, C.-J., Jaleel, A., Martonosi, M., Steely, S. C., Jr.., and Emer, J. 2011. Pacman: Prefetch-aware cache management for high performance caching. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 442–453. New York, NY, USA: ACM.

Xiang, L., Chen, T., Shi, Q., and Hu, W. 2009. Less reused filter: improving l2 cache performance via filtering less reused lines. In Proceedings of the 23rd international conference on Supercomputing, pp. 68–79. New York, NY, USA: ACM.

Yeh, T.-Y. and Patt, Y. N. 1991. Two-level adaptive training branch prediction. In Proceedings of the 24th annual international symposium on Microarchitecture, pp. 51–61. New York, NY, USA: ACM.

Zahran, M. 2007. Non-inclusion property in multi-level caches revisited.

Zhang, Z. and Torrellas, J. 1995. Speeding up irregular applications in shared-memory multiprocessors: memory binding and group prefetching. SIGARCH Comput. Archit. News 23.2 (May 1995): 188–199.

Zhou, H. 2005. Dual-core execution: building a highly scalable single-thread instruction window. In Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on , pp. 231 – 242.

Biography

Warisa Sritriratanarak was born in Bangkok, Thailand, on July, 1981. She graduated from Triamudom Suksa school in 1999. She received B.Eng. Computer Engineering, from Chulalongkorn University, Thailand, in 2003 and M.Sc. from California State University, Long Beach, USA, in 2006. She worked as a Lecturer at Bangkok University from 2006 to 2008. Her doctorate has been under the supervision of Prof. Prabhas Chongstitvatana and Asst. Prof. Mongkol Ekpanyapong. She was granted the CP Chulalongkorn Graduate Scholarship. During Oct 2012 - Aug 2013, she received an Erasmus Mundus scholarship Action 2 to perform an exchange study period at University of Minho, Portugal. Her field of interest includes various topics of Computer Architecture with emphasis on memory system, cache, and machine learning application on cache performance.