MEMORY SYSTEM OPTIMIZATIONS FOR CACHE MISS REDUCTION

by

Dindar Öz

BS, in CSE, Boğaziçi University, 2002

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2010

MEMORY SYSTEM OPTIMIZATIONS FOR CACHE MISS REDUCTION

APPROVED BY:

Prof. Dr. Oğuz Tosun (Thesis Supervisor)	.....
Asst. Prof. Dr. Alper Şen	.....
Assoc. Prof. Dr. Haluk Topçuoğlu	.....

DATE OF APPROVAL: 22.01.2010

ACKNOWLEDGEMENTS

Firstly, I would like to thank to my advisor Prof. Dr. Oğuz Tosun for his great support and encouragement. I also thank to Assoc. Prof. Dr. Haluk Topçuoğlu and Asst. Prof. Dr. Alper Şen for their support and serving my thesis committee.

I would like to thank to my friends, Fuat Geleri , Betül Demiröz, and Serdar Soran for their support and encouragement.

Finally, i am grateful to my family; my wife Işıl Öz, my mother , my father and my brothers for their love and great support during this study like as during my whole life.

ABSTRACT

MEMORY SYSTEM OPTIMIZATIONS FOR CACHE MISS REDUCTION

High processor utilization ,which has significant impact on the total system performance, is becoming the most popular target of many researchers in computer science. Since processors are extremely fast and much more expensive than other hardware components increasing their utilization is critical for efficiency of computer systems. One of the main reasons that cause processors to wait idle is memory stalls during a data or an instruction reference from the memory hierarchy. Although the fastest memory components and caching technologies are used to decrease access latency , the gap between memory system and cpu speed has been rapidly increasing. In addiditon to development of fast memory components which decrease miss latency and increase bandwidth, many techniques have been proposed to increase hit ratio. Prefetchers are one those which provides memory level parallelism by fetching blocks of data to the cache in advance of cpu requests hoping to increase cache hit ratio. Since caches are the fastest components in the memory hierarchy increasing hit ratio of the caches hides memory latency and therefore increase processor utilization. Scheduling threads according to their cache locality also increases data sharing and cache utilization in multithreaded systems. In this work, we focused these two areas of memory optimization techniques. We proposed a new hardware prefetcher model and a context switching heuristic among threads in multithreaded systems. We also implemented a multilevel cache simulator to test those ideas.

ÖZET

ÖNBELLEK BAŞARI ORANINI ARTTIRICI BELLEK SİSTEMİ OPTİMİZASYONLARI

Sistem performansı açısından çok önemli olan , işlemcilerin yüksek verimde kullanımı bilgisayar bilimindeki araştırmalarda oldukça popüler bir hedef haline geldi. Son derece hızlı ve pahalı olan işlemcileri mümkün olduğunca çalışır durumda tutmak bilgisayar sistemlerinin verimliliğini arrtıran en kritik faktörlerden birisidir. İşlemcilerin boş durumda beklemesine yol açan en önemli sebeplerden biri , bellekten herhangi bir veri yada komut okuma esnasında yaşanan gecikmelerdir. Bu gecikmeleri minimuma indirmek için en hızlı bellek bileşenleri kullanılsa bile, işlemci hızı ile bellek okuma yazma hızları arasındaki fark günden güne daha da açılmakta. Bu problemle ilgili, çalışmaların bir kısmı okuma yazma gecikmelerini azaltacak daha hızlı bileşenlerin üretilmesine yoğunlaşmış olmakla birlikte,diğer bir kısmı ise ön belleklerdeki isabet oranını arttırıcı teknikler geliştirmek üzerine devam etmekte. İşlemcinin bellekten okuyacağı bilgiyi daha öncesinden tahmin edip okumak ve bu şekilde bellek seviyesinde paralelizm sağlamak bu çalışmalardan birisi. Bu sayede önbellek isabet oranının yükseltilmesi ve önbellekler en hızlı bellek bileşeni oldukları için işlemci bekleme süresinin azaltılması hedeflenmektedir. Sistemde yapılacak iş parçacıklarının bellek kullanımları göz önüne alınarak çalıştırılma sırasının belirlenmesi bir diğer çalışma alanını oluşturmakta ve işler arası veri paylaşımını ve önbellek kullanımını arttırmayı amaçlamaktadır. Bu çalışmada bu iki çalışma alanıyla ilgili iki metod önerildi. Bu önerilerimizin denenmesi ve sonuçlarının incelenmesi amacıyla bir de çok katmanlı bellek simülatörü geliştirildi.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

CMP	Chip Multiprocessor
DSM	Distributed Shared Memory
ATP	Adaptive Temporal Prefetcher
PS	Pattern Selector
SK	Stream Knowledgebase
PE	Prefetcher Engine
PPA	Prefetcher Performance Analyzer
TSR	Total Success Ratio
RSR	Recent Success Ratio
LSR	Last Success Ratio
PL	Pattern Length
SL	Stream Length
AA	Adaptation Agility
PRC	Pattern Repetition Count

1. INTRODUCTION

1.1. Memory Hierarchy and Caches

Memory system stands at the core of the operation of a computer system and memory hierarchy is the most essential concept for development of the modern memory systems. Although , for simplicity, it is tempting to build a flat memory of a single technology a well-implemented memory hierarchy provides both the performance of the fastest component and the cost per bit of the cheapest and the least energy consuming one. Therefore, in process of designing memory systems , use of memory hierarchy has been very convenient since the early development of computer systems.

The aim of designing memory in a hierarchical manner can be summerized as to provide a memory system as fast as the fastest level with cost almost as the cheapest level of memory. In Figure 1.1 levels of memory hierarchy is depicted. The levels in the hierarchy are subset one another which means all data in each level can also be found in lower levels. the data addresses in each lower level, which is slower and larger, must be mapped to one upper level which are faster and smaller. This mapping and address checking is also one of the responsibilities of memory hierarchy.

Memory hierarchy has become more and more important as processor performance increased rapidly. In 1980s microprocessors were designed without caches whereas in 2001 two level on chip caches could be seen very often. Clearly, inspite of all the research on mitigating it , the gap between processor speed and the access speed of memory systems increasing. In Figure 1.2 starting with 1980 as a baseline the gap between performance of cpu an memory is plotted. Without highly engineered and sophisticated memory hierarchies and modern cache systems this gap would even be much larger.

As mentioned earlier, in memory hierarchy, from lower to the upper parts faster and more expensive components take place. The uppermost component is cache which

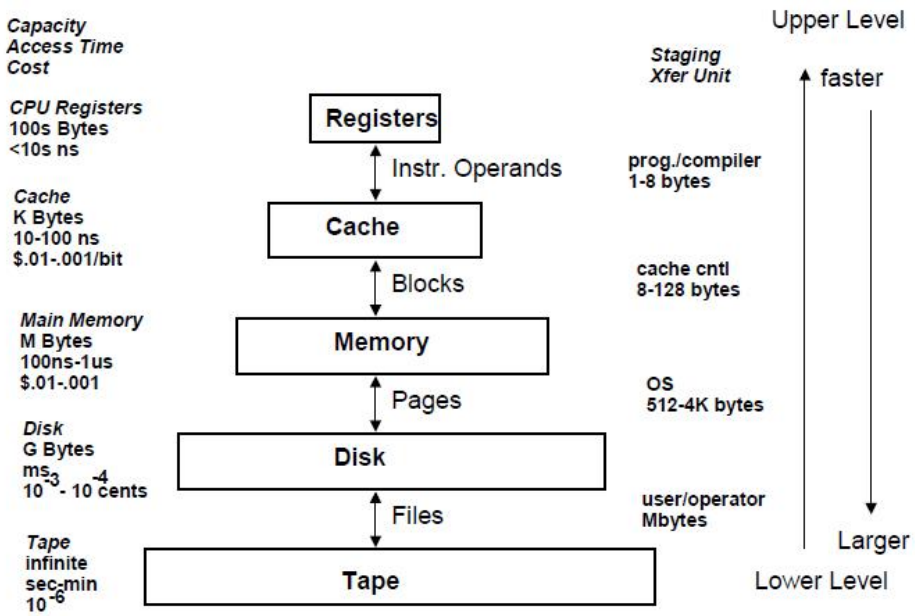

Figure 1.1. Memory Levels [10]

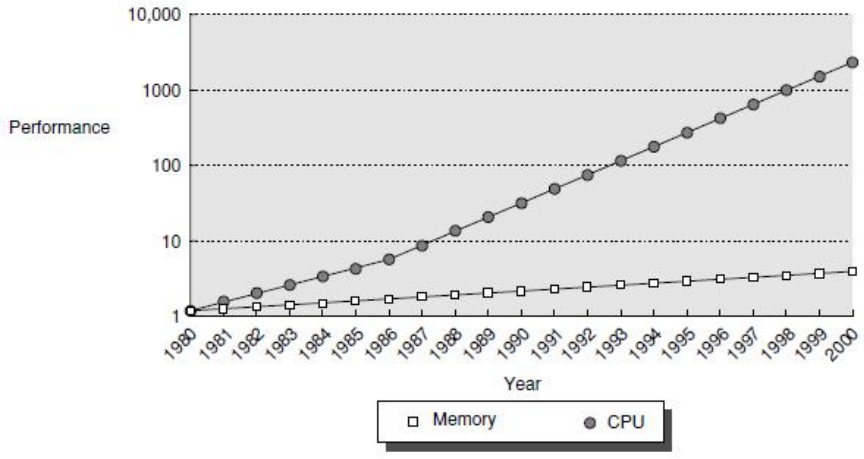

Figure 1.2. History of CPU and Memory Performance [10]

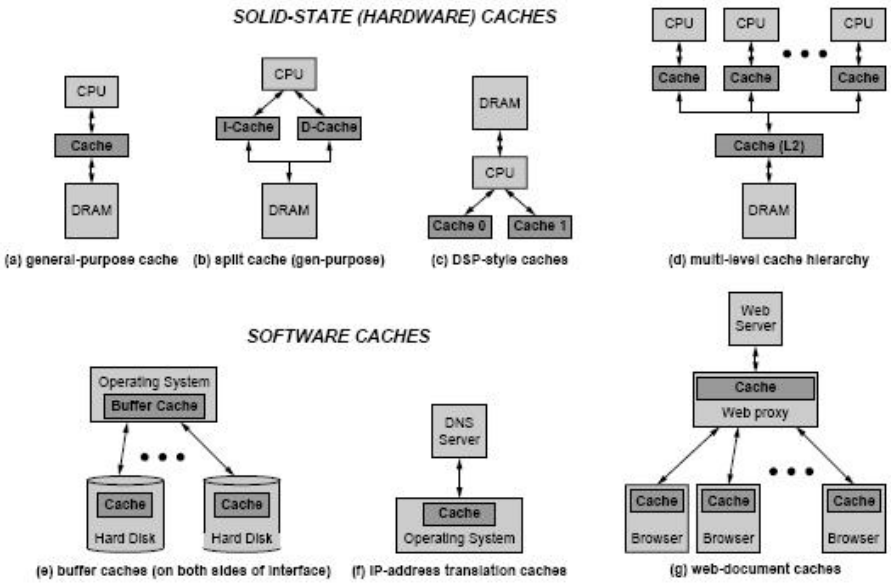

Figure 1.3. Applications of Cache [10]

provides extremely fast access and has limited area on the memory system. Not only in a memory system ,a cache can speed up accesses to all means of storage , including tapes, disk drives, software caches in operating systems ,servers on the network and even other caches. In Figure 1.3, different applications of caching concept can be seen. The idea is the principle of locality of reference, which is the tendency of applications to make reference a predictably small amount of data within a given window of time [Belady 1966, Denning 1970]. This principle implies that only small amount of the memory is used during an application's life cycle which is called working set. So caches , although its expense , need only be large enough to cover working set of applications.

1.2. Locality Principle

Starting from early days of computers, program behavior has been a research topic and computer engineers observed that programs do not access memory randomly. Accesses tend to repeat themselves in time and repeat in close region. The principle of exhibiting predictable, non-random memory-access behavior is called locality principle. The observation that if a memory address is referenced once then it is referenced in close future again is called temporal locality. The tendency of programs to make references near each other is called spatial locality. These principles are used to predict

future accesses of programs. Memory references has also some other regularities that help predictions. Some programs running special algorithms has memory references of regular patterns. The occurrence period of those patterns are too long to be considered as temporal locality and the addresses in the pattern are too away from each other to be considered as spatial locality. This can be considered as third kind of locality which is named "algorithmic locality" [9]. In the use of caches and data management heuristics these principles plays fundamental role. Program's tendency to use a data item over and over in a close time window,which is temporal locality, makes the idea of storing used data items in an easily accessible buffer for future use very efficient. If lower levels of memory hierarchy is main data store caches can be considered as that fast buffers. The tendency of programs to access memory in clusters, spatial locality, gives rise to the idea of retrieving data items as blocks of consecutive data items not just one by one.So using cache blocks of multiple words is the result of this spatial regularity of accesses.

1.3. Terminology

Throughout this work some memory system terminology is used which is briefly explained here. The name given to the first level(or levels in multilevel cache systems) is cache. Since taking advantage of locality has become very popular, the idea of buffering frequently used items in many other context is also called cache. File caches, web server caches are just a few examples. The address referenced by the working program first comes to caches just after leaving the cpu. If the referenced address can be found in current cache level it is called cache hit for that level. Upon a cache hit no further memory referencing is done , simply the data returned to the cpu. If the address is not in the cache it is called a cache miss. After a cache miss one lower level is searched in the same way. The data in caches is stored as fixed-size collection as mentioned before. This consecutive data items called as cache block. To renote , storing data as fixed-size collection of items exploits the spatial locality of memory references which tells that there is high probability that other parts of that block will be referenced soon. On a cache miss, the required minimum time to retrieve the first words of desired block from next level is called latency for that level. The time that

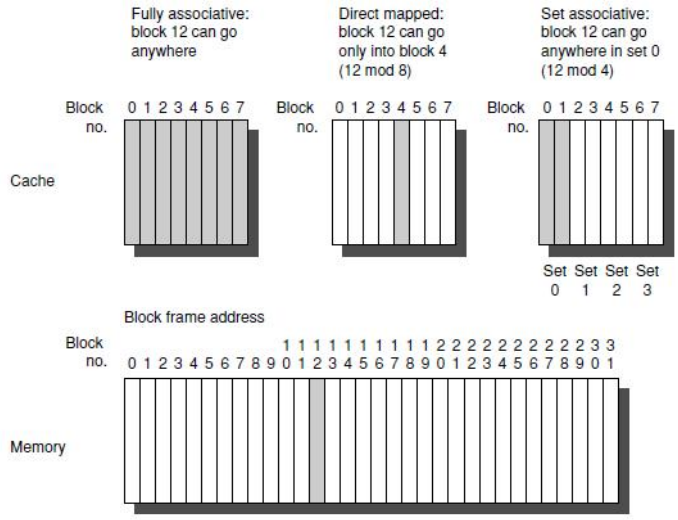

Figure 1.4. Cache classes according to the associativity [10]

passes through the retrieving of the whole block is determined by the bandwidth. If the processor working in-order fashion then it is stalled or paused by the operating system on misses. Caches can be categorized in many ways. Mainly according to the answer of three questions, they can be classified.

Firstly, where can a block be placed in the upper level? Direct mapped caches are the caches where each block has only one place to appear in. In the opposite case, where each block can appear anywhere in the cache it is called fully associative cache. In between, there is a third class according to block placement, which is set associative cache. In set associative cache, blocks can take place only in a restricted area , which is called set. Each block has their own set specified. To make it clear, we can think the first and the second class of caches as the extreme cases of set associative caches. If each set has only one block to be mapped, then it is direct mapped cache, and if there is only one set in the cache it becomes fully associative cache. Figure 1.4 summarizes the three cases via an example.

The second classifying question is which block is replaced on a cache miss. On a cache miss, one of the blocks in the cache should be selected as a victim to be replaced. If the cache is direct mapped, then there is nothing to decide on. The incoming block determines the outgoing one. Since each block has only one place to appear, the cache

block to be replaced is strictly determined just when the new block comes up. So direct mapped caches are simple to handle replacement of blocks. If the cache is fully associative then the selection problem arises. Although many different heuristics can be used , mainly three of them employed as replacement strategies. One is rondom replacement, which spreads allocation uniformly and involves no algorithm complexity. Since block are randomly selected, the statistics related to the performance of caches are nondeterministic in this case. The second aproach is to replace the unused block for the longest time, which is called Least Recently Used (LRU). This has the motivation arising from temporal locality and very sensible in many cases. However, LRU makes replacement job a little bit complicated since it requires storing of last usage time for each cache block at somewhere to determine the victim on replacement. To mitigate this complexity, there is a third popular method which is First in First out (FIFO). In FIFO replacement, the oldest block in the cache is replaced, without considering its last usage time. As it can be seen easily , this approach is more simple to implement comparing to LRU but it does not exploit locality as much as LRU does.

What happens on a write is the third question, which is called write policy. Write through and write back caches are two main classes corresponding to two different write policy. When a block is written into the cache, if it is simulataneously written to the lower level memory,it is called write through. If blocks are only written to the cache and they are not transferred to the memory until replacement, it is called write back policy. In write back scenario, cache blocks must be maintained with some freshness control bits, which is called dirty bit. Write back policy is harder to maintain and suffers from inconsistency problem but involves less write operation than write through policy. Multiple write operations on the same block results in multiple write signals in write through policy while in write back policy, only one write to the lower level is required if dirty bit is set, no write is necessary otherwise

To measure cache performance, some metrics are defined. The ratio of number of hits and total number of accesses is called cache hit rate and the ratio of the misses and total number of accesses is called cache miss rate. Hit and miss rates are also categorized into read hit rate , write hit rate, read miss rate and write miss rate. A

well known performance measure of memory hiearchy is average access time, which is expressed as follows.

$$Average\ memory\ access\ time = Hit\ time\ +\ Miss\ rate\ x\ Miss\ penalty$$

where hit time is the cache access time and miss penalty is the access latency in case of miss. Here looking at this formula, in order to improve memory system performance, hit time , miss rate or miss penalty can be reduced. Many researches on memory system design focus on decreasing one of these metrics.

Multi-level caches are mainly designed with motivation to reduce miss penalty. After a miss in the first level cache, instead of going to the memory , there are in between more levels of caches which are slower than the first level but much faster than the memory. Therefore with increasing the number of levels in the memory hierarchy, miss penalty can be significantly reduced. Using multilevel caches , since the upper levels gets faster and closer to the cpu, hit time also decreases, which in turn decreases average access time even more. Obviously increasing the number of levels ,besides those merits, requires extra hardware extra space and extra power. In modern memory systems, since multiple cpu and cores take place , memory hierarchy gets even more complex than just increasing the number of levels. To improve hit time and miss penalty in CMP, distributed memory hierachy models are introduced. in DSM(Distributed Shared Memory) systems, while some portion of the memory hierarchy are shared, there are specially allocated cache structures for each processor which can be considered as the first level of memory hierarchy. Although allocated caches introduce some coherence issues to the memory system , the gain stemming from reduced hit time can sufficiently compensate this complexity disadvantage.

Decreasing miss rate has become another crucial target of the studies on memory system design. There are many methods to reduce miss rate the most straightforward ones of which are using larger caches , larger cache block sizes, and higher associativity. Prefetchers can also be considered as an effort to reduce miss ratio. They are implemented both as software and hardware and the main idea is to provide Memory Level

Parallelism (MLP). By retrieving cache blocks in ahead of cpu requests , prefetchers , can significantly eliminate cache misses if they achieve highly correct prefetch ratio. Software prefetchers , usually implemented by inserting prefetching instructions to the programs source code which is statically determined before run time. On the other hand, hardware prefetchers try to exploit locality of references and discover repeating behaviour on the memory access pattern of programs dynamically at run time.

Tiling (also called blocking), a popular software optimization method [11], that reduces cache misses by restructuring a program to re-use certain blocks of data that fit in the cache. It can be either automatically done by compilers, or users can manually organize their programs.

In multi core , multi processor systems and multithreaded environments, increasing data sharing between cores and threads also increase hit rate. Clustering data sharing cores or threads together , locality aware thread allocation and scheduling are all very eligible methods currently employed in CMP(Chip MultiProcessor). Locality aware schedulers try to schedule threads to corun as they have geographically close working set on the memory. This improves data sharing and cache utilization.

In this work, we choose two of the methods as our focus and proposed two optimizations on each. Firstly , one result of our study is a new hardware prefetcher called Adaptive Temporal Prefetcher (ATP). Like many other hardware prefetchers , ATP stores repeating patterns in the memory references of programs to prefetch on further repetitions. The major difference from other proposed models , is the dynamic adaptation ability of ATP to changing memory bahaviour of different programs or even in different parts of a single program. The prefetcher adjusts its tendency to prefetch by adjusting prefetching parameters in order to adapt current programs memory access behaviour. By this, it reduces false prefetches decreasing prefetch tendency if prefetching is unavailable on current working application, does not show much of locality and on the other hand exploits the repeating patterns more if there is extremely regular access to the memory as in ,for example,heavy array processing of scientific applications.

Second optimization is about scheduling of threads in multithreaded applicaitons. We proposed a simple cache oriented heuristic to be used at context switching of previously scheduled threads. Like locality aware thread schedulers do, we try to maximize data sharing and henceforth decrease miss rate by switching to the thread that have most of its data ready in the cache. The method suggests dividing memory into fixed size regions and storing the last accessed region for threads. To replace the stalled or retired thread , the tread with the same last accessed region is selected. Therefore the optimization technique is called "region based context switching".

In order to check how effective is our proposed technique, we need to use a cache simulator. As we could not implement ATP and region based context switching on available cache simulators we developed our own simulator that we can easily modify and plug in new virtual memory system components and heuristics. In later sections , the cache simulator is described in more detail.

The rest of this work consists of section 2 the Related Work , section 3 Adaptive Temporal Prefetcher ,section 4 Methodology and section 5 Conclusions and suggested Future Work .

2. RELATED WORK

2.1. Prefetchers

Data prefetching has been used to provide memory level parallelism in many different ways for about fourty years. One of the very first attempts [1] was fetching multiple words as a block in one access from main memory into the cache which tries to make benefit from spatial locality of memory references in software behaviour. Afterwards prefetching of cache blocks in advance of cpu requests was implemented in the IBM 370 and AMdahl 470/V More recently, software-initiated prefetching are introduced.Porterfield [2] proposed the idea of a "cache load" instruction with several RISC implementations.

These techniques can be mainly categorized into two classes as Software-initiated and Hardware-initiated data prefetching. Also hybrid approaches that make use of both software and hardware support appear in some research more recently [6] [7] [8].

2.1.1. Software Initiated Prefetching

Having microprocessors that support a special form of fetch instruction is the main idea used in many sofware prefetching techniques. Fetches must be non-blocking allowing prefetches happen transparently not interfering with other memory operations. Moreover, in case of a failure a fetch instruction should not cause an exception or something else to interrupt the main program flow. Via this fetch instruction, placing the fetch calls in the program code in correct order and number is the essential work in software prefetching. This is called prefetch scheduling and , it is practically impossible to predict precisely when to schedule a prefetch so that the processor finds every block in the cache just when it requests.This stems from the different memory latencies of different hardware end the uncertainties in the program at compile time. Those fetch instructions can at best be placed by the programmer himself since he knows how the

program works best. However , programmers usually do not bother and compilers have to insert fetch instructions properly at compile-time.

Independent of who generated the prefetching instructions, sofware based initiation is most often used within loops of large array calculations. Loops usually have easily predictable array references which provides an excellent prefetching opportunity. It is just perfectly enough to establish the access pattern of the loop by examining loop iteration behaviour and to place fetch instruction in loop bodies accordingly.

In Table 2.1, it is basically shown how loop based prefetching can be adapted to a code easily. Further optimizations can also take place in order to make prefetching to be more precise. Since prefetching of the desired blocks has also some latency , it is important for the fetch instruction to be sufficiently ahead of the actual reference.

Table 2.1. Loop based prefetching

for $i = 0$ to N **do**
  $ip \leftarrow ip + a[i] * b[i]$
end for
for $i = 0$ to N **do**
  $fetch(\&a[i+1]);$
  $fetch(\&b[i+1]);$
  $ip \leftarrow ip + a[i] * b[i]$
end for

2.1.2. Hardware-Initiated Prefetching

Unlike software-initiated prefetching hardware-initiated prefetching does not require any programmer or compiler intervention. There are several hardware based tecnhiques proposed which generally take advantage of run-time information to discover possible prefetching opportunities.

In sequential prefetching , the spatial locality of memory accesses is exploited.

Just as using larger cache blocks (which can also be considered as a form of prefetching) , when a block requested from cache the adjacent blocks are prefetched in hope for a very soon access. Unlike enlargening the block size , this approach provides better granularity in replacement of cache blocks. With larger blocks, in case of capacity misses the replacing cache blocks also get larger. However in sequential prefetching , only fetching of the blocks is multiplied.

One-block loook ahead (OBL) [1] is the most primitive form of sequential prefetching. The implementations of OBL differ in the type of initiating access. In prefetch on miss algorithm, the block b+1 is prefetched just after the access of block b result in a cache miss.As for the tagged prefetch algorithm , every memory block is associated with a tag bit used to detect if a block is demand-fetched or a prefetched block is referenced for the first time. In both cases, the next sequential block is fetched. For a set of trace-driven simulations, %50 to %90 reduce in cache miss ratio with tagged prefetching is found by Smith [21]. As expected from the bahaviour of two algorithms , prefetch on miss and tagged prefetching , the formet is less than half as effective as the latter one. A regularly sequential access on an array will cause a cache miss for every other cache block in case of prefetch on miss algorithm whereas only one cache miss happens when tagged prefetching is used.

Some more complex prefetchers focus on the repeating patterns of the memory access both in time and in space to discover some prefetching opportunity. Locality principle is the main idea that stands behind most of modern hardware prefetchers.

Spatial Memory Streaming (SMS) [4] which is the prediction and streaming of spatially correlated access patterns to improve memory-level parallelism (MLP). In that work some terms have been defined to formalize the spatial correlation. Spatial Region is the name given to fixed-size portion of address space consisting of multiple consecutive cache blocks The time interval that SMS records accesses within a spatial record is called spatial region generation. The bit vector representing the set of blocks in a region accessed during a spatial region generation is called spatial pattern. Upon a trigger access, which is the first access that starts spatial region generation,

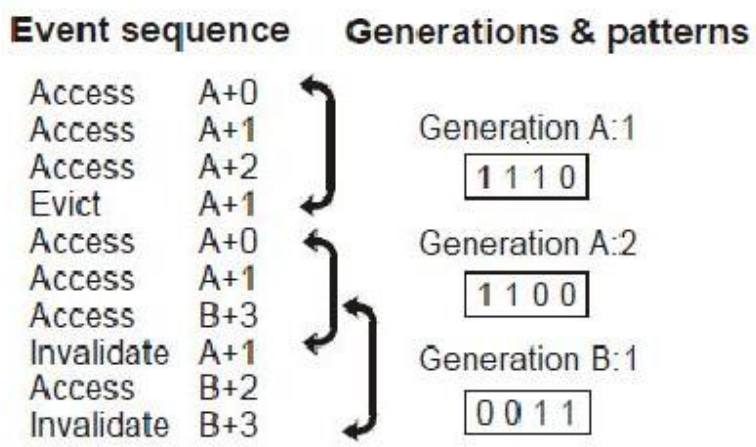

Figure 2.1. SMS Region Generation [4]

SMS predicts the spatial pattern that will be accessed over the course of the generation. Figure 2.1 shows the spatial region generations and their corresponding patterns. Previously observed patterns are stored in pattern history table for further use.

In another research titled TMS(Temporal Memory Streaming) [3] , coherent read misses on shared-memory multiprocessors are the main focus. They consider those misses consumes substantial fraction of execution time in many scientific and commercial workloads. Their study proposes Temporal Streaming (TMS) to eliminate coherent read misses by streaming data to a processor in advance of the corresponding memory accesses. TMS exploits two common phenomena in shared-memory access patterns including temporal address correlation ,which states that groups of shared addresses tend to be accessed together and in the same order and due to temporal stream locality, recently-accessed address streams are likely to recur. Figure 2.2 displays a temporal streaming in a DSM (Distributed shared memory). Node i incurs coherent read misses and records the sequence of misses A,B,C,D,E, Node j later misses on address B, and requests the data from the directory node. The directory node responds to this request through the baseline coherence mechanism and additionally requests a stream (following B) from the most recent consumer, Node i. Node i looks up address B in its order and assumes that requests to the subsequent addresses C,D,E are likely to follow. Thus, it forwards the stream C,D,E to Node j. Upon receipt of the stream, Node j retrieves the data for each block. Subsequent accesses to these addresses hit

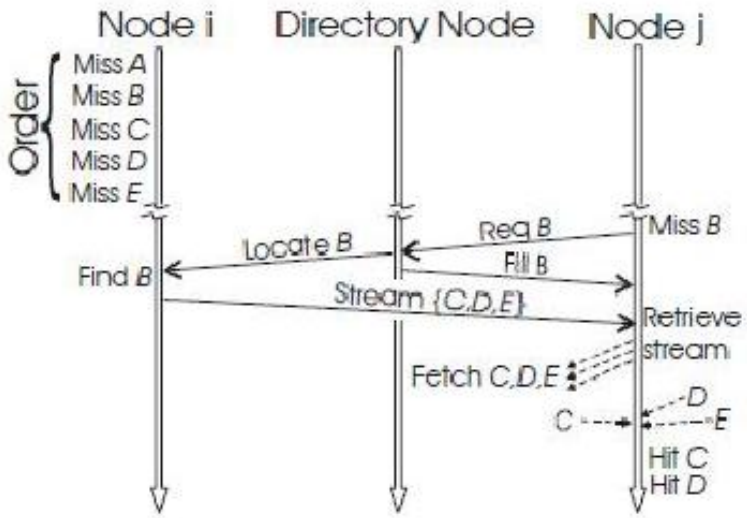

Figure 2.2. Temporal Streaming [3]

locally and avoid long-latency coherence misses.

TMS has three hardware structures. Coherent miss order buffer (CMOB) records nodes' miss order. There is a streaming engine which is called Temporal streaming engine (TSE). TSE locates stream and streams it to the processor. Possible stream alternatives for a specific pattern are stored in the Stream Value Buffer (SVB) In Figure 2.3 these components are depicted together with existing memory hierarchy.

To record the coherent read miss order, each node continuously appends the miss addresses, in program order, in its CMOB. Useful streamed blocks (i.e., resulting in accesses that hit in the SVB) are also recorded in the CMOB, as they replace coherent read misses that would have occurred without TSE. As misses are recorded, the recording node sends the corresponding CMOB pointer to the directory node for the block. The CMOB pointers stored in the directory allow TSE to find the correct CMOB locations efficiently given a stream head.TSE uses the information in each nodes CMOB to identify candidate addresses for streaming. When a node incurs a coherent read miss, TSE locates one or more streams on CMOBs across the system, and forwards them to the stream engine at the requesting node.

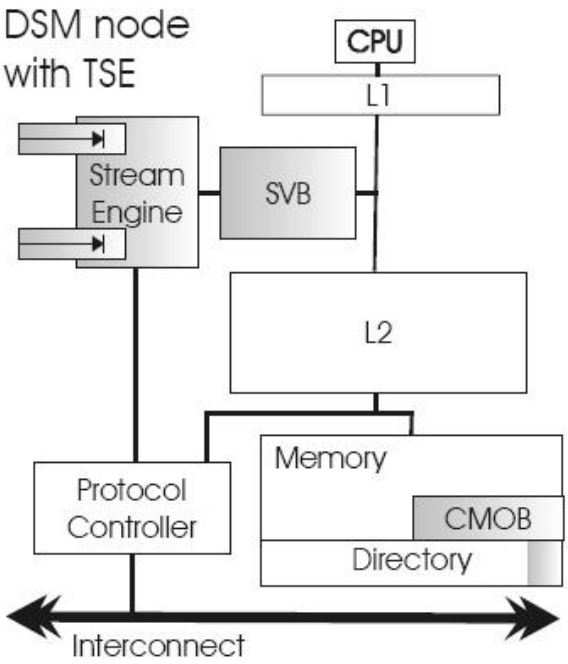

Figure 2.3. TMS Architecture [3]

The results of the experiments showed that temporal streaming has the potential to eliminate 98% of coherent read misses in scientific applications, and 43% to 60% in OLTP and web server applications. It gives speedups of 1.07 to 3.29 in scientific applications, and 1.06 to 1.21 in commercial workloads.

2.2. Thread Schedulers

Since computers started to work in multithreaded fashion , scheduling of threads has always been a targeted problem, which is NP-Complete for the time being. After development of multi processor and multicore systems , it bacame even more comlex and critical to order and distribute the working threads among the computational resources. In heterogenous and distributed architectures, thread allocation and scheduling can tremendously effect the total system performance. As a result of this complexity and importance many algorithms proposed until now [13], [14], [15], [16], [17].

List scheduling algorithms , assigns priority to threads, and keeps them in a list. They simply run threads according to the priorities of the threads.The priority is

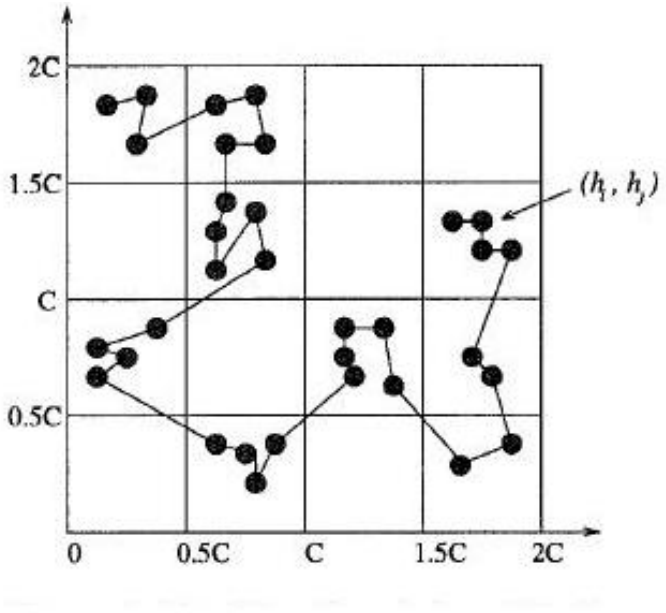

Figure 2.4. Clusters of Threads [18]

calculated based on different metrics such as dependency , process time, working set size etc.

In multiprocessor architecture, threads are handled as set of clusters which are supposed to run on the same proccessor or core. Firstly , clusters are formed and then mapped to processors.

Scheduling based on the data locality of threads is a technique used in compiler oriented methods [18], [19], [20]. Compiler optimization focus on the memory behaviour of the programs and use locality to improve code structure and thread granularity on compile time. Locality is also an important criteria in clustering algorithms. Threads are clustered according their data locality to improve data sharing and cache utilisation [19], [20]. The cooperative use of caches between threads called as constructive cache sharing in [20]. Multithreaded applications present significant opportunities for cunstructive cache sharing, where concurrently scheduled threads share largely overlapping working set. If threads are clustered according their working sets and clustered threads run concurrently , cache sharing gets improved.In Figure 2.4, thread clustering based on data locality is demonstrated.

3. PROPOSED SOLUTIONS

3.1. Adaptive Temporal Prefetcher (ATP)

Adaptive Temporal Prefetcher collects memory access information dynamically and stores repeating memory reference patterns in a table called pattern history table. When a specific patterns occurs more than certain times, the stream of addresses that is requested by the processor is stored in another hash table which is named Stream Knowledbase for future use. After storing a stream that follows a specific pattern in konwledgebase, if that patterns occurs again, the related stream is prefetched. The main difference of ATP from previously suggested techniques is the adaptation ability that provides dynamic modification of prefetch parameters.

As it is shown at Figure 3.1 there are mainly four components in the prefetcher architecture which are described in more detail in later sections.These are namely Pattern Selector (PS), Stream KnowledgeBase (SK), Prefetch Engine (PE) and Prefetcher Performance Analyzer (PPA).

3.1.1. Pattern Selector (PS)

Prefetch Engine keeps consecutive N memory references as access pattern in pattern selector. N is the pattern length which is one of the prefetch parameters that can be adjusted dynamically. In PS each access pattern is stored with repeat count as it is depicted in Figure 3.2. Any reoccurrences of an access pattern increases the repeat count of that pattern in PS. If repeat count exceeds Pattern Repetition Threshold, Pattern Selector decides that it is a usual pattern that must be learned and makes Prefetch Engine (PE)enter the learning state. At the learning state, PE waits for a number of access requests before storing them as a stream. This number is called fetch distance. Because prefetching itself has also some latency , storing the stream just after the pattern occurrence would result in late fetches which cause cache pollution. To prevent this, prefetcher must be ahead of the cpu with enough number of memory

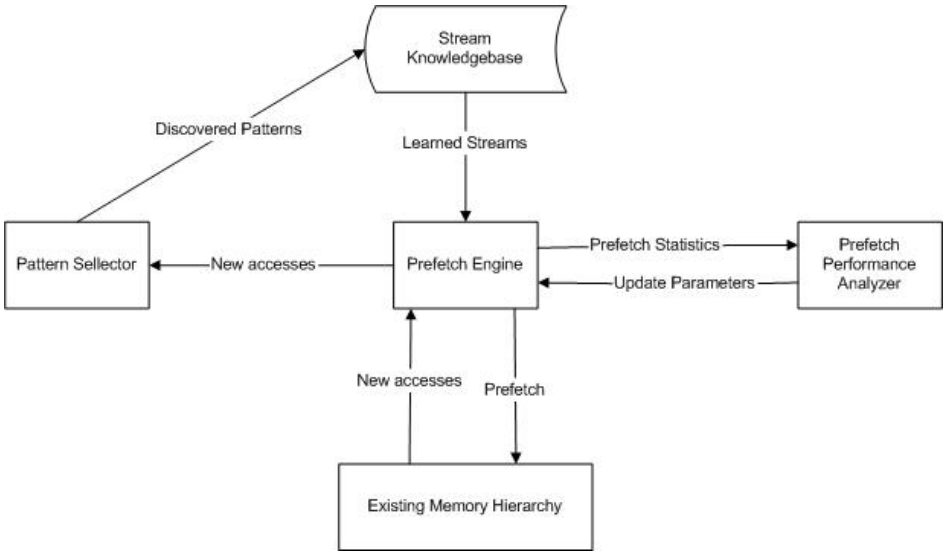

Figure 3.1. Main model of the prefetcher

references. After fetch distance passed, PE stores the stream of addresses referenced by cpu in Stream Knowledgebase. The number of the addresses on the stream is called stream length which determines the number of cache blocks prefetched after a pattern reoccurrence.Stream Length is another prefetch parameter. In short, PS is used to determine which patterns must be learned. Instead of storing all of the consecutive memory accesses as access pattern on Stream Knowledgebase only repeating patterns selected and only the streams referenced after selected patterns are stored in Stream Knowledgebase.

3.1.2. Stream Knowledgebase (SK)

As the name implies, the learned knowledge about the working program's memory access behaviour is stored in Stream Knowledgebase. It consists of the list of pattern stream pairs shown in Figure 3.3. Only limited amount of records can be stored in this table since it uses on-chip resource. When Stream Knowledgebase reaches its capacity, a victim is selected and removed from the table. For the sake of simplicity , the very first added record is removed. To make victim selection more efficient, additional information about the occurrences of patterns and streams in the table could also be used. Since this would make ST even larger and more complex, we avoided such extensions.

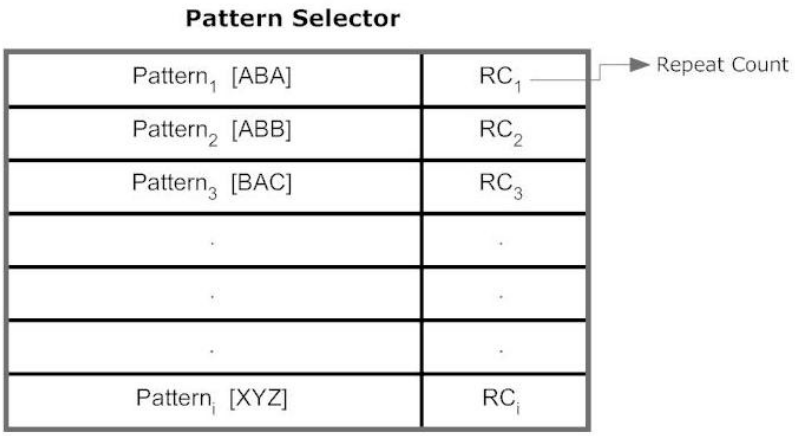

Figure 3.2. Pattern Selector

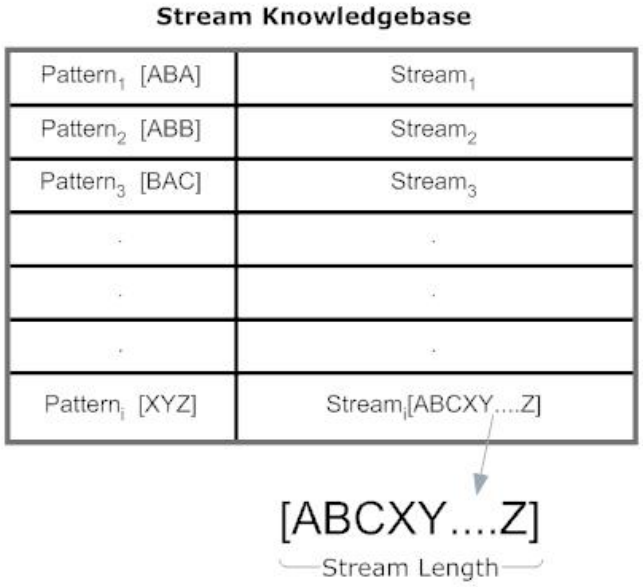

Figure 3.3. Stream Knowledgebase

3.1.3. Prefetch Engine (PE)

The working agent of the prefetcher is the prefetch engine which follows all of the memory requests made by the processor and update PS and ST accordingly. If any prefetch opportunity detected, prefetch engine fetches the memory addresses on the corresponding stream. Moreover, PE provides Prefetch Performance Analyzer with the prefetch success information by examining consecutive memory references after any stream prefetch operation. This success information is used to dynamically tune prefetching tendency of ATP which is explained in next section.

3.1.4. Prefetch Performance Analyzer (PPA)

With the success information given by PE , PPA maintains a statistical information about success ratios of streams and the prefetcher globally. Last Success Ratio(LSR) is the success ratio of the most recently prefetched stream. Total Success Ratio(TSR) is the global success ratio of the prefetcher. Recent Success Ratio (RSR) is the success ratio of the prefetcher that is being achieved recently. The formulas used to calculate these statistics are defined as in below:

$$LSR = \frac{number\ of\ correct\ fetches}{Stream\ Length} \tag{3.1}$$

$$TSR = \frac{number\ of\ total\ correct\ fetches}{number\ of\ total\ fetches} \tag{3.2}$$

where $number\ of\ correct\ fetches$ represents the correct fetches during the last prefetch operation and $number\ of\ total\ correct\ fetches$ is the total correct fetches throughout all prefecth operations.

Moreover, Recent Success Ratio (RSR) is observed in order to determine the

current performance of the prefetcher by the PPA. RSR is calculated by the following formula:

$$RSR = \frac{LSR + (RSR * AA)}{AA + 1} \tag{3.3}$$

where AA is the adaptation agility of the prefetcher which represents how fast ATP adapts new memory access behaviours. The greater value AA has, the slower ATP changes its prefetching parameters.

If any learned stream associated with a specific pattern causes an LSR to be less than of predefined minimum acceptable success ratio , this is considered by PPA as stream is learned wrong or no longer valid .Then PPA signals PE to remove that stream from Stream Knowledgebase to forget the no longer valid statistics. PPA also updates prefetch parameters according to the RSR values. If RSR drops below a predefined threshold PPA gradually adjusts prefetch parameters so that PE prefetches less frequently and shorter streams. If RSR exceeds a certain point the modification happens in the opposite way. The algorithm used to update parameters is described in Table 3.1

3.2. Region Based Context Switching

We propose an operating system optimization for multithreaded programs which includes a heuristic for the context switching among available threads. This method can be considered not a whole scheduling algorithm but an extension to the existing thread scheduling algorithm in the current architecture. After threads are scheduled, the operating system run them one at a time or multiple simultaneously if the architecture is a multi processor. Even when multi processing is the case,if we focus only in one processor , we see threads run serially. Each ready thread , which are ready to run according to the scheduler, runs through a specific amount of cycles. After that, a context switching occurs and the active thread changes. This time interval between

Table 3.1. Parameter update

```
if RSR < MinSuccessRatio then
  if SL > MinSL then
    SL ← SL − 1
  else if PL > MaxPL then
    PL ← PL + 1
  else if PRC < MaxPRC then
    PRC ← PRC + 1
  end if
end if
if RSR > MaxSuccessRatio then
  if PRC > MinPRC then
    PRC ← PRC − 1
  else if PL > MinPL then
    PL ← PL − 1
  else if SL < MaxSL then
    SL ← SL + 1
  end if
end if
```

context switches through which a thread runs called context switch interval. At context switching, the operating system has usually several ready threads to switch. Even the thread scheduler algorithm might be strictly stating the order of threads , the algorithm can be changed to be less strict offering a list of alternatives at each context switching. We propose a locality aware context switching heuristic for operating system that switches among available threads in such a way that data sharing increases. We call this as region based context switching. We call a fixed sized portion of the memory as memory region. The region size is defined in terms of the number of cache blocks it covers. A thread making reference in a specific region is considered to be working on that region. Due to temporal locality, threads do not change their memory region so often. Moreover, if thread granularity is well adjusted and region size is large enough ,

each threads working set covers only a few region on the memory. The last referenced memory region for each thread is stored in a table respectively.On context switching, operating system takes the list of ready threads and chooses a thread which has the same last referenced region as the the current thread's region. This switching strategy extends inner behaviour of temporal locality of each thread to locality among the threads. We hope this makes an improvement on the data sharing between threads and therefore, cache utilisation increases. If the threads are fine grained in very high numbers , since the opportunity to switch context in the region increases,we expect region based context switching gets more effective, which is validated in Chapter 5 which discusses the experimental results.

4. METHODOLOGY

4.1. Cache Simulators

Several cache simulators are examined during this study in order to simulate our proposed methods. Simics [22], Flexus [24] , SMPCache [25] are some of them which are described in more detail below.

Simics [22] is a full system simulation platform, capable of simulating high-end target systems with sufficient fidelity and speed to boot and run operating systems and commercial workloads. It is a commercial product of Virtutech but it is also freely available for academic use. To be able to get it for academic use , academic identity must be proven and usage is by simply registering on their website and providing the necessary academic information. Simics can simulate any digital system including simple processor and memory, a custom FPGA , a complete single mainboard, or a rack full of boards.Any custom heterogeneous architecture with any complexity can be modeled with its model definition language and several ready made components support model development including several processor families (e.g. ARM, Intel, MIPS, PowerPC, Tensilica, and TI DSP), hundreds of IO devices, and standard communications, backplane and network protocols. It is a full system simulator and allows firmware, device drivers, operating system, middleware stacks, and the application software. Although the simulator is extremeley capable of simulating almost any architectural model, to be able to develop even a simplest model in Simics requires lots of experience and knowledge about model development. Here in this study, i installed and get to develop the simplest model that is mentioned in the introductory tutorials [26]. To be able to develop and understand building the simplest component took about a week. Because of the time constraint of this study , we decided not to involve Simics model building and looked for other solutions.

Flexus [24] is a family of component-based C++ computer architecture simulators that is built on Virtutech Simics' Micro-Architecture Interface (MAI) to enable

full-system timing-accurate simulation of uniprocessor and multiprocessor systems running unmodified commercial applications and operating systems. Flexus encompasses both a simulation infrastructure and default simulation models. A simulator is composed of individual modules that are hooked together during compilation. A module is often the equivalent of a single hardware structure , for example, a branch predictor or a cache. A key strength of Flexus is its isolation of components: one implementation of a particular module can be swapped for a different implementation without requiring changes to any other modules. Like Simics, Flexus , which is an extension to Simics, is too complex for us to develop our models.

SMPCache [25] is a much more simple cache simulator, which runs only on Windows and simulates Multi Level memory systems on up to four-processor homogeneous arhictectures. It is a trace driven simulator , which means, a trace file of the simulated program must be provided to perfom simulation. Unlike Simics , SMPCache has very limited architectural customizations including only cache system parameters and the number of processors. The ability to develop custom virtual components is absent and only very standard computer systems can be simulated. No source code of SMPCache available publicly for further development.Since we are looking for a simulator that we can model our proposed prefetcher and context switching heuristic , SMPCache is too simple for our need.

LDASim [27] is a multilevel cache simulator which is based on Latency-of-Data-Access Model. In LDA, direct memory accesses to the various levels in the memory hierarchy is defined as data movements and the number of data movements multiplied by the cost of each access respectiveley provides and accurate estimate on the expected runtime.

The simulator gets two sources of input.The first one is the model of the target machines's memory architecture. With this configuration files you can define different levels of cache hierarchies, sizes of the caches and memory, size of cache blocks, replacement policies ,write policies etc. Secondly, the memory access pattern of the simulation which is meant to be declared as manually inserted function calls to the sim-

ulated source code must be provided. In that sense, LDASim is an execution-driven simulator that you have to enter the memory accesses in your source code programmatically. There is no GUI provided with the simulator. It is just a memory simulation engine and architecture not a stand alone application.

Although LDASim is the simplest simulator that we examined during this study, it is open source and publicly available for the development of new simulators. We take the cache engine of LDASim as the base for our own simulator and developed a new cache simulator that meets our requirements. In later section the simulator developed, called as BUSim , is explained in detail.

4.2. BUSim : The Developed Simulator

Firstly it should be noted that , our simulator ,unlike the LDASim, is a standalone simulator application that uses LDASim as memory architecture and base simulation engine and adds many other features that can facilitate simulation projects. It is written in C++ and Qt Programming Framework [28] is used to implement GUI of the application.

The simulated architecture is defined via configuration files a in LDASim. The format of the configuration files can be seen at Figure 4.1. Each memory level is defined as a block of structural definiton in the file and each block is linked to another block by next level property. A NULL valued next level in a block means that it is the last level in the memory hierarchy which is usually the main memory.

A major contribution to the base simulation architecture is to provide support for trace-driven simulation. BUSim can take memory access pattern as trace files of memory accesses like many other cache simulators do.Mainly ,simulator supports two kinds of trace files. It can take each trace file as a memory access pattern of a thread and take multiple of them to simulate in a multi-threaded manner.It allows for the definition of precedence releations among multiple threads so that some threads finish before other dependent threads. Thus, our simulator provides single processor multi-

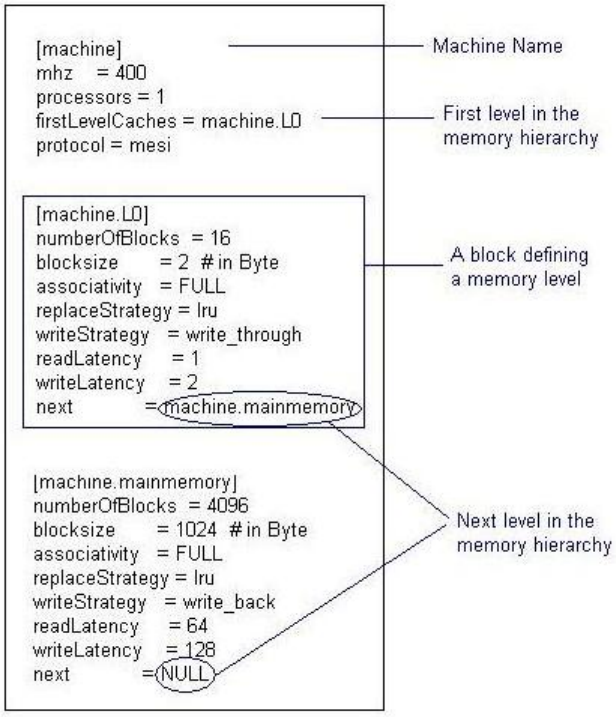

Figure 4.1. Machine Configuration File

threaded simulation. As for the other type of traces, in multiprocessing environments processor id's must be provided for each memory access in the trace file and there is only one trace file to define the global access pattern of the whole system. In this way, multi-processor simulation is possible also. The format of the trace file is very simple and the same as SMPCache [25]. Many other trace file formats are defined. A converter application can be developed to provides interoperability among other trace-driven simulators.

Moreover, as we mentioned above our simulator has multi-threaded simulation support with multiple thread-based trace files. The sheduling of the threads in the simulation process is performed by different scheduling heuristics which can be defined before the simulation. A Sheduler heuristic is a generic class , in OOP terminology, which provides a scheduling function. For the time being, our simulator has three types of sechedulers which can easily be increased by simply implementing other concrete scheduler classes. This allows for the analysis of the performance of different scheduling heuristics on a specific set of threads. For example ,by means of scheduler classes, we

got able to simulate region based context switching.

Another interesting feature of the simulator is the support for prefetchers. Like the scheduler classes, there is a generic prefetcher class from which several different prefetchers are implemented. The performance of different prefetching algorithms can be analyzed and a s pecific algorithm can be optimized for a specific domain. BUSim , if specified, provides the performance statistics of the prefetchers during the simulation. Currently, two different prefetching algorithms are implemented and many athers can be added by just class derivation from base abstract prefetcher class. ATP is modeled as a derived class of generic prefetcher class in BUSim.

Another important contribution is the graphical user interface that provides the users with a very user friendly management , configuration and monitoring features for their simulation projects. All of the input files and other simulation parameters can be defined easily by simply using the provided menus , buttons and very definitive input widgets. Furthermore, as the simulation goes on you can monitor all of the simulation statistics such as total latency. And as another contribution to LDASim, you can see hit and miss ratio of each cache level respectively during the simulation. Each cache level is presented in a self-refreshing table with their architectural information such as size , blocksize , policies etc. You can also modify the speed of the simulation as the simulation is in progress so that some specific portion of the smulation may be observed in desired detail. The simulation GUI consists of mainly three screen for configuration and monitoring of simulation experiments These screens are namely Simulation Configuration Screen (Figure 4.2), Memory Monitoring Screen (Figure 4.3) and Simulation Statistics Screen (Figure 4.4).

On Simulator Configuration Screen, threads to run can be defined as trace files and prefetchers and scheduling heuristics can be specified. On Memory Monitoring Screen, each level of memory components can be viewed and monitored during simulation. Hit and miss ratios of each cache level can be seen respectively. On the third screen, simulation and ,if exists, prefetcher statistics are displayed.

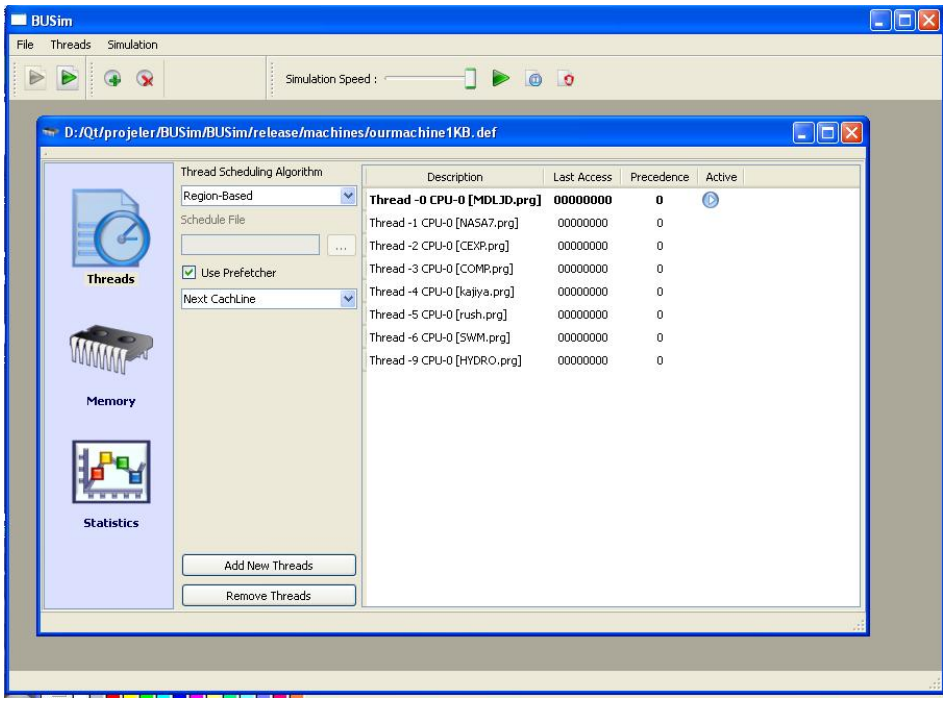

Figure 4.2. Simulation Configuration Screen

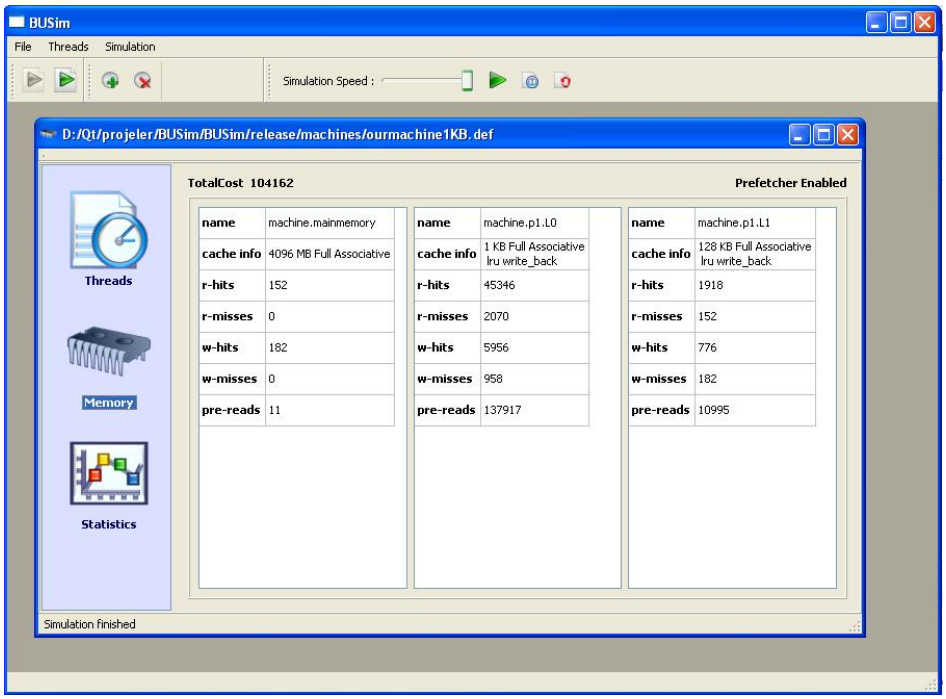

Figure 4.3. Memory Monitoring Screen

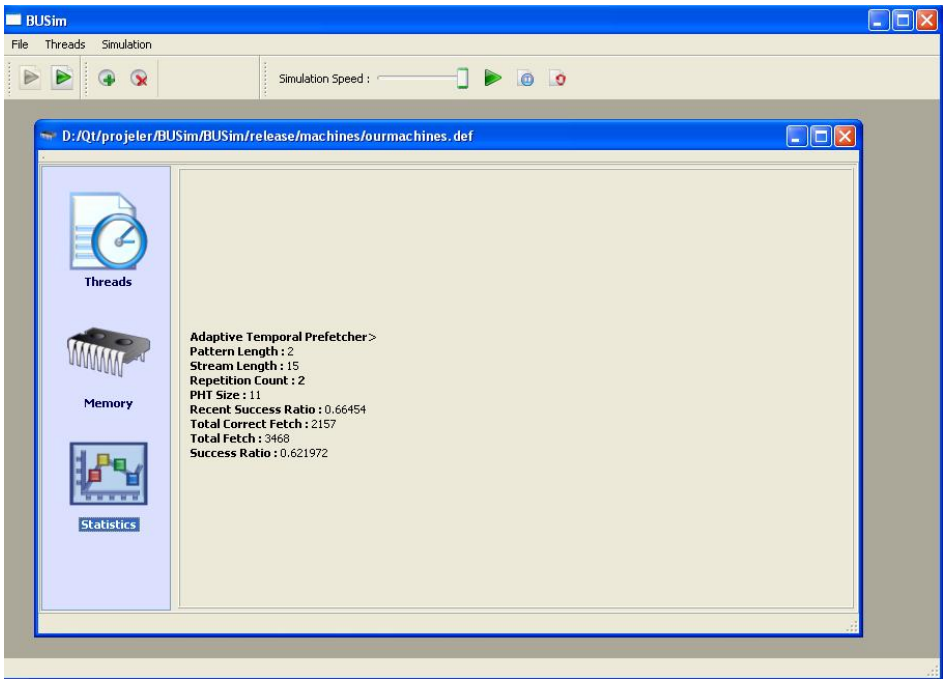

Figure 4.4. Simulation Statistics Screen

Finally, in order to perform planned experiment in a fast and noninteractive way, we added batch simulation mode to our simulator.In batch simulation mode, BUSim takes a batch file wich includes all of the parameters that define a specific experiment such as machine configuration file name, type of the prefetcher and scheduling heuristic, trace files , and if necessary prefetching parameters etc. Each experiment is defined by a single line, a batch line, in the batch file. BUSim parses the batch file line by line and perform experiments. After performing each experiment detailed statistics are written to an output file which is also defined in the batch line. By means of batch simulation , we tried to set up several experiment scenarios build a batch file including those experiments and just run BUSim in batch mode without any further interaction.

4.3. Simulator Validation

After implementing the BUSim, although its core cache engine is derived from an existing simulator LDASim [27], we made a validation with another trace driven simulator SMPCache [25]. We setup many different cache configurations on both simulators and run several experiments with the same trace files. We got exactly the same results from both SMPCache and BUSim when caches are direct mapped and

almost the same results whent they are fully associative. Some of the validation results are listed in Table 4.1

Table 4.1. Experimental Validation Results

Experiment	Stats	SMPCache	BUSim
8KB L1 Cache MESI Write Back FULL ASSOCIATIVE Trace : EAR.prg	Memory Access	5308	5308
	# of Misses	78	82
4KB L1 Cache MESI Write Back FULL ASSOCIATIVE Trace : EAR.prg	Memory Access	5308	5308
	# of Misses	120	127
2KB L1 Cache MESI Write Back FULL ASSOCIATIVE Trace : EAR.prg	Memory Access	5308	5308
	# of Misses	218	219
2KB L1 Cache MESI Write Back DIRECT MAPPED Trace : EAR.prg	Memory Access	5308	5308
	# of Misses	529	529
2KB L1 Cache MESI Write Back DIRECT MAPPED Trace : HYDRO.prg	Memory Access	2117	2117
	# of Misses	251	251
2KB L1 Cache MESI Write Back DIRECT MAPPED Trace : MDLJD.prg	Memory Access	20000	20000
	# of Misses	1664	1664
2KB L1 Cache MESI Write Back DIRECT MAPPED Trace : WAVE.prg	Memory Access	3427	3427
	# of Misses	354	354

4.4. Experimental Setup

In the experiments, BUSim has been used in batch mode. Batch files have been created for each set of experiments. The simulation environment was single processor single level cache hierarchy, since our simulator is not supporting multiproccessor simulation yet. We used 4 GB main memory and 8 KB of L1 cache and 1 MB of L2 cache with the block size of 64 bytes in the memory hierarchy. Since we think our proposed methods have similar on each level, we focused only on L1 cache performances. Several algorithms from different domains such as cook, rushmeier, kajiya, sixtrack, crafty, fma3d, equake , vortex, and art have been used as trace files which are the selected algorithms in SPEC 2000 benchmark suite. The trace files are downloaded from the web trace repository of Brigham Young University [29]. Since the format of the trace files did not comply with our trace format a , we developed a converter application. In the performance analysis of region based context switching, the example trace files

provided by SMPCache [25] are used, which are also the memory traces of different algorithms.

5. EXPERIMENTAL RESULTS

5.1. Experimental Study

5.1.1. Prefetcher

5.1.1.1. Pattern Selector Size. Firstly, we analyzed the effect of the size of pattern selector on the number of correct fetches. Unless the prefetcher success ratio changes significantly, the number of correct fetches indicates how well our prefetcher performed during the simulation. We simulated the system for Pattern Selector size of 4 up to 2000. If pattern selector gets smaller, PE discovers new repeating pattern less frequently. So we expect an increasing number of correct fetches for increasing pattern selector size. But this increase will stop at some point where it is already large enough to accommodate all repeating patterns. The optimum size of the pattern selector depends on the domain that we are working on. As it is seen from the graph illustrated in Figure 5.1 , up to 100 the number of correct fetches increasing with minor disorder. After 100 , no significant gain obtained because of redundancy.

5.1.1.2. Stream Knowledgebase Size. The effect of the size of Stream Knowledgebase is similar to PS size. The more pattern-stream pair SK stores the more likely PE

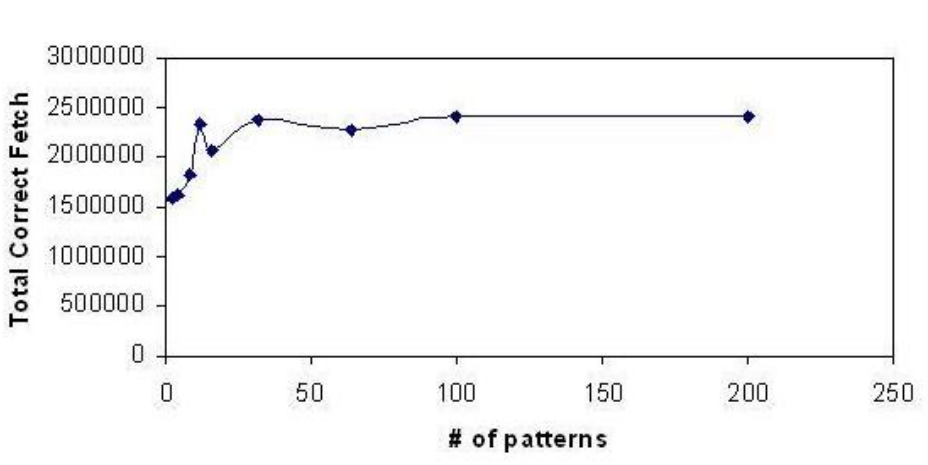

Figure 5.1. The effect of PS Size on Total Success Ratio

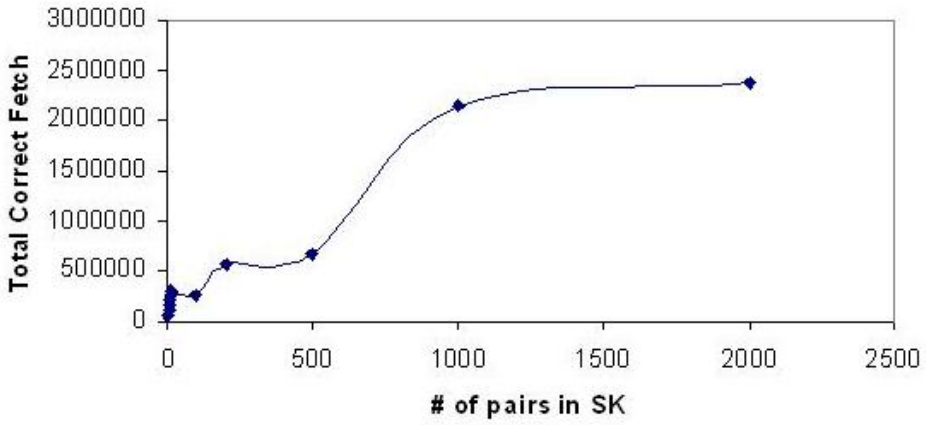

Figure 5.2. The effect of SK Size on Total Success Ratio

discovers prefetch opportunities. So increasing SK size results in an increase in correct fetch count. We again expect prefetch success ratio changes insignificantly since other prefetch parameters such as Pattern Length, agility, stream length and domain remains constant through simulations. We made simulations for SK size of 2 up to 1000. The result is illustrated in Figure 5.2.

5.1.1.3. Adaptation Agility. As it is mentioned 3.1, Adaptation Agility(AA) is a prefetcher parameter determining how fast ATP adapts new memory behaviour. The larger AA is the slower ATP changes prefetching parameters due to new conditions. Making prefetcher too agile, can be ineffective if the change itself is very temporary and adapting to this very short term condition is not needed actually. On the other hand, making ATP too reluctant to changes means disabling adaptation ability. We analyzed the value of AA on the performance of the prefetcher. The performance is measured in terms of Total Success ratio. It can be seen from Figure 5.3 that the performance slightly differs for different AA values. It has a maximum value at about 50 for the trace file, used in the experiment. Although the optimum value can change from program to program, we expect reduction in both ends which is also depicted in the Figure 5.3.

5.1.1.4. Prefetcher Performance Analysis. Based on the experiments,we could not see the expected performance gain on the total system performance. We think is mainly

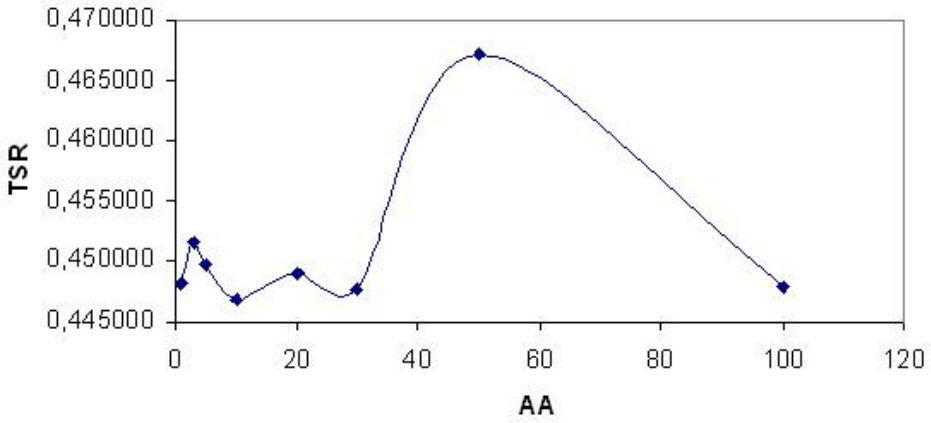

Figure 5.3. The effect of AA on Total Success Ratio

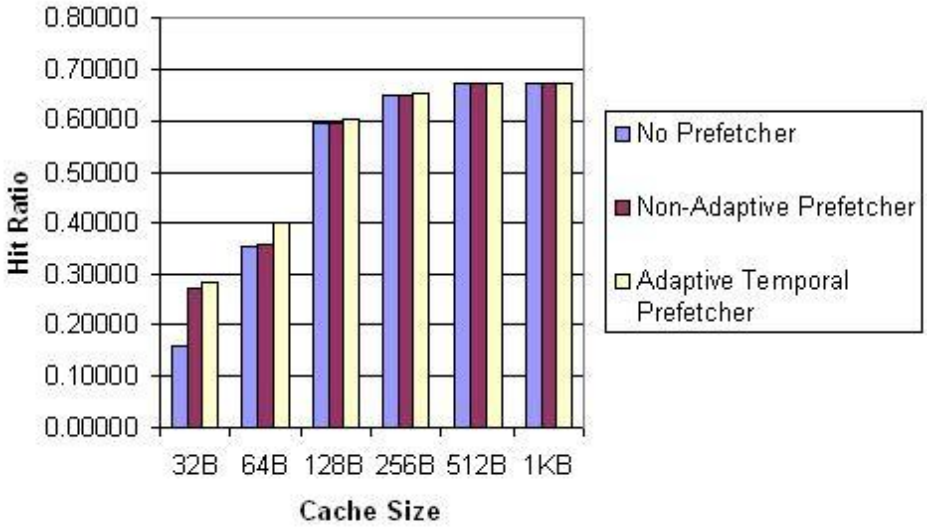

Figure 5.4. The prefetcher performance in very small caches

because of high hit ratio which hides the advantage of prefetched blocks. The cache pollution because of unnecessarily prefetched blocks also plays a diminishing role on the prefetcher performance.For practically unrealistic scenarios where hit ratio is about 0.2 a signinificant gain is observed, illustrated in Figure 5.4 , for more realistic cases, we saw very minor reduction in miss ratio eventhough we tested for differnet algorithm in different domains. In Figure 5.5, Figure 5.6, Figure 5.7 the percentage of covered misses can be seen for different cache sizes with different trace files.

After surprisingly inefficient performance results ,we searched for possible reasons that prevent covering misses. We checked how many of the prefetch attempts of ATP

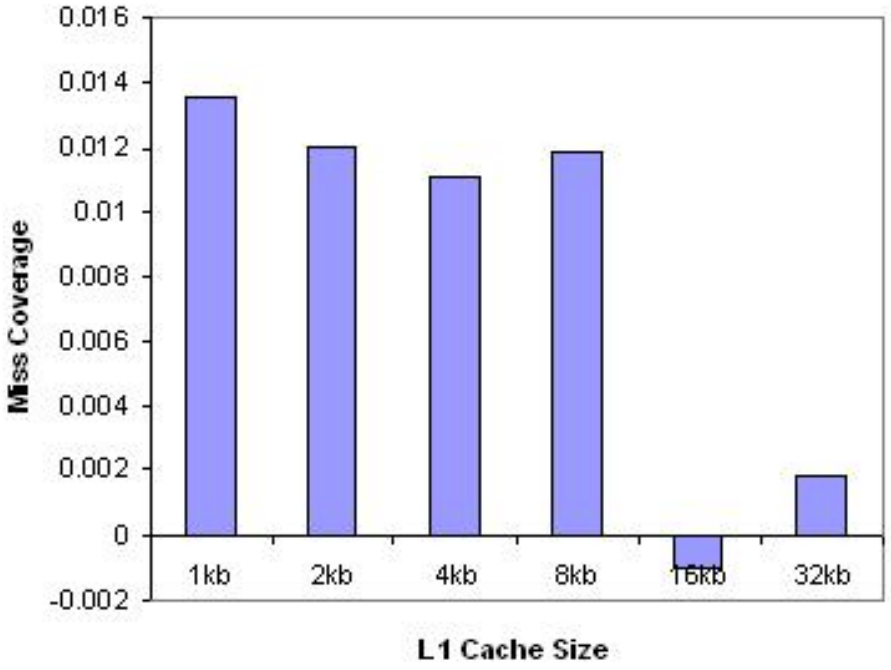

Figure 5.5. Miss coverage by ATP with equake

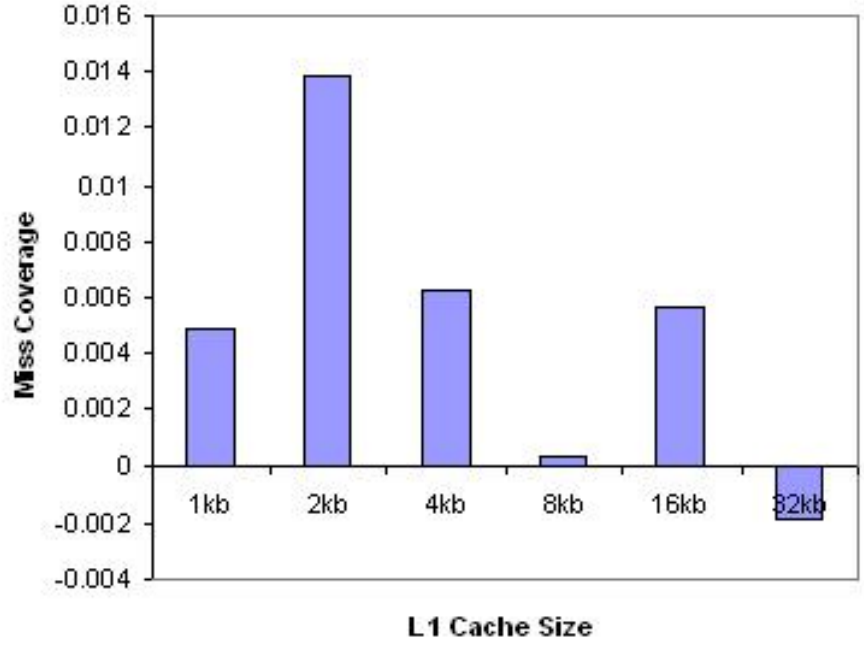

Figure 5.6. Miss coverage by ATP with sixtrack

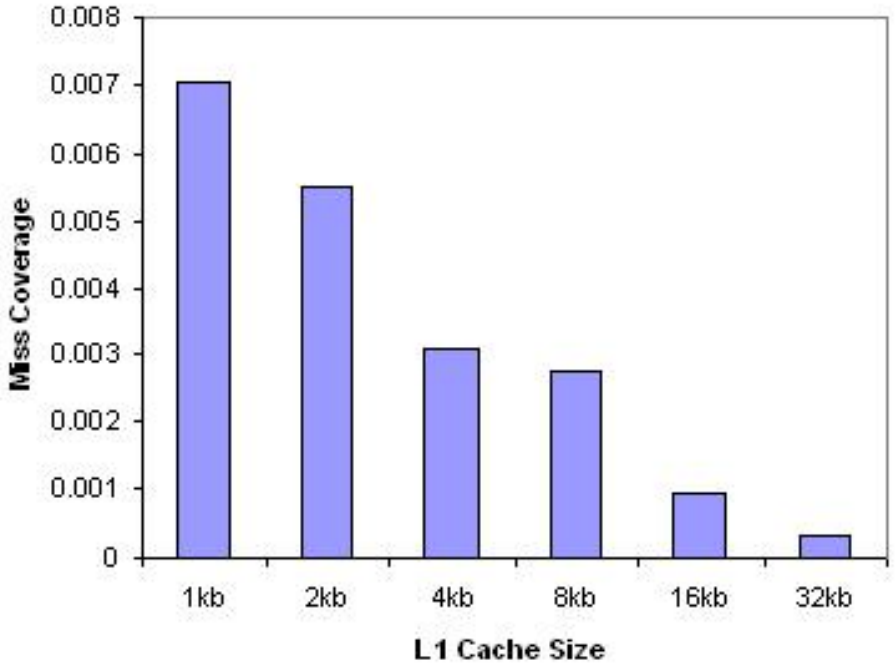

Figure 5.7. Miss coverage by ATP with rush

are really prefetched from lower levels of memory hierarchies. We counted only the blocks that cause a miss on prefetch operation. The Table 5.1 shows results of this investigation. As it is seen in the first row of the table, only about 5 percent of prefetch attempts are really prefetched and almost 95 percent of prefetch attemps are already in the cache. Furthermore only 20 percent of real prefetches are really used by the processor which are the real correct prefetches. This explains why miss reduction is so insignificant. But why the percentage of real prefetch to the prefetch attempts is so low? We suggest two resons for this? firstly the usage of 64 bytes block size provides an inner prefetching effect due to the spatial locality. If we consider word size as 4 bytes, when a word is referenced in the memory the next consecutive 15 words are already comes with it in the same block. Since our prefetcher prefetches streams of accesses that are usually consist of words close to each other, many of the prefetched words are all in the same cache block and already exist in the cache. To validate this we configured caches with block size of 1 byte just for experimental purposes and run the simulations again. The results was really supporting our guess, in which the real prefetch number is significantly higher than the previous experiment. But this time, since cache block size is just 1 byte, the granularity of cache blocks are so high that

prefetcher success ratio got diminished much. Another reason that causes very low real prefetch number, is that the footprint of the repeating patterns of the simulated programs are small enough to fit in the cache so that all of the repeatedly used cache blocks stay permanently in the cache. Therefore , since our prefetcher only tries to prefetch repeatedly referenced blocks, it usually tries to prefetch blocks that are already in the cache. When cache size is so small that the cache can not accomodate all of the repeating patterns simultaneously, then the prefetcher starts to play a significant role on the performance. The significant hit ratio improvement for extremely small cache configuration shown at Figure 5.4 validates this idea.

Table 5.1. Prefetcher statistics showing the percentage of real prefetches

L1 Cache Size	Total Prefetch Attempt	Total Real Prefetch	Total Real Correct Prefetch	Miss with NoPrefetcher	Miss with ATP	Reducton in miss count
2kb	2412000	144003	32380	1118635	1107589	11046
4kb	2412000	73907	16173	1046755	1037610	9145
8kb	2412100	39297	6952	971880	967873	4007
16kb	2412000	25790	3553	890999	889888	1111
32kb	2412000	25061	3478	860544	860162	382

5.1.2. Region Based context switching

The performance of region based context switching is measured in terms of miss coverage. In the graph, the reduction of misses for different L1 Cache size can be seen. The values are reduction of misses when region based context switching is used compared to the case where no heuristic is employed at context switching. About fourty percent reduction happens when cache sizes below 8KB As it can be seen from the graph in Figure 5.8,after cache size gets larger than a certain amount, the gain is hidden by high hit ratio and the miss reduction rate decreases.

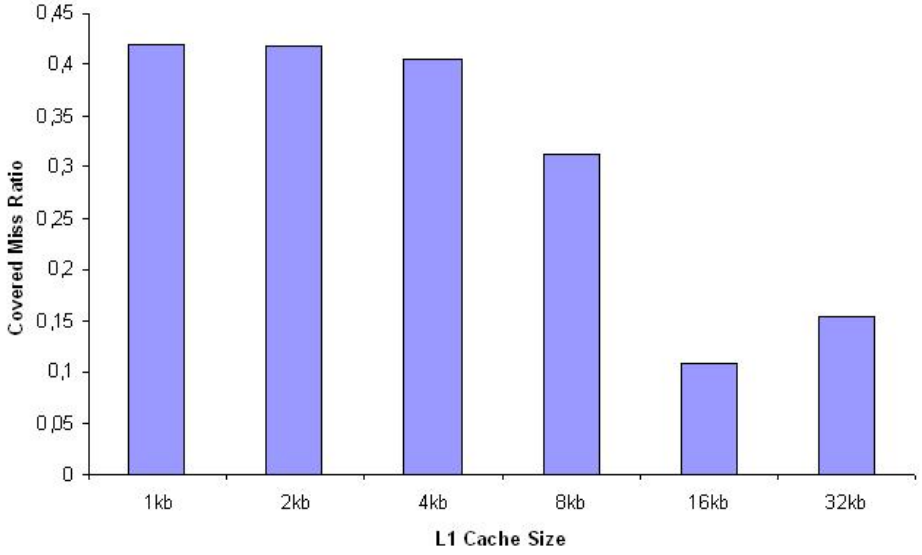

Figure 5.8. Region Based Contect Switching Miss Coverage

6. CONCLUSIONS

As processors became faster and faster day by day, memory systems have been becoming a more critical bottleneck that hinders total system performance in computer systems. This effect is even more detrimental on multiprocessor systems. Many studies focus on improving memory access speed by several means. Some of them such as data prefetchers, larger caches , and locality aware thread schedulers are trying to reduce miss rates in caches while some others such as distributed memory system design are making effort on decreasing miss latency. Locality of programs' memory access behaviour plays crucial role on most of the memory system optimizations. Spatial and temporal locality principles are the inspiring source of many currently used memory technologies today. In modern researches, memory access behaviour of programs is even more deeply analyzed to discover other types of localities that programs exhibit during their life cycles in order to exploit for further optimization in hit rates.

In this study, we also focused on the memory system otimizations based on the locality of accesses. First we proposed a new hardware data prefetcher , adaptive temporal prefetcher(ATP), which tries to exploit repeating patterns on memory accesses. ATP is designed to listen all memory accesses of programs and learn repeating access sequences to prefetch on further occurrences. One of the major properties of ATP that differs from previously proposed methods is its adaptation ability. ATP can adapt itself to changing memory behaviour of currently running applications by updating its prefetch parameters dynamically to tune its prefetching tendency. Thanks to this, we hope , ATP will prefetch more if the accesses are highly repetitive and there are high number of prefetch opportunities and will be more reluctant to prefetch if memory references are extremely irregular. Secondly we proposed a context switching heuristic which can be considered as a kind of short term thread scheduler for operating systems that focuses on locality. Region based context switching divides memory into regions and keeps last accesses of active threads in terms of region addresses. On context switching, it switches to the thread that has the same last accessed region as the current thread does. Due to temporal locality, this switching policy improves data sharing

and therefore cache utilization. In order to test our proposed techniques, we also implemented a trace driven multilevel cache simulator, BUSim. Since our simulator only supprorts single processor simulation we did our experiments on single processor systems. Although we could not get to have good results with our proposed prefetcher model, in our experiments, we hope it will be much more effective in multiprocessor systems where shared memory are subject to heavy usage. We left multiprocessor simulation in more capable simulators undone as future work due to time constraint of this study. We did have significant gain in the experiments of region based context switching. We hope the effectiveness of locality concerning scheduling increases even more with the increasing number of simultaneously running threads which is the current tendency in modern computer architectures.

Better measurement of prefetch performance by defining new prefetching metrics and dynamically collecting more statistics can also be a future work for ATP. We left also undone as future work the implementation of ATP in a full system simulator to see more realistic beaviour of the proposed system components with existing memory system hardware. As for the region based context switching, examining the whole cache content ,not just the last thread's region as the decision criteria can give better idea about the locality of current cache usage. Eventhough , because of temporal locality, we already took significant advantage of our context switching heuristic, defining current cache content better and decide accordingly can increase the gain which can be tested in future studies.

REFERENCES

1. Steve VanderWiel and David J. Lilja. A Survey of Data Prefetching Techniques.Technical Report No: HPPC-96-05, October 1996

2. Porterfield, A.K., Software Methods for Improvement of Cache Performance on Supercomputer Applications. Ph.D. Thesis, Rice University, May 1989.

3. Thomas F. Wenisch, Stephen Somogyi, Nikolaos Hardavellas,Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Temporal Streaming of Shared Memory. Proc. of the 32nd Annual ACM/IEEE International Symposium on Computer Architecture (ISCA), Jun 2005.

4. Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi and Andreas Moshovos. Spatial Memory Streaming. Proc. of the 33rd International Symposium on Computer Architecture (ISCA), Jun. 2006.

5. Stephen Somogyi, Thomas F. Wenisch, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Memory Coherence Activity Prediction in Commercial Workloads. Proc. of the 3rd Annual Workshop on Memory Performance Issues (WMPI), Jun 2004.

6. VanderWiel, S. and Lilja, D. . A compiler-assisted data prefetch controller. In IEEE International Conference on Computer Design (1999).

7. Chen, T.F.. An effective programmable prefetch engine for on-chip caches. In Proceedings of the 28th International Symposium on Microarchitecture, pages 237242 (1995).

8. Nir Oren, A Survey of prefetching techniques July 18, 2000

9. Bruce Jacob, Spencer Ng and and David Wang. Memory Systems: Cache, DRAM,

Disk.

10. John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach, 3rd Edition.

11. D. Gannon, W. Jalby, and K. Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation. Journal oj Parallel and Distributed Computing, 5:587616, October 1988.

12. Alper Köse. Locality aware task scheduling in heterogeneous computing environments. MS Thesis,2007

13. Topcuoglu,H., Hariri S., Wang M. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. IEEE Trans. Parallel Distrib. Syst. 13, 3, 260-274 (2002).

14. Sih, G. C. and Lee, E. A.. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. IEEE Trans. Parallel Distrib. Syst. 4, 2, 175-187 (1993).

15. El-Rewini, H. and Lewis, T. G. Scheduling parallel program task onto arbitrary target machines. J. Parallel Distrib. Comput. 9, 2, 138-153 (1990).

16. P. Shroff, D.W. Watson, N.S. Flann, and K. Freund. Genetic Simulated Annealing for Scheduling Data-Dependent Tasks in Heterogeneous Environments. Proc. Heterogeneous Computing Workshop, pp. 98-104. (1996)

17. Betül Demiröz, Haluk Rahmi Topçuoğlu. Static Task Scheduling with a Unified Objective on Time and Resource Domains. Comput. J. 49(6): 731-743 (2006)

18. Thread Scheduling for Cache Locality James Philbin and Jan Edler NEC Research Institute, 41ndependence Way, Princeton, NJ O854O Otto J. Anshus Department of Computer Science, Institute of Mathematical and Physical Sciences, University of Tromso, N-9o37 Tromso, Norway Craig C. Douglas IBM T.J. Watson Research

Center, P.O. Box 218, Yorktown Heights, NY 10598-0218; and Department of Computer Science, Yale University, P.O. Box 208285, New Haven, CT 06520-8285 Kai Li Department of Computer Science, Princeton University, Princeton, NJ 08540

19. Cache-Fair Thread Scheduling for Multicore Processors Alexandra Fedorova, Margo Seltzer and Michael D. Smith Harvard University, Sun Microsystems

20. Scheduling Threads for Constructive Cache Sharing on CMPs Shimin Chen Phillip B. Gibbons Michael Kozuch Vasileios Liaskovitis Anastassia Ailamaki Guy E. Blelloch Babak Falsafi Limor Fix Nikos Hardavellas Todd C. Mowry, Chris Wilkerson Carnegie Mellon University Intel Research Pittsburgh Intel Microprocessor Research Lab

21. Smith, A.J., Cache Memories, Computing Surveys, Vol.14, No.3, Sept. 1982, p. 473-530.

22. SIMICS, http://www.virtutech.com

23. Nikolaos Hardavellas, Stephen Somogyi, Thomas F. Wenisch, Roland E. Wunderlich, Shelley Chen, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. SIMFLEX: A Fast, Accurate, Flexible Full-System Simulation Framework for Performance Evaluation of Server Architecture. ACM SIGMETRICS Performance Evaluation Review, vol. 31, no. 4, Mar. 2004.

24. Flexus, http://www.ece.cmu.edu/ simflex

25. SMPCache, http://arco.unex.es/smpcache

26. Modeling Your System in Simics User Manual

27. Jens Simons and Jens-Michael Wierum. The Latency-of-Data-Access Model for Analyzing Parallel Computation. Information Processing Letters, 66(5):255-261,June 1998.

28. Qt, http://qt.nokia.com/

29. Trace Distribution Center, http://tds.cs.byu.edu/tds/