ABSTRACT

PANT, SALIL MOHAN  Slipstream-mode Prefetching in CMP’s: Peformance Comparison and Evaluation.  (Under the direction of Dr. Greg Byrd).

With the increasing gap between processor speeds and memory, many programs with large working sets do not derive significant benefits from caching. As a result, many architects have turned to prefetch mechanisms. Prefetching works by predicting data items that will be referenced by application programs in the future and fetching them into the cache \textit{a priori} to reduce cache misses.

The slipstream mode of execution was recently proposed for multiprocessors built from dual-processor CMP’s, as a means to improve performance of parallel programs that have reached their scalability limits. The prediction of future loads, in this case, is based on actual execution, rather than history-based prediction in conventional hardware prefetchers.

The contribution of this thesis is to evaluate the performance of prefetching while running an application in slipstream mode and comparing it with performance obtained from conventional hardware prefachers. The objective of this work is to find out whether the use of an available extra processor for the purposes of prefetching to reduce coherence misses, is justified.

We find that slipstream mode provides 10-13\% additional speedup, compared to a good hardware prefetcher, for two of the four applications we studied. We also observe that prefetching in slipstream mode has a higher accuracy than other prefachers. Due to this, we believe that slipstream mode will continue to perform well with an increasing number of CMP’s.
Slipstream-mode Prefetching in CMP’s: Performance Comparison and Evaluation

by

Salil Mohan Pant

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master of Science

Department of Computer Science

Raleigh
2004

Approved By:

Dr. Edward Davis
Dr. Vincent Freeh

Dr. Gregory Byrd
Chair of Advisory Committee
To

Mom and Dad

ii
Biography

Salil Mohan Pant was born on 27th May 1981 in Thane, India. He received the Bachelor of Engineering (B.E.) Degree in Electronics Engineering from K. J. Somaiya College of Engineering, University of Mumbai, in 2002.

Salil has been a graduate student in the Computer Science Department at North Carolina State University, Raleigh, NC since Fall 2002. He is a student member of the Institute of Electrical and Electronics Engineers (IEEE). Since Fall 2003, he has been working as a Research Assistant with Dr G. Byrd in the field of Parallel Processing.
Acknowledgements

First and foremost I would like to thank God because of the inner strength and perserverance that he has provided me with. It is because of these qualities that I have been able to carry out my research work and finish this thesis.

Next I would like to thank my parents and my sisters for everything they have given me in life. It is only due to their love, support and encouragement that I could achieve whatever I have achieved. Special thanks to my father for continuously inspiring me with immense hard work and dedication towards my goals.

I sincerely thank my advisor Dr Greg. Byrd for giving me the opportunity to work under his guidance. He has been very understanding, supportive and inspiring throughout my work.

I would also like to thank my committee members Dr. Vince Freeh and Dr. Edward Davis for agreeing to be on my committee and providing valuable insight to this work.
# Contents

List of Figures vi

List of Tables vii

1 Introduction 1

1.1 Contribution 4

2 Prefetching Background and Related Work 6

2.1 Hardware Prefetching 6

2.1.1 Sequential Prefetchers 7

2.1.2 Non-Sequential Prefetchers 9

2.2 Software Prefetching 16

2.3 Slipstream prefetching 20

2.4 Prefetch Metrics 25

3 Simulation methodology and Results 27

3.1 Simulation Environment 27

3.2 Benchmarks 29

3.3 Experiments 30

3.4 Experimental Results 31

3.5 Combining Slipstream with prefetchers 39

4 Conclusion and Future Work 43

4.1 Concluding Remarks 43

4.2 Future Work 44

Bibliography 45
## List of Figures

2.1 Sequential Prefetch Schemes .......................... 7  
2.2 Matrix multiplication example .......................... 10  
2.3 RPT Structure .......................................... 12  
2.4 Structure of the Global History Buffer ................. 14  
2.5 Example Software Prefetching for Loops ................. 18  
2.6 Execution modes for CMP ............................... 21  

3.1 SOR results ............................................ 32  
3.2 OCEAN results .......................................... 32  
3.3 SOR traffic ............................................ 33  
3.4 Ocean traffic ........................................... 34  
3.5 FFT results ............................................ 35  
3.6 FFT traffic ............................................ 36  
3.7 LU results ............................................. 37  
3.8 Summary of results ..................................... 38  
3.9 Combined Slipstream + prefetch results for FFT ......... 40  
3.10 Memory Traffic for FFT with combined scheme ........ 40  
3.11 Combined Slipstream + prefetch results for SOR ........ 41  
3.12 Memory Traffic for SOR with combined scheme ........ 42
List of Tables

3.1 SimOS Specifications .................................................. 28
3.2 Benchmark Specifications ............................................. 29
Chapter 1

Introduction

CPU speeds have been doubling every eighteen months following Moore’s law whereas memory speeds double only every seven to ten years. These diverging rates imply an impending memory wall [35] in which memory access time dominates code performance. This has necessitated the use of increasingly aggressive techniques designed to reduce or hide the latency of memory accesses. Traditionally, caches have been used to reduce memory access time by storing the recently used data locally. Modern uniprocessors typically have multiple levels of caches with a demand fetch memory model, where data are fetched into higher levels upon processor request. Unfortunately many programs still spend a lot of their run time stalled on memory requests [22]. Programs with irregular data accesses do not derive significant benefits from caching. The problem can become worse in a multiprocessor scenario, where the latencies are higher for remote memory accesses.

Data prefetching has been proposed as a technique for hiding the latency of memory accesses that defeat traditional caching strategies. Instead of waiting for a cache miss to initiate a memory access, prefetching anticipates such misses and issues a fetch to the memory system in advance of the actual memory reference. To be effective, however, prefetching should be implemented in such a way that the prefetches are timely, useful and introduce little overhead. Prefetches that are not accessed later are termed as useless prefetches. Such prefetches waste resources, increase bus busy
Chapter 1 Introduction

time and can degrade overall performance. Important issues concerning prefetching are what to prefetch — *analysis* and when to prefetch — *schedule*. Prefetching strategies are diverse for both uniprocessors as well as multiprocessors, and no single strategy has yet been shown to provide optimal performance for all type of programs. However, prefetching has the potential to provide significant benefits by overlapping memory accesses with computation.

One broad classification of prefetching techniques is whether prefetching is *hardware-controlled* or *software-controlled*. Hardware based prefetched typically use a special unit to dynamically detect recurring patterns among data accesses and generate prefetches assuming that the same patterns will continue in the future. Software based prefetching analyzes the program during compile time and inserts special load instructions to bring data into higher levels of memory (or registers) before it is accessed.

An important thing to consider while prefetching is the placement of prefetched data. Prefetched data can be placed anywhere in the memory hierarchy, from registers to lower level caches. Prefetching data into processor registers or the primary (L1) cache has an advantage that the data is available immediately when demanded. The disadvantage associated with bringing data into the primary cache is that generally the cache is small and holds only the current working set of the program. Generating too many prefetches, or generating prefetches too early, may result in useful data being evacuated from the cache. Similarly too many prefetches into registers would increase register pressure and cause *spills*. Prefetching into the secondary (L2) cache is more tolerant of such prefetches, because the secondary cache is able to hold more data. There have also been schemes proposed for uniprocessors [16] where the prefetched data is brought into special buffers, instead of caches, to avoid polluting cache lines with useless prefetch data. This is done to give prefetchers more flexibility in scheduling and make prefetching more aggressive. However, it increases the memory bandwidth requirements of the system.
Chapter 1  Introduction

Conventional research on Prefetching has focussed mainly on uniprocessors. As the number of processors increases, the time to access remote memory locations increases, especially in case of distributed shared memory machines. In such cases, prefetching data values before they are required may hide a significant amount of memory latency from the processor. However, there may be fewer opportunities for prefetching itself. This is because prefetching mainly attacks cold misses and capacity misses for an application. As the number of processors increases for the same data set size, the partition of data allocated per processor decreases, and thus there may be fewer capacity misses. Also, multiprocessor prefetching is considered more demanding [32] because:

1. Memory system bandwidth increases due to prefetching per processor. Useless prefetching will increase interconnect utilization, which may adversely affect the stall time of other processors.

2. In case of cache-coherent systems, generating prefetches for shared data too early may result in the prefetched data being invalidated, and thus increase coherence traffic.

3. Task scheduling and migration make prediction of future addresses, and hence history-based prefetching, more difficult.

Another issue that arises while prefetching in multiprocessors is whether prefetches are binding or non-binding. A binding prefetch means that the value of the prefetched data is bound at the time of prefetch. The processor reads the same value later through a regular read, even if the value has been modified by another processor and is currently visible. Prefetching directly into processor registers or special buffer for prefetch data [16] is an example of binding prefetch. A non-binding prefetch means that the value brought by a prefetch remains subject to modification/invalidation by other caches before it is actually read by the processor. Prefetching into caches in a cache coherent system is typically non-binding. Non-binding prefetches give the processor more flexibility and encourage aggressive prefetching techniques without
1.1 Contribution

affecting the semantics of the programs execution. Binding prefetches, on the other hand, are issued conservatively by the processor. Thus, there are many tradeoffs associated with prefetching in multiprocessors.

Recently Ibrahim et al. [14] proposed a new mode of execution for parallel programs using dual-processor Chip Multiprocessors with a shared L2 cache, called Slipstream mode. In this mode a task is allocated on one processor of each CMP node, while the other processor executes a reduced version of the task. This reduced task runs ahead and prefetches data on behalf of the other task. This type of prefetching is based on actual program execution rather than monitoring data access patterns. Slipstream mode mainly helps programs that have reached their scalability limit and show degraded performance with an increasing number of CPU’s. Running tasks in slipstream mode not only reduces the communication between tasks but also improves per task execution time. Other advantages of this mode include minimal modifications to existing hardware and no previous compiler analysis required for prefetching data. Also program-based prediction is potentially more accurate than history based techniques.

1.1 Contribution

Hardware prefetchers have the following drawbacks:

1. They require modifications to standard hardware in form of prefetch units with large history tables [9, 15, 13].

2. No single prefetcher has been shown to be optimal for all types of application programs.

In contrast, slipstream mode encourages the use of existing general purpose hardware for purposes of prefetching, reducing synchronization time, coherence misses etc. Ibrahim et al. [14] showed that slipstream mode is faster than running one or two tasks on a CMP, with no prefetching enabled. We would like to know if running an application in slipstream mode, is even better than including a hardware prefetcher
1.1 Contribution

with the system. Therefore, as part of this thesis, we evaluated slipstream mode of execution by comparing it against performance obtained by conventional prefetching on a CMP-based environment. We simulated a sequential \textit{one-block-lookahead} prefetcher (Section 2.1.1), and an aggressive stride prefetcher (Section 2.1.2), for comparison purposes.

We show that, with increasing number of CMP’s, slipstream prefetching is more accurate and generates less memory traffic than the other two prefetchers. Hardware prefetching provides speedup over single mode only up to 8 CMP’s. With 16 CMP’s, slipstream mode performs better than hardware prefetching by 10\% and 13\% respectively, in two of the four applications that we studied. Slipstream mode is second-best in the range from 2-8 CMP’s. With further increase in the number of processors, we expect slipstream to continue to perform well.

Most conventional microprocessor systems \cite{7,31} include a hardware prefetch unit. With this in mind, we also studied how slipstream mode performance is affected by a hardware prefetcher in the system. We also ran experiments with the prefetcher enabled only for the \textit{R-stream} to see if the overall prefetch coverage and efficiency improves, due to prefetching by both the \textit{A-stream} and the hardware prefetcher.

We found that having a prefetcher enabled while running applications in slipstream mode does not affect the performance of the application, as compared to slipstream mode alone. In our limited study, there were little or no performance gains due to combined prefetching. However, because there is no degradation, we can use this mode with systems already having a hardware prefetcher.
Chapter 2
Prefetching Background and Related Work

Prefetching, in some form, has existed since the mid-sixties. Early studies of cache design recognized the benefits of fetching multiple words from main memory \[4\]. Hardware prefetching was later implemented in the form of instruction prefetching in IBM 370/165 \[12\] and Amdahl 470V \[3\]. Software prefetching is more recent, first mentioned by Smith \[30, 29\] in his study of cache memories. Prefetching is not confined to fetching words from memory into the cache. Instead, it is a generally applicable technique for moving memory objects up the memory hierarchy before they are needed by the processor. Prefetching mechanisms for instructions \[26\] and file systems \[8\] are commonly used to prevent processor stalls. However for the sake of brevity only data objects residing in memory will be considered in this work.

2.1 Hardware Prefetching

Hardware-controlled prefetching methods use special hardware units to implement prefetching. These units dynamically track data references by the processor. Hardware prefetching methods can be sequential or non-sequential. A simple exam-
2.1 Hardware Prefetching

A sequential scheme is the one-block-lookahead (OBL) prefetcher [29], which generate a prefetch request to the next cache block every time a cache miss occurs. Non-sequential schemes are stride-based or pair-based. They track and store memory references and try to find correlations between them to generate prefetches. No changes to existing executables is required, so instruction overhead is completely eliminated. Hardware prefetching tries to take full advantage of runtime information to make prefetching as effective as possible.

2.1.1 Sequential Prefetchers

![Sequential Prefetch Schemes]

The sequential approach tries to take advantage of spatial locality in programs. In this scheme [29, 6], access to a cache line causes successive lines to be prefetched. An advantage associated with this scheme is that it does not require any complex
2.1 Hardware Prefetching

hardware in comparison to some other prefetching schemes. The simplest sequential prefetching schemes are based on the one-block-lookahead (OBL) approach [29] which initiates a prefetch for block $b+1$ when block $b$ is accessed. This differs from simply doubling the block size in that the prefetched blocks are treated as different blocks with respect to cache placement and coherence policies.

Variations of this scheme [29] are based on the type of access that initiates prefetch of successive blocks. In a prefetch-on-miss approach prefetching is triggered only if an access results in a cache miss (Figure 2.1a). In a tagged-prefetch scheme, a tag bit is associated with every memory block to identify whether a block was demand fetched or a prefetched. Only the above two cases result in a prefetch (Figure 2.1b). This is done to reduce unnecessary prefetches. Smith [29] found that prefetch-on-miss was less than half as effective in reducing cache misses as tagged-prefetch scheme.

A disadvantage of sequential prefetch methods is the timeliness of prefetches. Prefetches may not be generated early enough to avoid processor stalls. In such cases, the number of consecutive blocks being prefetched may be increased from one to $K$, where $K$ is known as the degree of prefetching. A program in which the data access pattern is mostly regular with high spatial locality will support a greater degree of prefetching. An increase in the degree of prefetching otherwise will result in a large number of useless prefetches being generated, which will increase memory traffic and degrade performance.

Dahlgren et al. [11] proposed a way to counter the above problem by allowing $K$ to vary during program execution, to match the amount of spatial locality during a particular phase of execution. They defined a useful prefetch as one which subsequently resulted in a cache hit. The degree of prefetching, $K$, is varied as follows: if the number of useful prefetches during any part of program execution falls below a specified lower threshold, the value of $K$ is decremented. The value is incremented whenever the number of useful prefetches is greater than a specified upper threshold. The paper called the scheme Adaptive Sequential prefetching. It has been shown to
2.1 Hardware Prefetching

work well even on shared memory multiprocessors. In general however, sequential
prefetchers are less demanding than other hardware prefetchers in terms of modifi-
cations to standard hardware, and they seem to perform reasonably well for many
applications with good spatial locality.

2.1.2 Non-Sequential Prefetchers

Several more elaborate schemes [13, 15, 9] have been proposed to reduce the
number of prefetches generated. Such schemes monitor memory accesses directly
from the processor or accesses arising due to cache misses, and they predict future
addresses by correlating past references. There are mainly two types of non-sequential
hardware prefetchers: Stride-based and Pair-based.

Stride-based schemes [13] try to detect accesses with a constant stride in the pro-
gram and calculate future accesses. Programs having a long stream of references with
a constant stride generally perform well using this prefetching technique since the
prefetcher would capture such references and issue prefetches for future addresses.

Pair-based schemes [15, 9] identify correlations between pairs or groups of accesses
for example, between a miss and subsequent misses. Later when a miss is observed,
all addresses that correlate to this miss are prefetched. Pair-based schemes are more
general than stride-based schemes. However, they require more hardware in the form
of tables to identify correlations between miss addresses. In this work we mainly look
into stride-based schemes since they are more commonly used in multiprocessors.

Placement of Prefetched data

Jouppi proposed the use of Stream Buffers [16] for placing data obtained from
prefetching. Whenever any entry in the buffer is referenced, it is brought into the
cache and the remaining blocks are moved up in the buffer as in a FIFO queue. This
2.1 Hardware Prefetching

is done to reduce the cache pollution caused by unnecessary prefetches. Whenever a memory reference occurs, it is checked in the cache as well as at the head of the stream buffer. Further, Palacharla and Kessler [24] studied replacing secondary caches with stream buffers. In their work they showed that using eight stream buffers and \( K = 2 \) provided adequate performance for most of their benchmarks. The advantage of using Stream Buffers instead of an L2 cache in this case was the tremendous savings in chip area. However one disadvantage associated with their scheme is the increase in memory bandwidth requirements due to the large number of useless prefetches being generated.

Stride Prefetchers

Stride-based schemes try to detect sequential as well as non-sequential accesses with a constant stride and then generate prefetches, expecting such a pattern to continue in the future. To illustrate some key concepts, let us consider a simple matrix multiplication program from Figure 2.2. Assuming the matrices A and B are allocated row-wise in memory. The read accesses from A have a constant stride equal to one vector element, whereas read accesses from B have a constant stride equal to the size of one row.

```c
for(i=0;i<N;i++){
    for(j=0;j<M;j++) {
        for (k=0;k<N;k++)
            C[i,j] = C[i,j] + A[i,k] * B[k,j];
    }
}
```

Figure 2.2: Example showing strided references from the matrix multiplication algorithm
2.1 Hardware Prefetching

There are two issues with stride prefetchers. First a constant stride must be identified during the stride detection phase. Identifying a constant stride is done by monitoring instruction address associated with a particular load-store reference. This way it is possible to filter out accesses involving same stride sequence and detect a constant stride sequence. These schemes are called I-detection schemes, and they assume that the program counter is available to the prefetch unit. Stride prefetchers use a table called Reference Prediction Table (RPT) indexed by program counter, to store references and calculate strides. Once a constant stride has been detected, prefetches have to be issued early enough so that memory latency is completely hidden from the processor. This phase is called the prefetching phase. The performance of a stride prefetcher depends on the number of misses caused by the strided references in the program. Also, since prefetching is not initiated until a stride has been detected, the length of strided references is also important.

A simple stride prefetcher works as follows. Whenever an address misses in the cache, an entry is added in the Reference Prediction Table which contains the program counter and the data address \( D_1 \) from the instruction. The RPT is organized as shown in the figure Figure 2.3. Initially the state is set to no prefetch. Subsequently if another reference for an address \( D_2 \) is encountered involving the same instruction address, the state is incremented to transient and a stride \( S = D_2 - D_1 \) is calculated. Both \( D_2 \) and \( S \) are inserted in the table. Potentially this is the beginning of a stride sequence. Another reference by the same program counter for a data address \( D_3 \) equal to \( D_2 + S \) results in a hit in the RPT and state is updated to steady. Also the RPT issues further prefetches for addresses \( D_3 + S, D_3 + 2S, \) and so on depending on the degree of prefetching. Any reference by an instruction to an address which no longer belongs to a stride sequence results in a miss in the RPT and the state is decremented to either transient or no prefetch depending upon implementation.

A shortcoming of the I-detection schemes is that they assume the availability of instruction address to the RPT. This may be true in cases where the prefetcher is monitoring accesses to the L1 cache. However recent research [19] has shown that a prefetcher situated in the L2 cache performs almost as well, has more flexibility in
2.1 Hardware Prefetching

terms of hardware and can be more aggressive, since L2 cache is more tolerant of useless prefetches due to its large size. In those cases the instruction address may not be available to the RPT. There have been schemes proposed which deal with this issue \[18, 17\]. Such schemes usually compare data addresses with previous data addresses to detect and classify stride sequences which makes the detection mechanism more complicated.

**GHB Prefetch**

A recent research paper \[23\] proposed a new structure to implement prefetching called the *Global History Buffer*. This buffer holds the list of most recent misses
2.1 Hardware Prefetching

in FIFO order. This structure is different from conventional table-based prefetchers, which can hold useless data over long periods of time and thus generate useless prefetches when accessed. This method not only reduces stale data but also more accurately describes the existing pattern of data accesses to prefetch as well. This structure can be used to implement either a stride-based prefetcher or a correlation prefetcher, or even a hybrid prefetcher built by combining different types of stride or correlation prefetchers.

A GHB-prefetcher structure has two levels. The key matching is separated from the prefetch related history information.
2.1 Hardware Prefetching

Figure 2.4: Index Table and the Global History Buffer (GHB)
2.1 Hardware Prefetching

* An Index Table is accessed with a key as in conventional table-based prefetchers. The key may be the PC of an instruction, its cache miss address, tag etc. Each entry in the table consists of a pointer to an entry in the GHB. This is used to link to address lists formed within the GHB.

* A Global History Buffer is an n entry FIFO buffer holding the most recent n L2 miss addresses. Each entry also consists of a link pointer, which is used to chain the different entries together. For example, in a stride based-prefetcher all the addresses that were generated by the same PC will be linked together.

We now illustrate how to use the GHB as a stride prefetcher. As in conventional stride prefetchers, the GHB detects references with different strides by using the PC of the instruction causing the miss. In this case, an address list is formed in the GHB which consists of all miss addresses from the same PC. Stride detection is done by calculating the difference between consecutive addresses in the address list. If the difference between the first $x$ addresses is the same, then prefetches may be issued for addresses $a + s$, $a + 2s$ and so on, where $a$ is the current miss address and $s$ is the stride distance, depending on the degree of prefetching. The state machine for the prefetcher is implemented follows: whenever an address misses in the L2 cache, the GHB is updated in a FIFO fashion with the latest miss address added to the entry pointed to by the head pointer. The link entry for the new GHB entry is updated from the Index Table entry corresponding to the same PC. The Index table entry is then updated to point to this new GHB entry. In this way all the addresses from the same PC are chained together.

The GHB eliminates many of the problems associated with table-based methods. It allocates more space to recent table entries and thus reduces the stale data problem. Also the index table need only be large enough to hold the working set of prefetch-keys, whereas the GHB can be made larger to hold as many addresses as possible to hold a representative portion of the miss address stream.
2.2 Software Prefetching

A possible drawback to the GHB can arise due to the fact that it stores information globally; thus a stream of miss addresses can invalidate information about other prefetch streams. Nesbit and Smith [23] showed that implementing stride prefetcher using a GHB resulted in a 6% improvement in program execution time over conventional stride prefetching methods.

2.2 Software Prefetching

Compiler-directed prefetching or Software prefetching was first implemented by Potterfield et al. [25]. It consists of analyzing the program at compile time and inserting special prefetch instructions. Compared to hardware prefetching it has a higher potential for detecting future accesses by the program. However the prefetch instructions have to be issued early enough so that the data is available when the processor needs it. This type of prefetcher requires a little support from the hardware in the form of special non-blocking prefetch instructions provided by the ISA with the following characteristics. Prefetch instructions should not block demand loads from the processor. They should not cause interrupts or faults. Prefetch instructions should not stall the processor pipeline once the prefetch request has been issued to memory since the data is not required immediately by the processor. All these characteristics are generally provided so that prefetch instruction remains an optimization feature and does not affect program correctness or generate large or potentially unnecessary overhead. Modern ISA’s provide prefetch instructions which load data either directly into the processor registers or at various levels of memory hierarchy thus giving the compiler more flexibility while scheduling prefetches. In the multiprocessor case, software prefetching has tighter constraints than uniprocessors due to data sharing and cache coherency mechanisms between processors [32].

When compared to hardware prefetching techniques, software prefetching has different issues. It is difficult for the compiler to predict whether a particular cache line has been evicted due to a conflict miss. Furthermore scheduling prefetches is also difficult because the prefetch data must arrive just in time when the processor needs
2.2 Software Prefetching

It. The execution time between prefetch and matching memory reference may vary, as will memory latencies. These uncertainties are not predictable at compile time and thus require careful consideration when scheduling prefetch instructions. Prefetch instructions may be added by the programmer or by the compiler during optimization phase. Unlike other optimizations which are too tedious to implement by hand, prefetch scheduling can be effectively done by the programmer, especially in case of uniprocessors, where the memory latencies have less variability. The most common case where prefetching can be easily accomplished is within loops responsible for large array calculations. Such loops are common in scientific codes, exhibit poor cache utilization and often have predictable array referencing patterns. By establishing these patterns at compile time a prefetch instruction can be placed inside loop bodies, so that data for future iterations may be requested during the current iteration.
2.2 Software Prefetching

![Code examples showing how prefetch instructions will be inserted for loops](image)

```c
for(i= 0;i < N; i++)
ip = ip + a[i] * b[i];
```

(a) No prefetching

```c
for (i=0;i < N; i++) {
    fetch (&a[i+1]);
    fetch (&b[i+1]);
ip = ip + a[i] * b[i];
}
```

(b) Simple prefetching

```c
for (i=0;i < N; i+=4) {
    fetch ( &a[i+4]);
    fetch ( &b[i+4]);
ip = ip + a[i] * b[i];
ip = ip + a[i+1]* b[i+1];
ip = ip + a[i+2]* b[i+2];
ip = ip + a[i+3]* b[i+3];
}
```

(c) Prefetching with loop unrolling

Figure 2.5: Examples showing how prefetch instructions will be inserted for loops
2.2 Software Prefetching

As an example of how loop based prefetching works, consider a code segment shown in Figure 2.5. This loop calculates the inner product of two vectors a and b in a manner similar to the innermost loop of matrix multiplication calculation. If we assume that a single cache line holds four elements then every fourth reference will lead to a cache miss. We can avoid this by adding prefetch instructions as shown in the Figure 2.5. The number of iterations to look ahead is called the *prefetch distance* $\delta$ and it is expressed in terms of loop iterations as: $\delta = \lfloor l/s \rfloor$ where $l$ is the memory latency measured in terms of processor cycles, and $s$ is the time taken for the shortest path in a single loop iteration.

The previous example is a simple approach to prefetching and has a lot of drawbacks. One of the main drawbacks is that it would continue to prefetch data towards the end of the loop even if that data is not required. Although it may not affect program semantics extra prefetch instructions may degrade performance by stalling other loads or evicting useful data from the cache. Thus it needs to be modified to reduce unnecessary prefetches. However, most loop transformations for software prefetching are fairly mechanical and, with some refinements, can be applied recursively to nested loops.

One thing to be particularly noticed in the above example is the addition of a single prefetch instruction per loop iteration for prefetching. This overhead may be large if the number of loop iterations is high and may negate any benefits associated with prefetching. Increased instruction overhead is of more concern for processors with in-order pipelines, rather than today’s out-of-order superscalar processors. Also, software prefetching is effective for loops only if the array indices are linear functions of loop iterations. Such loops may be common in scientific codes, but are far less common in general applications. In many cases, prefetching is also hampered by irregular referencing patterns or the inability of the compiler to generate memory addresses correctly. To aggravate the problem further, a compiler may not be able to guess whether a prefetched line was evacuated due to conflict miss, and may need to be loaded again. In short, it cannot exactly predict the presence of an address in the
2.3 Slipstream prefetching

Most of the studies on prefetching have shown that both hardware and software prefetching perform well on certain applications with regular accesses and provide little benefit for applications with irregular reference patterns. In the latter case, both the above techniques fail to predict future accesses and generate useless or no prefetches. Compared to software prefetching, hardware prefetching is more dynamic and sees addresses as they come from the processor. However it has no idea of the overall program behaviour and can predict addresses only by finding correlations with previous addresses. Software prefetching has a better idea of overall program behaviour, it is either unable to guess addresses at compile time or predict the execution latency correctly resulting in poor timeliness of prefetches. To overcome this problem many hybrid approaches have been proposed [28, 33]. In case of multiprocessors, bus-based or otherwise, prefetching is more restrained because such systems have higher remote memory latencies and the memory system interconnects are more sensitive to excessive traffic generated by prefetching.

2.3 Slipstream prefetching

Recently Ibrahim et al. [14] introduced a new mode of execution for dual processor CMP-based distributed shared memory systems called Slipstream mode. This mode of execution is based on the observation that increasing the number of processors does not always improve execution time. As a task is divided into smaller regions to increase parallelism, the amount of synchronization and communication overhead begins to increase and form a significant part of overall program execution time. In such a scenario, any reduction in the overall execution time due to increased parallelism may be offset by the increased overhead. Instead, slipstream mode uses additional processors to reduce overhead due to communication and synchronization rather than increase parallelism. The resulting gains in efficiency can result in better performance than using the two processors for increased parallelism. The paper [14] showed that running in slipstream mode provides a speed-up of 12-29%, compared
2.3 Slipstream prefetching

to single mode, with one task per CMP, or double mode, with two tasks per CMP. Figure 2.6 shows the various modes of execution for a CMP.

Figure 2.6: Single, Double and Slipstream modes for CMP’s
2.3 Slipstream prefetching

Slipstream is a mode of execution which can be selected at run time by the programmer. However it does require the libraries to be slipstream aware for synchronization, task creation purposes etc. If the programmer enables this mode during run time, then the slipstream library routine creates two copies of each task and assigns the two on different processors on the same CMP node. Each copy has its own private data but shared data is not replicated. One of the copies is called the A-stream, while the other unreduced copy is called the R-stream.

The A-stream speculatively runs ahead of the original program and not only provides an accurate view of the future for the R-stream but also prefetches data for it into the shared L2 cache. If the data is still valid when the R-stream reaches its load, it will hit in the L2 cache. This view of the future provided by the A-stream is more accurate than conventional history-based predictors because it is based on actual program execution. This technique is similar to the one implemented by Chen and Baer [5] using a lookahead PC. However, their scheme requires significant modifications to the hardware in form of a Reference Prediction Table and a lookahead program counter to stay ahead of the normal program counter. Slipstream mode, instead, encourages the use of an available extra processor for prefetching purposes. Using slipstream mode for prefetching requires no changes to the coherence protocol or the memory subsystem. With some additional hardware support, the A-stream can also be used to provide hints for future sharing behavior at the directory. These hints can be used for coherence optimizations such as self-invalidation [21, 20].

One of the requirements for running an application in slipstream mode is that the A-stream must be reduced so that it can go ahead of the R-stream. One way to do it is to avoid executing long latency communication events. For many shared memory programs, synchronization events (barriers, locks etc) and shared memory stores can be avoided without affecting the A-stream’s ability to make correct forward progress. In order to skip synchronization operations, the system routines provided for these operations, for Splash-2 [34] are modified to support tasks running in slipstream mode. The A-stream does not perform these synchronization events, but the R-
2.3 Slipstream prefetching

stream executes these events as usual. This makes A-stream execution speculative. To prevent corruption of shared data by the A-stream, shared memory stores by the A-stream are executed by the A-stream however, the values produced are not committed to the shared L2 cache. Shared data is identified by marking the portion of the shared data accessed by the program with a unique task ID.

The paper also describes a mechanism for controlling how far the A-stream gets from the R-stream, called A-R synchronization. Letting the A-stream run far ahead of the R-stream may hide longer latencies, but it also may result in premature migration of data from processors, thus increasing the number of invalidations. This is analogous to prefetch distance in standard prefetchers. The mechanism for synchronization is coupled with barriers and other event-wait synchronizations. Thus program execution in slipstream mode occurs in sessions that end with barriers or other synchronization events. In each session the A-stream runs ahead and prefetches data for the R-stream. At the end of the session the A-stream either skips the synchronization event or waits for permission from the R-stream to continue further.

For controlling the distance that the A-stream can go ahead, the authors have used a token-bucket type of an algorithm, wherein every A-stream, upon completing a session, consumes a token and an R-stream, upon entering, inserts a token. To implement this method, the authors have used a shared hardware register. However this can also be easily established by using a single shared memory location and standard read-modify-write synchronization primitives. This register acts as a semaphore to control synchronization between an A-stream R-stream pair. The initial value of the semaphore indicates how many sessions can the A-stream get ahead without waiting for the R-stream. Synchronization can be local or global, indicating whether the A-stream progress is dependent on only the local R-stream or all R-streams during program execution. Synchronization is also required to check for a deviating A-stream. Since the A-stream is executing speculatively it may go astray; in those cases the R-stream serves as a checker for the A-stream and redirects it when needed. The checking is easy — if the R-stream reaches the end of the session before the
2.3 Slipstream prefetching

A-stream, they assume that the A-stream has deviated. This is a software-only check and is not based on the predictions made by the A-stream. The recovery mechanism is simple: the A-stream is killed and a new one is forked.

Because of the time-sensitive nature of prefetching, the choice if the A-R synchronization method has a significant effect on program performance. Using global synchronization with zero tokens initially limits the A-stream from moving on to the next session until all the participating R-stream’s have reached the barrier/event. This reduces number of premature prefetches by the A-stream but it also reduces the opportunities for prefetching early enough to fully hide the latency from the R-stream. This may be useful for programs with significant producer-consumer dependencies. Local synchronization encourages the A-stream to move further ahead, depending only on the number of tokens. This scheme may be good for programs with little or no sharing, and thus few conflicting accesses. Thus, the choice of A-R synchronization mode heavily affects the performance of slipstream prefetching. This choice is made once, along with the choice of running an application in slipstream mode, once, at the beginning of the program.

For the same program, different phases may perform well with different synchronization methods. Instead of choosing the appropriate mode at runtime, a program can benefit by executing different sessions of the program with different synchronization modes. One way to achieve this, implemented by Sivagnanam [27], is to change the A-R synchronization mode dynamically, during the execution of the program. Subhashini reported speedups of 10% for SOR and 7.9% for OCEAN using dynamic synchronization. Another way to implement this is to statically fix modes of execution during compile time or by profiling using traces [10]. This is an area of ongoing research.
2.4 Prefetch Metrics

The performance of a prefetching implementation can be measured in many ways. One way to do it would be to look at the speedup due to prefetching alone. Another metric to look at is the excessive memory traffic generated due to prefetching. However generally there are three popular metrics used for quantitatively analyzing the performance of a prefetcher and comparing prefetching techniques for different applications. We call *useful prefetches* as those prefetches that were available to deliver data timely to the processor. *Useless* prefetches are those that were not subsequently demanded by processor. The metrics used for performance evaluation are:

**Prefetch Efficiency** This is defined as the ratio of *useful* prefetches to the *total prefetches* issued. This metric tries to evaluate the accuracy of prediction of the prefetching implementation. A prefetcher with a higher efficiency will generate fewer useless prefetches.

**Prefetch Coverage** This metric tries to measure the ability of the prefetcher to correctly predict prefetch addresses. It is defined as the ratio of *useful* prefetches to the *total misses* that would have occurred in absence of prefetching. The better the coverage of a prefetcher, the greater will be the improvement in execution time.

**Timeliness** This metric measures the ability of the prefetcher to provide data when required to the processor. A prefetcher may have high efficiency and coverage, however it fails to deliver data soon enough to the processor. It is given as the ratio of the number of delayed prefetches to the total number of prefetches.

Generally providing one metric is not sufficient, since a prefetcher may have high efficiency but it may not generate sufficient number of prefetches to cover all misses. At the same time a high coverage doesn’t necessarily translate into good performance unless the prefetch efficiency is also high. Improving the overall execution time of the program is the ultimate goal of the prefetcher and these metrics often help explain the reasons behind any improvement or degradation in execution time due to prefetching.
2.4 Prefetch Metrics

In this work for performance evaluation purposes, we look at *prefetch efficiency* as well as *coverage*. We also look at the overall memory traffic due to prefetching, since performance in a distributed shared memory environment is sensitive to memory traffic. In the next chapter we explain the simulation methodology used and results obtained.
Chapter 3

Simulation methodology and Results

3.1 Simulation Environment

Our aim in this work is to evaluate the performance while running an application in slipstream mode as against running it in single or double mode with prefetching enabled in a CMP based system. In his work Ibrahim used a simulator SimOS \[1\] for simulating a CMP based system and running benchmarks in slipstream mode. We used the same simulator for simulating the different prefetchers. For each benchmark, we compared the best of the slipstream mode with other prefetchers. SimOS is a full system simulator that can be used for studying uniprocessor as well as multiprocessor systems. SimOS currently models MIPS R4000, R10000 and Digital Alpha microprocessor families. It simulates hardware in sufficient detail so as to boot and run commercial operating systems. It also provides better support for running realistic workloads than other traditional simulators. It provides three CPU models EMBRA, MIPSY and MXS for running simulations based depending on the level of detail required. The EMBRA model is the fastest with full simulation of devices and operating system execution, but no memory hierarchy simulation and hence workloads incurring a 10 times slowdown. The MIPSY model is more detailed and can be used for simulating memory hierarchies and cache models. Workloads run 100-200
3.1 Simulation Environment

<table>
<thead>
<tr>
<th>CPU: MIPSY-based CMP model, 1GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 Caches split:</strong></td>
</tr>
<tr>
<td>32KB, 2-way assoc, 1-cycle hit</td>
</tr>
<tr>
<td><strong>Memory cycles</strong></td>
</tr>
<tr>
<td>Bus Time: 30</td>
</tr>
<tr>
<td>PILocalDCTime: 60</td>
</tr>
<tr>
<td>PIRemoteDCTime: 10</td>
</tr>
<tr>
<td>NILocalDCTime: 60</td>
</tr>
<tr>
<td>NetTime: 50</td>
</tr>
<tr>
<td>MemTime: 50</td>
</tr>
</tbody>
</table>

Table 3.1: Simulation parameters used for experiments

times slower in this model. The last model MXS is used for accurately simulating processor pipelines, out-of-order execution etc.

In this work we carried out simulations using the MIPSY model running operating system IRIX 5.3. The target systems were dual-processor CMP-based distributed shared memory systems with 1-16 CMP’s. Each of the CMP nodes includes 2 processors with separate primary (L1) caches, a shared secondary (L2) cache and some portion of the globally shared memory. The shared L2 cache manages coherency among the two private L1 caches on each CMP node and can also merge requests from the two caches. Global Cache coherency is enforced through a fully-mapped invalidation-based directory protocol. The processor interconnect is modelled as a fixed delay interconnect. Contention is modelled at the network inputs and outputs and also at the memory controllers. The memory system latency and bandwidth parameters are as shown in table. They have been modelled so as to approximate an Origin 3000 memory system [2].
3.2 Benchmarks

SimOS provides a way to introduce workloads inside of the simulator and run them on top of simulated architectural models. For the purpose of this study we used benchmarks from the Splash-2 [34] suite. We ran experiments using three application programs from the Splash suite namely FFT, LU and OCEAN. One more application SOR was used from the NAS parallel benchmarks. The benchmark and their sizes are listed in Table 3.2.

The FFT kernel is a complex 1-D version of the radix-$\sqrt{n}$ six-step FFT algorithm, which is optimized to minimize interprocessor communication. The data set consists of $n$ complex data points to be transformed, and another $n$ complex data points referred to as roots of unity. Both the sets of data are organized as $\sqrt{n} \times \sqrt{n}$ matrices, partitioned so that every processor is assigned a contiguous set of rows allocated in its local memory. Communication in FFT occurs during the matrix-transpose phase, which requires an all-to-all interprocessor communication.

The LU kernel factors a dense matrix into a product of lower and upper triangular matrix. To improve memory performance, the dense $n \times n$ matrix is divided into $N \times N$ array of $B \times B$ blocks. To reduce communication block ownership is assigned using a 2-D scatter decomposition, with size $B$ large enough to keep the cache miss rate low and small enough to maintain a good balance. Elements within a block are contiguously allocated to improve spatial locality of the program.

<table>
<thead>
<tr>
<th>Application</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>64K complex double</td>
</tr>
<tr>
<td>Ocean</td>
<td>$258 \times 258$</td>
</tr>
<tr>
<td>SOR</td>
<td>$1024 \times 1024$</td>
</tr>
<tr>
<td>LU</td>
<td>$512 \times 512$</td>
</tr>
</tbody>
</table>

Table 3.2: Benchmarks and data set sizes
3.3 Experiments

The Ocean application studies large-scale ocean movements. It is representative of many applications that stream through a large data structure and perform little computation at each data point. At each horizontal cross section, several variables are modelled, including current temperature, pressure and friction. Each variable is discretized and represented by a regular two-dimensional grid. The current application in the Splash-2 suite uses a red black Gauss-Seidel multigrid equation solver for computation.

SOR is a nearest neighbor algorithm used to solve differential equations. The original algorithm has been discretized into two-alternate-nodal scheme known as red-black ordering to improve parallel performance.

3.3 Experiments

To select a prefetcher to compare with slipstream mode we looked at how prefetching occurs in slipstream mode. Since the $R$-stream is the main executing task, any reference by $A$-stream made to the secondary cache is analogous to a prefetch request made by an ordinary prefetch unit to the secondary cache. However since the $A$-stream places a request to the secondary cache only if there is a miss in the primary cache, this indicates that prefetching is triggered due to misses in the L1 cache. Another issue is to choose some prefetchers to compare against, among the different types of prefetchers available. One of the primary objectives of this study is to understand how good is the prediction accuracy of the $A$-stream for prefetching. A natural comparison would be against similar hardware prefetchers which predict prefetch addresses based on previously observed memory reference patterns. The simulated prefetchers have the following characteristics: prefetching is triggered due to misses arising in the primary cache and the prefetched-data is placed in the secondary cache. This action resembles the behavior of $A$-stream in slipstream mode.

The first of the prefetchers is a simple sequential one block lookahead prefetcher which prefetches successive cache lines upon misses in the primary cache. The degree
of prefetching for the experiments is 2. We conducted simulations to see how prefetch efficiency and coverage varies with the degree of prefetching. It is found that for the OBL prefetcher best results are obtained when the degree of prefetching is kept at 2 and so from now on for all experiments, the degree of prefetching is kept at 2 for the OBL prefetcher. The next prefetcher simulated is an aggressive stride GHB prefetcher described in Section 2.1.2. The size of the index table is 256 entries while the size of the GHB is kept at 512 entries. These are the same values used in the original work [23] for studying the prefetcher for uniprocessors. While simulating the GHB prefetcher we have not included the latencies for table access and adding or deleting a node into the GHB. We don’t expect them to be very large, though, especially when compared to the latency of a miss. Another stride prefetcher was also simulated; however early results showed its performance to be not as good as the first two and hence its results are not discussed in this work.

3.4 Experimental Results

The first set of experiments were conducted to measure and compare speedups due to prefetching with that obtained against the best slipstream A-R synchronization mode for that benchmark. We ran the four benchmarks discussed in section Section 3.2 for 1,2,4,8 and 16 CMP configurations and observed speedups obtained due to prefetching over single mode no-prefetch for the same number of CMP’s.
3.4 Experimental Results

Figure 3.1: SOR Speedup due to increasing number of processors

Figure 3.2: OCEAN Speedup due to increasing number of processors
3.4 Experimental Results

For SOR and OCEAN benchmarks, the results show that slipstream outperforms both, single and double modes, with 16 CMP’s. From Figure 3.1 double mode performs better than single up to 4 CMP’s. Beyond that slipstream mode seems to perform better than others. This can be attributed to the fact that for both these benchmarks, slipstream mode has very high prefetch efficiency, see Figure 3.3 and 3.4, so the number of prefetch messages is low. However at 2 CMP’s slipstream has poor coverage, resulting in missed opportunities for prefetching which results in worse performance than the other two for 2 to 4 CMP’s. As the number of CMP’s is increased, the coverage for slipstream improves, resulting in better performance. Also a high efficiency of prefetch means that there are not many invalidations going around and thus it provides some speedup even at 16 CMP’s.

![SOR traffic graph]

Figure 3.3: Traffic generated at the secondary cache level for SOR for different prefetchers
Figure 3.4: Traffic generated at the secondary cache level for OCEAN for different prefetchers
3.4 Experimental Results

For FFT benchmark, from Figure 3.5, it can be seen that beyond 2 CMP’s, there is no speedup obtained in double mode, indicating that FFT is not scalable beyond 2 CMP’s. However beyond 2 CMP’s there is no speedup obtained by prefetching as well for FFT. This can be explained as follows. Due to premature prefetching of data items, there are a lot of invalidation messages being passed around in FFT, which degrade performance. This can be seen from the memory traffic chart for FFT, Figure 3.6. It shows that beyond 2 CMP’s invalidation messages form a significant part of the memory traffic at the secondary cache level. At 8 CMP’s for OBL prefetcher, invalidation messages contribute 50% of the network traffic, resulting in poor performance. GHB prefetcher performs the best with approximately 20% network traffic consisting of invalidations. Slipstream is second best with 25% network traffic from invalidations. None of the prefetch schemes including slipstream, provide any speedup beyond 4 CMP’s and thus we will not use this for comparison purposes.

![FFT Speedup](image)

Figure 3.5: FFT Speedup due to increasing number of processors
3.4 Experimental Results

Figure 3.6: Traffic generated at the secondary cache for FFT for different prefetchers
3.4 Experimental Results

From Figure 3.7, it can be seen that prefetching doesn’t provide any benefit for LU. Double mode seems to perform much better than single as well as slipstream mode which means that LU is scalable right up to 16 CMP’s. The unusual behavior of LU is mainly due to two reasons. The size of the L2 cache being 1 MB, it seems that the entire working set of LU fits in the cache and since all schemes prefetch into L2 cache, there is no benefit obtained from prefetching at all. Another reason may be the fact that LU kernel uses blocking to exploit temporal locality and reduce communication. Thus very little or no benefit is obtained from prefetching and hence we shall not study LU in further detail.

![Figure 3.7: LU Speedup due to increasing number of processors](image)

Figure 3.7: LU Speedup due to increasing number of processors
3.4 Experimental Results

In Figure 3.8 we summarize our observations and show that overall, slipstream prefetching has a higher efficiency and coverage in the two cases that showed speedup due to prefetching. Slipstream prefetching remains fairly accurate, close to 80% in both the benchmarks, even at 8 CMP’s, which helps reduce unnecessary prefetches and improve program execution time. In contrast, the accuracy of OBL prefetches drops to almost half of its value for uniprocessors, generating a large number of invalidations and degrading overall performance.

![Figure 3.8: Prefetch efficiency and coverage values for SOR and OCEAN for different configurations](image-url)
We are also interested in knowing whether performance in slipstream mode is affected if a conventional hardware prefetching is also enabled along with slipstream mode. This study is important because most microprocessor systems come with standard prefetching hardware already included. Thus we expect prefetching to be already enabled in most systems running applications in slipstream mode.

We used GHB as an example of an aggressive prefetcher that may be present in modern systems. We looked at two scenarios: In the first case the prefetcher is monitoring only R-stream requests and prefetches data based on R-stream accesses only. This is equivalent to assuming that the hardware for prefetching is available only on the processors executing the R-stream. In the next case we enabled prefetching for both A and R streams, which is similar to a scenario when the prefetching hardware is present in all the processing nodes within the CMP chip. In both the above cases however, the prefetcher brings data into the secondary cache.

We ran benchmarks FFT and SOR for the two cases above. From the results, from Figure 3.9 and 3.11, we see that prefetching doesn’t seem to benefit slipstream mode performance, significantly. However, the important thing to notice is that for both the benchmarks, there is little or no degradation obtained in performance of slipstream mode with prefetcher attached, except one. For FFT for 8 CMP’s, the performance degrades due to large number of invalidation messages, as can be seen from Figure 3.10, being passed around when slipstream is used with prefetching enabled. In case of SOR (Figure 3.12), the combined mode performs better for up to 8 CMP’s and beyond that it provides no performance gains. Thus slipstream mode can be used for CMP’s with hardware prefetchers without affecting performance significantly.
3.5 Combining Slipstream with prefachers

Figure 3.9: Effect of prefetching in slipstream mode for FFT

Figure 3.10: Memory Traffic values for FFT for Slipstream and combined schemes
3.5 Combining Slipstream with prefetchers

Figure 3.11: Effect of prefetching in slipstream mode for SOR
3.5 Combining Slipstream with prefetchers

![Traffic for SOR - Combined prefetching](image)

Figure 3.12: Memory Traffic values for SOR for Slipstream and combined schemes
Chapter 4

Conclusion and Future Work

4.1 Concluding Remarks

Slipstream mode of execution is used for CMP’s to speed up applications that have already reached their scalability limit and would show degraded performance with a further increase in the number of processors. We expect that as the number of processors increases, slipstream prefetching will continue to be accurate, because its prediction is based on the future view provided by the A-stream. In all of his experiments, Ibrahim [14] found that A-stream never goes wrong. So we believe that slipstream prefetching will continue to be accurate even with an increased number of processors. However, the accuracy of history-based prefetchers may reduce as the number of processors is increased, because of factors such as task scheduling, migration, etc.

We show that slipstream mode prefetching provides better speedup, than hardware prefetching, for two of the four applications that we studied. For the other two applications, slipstream mode was second-best, behind the aggressive prefetcher. Slipstream mode has a higher accuracy of prefetching and generates less traffic than the other two prefetchers. Therefore we expect its performance to dominate even as the number of CMP’s are increased beyond 16. However, in general, speedup due to prefetching reduces with increased number of processors. This reduction in
performance can be due to fewer opportunities for prefetching as the partition size reduces. We expect capacity misses to reduce as the number of processors increases, and thus prefetching performance falls. Another reason may be that we modelled fixed delay networks using SimOS, even as the number of processors increases, the delay for remote misses remains the same, so there is no extra memory latency hiding with increased number of processors.

Slipstream can also be used along with standard prefetchers as seen from results obtained from Figures 3.9 and 3.11. There is little or no degradation in performance when prefetching is enabled for either the \textit{A-stream} or \textit{R-stream}.

\section*{4.2 Future Work}

As part of future work, there are three areas we would like to explore:

\textbf{Number of benchmarks} Currently our simulator, SimOS, doesn’t have as many benchmarks as we would like to use. It is difficult to port programs into SimOS because of which we were not able to use more benchmarks. However we would like to see if our observations concur with a larger set of benchmarks.

\textbf{Number of processors} Currently SimOS supports simulations only up to 32 processors or 16 CMP’s. We would like to see if the performance of slipstream mode is better than using prefetchers even with increasing number of CMP’s.

\textbf{Combining slipstream mode and prefetchers} Since using slipstream with other prefetchers doesn’t degrade performance, we would like to see the effect it has, if any, on the \textit{A-R synchronization} mode used being used. We would also like to see the performance obtained with other slipstream-based optimizations like \textit{self-invalidation} along with a standard prefetcher.
Bibliography


BIBLIOGRAPHY


