OUT OF CONTEXT CACHE PREFETCHING

By

DAVID JAKOB FRITZ

Bachelor of Science in Electrical and Computer

Engineering

Oklahoma State University

Stillwater, Oklahoma

2008

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2008

OUT OF CONTEXT CACHE PREFETCHING

Thesis Approved:

Dr. Sohum Sohoni

Thesis Adviser

Dr. James Stine

Dr. John Acken

Dr. A. Gordon Emslie

Dean of the Graduate College

ACKNOWLEDGMENTS

I thank my advisor and friend, Dr. Sohum Sohoni, for the support and opportunity he has given me since joining the CAESAR Lab. Dr. Sohoni has accommodated me in every way for 2 years now, and a great deal of thanks is due. The experience has been extraordinary and I feel honored to have worked with him on this research.

I would also like to thank Dr. James Stine and Dr. John Acken for agreeing to serve on my committee and offering their support whenever needed.

Sincere thanks to Wira "Spencer" Mulia, who has assisted me in nearly every step of this work. Wira has made an enormous and often overwhelming task into something tractable, and has always been willing to work at a moment's notice to help solve any problems we encountered.

I also thank the National Science Foundation, who has supported this work through NSF grant #0720741.

Finally, I dedicate this work to my mentor and role model, my father.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

1.1 Motivation 2
1.2 Out of Context Prefetching Overview 3
1.3 Contributions 5
1.4 Thesis Outline 5

II. RELATED WORK 8

2.1 Prefetching 8
2.2 Operating System Interactions with Memory 9
2.3 Context Aware Systems 10

III. OUT OF CONTEXT PREFETCHING 12

3.1 Design Space Exploration 12
3.2 An Example 13
3.3 When to Begin Prefetching 14
3.4 Predicting Context Switches 16
3.5 The Case for CPU-Bound Processes 19
3.6 The Case for I/O-Bound Processes 21
3.7 Considerations for SMT and Multi-core Systems 23

IV. SIMULATION METHODOLOGY 25

4.1 Full System Simulation 25
4.2 Trace Driven Simulation 26
4.3 Workload 30

V. RESULTS 33

5.1 Context Switch Prediction 33

5.2 CPU-Bound Processes 35
5.3 I/O-Bound Processes 39
5.4 Optimum ℓ 42

VI. CONCLUSIONS 45

6.1 Conclusions 45
6.2 Future Work 48

REFERENCES 51

APPENDIX A 55

APPENDIX B 57

LIST OF TABLES

Table Page

4.1 Simics configuration 26
4.2 Dinero configurations 29
4.1 SPECcpu2006 benchmark descriptions 31
5.1 Time prediction errors for the predict last and hysteresis algorithms..........35
5.2 Process specific miss rate improvements..........37
5.3 Percentage of total accesses to the L2 belonging to each SPEC benchmark......38
5.4 Process specific miss rate improvements for I/O bound processes 41

LIST OF FIGURES

Figure Page

1.1 The CPU-Memory gap 1
3.1 Possible outcomes of out of context prefetching 14
3.2 The working set as a function of time 16
3.3 Context switches over time 18
3.4 Average memory bus utilization for the MCF SPEC2006 benchmark 20
3.5 Performance vs. Multi-programming 21
3.6 Average memory bus utilization for the X Window System 22
4.1 FSM representing hysteresis based predictor 27
4.2 FCM with N=2 28
5.1 Context predictor accuracy 34
5.2 Average bus utilization for MCF with no prefetching 36
5.3 Average bus utilization for MCF with L=10000 36
5.4 Maximum miss rate improvement for each SPECcpu benchmark 38
5.5 Average bus utilization for Xorg with no prefetching 40
5.6 Average bus utilization for Xorg with L=10000 40
5.7 Miss rate improvement for Xorg and XMMS with varied L and 256kb 43
5.8 Miss rate improvement for Xorg and XMMS with varied L and 2048kb 43

CHAPTER I

INTRODUCTION

In the last three decades, dramatic increases in CPU performance have led to the formation of the CPU-Memory gap, the difference in performance growth between the CPU and the memory. With each generation of CPU architectures, this gap widens, and has quickly become one of the most salient bottlenecks in contemporary computing. An overwhelming amount of research has been dedicated to closing the CPU-Memory gap [1, 30, 31, 32, 33], including work that attempts to mask its effects on performance [2, 3, 15]. Much of this work has been widely accepted and is in use in current architectures.

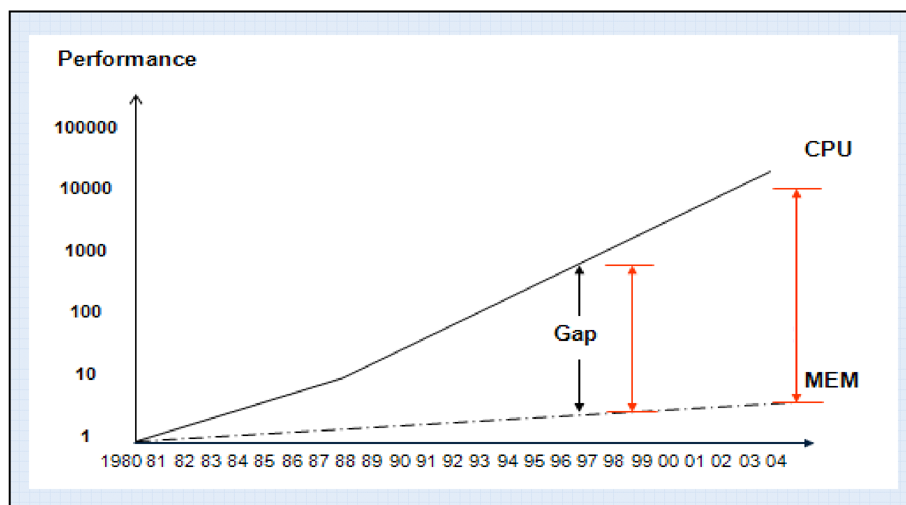

Figure 1.1- The CPU-Memory gap. Over time, CPU performance increases at a much higher rate than DRAM performance. This characteristic gap has motivated a great deal of research in computer architecture.

One effective way to mitigate the effects of the CPU-Memory gap is to prefetch data into the cache before it is used. Numerous prefetching techniques have been presented [4, 5, 6, 7, 8, 9, 10, 11, 12, 13], with varying degrees of effectiveness, mainly dependent on workload. This work does not propose a new prefetch algorithm, but rather attempts to explore the potential of leveraging a prefetch unit to work in a more intelligent way, as described in section 1.2 and chapter 3.

1.1 Motivation

All prefetching schemes operate on one premise – that regular patterns exist in data and instruction streams that can be exploited. While this is true, all prefetch units have 2 major shortcomings.

1. Even if a prefetch unit knows what to prefetch, there exists a case where the memory bus may be so busy that it cannot function. Prefetcher starvation is especially prominent when attempting to prefetch into small, busy caches.
2. Hardware prefetch units are "dumb" – they only look at the demand fetch access stream, instead of taking into account the wealth of information about the workload available at runtime. The case addressed here is that traditional prefetch schemes do not take into account the effects of multi-programmed systems. Prefetchers are interrupted at each context switch and must be re-trained for the current process.

This research introduces a system that attempts to address both of these issues.

With the advent of the multi-core era, data demands will increase. "Feeding the beast" will continue to be a problem, and as the degree of multi-programming increases

with multi-core machines, more intelligent prefetch paradigms will need to be in place to continue to be effective. Some studies have quantified the effects of multi-programming and operating systems on caches [14, 18] and conclude that these effects are significant and are growing.

1.2 Out of Context Prefetching Overview

The overall goal of this work is to provide a first step in creating a **less transparent architecture that can interact with the operating system in more meaningful ways**. This work focuses on creating a *context aware* architecture, specifically to prefetch data for an incoming process before that process is switched in by the operating system scheduler. By allowing the architecture to know and track the current running process, and potentially predict the next process, new areas of research are introduced, including context aware caching, process selective cache replacement algorithms, process predictive context switching, and others.

Out of context cache prefetching operates by first predicting the ID of an incoming process, and some time before the context switch occurs, it enables a prefetch unit which gathers data for the incoming process. The initial focus of this work is on the last on-chip cache assumed to be the level 2 cache, unless otherwise stated. Thus, the out of context prefetcher brings data into the L2 cache from main memory. The ideas explored in this research may be extended further down the memory hierarchy.

There are two classes of processes that out of context prefetching targets. Both classes of applications are explored in this work, and details are provided in sections 3.5 and 3.6.

1. ***CPU-bound, memory intensive processes***. These are processes that tax the main memory so much during their timeslice that traditional prefetch units simply do not have enough time to initiate transactions. This is true even if they know what to fetch. By beginning prefetching out of context, near the end of the timeslice for the current process when pressure on the memory is low, there is a potential for an aggressive prefetcher to get a head start on fetches and stay ahead of the demand fetches initiated by the process when in context. In order to ensure that less memory intensive processes are scheduled before more memory intensive ones, the operating system scheduler can be modified to identify and reschedule based on memory demands.
2. ***I/O-bound, highly interactive processes***. These are processes with relatively short timeslices, which display memory activity "bursts" at the beginning of each timeslice. These bursts represent a process rebuilding its working set, and are a primary symptom of cache thrashing due to aggressive multi-programming. These processes work on smaller sets of data and most of the cache misses occur at the beginning of each timeslice, so it may be possible to prefetch most of the blocks for the incoming processes. This has the effect of making the process more interactive and masks the effects of cache thrashing. Reducing the amount of thrashing may allow for higher degrees of multi-programming and greater total CPU utilization.

1.3 Contributions

This thesis provides contributions to computer architecture and computer science. It provides an initial step towards creating less transparent architecture that the operating system may then use to leverage greater performance from the hardware. More specifically, by creating a context aware memory system, the operating system has the potential to schedule memory much like current systems schedule the CPU. In a broader context, exploring context aware architectures opens up many interesting research topics for improving system performance for engineers and computer scientists alike.

In the area of computer science, this thesis suggests several interesting research topics in operating system interactions and especially scheduler algorithms. By creating a less transparent architecture, computer scientists have greater opportunities to create systems that better utilize system hardware.

The system described in this thesis is relatively inexpensive (in terms of area and power) to implement in modern memory systems, and does not require tampering with the critical path of the CPU. It leverages greater system performance through better utilization of the memory system.

1.4 Thesis Outline

The remainder of this thesis is as follows. We begin by discussing related work in the areas of prefetching, operating system interactions with caches, and other context aware software and hardware systems in Chapter 2. Chapter 3 provides a detailed description of the out of context prefetch system, including a discussion on when to schedule prefetching, predicting context switches, and considerations for different classes of

processes and multi-core systems. Chapter 4 describes our simulation methodology including a description of the simulator configurations, data analysis techniques, and CPU/memory configurations. Chapter 4 also provides a discussion and justification of certain configuration parameters, primarily in regards to simulation feasibility. Chapter 5 presents a discussion of simulation results. Data is presented for a simple context switch prediction unit, as well as a detailed discussion of the design space exploration. Finally, Chapter 6 provides conclusions and a discussion of future work in context aware architectures and out of context cache prefetching.

CHAPTER II

RELATED WORK

This chapter describes several of the most pertinent works in hardware and software data prefetching, operating system interactions with the memory hierarchy, and context aware architectures.

2.1 Prefetching

Jouppi's early work in prefetching [5] proposed the addition of simple stream buffers that identified unit length streams and prefetched several blocks ahead in a stream. For example, if block α is demand fetched, the stream buffer fetches $\alpha+1, \alpha+2, \alpha+3$, etc. Palacharla and Kessler extended this work to include stride directed prefetching [7], where non-unit strides can be fetched, i.e. $\alpha, \alpha+2, \alpha+4$, etc. Stream prefetching is especially effective and inexpensive to implement.

Another prefetch technique is Correlation based prefetching which, first introduced by Baer [11], associated prefetch addresses with demand fetches. When a demand fetch is initiated, the prefetch candidate is also fetched. Joseph and Grunwald [4], extend correlation-based prefetching by adding a Markov model to the reference stream.

Reinhardt et al. [12] proposed guided region prefetching, that uses compiler supplied prefetch hints to regulate a hardware prefetcher. Guided region prefetching aims to minimize the extra memory traffic introduced by hardware only scheduled region prefetchers.

Zucker et al. [13] proposed profiling software with an emulated hardware stream prefetcher, identifying candidate prefetches. These candidates would then be used to recompile the software with appropriately placed software hints. This method was quite effective in achieving a reduction in miss rate similar to the equivalent hardware.

This work does not call for a specific type of prefetch unit, just that it be capable of maintaining training data for different processes running on the system. Any of the techniques proposed above can be applied to the topics introduced here.

2.2 Operating System Interactions with Memory

Process scheduling and other operating system mechanisms can have a dramatic effect on cache performance. Increasing the degree of multi-programming increases performance in general by supporting greater utilization of the CPU. However, an upper limit exists where raising the degree of multi-programming will result in cache thrashing [19]. Chen and Bershad [14] assert that operating systems interfere with cache performance because of lower locality of reference, conflicts between the OS and user applications, poor page mapping algorithms, and other various issues. Chen notes that context switching interference is amortized over relatively long timeslices, but as timeslices get shorter and multi-programming increases, context switching interference becomes more significant.

2.3 Context Aware Systems

Koka and Lipasti [15] suggest scheduler modifications that include scheduling memory hierarchy. This is accomplished by scheduling threads that share memory one after another, which in effect warms the cache for the next process in the runqueue. This concept is similar to out of context prefetching (in fact Koka in passing suggests prefetching before a context switch occurs for an incoming process as an alternative design).

Suh et al. [16] has presented work detailing job-speculative page fetching from virtual memory. The approach presented is entirely software driven, controlled by the operating system, and employs a design very similar to the one proposed in this thesis.

Chiou et at. [17] has proposed scheduler based prefetching for various levels of the memory hierarchy, in very much the same way as Suh et al. [16] and this work. This system issues prefetch instructions in the scheduler and the potential exists for significant additional overhead to occur during context switching

All three of these techniques employ entirely software driven solutions to mitigating the effects of context switching on memory. The method described in this thesis differs primarily by being implemented in hardware, without any explicit requirement to modify well established scheduling algorithms and time critical code sections. It does not however, preclude such modifications. Out of context prefetching may also exist in current architectures, without the need to augment the ISA or require new versions of operating system code.

CHAPTER III

OUT OF CONTEXT PREFETCHING

The out of context prefetch system consists of two major components, the context switch prediction unit and the prefetcher. The context switch prediction unit handles tracking and predicting the incoming process ID and when the context switch will occur as well as controlling the prefetch unit. The prefetch unit behaves as it would in a traditional system, but since it must track one or more running processes, it must keep one or more context records that hold training data for each process. When the context switch prediction unit enables the prefetcher, the predicted process ID's context record will be used for prefetching.

3.1 Design Space Exploration

Like all prefetch schemes, out of context prefetching is sensitive to certain configuration parameters, such as cache size, predictor accuracy, and especially the workload. Out of context prefetching specifically addresses reducing sensitivity to workload, but also introduces new factors that need to be studied before an assessment on its efficacy can be made. The design space is enormous, with key parameters such as cache size, predictor accuracy, and context prediction accuracy, also practical implementation issues exist, such as the number of processes to track and system scheduler modifications.

Furthermore, since out of context prefetching can be viewed as an extremely aggressive prefetcher, the potential for pollution and interference with other prefetch schemes exists. This work focuses on exploring the parameters associated with out of context prefetching and their interactions with various cache configurations.

3.2 An Example

To better illustrate out of context prefetching, consider an example. Assume Process A is given the CPU, that is, a context switch occurs and Process A is now running. The context switch prediction unit predicts that when Process A is running, Process B will become the next run process 10 million cycles from the beginning of Process A's timeslice. A certain amount of cycles, ℓ, before the predicted context switch, the out of context prefetch unit begins to issue prefetch transactions for Process B. When the context switch occurs, the prefetch unit either continues prefetching in-context (a context hit), or begins to prefetch for the new process (a context miss). If no context switch occurs after ℓ cycles (and possibly some grace period), the out of context prefetch unit stops issuing prefetches, in order to prevent cache pollution. Note that the in-context and out of context prefetch units do not have to be the same. Figure 3.1 illustrates this and other possible outcomes of out of context prefetching.

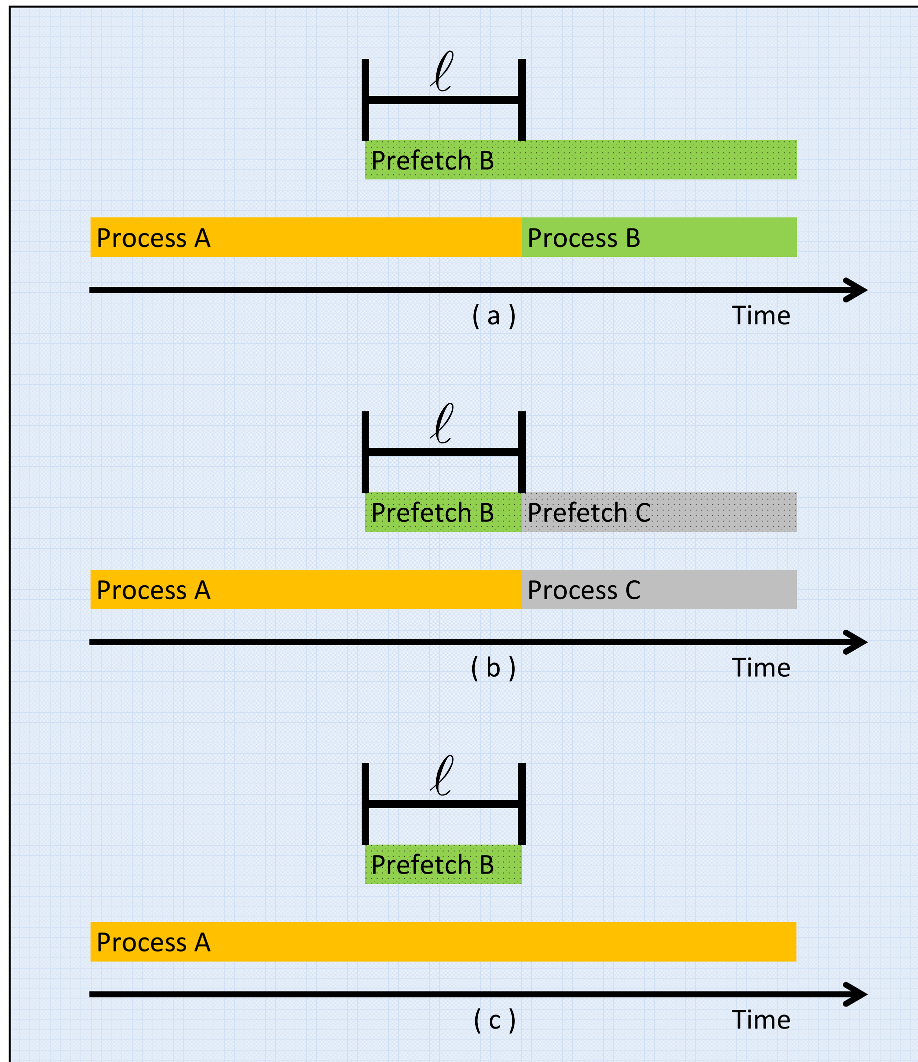

Figure 3.1 - Possible outcomes of out of context prefetching. (a) A context hit. Prefetching continues uninterrupted. **(b) A context miss.** The prefetcher must begin prefetching for the correct process, pollution occurs. **(c) A time prediction miss.** Pollution may or may not occur, depending on the future context switch.

3.3 When to Begin Prefetching

Accurately determining the time ℓ before a context switch occurs to begin prefetching is important both to ensure that enough data is prefetched to make out of context prefetching worthwhile and to prevent interfering with the memory footprint of the

current process. In order to determine ℓ, we must know how many blocks we intend to prefetch (or at least a range), and what the working set of the currently running process is.

The first of these measures is quite simple to calculate and can be determined at each context switch. The prefetcher's aggressiveness determines how many blocks to fetch. This number, multiplied by the average latency required to prefetch one block gives the time required to complete out of context prefetching. For example, if the average (based on average available memory bandwidth) latency to prefetch one block is 200 cycles, and we intend to prefetch 1000 blocks, we would need at least $\ell = 200 * 1000 = 200000$ cycles before the context switch occurs to complete the intended prefetching.

The second measure (the working set) is quite a bit more difficult to quantify on a real system. Out of context prefetching functions partially on the assumption that fetching into the cache will not interfere with the current running process. The current working set of a process decreases to zero as the process nears the end of its timeslice, as illustrated in figure 3.2. This is because a process can only work on so much data in a unit of time, and as the timeslice of a process runs out, the total amount of data it can work on also decreases. Knowing the working set of the current process is important because we do not want to interfere with data currently in use. In other words, prefetching too early, or inside the working set of the current process, can pollute the cache and degrade performance for the current process. It should be noted that even if we have knowledge of the working set of the current process, we must take care to ensure that certain cache lines in the current working set were protected during prefetching. This implies a modification to the replacement algorithm, and is not addressed in this paper.

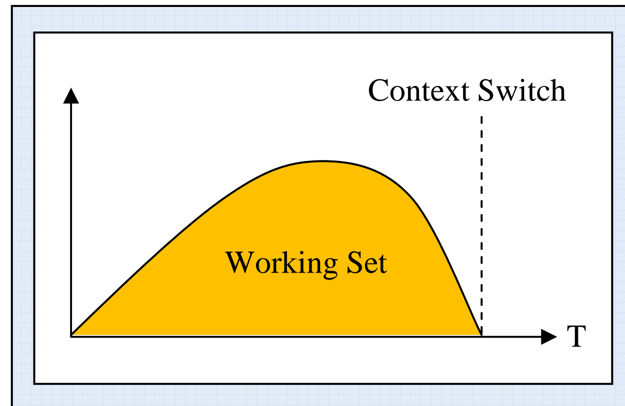

Figure 3.2 – The working set as a function of time. As a process nears the end of its timeslice, its effective working set approaches zero. It may be possible to prefetch into the cache at the end of the current timeslice without interfering with the current working set.

For this study, the value ℓ is measured in L2 accesses. This is done because performance indicators such as miss rate and cache pollution are functions of cache accesses rather than time. In a real system, ℓ would likely be calculated in cycles for the sake of practicality. Dedicated performance counters can be used to calculate average cycles per L2 access in real time, making a dynamic calculation of when to begin prefetching possible.

3.4 Predicting Context Switches

The efficacy of out of context prefetching is bounded by its ability to accurately predict an incoming process and when a context switch will occur. Context switching behavior, and thus the ability to predict a context switch, is driven entirely by the operating system scheduler. CPU scheduling occurs under one of four possible events [19]:

1. When a process switches from a running state into a waiting state, usually because of an I/O request.

2. When a process switches from a running state into a ready state, usually due to an interrupt.
3. When a process switches from a waiting state into a ready state. For example, when an I/O request completes.
4. When a process terminates.

For events 1 and 4, scheduling must take place. This is known as *cooperative scheduling*, because the process itself relinquishes control of the CPU. Scheduling during events 2 and 3 is known as *preemptive scheduling*.

In a system that only supports cooperative scheduling, predicting context is simple, because process runqueues are decided upon before the runqueue is executed, as in the case of round-robin scheduling. Preemptive scheduling poses a difficult problem because certain events such as interrupts and I/O completion can be difficult to predict. Examples of this include keyboard/mouse input, network I/O, DMA completion, and so on. However, as chapter 5 will show, a high degree of regular patterns and timing still exists and can be exploited.

Ideally, system schedulers attempt to optimize CPU utilization, throughput, or response time. Although, most systems implement simple priority based algorithms that allow for dynamic runqueue adjustment and user priority overrides. For example, the Linux kernel implements a preemptive, priority based algorithm. Processes are assigned priorities ranging from 0 to 140, with 0 being the highest priority. Higher priority processes are awarded longer timeslices, and lower priority processes shorter timeslices, ranging from 10ms to 200ms. During execution, the highest priority task that still has

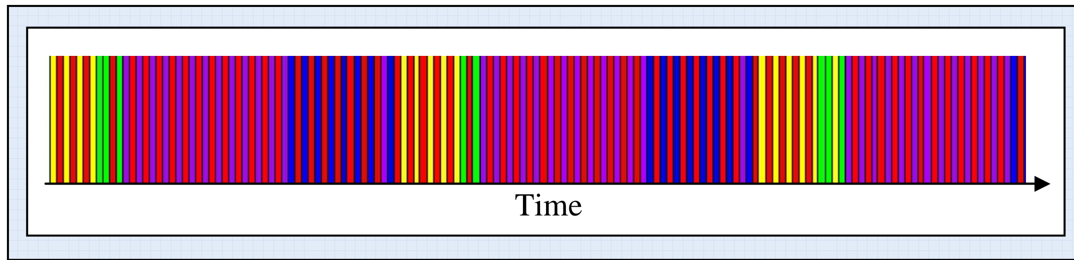

Figure 3.3 – Context switches over time. Each process is represented by a color. The most frequent timeslice represents a window manager application servicing requests to redraw the screen and handle mouse I/O.

time remaining in its timeslice, and is not waiting on I/O, will be scheduled. When a process exhausts its timeslice, it cannot run again until all other processes have exhausted their timeslices. The Linux scheduler measures the interactivity level of a process by detmining how long it has been waiting for I/O. It favors interactive processes and will increase their priority (by lowering the priority value) over time. Conversely, CPU-bound processes will have their priority lowered (by raising the priority value).

Figure 3.3 shows the context switching behavior of a Linux workstation. Several active processes are running, including a window manager, web browser, media player, and a CPU-bound process (in this case bzip2 from the SPECcpu2006 benchmark suite). Figure 3.3 indicates that even in a preemptive multi-tasking system, a significant amount of *exploitable* regular patterns exist for context switching behavior.

It should be noted that it is possible to avoid having to predict the ID of an incoming process by either snooping into the runqueue or having the scheduler inform the architecture as to the next incoming process. Either of these implementations will account for all non-preemptive context switches. However, snooping into the runqueue is expensive and dangerous, since operating system data structures can change between software versions. Requiring the scheduler to inform the architecture about the runqueue

can also be expensive, at least on a small scale, where process dispatching (the actual context switch) latency is critical. Furthermore, neither implementation informs the architecture *when* the context switch will occur on any useable timescale (timeslices are typically measured in milliseconds, far too course-grained for out of context prefetching). Future work may lead to an investigation of the potential merits of modifying the system scheduler to better accommodate out of context prefetching. This work explores the possibility of predicting context switches without the aid of the system scheduler.

3.5 The Case for CPU-Bound Processes

CPU-bound processes, those that have relatively few I/O requests, are the process type that most prefetchers target. This is because CPU-bound processes have longer contiguous timeslices, which promotes exploitation of process specific locality. Certainly, as least in part, early research in data prefetching was limited to single processes because full system simulators were too slow, inaccurate, or were not available to a majority of the research community. Furthermore, the *de facto* simulation benchmark suite, SPECcpu, which consists of CPU-bound applications by design. Contemporary cache prefetching techniques work well for CPU-bound processes [9, 12, 13, 20, 21], and since they tend to consume a majority of CPU time, Amdahl's law directs us to focus on them.

However, as the CPU-Memory gap widens, because of bandwidth limitations, it becomes increasingly difficult to prefetch, even if the prefetcher knows what data to fetch [22]. Figure 3.4 shows the average memory bus utilization for MCF, a SPEC2006

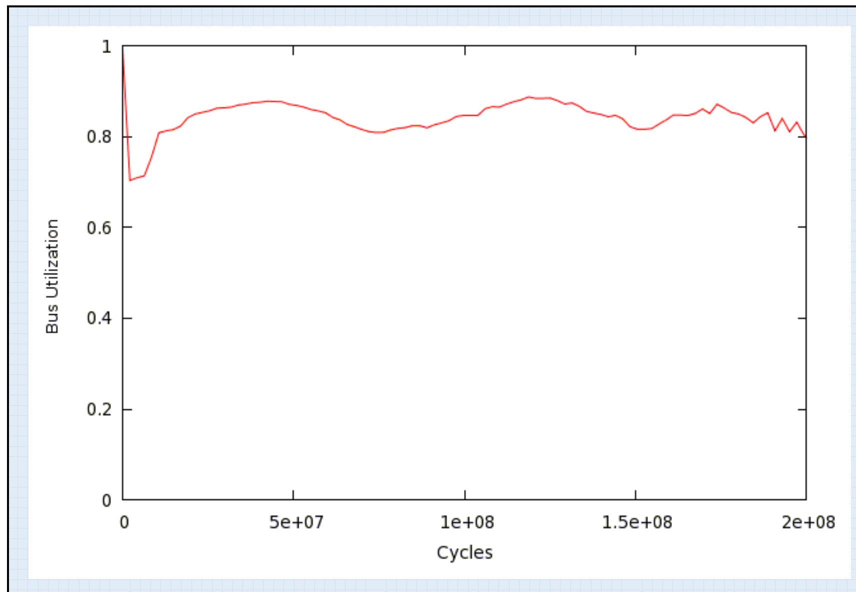

Figure 3.4 – Average memory bus utilization for the MCF SPEC2006 benchmark. For processes that maintain high memory bus utilization, traditional prefetch schemes may not be able to issue fetches even if they know what to fetch.

benchmark, on a scalar, in-order CPU with a blocking memory hierarchy. At above 80% for nearly its entire run, finding time to prefetch without interrupting demand fetches is clearly difficult. On a modern out-of-order, superscalar CPU, memory requirements increase, and it becomes even more difficult to prefetch, even if the prefetcher is highly accurate.

Out of context prefetching attempts to minimize the bus utilization caused by demand fetches by issuing prefetches out of context, when memory pressure is low. This reduction in demand misses potentially gives a regular prefetch unit a greater chance of injecting prefetches. The regular prefetcher then, depending on its accuracy, may be able to stay ahead of the demand fetches to the cache. A far more likely scenario is that the demand fetches will catch up to the prefetcher, and the effect will be a “wedge” of cache misses removed from the front end of the timeslice of the process.

3.6 The Case for I/O-Bound Processes

I/O-bound processes are those that spend more time waiting on I/O than actually computing data. I/O-bound processes can create significant performance loss because of exceedingly long latency on certain I/O events such as keyboard/mouse input, network I/O, etc. In part to work around this bottleneck, systems began time-sharing resources to mask the latency of I/O. While multitasking has allowed for significant performance improvements, as the number of processes running on a system and the degree of multi-programming increase, *thrashing* becomes a problem.

Thrashing is a condition where a system performs less and less meaningful work (progress) because resources spend more time working on non-progress related tasks. A classic example is page thrashing, where the working sets of all of the processes on the system do not fit into physical memory. Pages must be "swapped" to a higher level memory, usually the hard disk, and brought back in when needed. As multi-programming increases, the operating system spends a disproportionate amount of time "swapping" pages, causing an overall loss in system performance.

Cache thrashing occurs in much the same way and has the same symptoms. As processes contend for space in the cache, the cache becomes increasingly less capable of masking main memory access latency. As the degree of multi-programming increases,

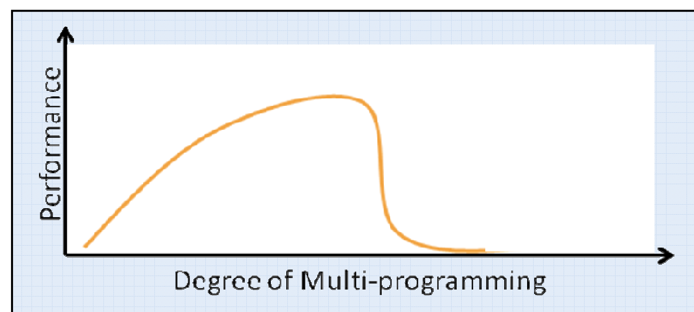

Figure 3.5 – Performance vs. Multi-programming. As the degree of multi-programming increases, thrashing can quickly degrade system performance.

locality of reference becomes less apparent, and latencies increase.

Unfortunately, prefetch units suffer just as caches do. At every context switch, prefetch units have to spend time retraining. Even after the prefetch unit retrains, bandwidth limitations may exist because the running process is busy refilling the cache with its working set. Figure 3.6 shows the average bus utilization for an I/O-bound process. Note the exponential decay of bus utilization, which implies the process is able to fit its working set into the cache, but thrashing causes it to refill the cache every time it is switched in. Even if a prefetch unit knows what to fetch when the context switch occurs, it may not have any available bandwidth to work with, and by the time bandwidth becomes available, prefetching in the current timeslice may yield little performance gain. Furthermore, it is unlikely that a traditional prefetch unit would know what to fetch at the beginning of the timeslice, since locality of reference has been compromised, causing the prefetcher to waste time retraining. The type of prefetcher determines the amount of accesses required to retrain.

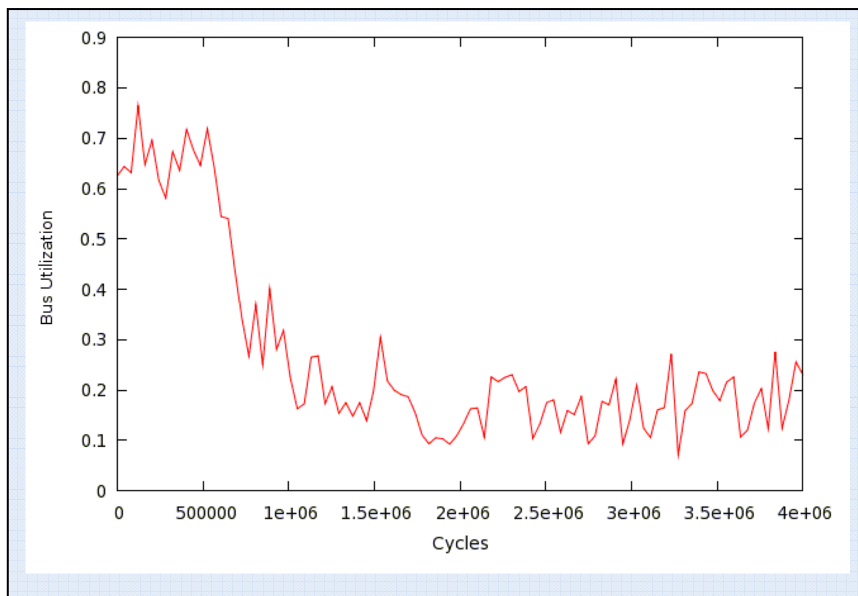

Figure 3.6 – Average memory bus utilization per timeslice for the X Window System. I/O-bound processes tend to display a memory “burst” at the beginning of each timeslice.

3.7 Considerations for SMT and Multi-core Systems

With the multi-core era well in place, memory bandwidth is becoming a scarce commodity [22, 23, 24]. Shared memory resources must divide their bandwidth between all of the cores that access them. Process scheduling becomes a significant bottleneck. When a core "steals" a process from another core to balance the load, the entire working set of that process must move with it, incurring an enormous amount of memory traffic. Even prefetching can create issues in multi-core systems. When a "dumb" prefetcher is too aggressive, it may inadvertently create false sharing and increase coherency traffic [34].

Out of context prefetching is an intelligent prefetching scheme that adapts well to multi-core paradigms. As these systems grow in number of cores, and parallel programming becomes more commonplace, the number of active threads on a system will also grow. Traditional cache hierarchies and prefetch schemes will have trouble scaling, especially as the degree of multi-programming of these systems increases.

CHAPTER IV

SIMULATION METHODOLOGY

This work employs both full system, execution driven simulation, and trace driven simulation to generate data. Since this work is preliminary in nature, and simply attempts to explore the efficacy of out of context prefetching in a broad sense, this work bases the simulations on simple models to reduce simulation time. This allows for performing a wide range of simulations covering several key parameters. In total, this work presents results from more than 350 simulations.

4.1 Full System Simulation

The Virtutech Simics Full System Simulator [25] generates various trace files that are used in the trace driven simulators. Simics is a full system execution driven simulator that can simulate Alpha, ARM, IA-64, MIPS, PowerPC, SPARC v9, x86, and x86-64 families of processors. The overall design of Simics is geared to be generic and flexible (it can simulate embedded systems, desktops, multiprocessors systems and clusters). Its modular, object-oriented design allows for great ease of use, despite its large size and complexity. We use Simics/x86 with the *Tango* target. *Tango* is a simplified single, scalar, in-order x86 CPU running Fedora 5 GNU/Linux. While Simics is capable of simulating more complex, out-of-order, super-scalar machines, the simulation time in

these systems grows by several orders of magnitude. The prohibitive time requirement for running these simulations restricts the level of detail we are able to simulate in any tractable amount of time. Since memory demands increase with the addition of technologies such as super-scalar processing, any improvement in performance for a simplified model will also yield improvement in a super-scalar, or even multi-core environment. Table 4.1 details the simulator and cache configuration.

This work uses two sets of trace files. The first contains an annotated trace of every context switch that occurs over a 10 billion instruction simulation. The trace file contains the PID and timestamp (in cycles) of each context switch. These files are then fed into a several simple prediction units to measure the efficacy of predicting context switches.

The second file type is an annotated memory reference trace. Every L2 access is written with the current running PID, and timestamp (in cycles). These files are fed into a trace driven simulator to measure the effects of various L2 cache configurations and other metrics, described in the next section.

4.2 Trace Driven Simulation

As described in section 4.1, two types of trace files were generated by Simics. The first type is a timestamp annotated trace of every context switch over a 10 billion instruction

CPU	2GHz single core, scalar, in-order x86 ISA.
L1 I-Cache	64kb, 4-way, 64 byte block, LRU, 2-cycle read penalty.
L1 D-Cache	64kb, 4-way, 64 byte block, Write-back, LRU, 2-cycle
L2 Cache	2048kb, 8-way, 128 byte block LRU, 10-cycle read/write
Main Memory	200 cycle access penalty

Table 4.1 - Simics configuration. The L2 and main memory configurations are used only to generate traces for the context switch prediction traces.

simulation. This file is then parsed by three simple context switch prediction units, written in Perl.

The first predictor employs a simple predict-last algorithm, where an entry in a table is made for each PID on the system containing the last context switch from that process. So, for example, if Process A is switched out for Process B, the next time Process A is switched in, the unit will predict that Process B will come next.

The second predictor builds on the predict-last algorithm, but attempts to filter out switches that occur due to preemption. This is known as hysteresis based prediction. In this algorithm, a pattern must be repeated twice before the table entry is overwritten. For example, if the prediction for the PID after Process A is currently Process B, another pattern, such as Process A to Process C, must occur twice before the prediction is changed. This can be implemented using a simple 2-state finite state machine, as shown in Figure 4.1.

The third and final predictor evaluated is a finite context method predictor [26], common in the use of text compression [27]. A finite context method predictor, FCM, generates predictions based on a sequence of previous values. FCM's implement counters for each possible next value after a sequence of order N, as shown in Figure 4.2.

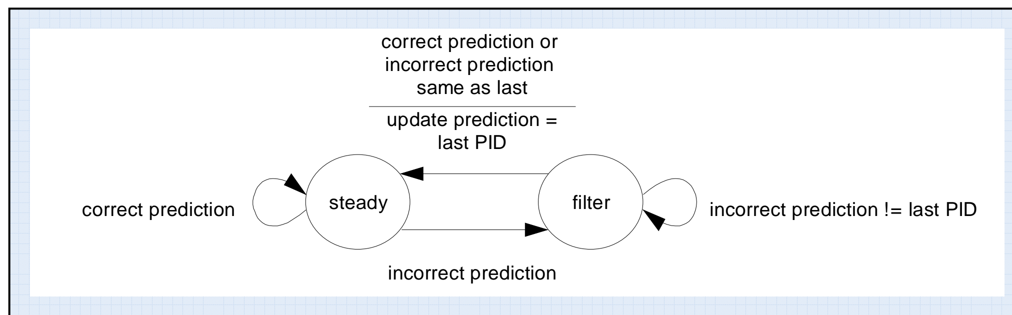

Figure 4.1 – FSM representing hysteresis based predictor. A hysteresis predictor filters out single occurrences of context switch misprediction. This is effective for ignoring preemptive context switches.

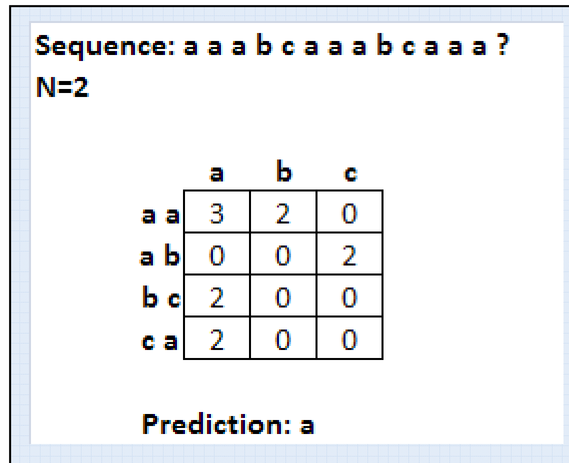

Figure 4.2 – FCM with N=2. As the sequence is fed into the FCM algorithm, a scoreboard of next values for each N length pattern is kept. The highest scoring value for a given pattern will be predicted.

When a particular sequence is encountered, the value with the highest count will be predicted. This type of predictor is difficult to implement in hardware, as the table size needed to keep track of a large number of PID's can be prohibitively large.

The second trace file set contains annotated memory references over a 10 billion instruction run. An extensively modified version of the DineroIV cache simulator [28] is used to generate several performance indicators. Dinero is a trace driven cache simulator originally developed at the University of Wisconsin as part of the Wisconsin Architecture Tool Set. It is written in C, with the full source available, and is easily modified for various types of cache simulation.

The role of Dinero in this study is to explore the efficacy of out of context prefetching by simulating over a range of parameters likely to affect performance. In most studies of cache configurations and prefetching, the primary performance indicator is miss rate. This work uses both overall and process specific miss rate as the primary performance indicators. By using process specific miss rates, we may evaluate the

L1 I-Cache	64kb, 4-way, 64 byte block, 2 cycle read
L1 D-Cache	64kb, 4-way, 64 byte block, Write-back, 2 cycle read/write
L2 Cache	{256kb, 512kb, 1024kb, 2048kb}, 8-way, 128 byte block, 15 cycle read/write
Main Memory	400 cycle access
ℓ	{0 (no OOC prefetching), 100, 500, 1000, 1500, 2000, 2500, 5000, 10000}
Workload	9 configurations, see section 4.3

Table 4.2 – Dinero configurations. Nine simulated workloads with nine values for ℓ and 4 cache sizes creates 324 DineroIV configurations.

efficacy of out of context prefetching for each class of process (CPU-bound vs. I/O-bound). Process specific miss rate is simply the average miss rate for a process per timeslice.

The focus is on exploring the potential benefits and pitfalls of varying *when* to prefetch, rather than what kind of prefetch mechanism to use. Therefore, a "perfect" prefetch unit is simulated by maintaining a buffer of future block accesses for the next process. When the simulator triggers the out of context prefetcher, references are prefetched from this buffer, until either the buffer runs out (which may occur for a large ℓ and short timeslice), or a context switch occurs. Furthermore, the prefetcher operates as conservatively as possible, issuing prefetches only when no other demand miss will occur (this is only possible by looking ahead in the reference stream), thus preventing out of context prefetching from directly negatively altering the total stall time in the cache (cache pollution may still occur, indirectly causing additional stall time). This idealized model simplifies simulation and minimizes the impact of variability of factors not relevant to this study – prefetcher accuracy and bus interference.

As a result of implementing a "perfect" prefetch unit, "perfect" context prediction is also implemented. This is done simply because the output of the prefetch unit is undefined when context switches are incorrectly predicted. An incorrectly predicted

context switch implies that the prefetch unit brings invalid data into the cache, which is by design not possible.

More than 350 simulations are conducted for this study (roughly 600 hours of CPU time to complete), consisting of combinations of varying cache size and ℓ (how early to begin prefetching before a context switch occurs), compared against 9 different workload configurations, detailed in section 4.3. Table 4.2 presents the simulation configurations.

4.3 Workload

Out of context prefetching requires a unique workload that represents a "typical" workstation, running multiple processes of various class (CPU-bound vs I/O-bound). To provide CPU-bound processes, nine of the SPECcpu2006 benchmarks are used, as shown in Table 4.3. The SPECcpu benchmark suite is comprised of several CPU-bound programs designed to stress the CPU. Originally developed to compare the relative performance of real computing systems, it has been adopted as a *de facto* standard set of benchmarks for computer architecture simulation. It should be noted that SPEC benchmarks do not tax the memory subsystem [29], and a future study into out of context prefetch will require a more robust workload. The rest of the system is comprised of a standard Fedora Core 5 GNU/Linux operating system running Xorg, a windowing environment, XMMS, a multimedia player (playing F.D. Roosevelt's "Infamy" speech), and Firefox, a web browser (loading several websites in a scripted loop). Several other relatively idle processes are also running, including the window manager, swap daemon, and so on. Nine total workloads are created by running one of nine different SPEC

applications with the system described above. Each of the nine configurations is run for 10 billion instructions.

Benchmark	Description
bzip2	A compression utility.
GCC	The GNU C Compiler, version 3.2. Compiles code for an Opteron target.
MCF	Combinatorial optimization for vehicle scheduling. Uses a network simplex algorithm.
soplex	Solves a linear program using a simplex algorithm.
povray	Image rendering using ray tracing.
h264ref	Video compression using the H.264 standard.
astar	Path finding using the A* algorithm.
sphinx3	Speech recognition system from Carnegie Mellon.
xalancbmk	XML processing. Converts XML to other document types.

Table 4.3 – SPECcpu2006 benchmark descriptions. SPECcpu2006 applications represent a diverse selection of scientific workloads.

CHAPTER V

RESULTS

This chapter presents results from simulations as described in the previous chapter. In this chapter we present four sets of results. The first section discusses context switch prediction as well as predicting context switch timing. The second and third sections discuss the impact out of context prefetching has on CPU-bound and I/O-bound processes, respectively. The fourth and final section covers results pertaining to finding an optimum value for ℓ .

5.1 Context Switch Prediction

As mentioned before, the ability to accurately predict both the ID of the next process to be switched in, and when that will occur, creates an upper bound on the effectiveness of out of context prefetching. Preemptable scheduling algorithms create entropy in an otherwise perfectly predictable system. It appears however, that even with a preemptable scheduler, predictable regular patterns still exist, as shown in figure 3.3. Figure 5.1 shows the results of the three context prediction units described in section 4.2. Recall that the first prediction unit employs a simple "predict last" algorithm, the second unit uses a second order hysteresis algorithm to filter out one time preemption based context

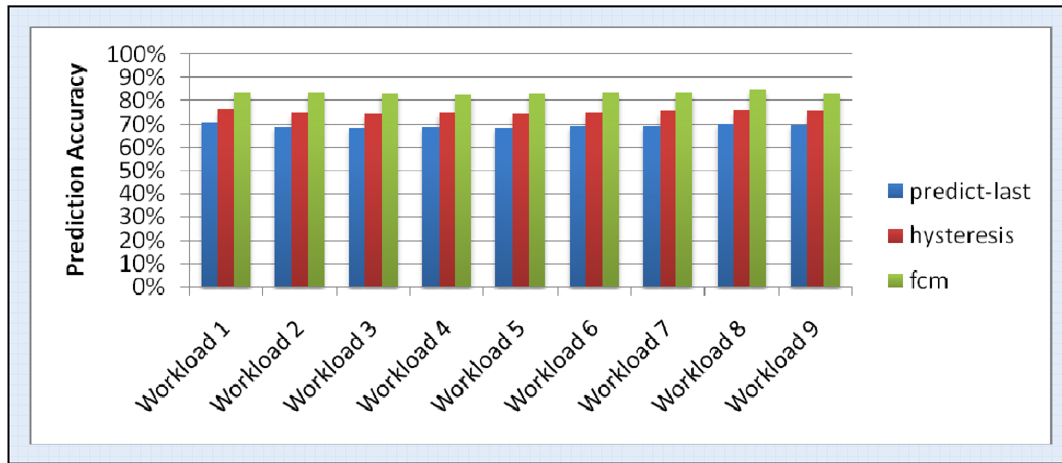

Figure 5.1 – Context predictor accuracy. Even with simple and inexpensive algorithms, a high degree of prediction accuracy is easy to obtain. Although fcm has the highest accuracy, it is impracticle to implement in hardware, although the hysteresis algorithm is relatively inexpensive.

switches, and the third unit employs a third order finite context method algorithm. Considering their simplicity, all three algorithms performed surprisingly well. The hysteresis algorithm, for example, has a prediction accuracy of 75.3% on average, and correctly predicts the time of each correctly predicted context switch to within 0.2% of the average timeslice length (in cycles). Table 5.1 presents additional data regarding context switch prediction. The ease with which context can be accurately predicted is a significant observation, not only for out of context prefetching but for all context aware architectural enhancements.

Greater prediction accuracy can be obtained with a more in depth study of context switching behavior. An obvious means to increase context switch prediction is to modify the system scheduler to provide details about the runqueue to the architecture. As mentioned in section 3.4, this does not enable the architecture to predict the time of the context switch any more accurately, and still does not account for processes that may preempt the runqueue. It may be possible to predict when certain processes will preempt the runqueue by measuring a history of past preemptions or using

Benchmark	predict-last time prediction	hysteresis time prediction
bzip2	0.32%	0.17%
gcc	0.21%	0.31%
mcf	0.21%	0.28%
soplex	0.23%	0.32%
povray	0.31%	0.19%
h264	0.25%	0.28%
astar	0.25%	0.29%
sphinx3	0.23%	0.25%
xalanbmk	0.27%	0.28%
average	**0.25%**	**0.26%**

Table 5.1 – Time prediction errors for the predict last and hysteresis algorithms. Time error for fcm is not calculated.

hints from hardware that assert interrupts that drive preemptive context switches. Predicting preemptions may be reserved for very high priority processes, as the probability of incorrectly predicting may be greater, and maintaining high priority process specific data in the cache, even at the sake of performance loss for other processes, may be tolerated.

5.2 CPU-Bound Processes

The primary purpose of applying out of context prefetching to CPU-bound processes, and in fact the original hypothesis of this entire work, is to reduce demand-fetch driven bus utilization for memory intensive processes, in order to allow a traditional prefetch unit to drive more prefetches while in context. This is accomplished by fetching blocks into the cache out of context, when pressure on the memory subsystem is low. In effect, the memory accesses become more uniformly distributed.

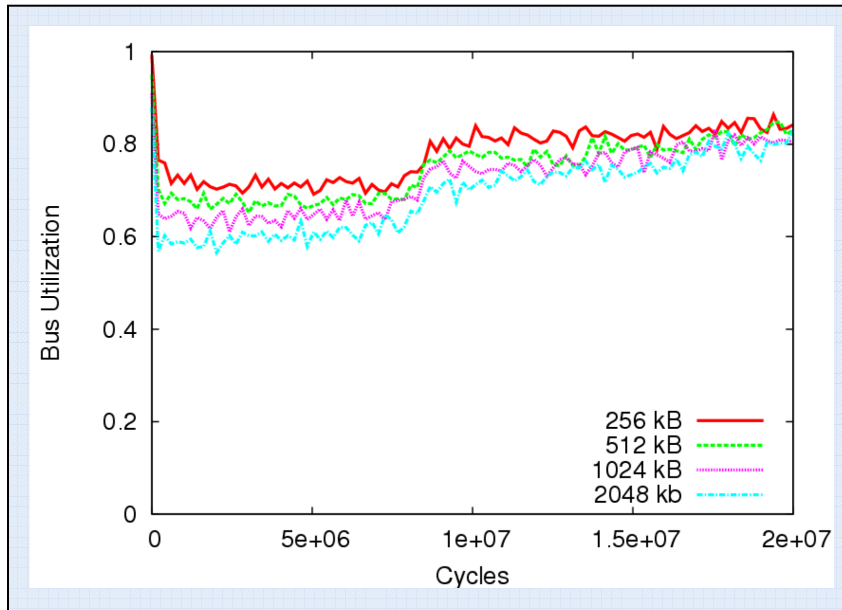

Figure 5.2 –Average Bus utilization per timeslice for MCF with no prefetching. Bus utilization is high for each configuration, indicating that most of the misses are compulsory.

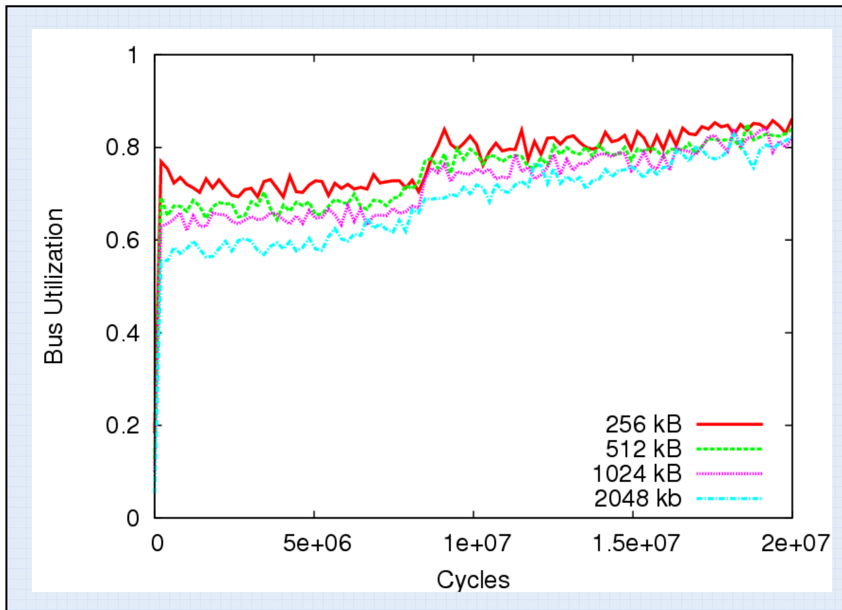

Figure 5.3 –Average Bus utilization per timeslice for MCF with $\ell = 10000$. Even with an aggressive out of context prefetching scheme, little demand-fetch bus utilization reduction occurs.

However, two factors prevent out of context prefetching from making any meaningful increase in performance for CPU-bound processes – low memory bus utilization for the applications studied, and long, contiguous timeslices. The first factor, low memory bus utilization, may in fact just be an artifact of the particular workloads chosen for this

Cache Size	256kb	512kb	1024kb	2048kb
bzip2	0.00%	0.62%	1.40%	2.07%
gcc	0.06%	0.61%	2.05%	2.52%
mfc	0.00%	0.06%	0.15%	0.31%
soplex	0.01%	0.10%	0.45%	2.19%
povray	8.60%	31.39%	48.92%	55.42%
h264ref	0.01%	4.67%	24.52%	54.15%
astar	1.40%	4.22%	5.92%	8.50%
sphinx3	3.58%	10.33%	28.65%	35.69%
xalancbmk	-0.08%	0.14%	1.36%	5.61%

Table 5.2 – Process specific miss rate improvements for $\ell = 10000$. Xalancbmk actually has a hit rate loss with a 256kb cache.

study. As mentioned before, Sair and Charney conclude that SPECcpu does not overexert the memory subsystem [29]. This does not imply that memory intensive workloads do not exist, or are even rare. Rather, it simply suggests that SPECcpu may not be a valid workload for memory intensive related studies. Figure 5.2 illustrates the bus utilization over time for the SPECcpu2006 benchmark MCF, with no prefetching, with varying cache size. MCF is the most memory intensive application in simulated in this study, and shows a worst case for out of context prefetching. Similar results exist for bzip2, gcc, and soplex. An entire catalog consisting of approximately 11,000 bus utilization plots generated for this study is referenced in Appendix A.

Even if a very memory intensive workload were simulated in this thesis, the second limiting factor, contiguous timeslice length, mitigates the effects of out of context prefetching by issuing many more, in some cases by several orders of magnitude, demand fetches. The number of demand fetches simply overshadows the number of blocks that can be practically fetched out of context. Figure 5.3 shows the memory bus utilization for MCF over time with $\ell = 10000$ (the maximum value simulated). Note that only a slight change is noticeable at the very beginning of the timeslice. Table 5.2 shows the process

	% References
bzip2	55.45%
gcc	52.06%
mfc	79.58%
soplex	62.64%
povray	8.62%
h264ref	11.42%
astar	29.01%
sphinx3	3.18%
xalancbmk	44.21%

Table 5.3 – Percentage of total accesses to the L2 belonging to each SPECcpu2006 benchmark. Notice that the processes with the greatest ratio of references also have the least performance improvement.

specific miss rate improvement for each of the nine SPECcpu benchmarks used in this study. All miss rate improvement calculations are defined as $1 - \frac{\textit{miss rate with ooc}}{\textit{miss rate without ooc}}$.

CPU-bound processes also dominate CPU time, and as such, have the greatest influence on overall miss rate in the cache. Table 5.3 shows the percentage of memory references originating from each of the SPECcpu2006 benchmarks in their respective

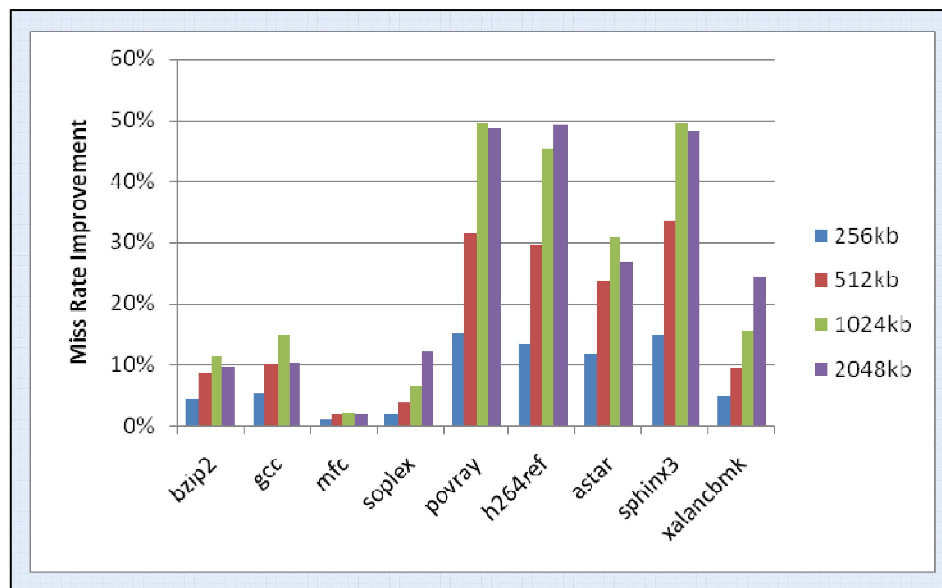

Figure 5.4 – Maximum miss rate improvement for each SPECcpu2006 benchmark simulated in this study. The processes with the most overall references – bzip2, gcc, mfc, and soplex, have the least improvement.

workloads. Figure 5.4 shows maximum overall miss rate improvements in the L2 for each configuration, each workload is identified by the SPECcpu2006 benchmark that ran as part of that workload. Complete results of overall miss rate improvements for every configuration is given in Appendix B.

Notice that the CPU-bound processes that have the greatest percentage of total references, as shown in Table 5.3, have the lowest miss rate improvement, shown in Figure 5.4. This indicates that these processes have long, contiguous timeslices that do not benefit from out of context prefetching. Also note that miss rate improvements decrease for the 2048kb cache over the 1024kb cache. This is due to capacity misses becoming low enough to mitigate the effects of out of context prefetching. The remaining misses are mainly compulsory.

5.3 I/O-Bound Processes

An I/O-bound process, and any process that is not memory intensive, will display a decrease of cache misses over time in general, as shown in Figure 5.5. In a system with a sufficiently large cache and few running processes, this characteristic may only occur once, when misses are compulsory. In an active system this characteristic may occur at each timeslice for each process because of limited cache capacity. If the degree of multi-programming on the system is increased, the amount of cache misses per timeslice may become prohibitive, causing a decrease in CPU utilization. A traditional prefetcher may not be able to prefetch these blocks because of the loss of locality of reference (causing the prefetch unit to re-train), lack of bus availability at the beginning of the timeslice, or both.

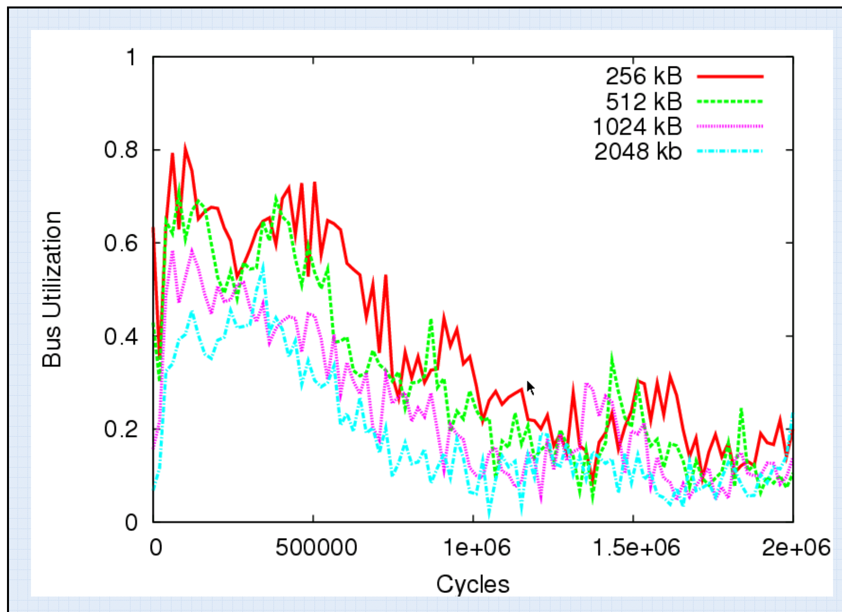

Figure 5.5 –Average Bus utilization per timeslice for Xorg with no prefetching. The memory burst at the beginning of each timeslice is apparent even with a large cache.

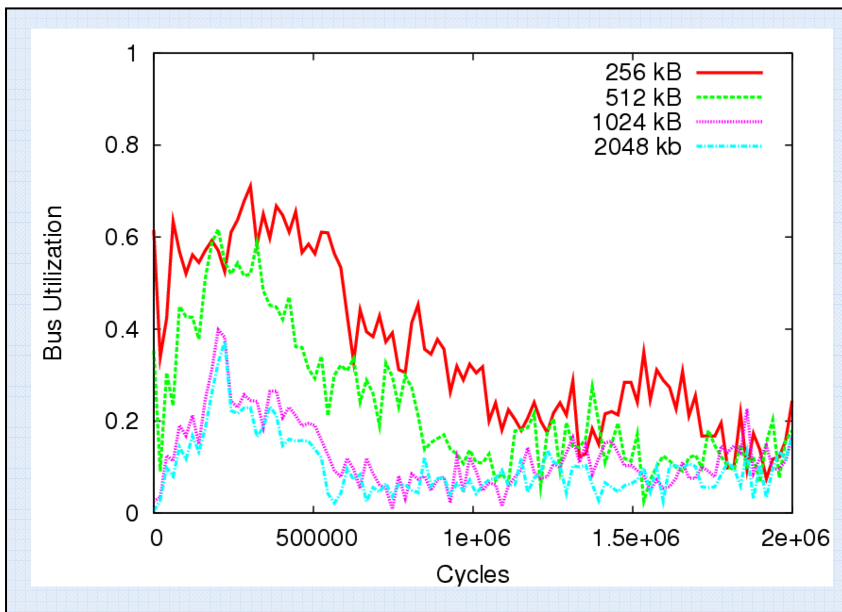

Figure 5.6 –Average Bus utilization per timeslice for Xorg with $\ell = 10000$. Out of context prefetching causes a significant reduction in bus utilization, dramatically reducing the characteristic exponential decay of bus accesses.

As for CPU-bound processes, out of context prefetching attempts to minimize cache misses by fetching blocks out of context. Unlike the case for CPU-bound processes however, the goal for out of context prefetching for I/O-bound processes is to reduce the

effects of thrashing, potentially allowing for higher degrees of multi-programming, and possibly making these processes more responsive.

Figure 5.6 shows the average bus utilization per timeslice for Xorg with $\ell = 10000$, and Table 5.5 lists the corresponding improvement in process specific miss rate. Notice that process specific miss rate improvements are significantly greater for I/O-bound processes than for CPU-bound. While this is simply because the total amount of memory references that constitute I/O-bound processes is lower, constituting a greater ratio of prefetches to demand fetches, it does indicate that out of context prefetching works well for this class of processes.

Although the workloads simulated for this thesis are dominated by CPU-bound processes, highly interactive systems may see significant overall miss rate improvements from out of context prefetching, particularly when no one process consumes the majority of CPU time. Future work for out of context prefetching begins with creating a more robust and representative workload.

Cache Size	256kb	512kb	1024kb	2048kb
Xorg	33.93%	58.67%	67.48%	61.20%
xmms	24.88%	44.88%	54.18%	46.59%
Firefox	1.14%	4.06%	9.54%	16.77%

Table 5.5 –Process specific miss rate improvements for I/O bound processes with best ℓ. Miss rate improvements are dramatically better than overall improvements in the cache. This is due to the CPU-bound processes dominating CPU time.

5.4 Optimum ℓ

The value ℓ determines the number of references to the L2 cache before a context switch occurs to begin out of context prefetching. It may seem intuitive to try to make ℓ as large as possible in order to ensure that as many blocks as possible are brought into the cache. However, making ℓ too large may cause the prefetch unit to bring data in while the currently running process still has a large working set, causing pollution for the current process. Furthermore, a large ℓ in a small cache may interfere with itself, although no practical implementation of this system would allow for more speculative loads than blocks in the cache. Even still, having a large ℓ when only a few blocks need to be prefetched may still cause pollution, and if ℓ is large enough, the prefetched blocks may be evicted by the current process before a context switch occurs.

Figure 5.7 shows process specific miss rate improvements for Xorg and xmms, with a 256kb cache and varying values of ℓ. Figure 5.8 shows the same data for a 2048kb cache. Notice that not only is miss reduction over ℓ non-monotonic, but the optimum ℓ varies between Xorg and xmms. This implies that to provide enough time to the out of context prefetcher to produce significant results and prevent pollution from beginning fetching too early, ℓ should ideally be variable for each process tracked. This work simulates a constant value of ℓ for each simulation, and future work will include a study of maximizing performance through variable, run-time calculated optimum values for ℓ. Notice in Figure 5.8 any negative slope is absent over the range simulated. This occurs because the larger cache size creates fewer capacity misses at the end of each timeslice, making ℓ less likely to interfere with the current process.

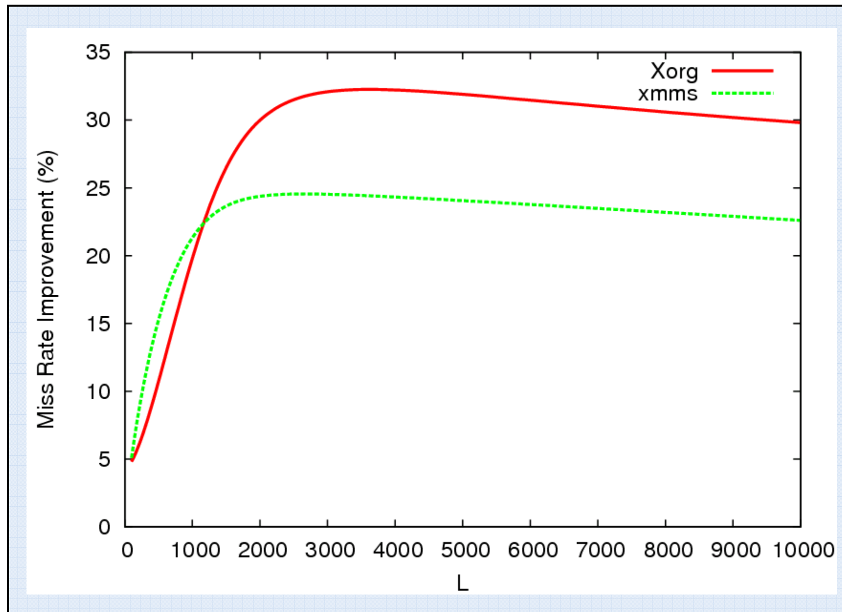

Figure 5.7 –Miss rate improvement for Xorg and xmms with a 256kb cache and varied ℓ. The maximum point for Xorg and xmms are different, implying that an optimum ℓ is based on the memory behavior of the incoming process.

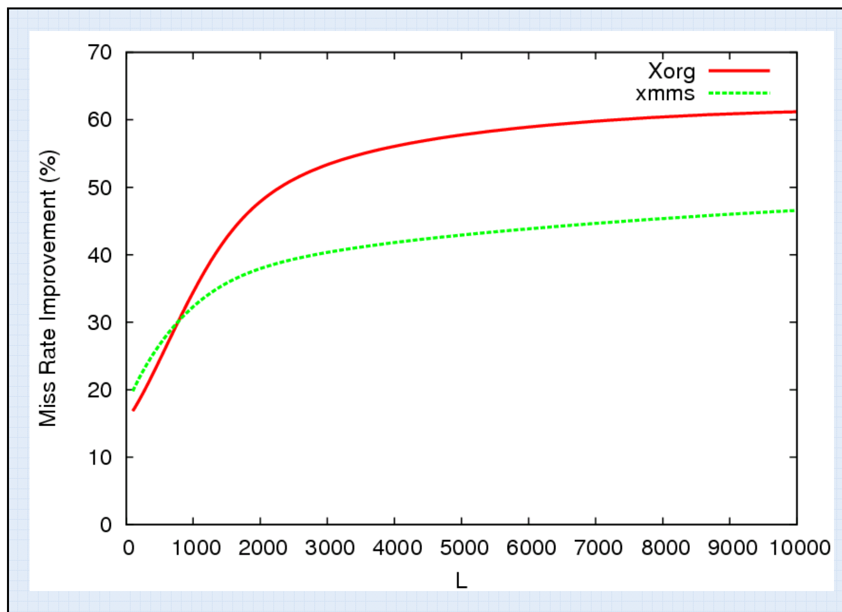

Figure 5.8 –Miss rate improvement for Xorg and xmms with a 2048kb cache and varied ℓ. The larger cache creates fewer capacity misses at the end of each timeslice, making performance less sensitive to large ℓ.

CHAPTER VI

CONCLUSIONS

This chapter provides a brief summary and concluding remarks for this thesis, followed by a discussion of future work building on the concepts introduced here.

6.1 Conclusions

Context switching effects can degrade performance by compromising locality of reference, causing additional cache misses not present in systems that do not employ multi-programming. These effects magnify when the degree of multi-programming is increased. Traditional cache prefetch schemes do not take into account the effects of context switching on cache behavior, which limits their effectiveness. Because of the memory "burst" typical of a process at the beginning of each timeslice, it may not be possible for a traditional prefetch unit to fetch these blocks after the context switch occurs, either because the bus is too busy, the prefetcher must re-train, or both.

Out of context prefetching combines a prefetcher with a context prediction unit, allowing for context-aware prefetching. This enables prefetchers to fetch for future localities, potentially making them more effective. This thesis provides an initial design exploration for out of context prefetching, as well as a first step in creating a more

transparent architecture that can interact with the operating system in more meaningful ways.

Accurately predicting context switching behavior sets an upper bound on the efficacy of out of context prefetching. If a context switch can be correctly predicted, an aggressive prefetch unit may be able to fetch blocks into the cache before the context switch occurs. A brief test of three simple, common prediction algorithms is presented, all of which correctly predicted context switch above 65% for each workload simulated. Timing predictions for correctly predicted context switches had an error of less than 1% for each workload. It is likely that more accurate context prediction algorithms can be developed, possibly using hints from the system scheduler to gain high levels of accuracy.

CPU-bound processes exhibit little I/O relative to the amount of compute activity, and as such receive longer, contiguous timeslices on multi-programmed systems. Some CPU-bound processes may tax the memory subsystem so much that traditional prefetch schemes may not be able to function because of bandwidth restrictions. In a worst-case system, out of context prefetching may be able to fetch enough blocks out of context, when pressure on the memory subsystem is low, to give a traditional prefetcher enough bandwidth to operate while in context. In reality, the overwhelming length of CPU-bound timeslices tends to mask any benefit out of context prefetching may produce. This study uses SPECcpu2006 benchmarks to provide a CPU-bound workload. SPECcpu2006 applications do not tax the memory system as much as needed to illustrate the effect of an over-burdened memory system, and more work needs to be done with more representative workloads before this aspect of out of context prefetching is discarded.

I/O-bound processes are those which display more I/O than compute activity. As a result, I/O-bound processes tend to have shorter and more frequent timeslices in an effort to increase CPU utilization and system responsiveness. On highly interactive systems, the operating system may dramatically increase the degree of multi-programming in an effort to increase CPU utilization. This can actually lead to less CPU-utilization due to thrashing in the memory system. Because locality of reference is compromised in highly multi-programmed systems, traditional prefetch schemes may not be effective. Out of context prefetching specifically addresses these effects, preserving process specific locality, and prefetching out of context. Results from this study clearly show that out of context prefetching can have a significant positive effect on I/O-bound process miss rates, which indicates it may be possible to significantly reduce the effects of thrashing, and possibly make the system more responsive.

This study examines varying cache sizes and values of ℓ, the time before a context switch, and naively assumes that there is one value of ℓ to serve each process on the system. It is clear that the value of ℓ is dependent on the specific memory and timeslice behavior of the target process, and a great deal of variation may exist between processes on a real system. As such, out of context prefetching needs to support a variable value of ℓ for each process in the system in order to maximize performance gains.

In summary, this thesis provides an initial exploration of a more intelligent prefetching scheme, with results that indicate out of context prefetching is a worthwhile avenue for additional research.

6.2 Future Work

This thesis serves as an introduction to a greater body of research that deals with context aware microarchitectures. Several worthwhile topics relating to out of context prefetching exist, including exploring the effects of out of context prefetching on multi-core systems, examining the potential of modifying the system scheduler to favor context aware memory, examining out of context prefetching for other levels of the memory subsystem.

With all of the new challenges raised by multi-core systems, is becomes imperative to create more intelligent architectures that work for, not in spite, of the software that runs on it. Out of context prefetching may provide the means necessary for prefetch units to continue to be effective in systems that are increasingly sensitive to the effects of context switching and other operating system concepts.

A future study into the potential of modifying the system scheduler to both arrange the runqueue in a way that promotes out of context prefetching, as well as provide hints to the context prediction unit, is already planned and funded. Since the upper bound on efficacy of out of context prefetching is set by the ability to predict context switches, it is certainly worthwhile to invest resources into maximizing prediction rate.

Out of context prefetching may be effective for other levels of the memory system, especially for the main memory, where page fault latencies are measured in milliseconds, with little promise of decreasing permanent storage access time in the near future. Examining out of context prefetching for memory pages has already been suggested [16], but more research needs to be performed.

Finally, simple, context-aware prefetching, needs to be investigated. Context-aware prefetching does not perform prefetching out of context, and therefore is not sensitive to the ability to predict context switches. In context-aware prefetching, a traditional prefetch unit maintains training data for each active process on the system. When a context switch occurs, the prefetcher simply uses the training data for that process, potentially maintaining greater degrees of locality in a multi-programmed system.

REFERENCES

[1] R. Crisp, "Direct Rambus Technology: The New Main Memory Standard," *IEEE Micro,* vol. 17, no. 6, pp. 18-27, Dec. 1997

[2] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, J.D. Owens, "Memory access scheduling," *Proc. 27th Ann. Int'l Symp. Computer Architecture,* pp. 128-138, June 2000

[3] A.J. Smith, "Cache Memories", *Computing Surveys,* vol. 14, no. 3, pp. 473-530, Sept. 1982.

[4] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *Proc. 24th Ann. Int'l Symp. Computer Architecture,* pp. 252-263, June 1997.

[5] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefech buffers," *Proc. 17th Ann. Int'l Symp. Computer Architecture,* pp 388-397, 1990.

[6] T. Mowry, M. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," *Proc. 5th Int'l Conf. Architectural Support for Programming Languages and OS*, pp. 62-73, Oct. 1992.

[7] S. Palacharla and R. Kessler, "Evaluating stream buffers as a secondary cache replacement," *Proc. 21st Ann. Int'l Symp. Computer Architecture,* pp. 24-33, April 1994.

[8] R. Cooksey, S. Jourdan, D. Grunwald, "A stateless, content-directed data prefetching mechanism," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and OS,* pp. 279-290, Oct. 2002.

[9] W. Lin, S. Reinhardt, and D. Burger, "Designing a modern memory hierarchy with hardware prefetching," *IEEE Transactions on Computers,* vol. 50, no. 11, pp. 1202-1218, Nov. 2001.

[10] G.S. Manku, M.R. Prasad, and D.A. Patterson, "A new voting based hardware data prefetch scheme," *Proc. Int'l Conf. High Performance Computing,* 1997.

[11] J.L. Baier and G.R. Sager, "Dynamic improvement of locality in virtual memory systems," *IEEE Transactions on Software Engineering,* vol. SE-2, no. 1, pp. 54-62, Mar. 1976.

[12] Z. Wang, D. Burger, K.S. McKinley, S.K. Reinhardt, and C.C. Weems, "Guided region prefetching: A cooperative hardware/software approach," *Proc. Int'l Symp. Computer Architecture,* 2003.

[13] D.F. Zucker, R.B. Lee, and M.J. Flynn, "An automated method for software controlled cache prefetching," *Proc. Int'l Conf. System Sciences*, 1998.

[14] J.B. Chen and B.N. Bershad, "The impact of operating system structure on memory system performance," *Proc. 14th ACM Symp. Operating System Principles,* pp. 120-133, Dec. 1993.

[15] P. Koka and M.H. Lipasti, "Opportunities for cache friendly process scheduling," *Workshop on Interaction between Operating Systems and Computer Architecture,* Oct. 2005.

[16] E. Suh, E. Peserico, S. Devadas, and L. Rudolph, "Job-speculative prefetching: Eliminating page faults from context switches in time-shared systems," *Technical report, Massachusetts Institute of Technology,* Memo 442, June 2001.

[17] D. Chiou, S. Devadas, J. Jacobs, P Jain, V. Lee, E. Peserico, P. Portante, and L. Rudolph, "Scheduler-based prefetching for multilevel memories," *Technical report, Massachusetts Institute of Technology,* Memo 444, July 2001.

[18] J.C. Mogul and A. Borg, "The effect of context switches on cache performance," *Proc. 4th Int'l Conf. on Architectural Support for Programming Languages and OS,* 1991.

[19] A. Silberschatz, J. Peterson, and P. Galvin, *Operating System Concepts*, Addison-Wesley, 1991.

[20] C. Zhang, S.A. McKee, "Hardware-only stream prefetching and dynamic access ordering," *Proc. Int'l Conf. Supercomputing*, pp. 167-175, 2000.

[21] J. Lee, S. Jeong, S. Kim, C.C. Weems, "An intelligent cache system with hardware prefetching for high performance," *IEEE Transactions on Computers*, vol. 52, no. 5, pp. 607-616, May 2003.

[22] D. Burger, J.R. Goodman, A. Kagi, "Memory bandwidth limitations of future microprocessors," *Proc. Int'l Symp. Computer Architecture*, pp. 78-89, May 1996.

[23] D. Burger, J.R. Goodman, A. Kagi, "Limited bandwidth to affect processor design," *IEEE Micro*, pp. 55-62, Nov. 1997.

[24] L. Spracklen and S.G. Abraham, "Chip multithreading: opportunities and challenges," *Proc. 11th Int'l Symp. Computer Architecture*, 2005.

[25] Virtutech. http://www.simics.net/

[26] Y. Sazeides, J.E. Smith, "The predictability of data values," *Proc. 30th Int'l Symp. Microarchitecture*, pp. 248-258, 1997.

[27] T.C. Bell, J.G. Cleary, I.H. Witten, *Text Compression*, Prentice-Hall, 1990.

[28] J. Edler and M. Hill, "DineroIV trace-driven uniprocessor cache simulator," http://www.cs.wisc.edu/markhill/dineroIV/

[29] S. Sair and M. Charney, "Memory behavior of the spec2000 benchmark suite," Technical report, IBM T.J. Watson Research Center, Oct. 2000.

[30] D. Psaltis, G.W. Burr, X. An, M. Levene, G. Barbastathis, A. Pu, "Holographic memories," *Lasers and Electro Optics Ann. Meeting,* 1997.

[31] S. Tehrani, E. Chen, M. Durlam, T. Zhu, H. Goronkin, "High density nonvolatile magnetoresistive RAM," *IEEE Electronic Devices Meeting*, pp. 193-196, 1996.

[32] A.B. Cosoroaba, "Double data rate synchronous DRAMs in high performance applications," *Proc. WESCON*, pp. 387-391, 1997.

[33] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, K. Yelick, "A case for intelligent RAM: IRAM," *IEEE Micro,* April 1997.

[34] A. Fedorova, C. Small, D. Nussbaum, M. Seltzer, "Chip multithreading systems need a new operating system scheduler," *Proc. 11th Workshop ACM SIGOPS*, 2004.

APPENDIX A

One of the goals outlined in the terms of the NSF grant funding this project is to provide all data, source code, scripts, configuration files, and any other generated data to researchers in an effort to maintain a high degree of reproducibility. A project website is maintained for this project to serve this need. Among the data and simulator source code is a catalog of nearly 11,000 plots generated for each PID in each cache configuration for each workload simulated in this study. The reader is encouraged to visit the project website to learn more about out of context prefetching.

Project website: **http://rome.ceat.okstate.edu/ooc/**

APPENDIX B

This section provides tables detailing overall cache miss rate improvement for each process under each simulated configuration. Cache miss rate improvement is defined as $1-\frac{miss\ rate\ with\ ooc}{miss\ rate\ without\ ooc}$. Each workload is identified by the SPECcpu2006 benchmark that ran in that workload.

bzip2				
Cache Size				
L	256kb	512kb	1024kb	2048kb
100	1.13%	2.94%	4.37%	4.15%
500	4.30%	8.60%	10.30%	8.30%
1000	4.09%	7.14%	8.47%	7.14%
1500	4.34%	7.89%	9.32%	7.72%
2000	4.41%	8.50%	10.10%	8.13%
2500	4.30%	8.60%	10.30%	8.30%
5000	3.94%	8.55%	10.63%	8.71%
10000	3.54%	8.50%	11.34%	9.79%

gcc				
Cache Size				
L	256kb	512kb	1024kb	2048kb
100	1.42%	3.51%	6.84%	5.83%
500	5.24%	10.24%	14.64%	9.97%
1000	4.87%	8.46%	12.53%	8.59%
1500	5.32%	9.63%	13.76%	9.33%
2000	5.36%	10.13%	14.37%	9.76%
2500	5.24%	10.24%	14.64%	9.97%
5000	4.71%	10.13%	14.99%	10.39%
10000	4.26%	9.63%	14.99%	10.50%

mfc				
Cache Size				
L	**256kb**	**512kb**	**1024kb**	**2048kb**
100	0.32%	0.74%	0.92%	0.91%
500	1.21%	2.17%	2.23%	1.73%
1000	1.10%	1.73%	1.85%	1.42%
1500	1.23%	2.03%	2.04%	1.54%
2000	1.23%	2.14%	2.16%	1.65%
2500	1.21%	2.17%	2.23%	1.73%
5000	1.12%	2.17%	2.32%	1.81%
10000	0.98%	2.06%	2.32%	1.85%

soplex				
Cache Size				
L	**256kb**	**512kb**	**1024kb**	**2048kb**
100	0.52%	1.32%	2.46%	6.03%
500	1.91%	3.91%	6.49%	12.07%
1000	1.78%	3.17%	5.36%	10.41%
1500	1.95%	3.67%	6.00%	11.31%
2000	1.95%	3.88%	6.39%	11.92%
2500	1.91%	3.91%	6.49%	12.07%
5000	1.70%	3.86%	6.59%	12.37%
10000	1.55%	3.59%	6.54%	12.37%

povray				
Cache Size				
L	**256kb**	**512kb**	**1024kb**	**2048kb**
100	4.17%	12.72%	23.71%	32.97%
500	15.10%	31.41%	48.58%	46.01%
1000	13.77%	25.45%	41.74%	41.30%
1500	15.16%	29.32%	45.91%	43.84%
2000	15.28%	31.11%	47.91%	45.29%
2500	15.10%	31.41%	48.58%	46.01%
5000	14.07%	31.41%	49.58%	48.19%
10000	12.98%	30.02%	49.58%	48.55%

h264ref				
Cache Size				
L	**256kb**	**512kb**	**1024kb**	**2048kb**
100	3.39%	10.05%	20.06%	31.61%
500	13.12%	29.65%	44.76%	47.74%
1000	12.53%	24.66%	38.32%	43.55%
1500	13.39%	27.79%	41.92%	45.81%
2000	13.44%	29.48%	44.01%	47.10%
2500	13.12%	29.65%	44.76%	47.74%
5000	11.67%	29.14%	45.36%	49.03%
10000	10.48%	27.45%	44.91%	49.35%

astar				
Cache Size				
L	**256kb**	**512kb**	**1024kb**	**2048kb**
100	3.19%	8.27%	13.71%	14.29%
500	11.72%	23.55%	29.63%	25.05%
1000	10.80%	19.64%	25.59%	21.92%
1500	11.72%	22.09%	27.94%	23.68%
2000	11.91%	23.18%	29.11%	24.66%
2500	11.72%	23.55%	29.63%	25.05%
5000	10.93%	23.64%	30.68%	26.42%
10000	9.82%	22.82%	30.94%	26.81%

sphinx3				
Cache Size				
L	**256kb**	**512kb**	**1024kb**	**2048kb**
100	4.13%	12.18%	23.22%	32.53%
500	14.76%	33.55%	48.52%	46.39%
1000	13.71%	27.82%	42.60%	42.47%
1500	15.11%	32.18%	46.60%	44.58%
2000	15.05%	33.27%	47.93%	45.78%
2500	14.76%	33.55%	48.52%	46.39%
5000	13.71%	33.45%	49.26%	47.59%
10000	12.73%	32.27%	49.56%	48.19%

xalancbmk				
Cache Size				
L	**256kb**	**512kb**	**1024kb**	**2048kb**
100	1.29%	3.35%	6.58%	12.27%
500	4.84%	9.73%	15.10%	23.72%
1000	4.45%	7.85%	12.32%	19.84%
1500	4.84%	8.91%	13.79%	21.88%
2000	4.97%	9.64%	14.87%	23.31%
2500	4.84%	9.73%	15.10%	23.72%
5000	4.23%	9.55%	15.57%	24.54%
10000	3.68%	8.63%	15.10%	24.34%

VITA

David Jakob Fritz

Candidate for the Degree of

Master of Science

Thesis: OUT OF CONTEXT CACHE PREFETCHING

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Tulsa, Oklahoma, on February 4, 1983, the son of Herbert and Debbi Fritz.

Education: Received a Bachelor of Science in Electrical Engineering from Oklahoma State University, Stillwater, Oklahoma in December 2006. Completed the requirements for the Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in May, 2008.

Experience: Employed by Oklahoma State University, School of Electrical and Computer Engineering, as research assistant, undergraduate research assistant, and teaching assistant, 2004-Present.

Professional Memberships: Eta Kappa Nu Honor Society, Institute of Electrical and Electronics Engineers

Name: David Jakob Fritz Date of Degree: May, 2008

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: OUT OF CONTEXT CACHE PREFETCHING

Pages in Study: 60 Candidate for the Degree of Master of Science

Major Field: Electrical Engineering

Scope and Method of Study: The purpose of this study was to examine the efficacy of modifying a hardware cache prefetcher to track and predict context switches and prefetch for incoming processes before they are switched in. The study was composed of three major components – quantifying the amount of context switches that can be correctly predicted, examining out of context prefetching on CPU-bound processes and I/O-bound processes, and examining the effects of varying how early before a context switch to begin prefetching.

Findings and Conclusions: Data suggests that highly accurate context switch prediction is viable, with our own simple prediction unit correctly predicting over 75% of context switches. The study shows that out of context prefetching may not work well with CPU-bound processes, as the positive effects are masked by the relatively long timeslice lengths. Finally, performing out of context prefetching on I/O-bound processes exhibits significant performance improvements over traditional cache prefetching.

ADVISER'S APPROVAL: Dr. Sohum Sohoni