ABSTRACT

CUI, HANYU. Extending Data Prefetching to Cope with Context Switch Misses. (Under the direction of Dr. Suleyman Sair).

Among the various costs of a context switch, its impact on the performance of L2 caches is the most significant because of the resulting high miss penalty. To mitigate the impact of context switches, several OS approaches have been proposed to reduce the number of context switches. Nevertheless, frequent context switches are inevitable in certain cases and result in severe L2 cache performance degradation. Moreover, traditional prefetching techniques are ineffective in the face of context switches as their prediction tables are also subject to loss of content during a context switch.

To reduce the impact of frequent context switches, we propose restoring a program's locality by prefetching into the L2 cache the data a program was using before it was swapped out. A Global History List is used to record a process' L2 read accesses in LRU order. These accesses are saved along with the process' context when the process is swapped out and loaded to guide prefetching when it is swapped in. We also propose a feedback mechanism that greatly reduces memory traffic incurred by our prefetching scheme. A phase guided prefetching scheme was also proposed to complement GHL prefetching. Experiments show significant speedup over baseline architectures with and without traditional prefetching in the presence of frequent context switches.

Extending Data Prefetching to Cope with Context Switch Misses

by
Hanyu Cui

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fullfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2009

APPROVED BY:

Dr. Edward Gehringer	Dr. Eric Rotenberg
Dr. Suleyman Sair Chair of Advisory Committee	Dr. Yan Solihin

DEDICATION

To my parents.

BIOGRAPHY

Hanyu Cui was born in Guangzhou, P.R. China. He has been very interested in natural science since childhood. He received his bachelor degree from Sun Yat-Sen University (formerly Zhongshan University) in 2003. He joined the Ph.D. program in computer engineering at North Carolina State University in 2004. He has been working on program phase analysis and prefetching under the direction of Dr. Suleyman Sair.

ACKNOWLEDGMENTS

Firstly, I would like to thank my parents, who love me, teach me and always believe in me. No matter what happens, they are always encouraging and supporting me.

I would also like to express my gratitude to my advisor, Dr. Suleyman Sair. I could not have completed my dissertation without his guidance. And he always encouraged me when I had difficulties. Besides research, he also shared with me his experience on pursuing a career and adapting to the U.S. society as an international student.

My gratitude also goes to my advisory committee, for their invaluable feedback and suggestions on my dissertation and research. And I am very grateful for their help on my job searching, without which I could not have received my current offer under in such a economy.

I want to take this opportunity to say "thank you" to my girlfriend. She has been always encouraging and supporting me. And she never hesitated to spend the time to help me out when I was on tight deadlines with my research, dissertation and interviews. We have been in perfect harmony ever since we met. I am so lucky to have her with me along the way.

I am also very thankful to the people in Center for Efficient Scalable and Reliable Computer (CESR), for the inspiring conversations we had and the help they offered countless times.

I also want to thank all the people who have offered help to me in my study and life at North Carolina State University.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Most time-shared operating systems use context switching since it leads to faster response times, higher throughput and fairness as well as higher utilization of system resources. However, context switches hurt cache performance as cache state belonging to different programs compete for cache space and replace each other's lines. Since most of these interfering programs do not share any data or instructions, the cache content built up by the swapped out process is of no use to the new process. The new one would have to bring all its data and instructions into the cache all the way from memory and results in many cold misses. We call these misses *context switch misses*. Given the speed gap between memory and on chip caches, this creates a significant bottleneck.

Because context switches are so expensive, various techniques have been proposed to reduce their impact. Most operating systems (OS) are designed to minimize the number of context switches whenever possible. For instance, the minimum quantum for a process is 5 ms in Linux Kernel 2.6 [4], which makes context switches rare enough to have small overhead in many cases. Alternatively, affinity scheduling [34, 55] also helps reduce the number of context switches by giving higher priority to the currently running process. While this gives the current process a better chance of retaining the processor, it hurts the responsiveness and fairness [1] of the system. Cache partitioning is another solution to prevent processes from clobbering each other's state [53]. However, this reduces the usable cache space for each process. Furthermore, if the number of processes exceed the number of available partitions, even the partitioned cache will suffer from context switches.

Our approach is to use prefetching to eliminate context switch misses. There are various prefetching schemes in the literature. An early work would be Smith's Nextline Prefetching (NLP) [51], where each cache block was tagged with a bit indicating if the

next sequential block should be prefetched. A stride prefetcher [7] captures access patterns that have a fixed stride. Markov prefetchers [23] try to discover global correlation between addresses. Solihin et al. [52] proposed using a User-Level Memory Thread (ULMT) for correlation prefetching, which takes advantage of cheap memory and software's flexibility. However, the prefetchers mentioned above do not specifically target context switch misses. Nesbit et al. [39, 40] use the Global History Buffer (GHB) for holding the most recent miss addresses in FIFO order for subsequent prefetching. Their approach is close to our scheme, but with different goals. They are also different in several aspects, which are discussed in section 2.

In this work, we propose eliminating context switch misses via prefetching to mitigate the impact of context switches. Prefetching is advantageous over previously mentioned schemes: (1) Prefetching frees the OS to focus on other scheduling criteria (responsiveness, fairness etc.) by removing restrictions such as minimum time quantum limits or increased the priority of the current thread; (2) In cases where frequent context switches are inevitable, e.g. in the presence of network streaming, multimedia processing or inter-process piping, OS based techniques do not perform well; (3) Cache partitioning is fundamentally geared towards a different problem (resolving conflicts among simultaneously running threads in a CMP) and is ineffective when there are more threads than partitions.

To eliminate context switch misses, we use a list to save the addresses of blocks accessed by the program when it is running. At the time it is swapped out, these addresses are saved with the context of the program. The next time the program is swapped in, these addresses are loaded into a buffer to guide prefetching. This effectively restores the cache contents of the program before it was swapped out. Compared with prefetching techniques

which rely on arithmetic patterns in address streams, our technique can capture irregular accesses. And unlike common context-based prefetchers (e.g. Markov [23]) which are limited by predictor table size and are themselves subject to loss of content during context switches, our technique stores the contents of the prefetcher along with the program state so that it can be restored upon being swapped in.

Given the size of caches and program's working set nowadays, our scheme needs to prefetch a large number of memory blocks following a context switch, which could potentially consume too much memory bandwidth and thus offset its benefits. Based on our design mentioned above, we come up with two schemes that aim at improving accuracy of the prefetcher. The first one is a feedback mechanism that tracks which prefetches are used and make predictions for future prefetch uses accordingly. Only the blocks that are predicted to be used will be prefetched. The second one is a phase-guided prefetching scheme, which captures the unique phase behavior certain benchmarks have. These two schemes effectively eliminate a significant number of useless prefetches, improving performance while preserving precious memory bandwidth.

The contributions of this work include: (1) tracking which blocks are accessed before a program is swapped out using an LRU-ordered doubly-linked list; (2) saving the blocks along with a process' state and prefetching them the next time the process is swapped in; (3) presenting a placement policy which is tailored to increase the lifetime of prefetches in the cache; (4) designing a feedback mechanism which significantly reduces memory traffic incurred by prefetching; (5) examining a hybrid scheme that selects between GHL-prefetching and NLP; (6) complementing GHL-prefetching with phase-guided prefetching; (7) evaluating our design in CMP environment; (8) Comparing our design directly to NLP and Stride

prefetcher with or without saving tables across context switches, and saving tags of the L2; and (9) attaining 36% average speedup over no prefetching, and 11% and 24% average speedup in the presence of other prefetchers.

The rest of the dissertation is organized as follows. In Chapter 2, prior work on reducing the impact of context switches and prefetching are discussed. Chapter 3 describes our prefetching technique while our simulation methodology can be found in Chapter 4. Chapter 5 presents an evaluation of the merits of the proposed prefetching scheme. Finally, our conclusions are summarized in Chapter 6.

Chapter 2

Related Work

In this chapter, we present research done in context switches and its impact on system performance, which demonstrates the need to cope with L2 cache context switch misses. We discuss various OS studies and scheduling techniques related to the costs associated with context switches. We also compare our design to various related prefetching and cache management schemes. Research done in phase analysis is discussed and compared to our phase-guided prefetching scheme. Furthermore, we discuss potential bandwidth consumption of our scheme.

2.1 Context Switches

2.1.1 Studies of Context Switches

Context switching is an indispensable component of modern operating system. However, it also comes at a cost. In this section, we are going to present prior research in the impact of context switches.

To estimate the effect of a context switch on cache performance, Mogul et al. [37] obtained address traces from a multi-tasking system, marked them with context switch information and fed them through a cache simulator. They showed that the overhead of a context switch can be up to hundreds of microseconds or thousands of cycles. On modern processors and OSs, this amounts to hundreds of thousands of cycles.

Li et al. [29] performed a detailed quantitative study of the cost of a context switch on an IBM eServer with dual 2.0 GHz Intel Pentium Xeon CPUs. They showed that the total cost of a context switch could be more than a thousand microseconds. It is also shown that the cost of refilling the L2 cache is substantial for a program with a large working set.

Liu et al. [31] studied the impact of context switches on cache misses. They characterized previously-unreported cache misses caused by context switches, which they call reordered misses. They are different from the commonly known context switch misses (they call them replaced misses), which are caused by one program's working set being replaced by another program's between two context switches. Reordered misses are not caused by displacement of cache blocks, rather, the blocks' recency was changed by another program's working set and thus has a higher chance to be replaced by the originally-running program when it is swapped in. In this dissertation, we adopt the commonly known "context switch miss" which is equivalent to their replaced miss. They also proposed an analytical cache model that accurately predicts context switch misses.

Fromm et al. [15] showed that the impact of context switches on L1 caches is insignificant. While they mentioned reducing the impact of context switches via duplicating caches or prefetching, they did not investigate them. In any case, duplicating the caches (or the tag arrays to guide prefetching) only addresses the issue of context switches among two processes. It would not be helpful in a more heavily loaded system.

Tsafrir [56] showed periodic hardware interrupts have a non-negligible overhead. It should also be noticed that interrupt handlers usually do not perform much work, but they may wake up a sleeping process which could preempt the current one. And since hardware interrupts are asynchronous, the operating system or user has no control over the number or frequency of the interrupts. Large numbers of or frequent interrupts could lead to frequent context switches and thus degrade system performance.

Prior research on context switches reveals that they have a significant impact on system performance, especially the lowest level caches, which necessitates further research

on mechanisms to mitigate such impact.

2.1.2 Reducing the Impact of Context Switches

Since context switching has significant impact on lower level cache, various schemes have been proposed to mitigate it.

Suh et al. [53] established an analytical cache model for a time-shared system and proposed cache partitioning to eliminate context switch misses. Compared with partitioning, our scheme utilizes cache more efficiently because at any moment, the running process always has the entire cache to itself instead of a small partition. Furthermore, partitioning is ineffective when there are more processes than the number of partitions or the size of a partition becomes too small to be useful.

A processor with fast context switches enabled [2, 16, 26, 50, 54] can store the contexts of multiple threads/processes in hardware. It mitigates the impact of long latency operations by context-switching to another thread/process that is ready to execute. Since its context switches happen at instruction granularity, multiple threads are actually competing for the shared caches simultaneously. This is very similar to the threads running on a CMP. In contrast, our research focuses on threads/processes time-sharing the processor, which complements their studies.

Affinity scheduling [34, 55] was proposed for shared-memory multiprocessors. The premise is giving a higher priority to processes with affinity. Affinity has two components: temporal affinity and, processor affinity. Temporal affinity refers to a process that just swapped out having a greater chance of being selected the next to run. Processor affinity on the other hand applies to a process having a greater chance to be selected to run on

the same processor. Temporal affinity scheduling can be applied to context switches in a uniprocessor. However, its effectiveness would be reduced in heavily loaded systems because delaying scheduling other processes in favor of the one with temporal affinity could result in significant performance degradation, for example, due to lost packets. In general, our approach provides the OS the scheduling flexibility to address the needs of the current workload.

2.2 Prefetching Schemes

Our approach is to use prefetching to eliminate context switch misses. There have been many related prefetching schemes in the literature.

An early example of a prefetching architecture is Nextline Prefetching (NLP) by Smith [51], where each cache block was tagged with a bit indicating a prefetch should be issued. Using this bit, when a prefetched block is accessed by the program, a prefetch of the next sequential block is triggered. This scheme is simple yet effective, especially for programs with sequential access patterns.

A stride prefetcher [7] keeps track of the difference between the last address of a load and the address before that, which is called the stride. The prefetcher speculates that the new address seen by the load will be the sum of the last address value and the stride. This type of prefetcher is very effective for programs with regular array accesses, e.g. scientific program that perform matrix manipulations.

In Markov prefetchers [23], each missing address would index into a Markov prediction table to provide the set of cache addresses that have followed this address before. Prefetches are issued for these addresses under the heuristic that a miss is likely be followed

by the same set of misses.

Kandiraju et al. [24] proposed distance prefetching by generalizing Markov prefetching. It was originally proposed for TLB prefetching and later on turned out to be also effective for data cache prefetching. It has the same idea as Markov prefetching, which is to capture the global correlation between addresses and use similar tables to store prefetcher states. Different from Markov prefetching, instead of capturing the correlation between actual addresses, it tries to capture the correlation between the distances of successive addresses. The distances, instead of the actual addresses, are stored in the prefetcher tables. If a distance x is predicted, a prefetch is issued with a target computed by adding x to the current address. (or previously issued prefetch if the prefetcher goes further down the correlation chain) Using the distances enables more compact tables and can potentially cover more addresses. However, it could also have more aliasing since the distance space is smaller than the address space by far.

Solihin et al. [52] proposed using a User-Level Memory Thread (ULMT) for correlation prefetching. It needs a general purpose processor at the memory side, with minimal changes to the main processor and other architectural components. Since it is closer to the main memory, it can prefetch farther ahead with shorter latency. Since it is using a software thread, it is more customizable than a pure hardware scheme. They also proposed an improved correlation prefetching scheme that exploits that fact that memory is cheap and the prefetcher is closer to the memory.

Scheduled Region Prefetching [30] (SRP) is an aggressive technique that fetches the data surrounding an L2 cache miss. This surrounding region is typically the same size as a memory page. Subsequently, Wang et al. built on the SRP framework to implement a

cooperative software-hardware prefetcher [30]. In this scheme, the compiler inserts prefetch hints for load instructions and the region prefetcher would issue the prefetches in hardware.

The above prefetching schemes are usually effective for the type of misses they are targeting. However, none of them specifically target context switch misses. On the other hand, context switch misses are special in that they are not inherent to the program, but they are also significantly affected by hardware, how frequently context switches take place, and what programs are run between context switches. These factors make the mentioned schemes unsuitable to cope with context switch misses. Moreover, none of them save any state in the main memory and thus are subject to meta data loss across context switches.

Nesbit et al. [39, 40] proposed the Global History Buffer (GHB) for holding the most recent miss addresses in FIFO order for subsequent prefetching. The GHB has two advantages over traditional predictor tables: it eliminates stale prediction data, and it maintains a complete picture of cache miss history. The GHB is similar to our approach in that both are trying to record the global history. However, they have three fundamental distinctions: (1) The GHB only records misses while our scheme records all read accesses; (2) Our scheme tries to remove duplicates and compact the history while the GHB does not because doing so would break how it operates; (3) We save the access history along with the state of the process while it is being swapped out so that it can later be restored. As a result, our scheme is more suited to warming up cache state after a context switch.

2.3 Shared Cache Management

Our design targets processes time-sharing the processor while various shared cache management schemes target simultaneous sharing. In a CMP environment, both could

happen if the number of processes is more than the number of physical cores, which make the two schemes complementary.

Qureshi and Patt [43] proposed utility-based cache partitioning (UCP), which partitions a shared cache among multiple applications depending on how many misses could potentially be reduced for a given amount of cache resources. It achieves higher throughput than LRU replacement policy because it partitions resources based on whether the resource is going to bring the highest return, not whether the requesting application has the highest demand. They also design an efficient mechanism that monitors all running applications and make partition decisions.

Kim [25] et al. studied the fairness in cache sharing among multiple threads in a CMP environment in detail. Their work complements prior research on cache sharing, which only focuses on characterization and optimization techniques on throughput. They proposed several metrics that measure fairness and cache partitioning algorithms that improves fairness. They also studied the relation between fairness and throughput and indicate that increasing fairness usually result in improved throughput while the opposite is not always true.

Chang and Sohi [6] proposed cooperative cache partitioning (CCP). Instead of partitioning the shared cache into multiple partitions, they also time-shared a single partition. It provides more opportunities for optimization than prior cache management schemes, especially in the case when there are many cache-intensive threads in the processor and any partition alone is far from enough. In this case, CCP would allow a single thread use most of the cache for a period and all other threads have extremely small partitions. After a certain period, another thread will be selected to replace the currently running thread. By

alternating the biggest partition among multiple threads, every thread has the chance to progress as fast as possible instead of being slowed down constantly by a small partition.

Guo [17] et al. proposed a frame work to provide quality of service (QoS) to applications on CMPs based on shared cache management. They proposed a way to specify QoS targets, as well as an admission policy that accepts jobs according to their QoS targets. They found that enforcing a strict QoS target has significant impact on system performance because of resource fragmentation and proposed a mechanism that solves this problem by "stealing" excess resources from some applications.

All of the above schemes only focus on sharing caches among threads that are executing simultaneously but none of them studied threads time-sharing the processor(s). Our research complements them by studying threads/processes time-sharing the processor, and by using prefetching instead of partitioning the shared cache.

2.4 Memory Bandwidth

There has been extensive research on techniques and architectures that enhance performance of the memory bus and DRAM memory in the literature [19, 35, 36, 45, 57]. Since our scheme needs to prefetch a large number of memory blocks into the processor, high speed bus and memory architectures can boost its performance in some cases.

An orthogonal but symbiotic technique to ours was studied by Rixner et al. [46] who looked at increasing memory bandwidth via scheduling memory accesses. A contemporary DRAM chip is actually a "3-D" structure of banks, rows and columns. Their results showed that there is nearly an order of magnitude access latency difference between successive references to different columns within the same row and different rows in the same

bank. They proposed several scheduling policies and attained close to peak bandwidth via aggressive reordering. Since our scheme needs to prefetch many blocks right after a process is swapped in, it would be advantageous to implement our scheme on a system with memory scheduling. By scheduling prefetches issued by GHL according to the memory architecture, higher bandwidth and smaller delay can be expected. Our design simply give higher priority to demand misses but does not perform any memory scheduling. This could be one of our future directions.

2.5 Phase Analysis

In this section, we are going to discuss prior research in phase analysis, which is related to our phase-guided prefetching scheme in section 3.8.

In order to guide dynamic cache reconfiguration for energy conservation, Balasubramonian et al. [3] proposed collecting hardware counters to observe miss rate, CPI and branch frequency information for every 100K instructions. Their algorithm used these hardware counter values to determine if a reconfiguration needs to be triggered because of a drastic change in program behavior. Similarly, Isci and Martonosi [21, 22] observed that a program's power consumption can exhibit phase behavior. They proposed using power vectors to identify this behavior. Duesterwald et al. also observed periodic phase behavior across multiple metrics measured with hardware performance counters [14]. They introduced cross-metric predictors that use one metric to predict another, thus enabling an efficient coupling of multiple predictors.

Since the code that is being executed is the major factor on program behavior, Dhodapkar and Smith [9, 10, 8] proposed associating phase changes with the changes in the

instruction the working set. They then employ these changes in program working sets for initiating instruction cache, data cache and branch predictor reconfiguration [9, 10].

Sherwood et al. [47, 48] showed that periodic phase behavior in programs can be automatically identified by profiling the code executed. For this purpose, they proposed associating a *Basic Block Vector* (BBV) to each fixed length execution interval. BBVs are one dimensional arrays where each element in the array corresponds to one static basic block in the program. The execution starts with a BBV containing all zeroes at the beginning of each interval. During each interval, for each basic block that is executed, they increment the corresponding vector element by the number of instructions in the basic block. This ensures that blocks containing more instructions will have more weight in the BBV. In a nutshell, a BBV can be considered as the fingerprint for the interval it was collected for. Subsequently, the k-means clustering algorithm [33] is used over all the BBVs to group intervals of execution that were alike into the same phase. The authors observed that phases formed in this manner exhibited similar behavior across a variety of architectural metrics. Based on this analysis, they presented the SimPoint toolset that identifies a small set of representative simulation points for detailed architectural simulation. They also extended this approach to perform hardware phase classification and prediction [49].

Another means of extracting phase behavior was proposed by Huang et al. [20]. They examined tracking procedure calls via a call stack, which can be used to dynamically identify phase changes. More recently they proposed using procedure call boundaries for creating samples to guide statistical simulation [32].

Shen et al. use wavelets and the Sequitur string matching algorithm to build a hierarchy of phase information to guide the prediction of phases in data access behavior.

They analyze the data reuse distance trace. They analyze the data reuse distance trace, which is analogous to our LRU-stack based technique described in Section 4.4.3.2.

Lau et al. analyze the program behavior over a hierarchy of interval lengths [27]. This framework is built on top of the SimPoint toolset and uses the same basic block vectors. Two changes to SimPoint were needed to support their work. The first was the explosion in the number of intervals that needed to be analyzed due to the need to experiment with a hierarchy of interval lengths. They addressed this issue by randomly sampling intervals and performing clustering on those. The second change was supporting non-uniform weights in their k-means clustering algorithm. They also presented initial results using SimPoint and Sequitur with variable length intervals for creating a hierarchy of variable length intervals. Both of these techniques use the Sequitur string matching algorithm to identify hierarchical phase behavior.

We extend Sherwood et al. [49]'s work to capture data access phases by calculating signatures with a new algorithm from data accesses and tuning all parameters accordingly. By utilizing phase analysis and prediction, prefetching is more accurate. Combing phase-guided prefetching with GHL-prefetching yields higher speedup.

Mukundan [38] also studied applying phase analysis to instruction prefetching. In her scheme, checkpoints of the L1 instruction cache (IL1) are made for each phase. When the phase analyzer successfully predicts a phase change, the corresponding checkpoint is loaded into the IL1. However, she did not study data phases, which are much more irregular. Neither did she study prefetching for L2, which is more challenging and realistic given the increasing latency to access off-chip memory.

Chapter 3

GHL Prefetching

In this chapter, we present the details of our *GHL-prefetching* architecture and the motivating forces behind its design.

3.1 Case Study

As mentioned in Section 1, frequent context switches are inevitable in certain cases. We used SystemTap [44] to collect the context switch trace from a real machine. The machine is a single core Pentium 4 3.0 GHz machine running Linux Kernel 2.6.17. Part of the collected trace is shown in Figure 3.1. Each entry of the trace has three fields: a sequence number, process ID with the name of the executable, and the duration (in μs) a process runs before being swapped out.

The case shown here is typical. The user is running two tasks. One application is `scp` copying a large file to another computer. The other is `cc1`, which is invoked by `gcc` to compile some code. We can see that it is actually `sshd` that does the actual work while copying files over an ssh tunnel. The trace shows that sshd and cc1 interrupt each other frequently. Most of the time, they can run no more than 600 μs before a context switch occurs. The average runtime without context switches for `cc1` is approximately 300 μs and 100 μs for `sshd`, which translate into 900K and 300K cycles for a 3GHz machine respectively. Our experiments show that such frequent context switches have a significant impact on the performance of the L2 cache and the entire system.

3.2 Architecture Overview

Given the pressing need to mitigate the impact of context switches, we propose restoring the L2 cache content via prefetching. Our prefetching infrastructure centers around a *Global History List* (GHL), along with a *prefetch buffer* and certain block-specific

```
01  3071(scp):        6(us)   |   17  3096(cc1):     2192(us)
02  2440(upsd):      17(us)   |   18  2440(upsd):      19(us)
03  3096(cc1):      509(us)   |   19  3096(cc1):      133(us)
04  3069(sshd):      23(us)   |   20  3069(sshd):      23(us)
05  3096(cc1):      103(us)   |   21  3096(cc1):       73(us)
06  3069(sshd):      18(us)   |   22  3069(sshd):      17(us)
07  3096(cc1):       98(us)   |   23  3096(cc1):      128(us)
08  3069(sshd):      18(us)   |   24  3069(sshd):      16(us)
09  3096(cc1):      112(us)   |   25  3096(cc1):      102(us)
10  3069(sshd):      17(us)   |   26  3069(sshd):      16(us)
11  3096(cc1):      100(us)   |   27  3096(cc1):      113(us)
12  3069(sshd):      18(us)   |   28  3069(sshd):      17(us)
13  3096(cc1):      138(us)   |   29  3096(cc1):      123(us)
14  3069(sshd):     550(us)   |   30  3069(sshd):     559(us)
15  3071(scp):       75(us)   |   31  3071(scp):       77(us)
16  3069(sshd):       6(us)   |
```

Figure 3.1: Context switch trace collected with SystemTap.

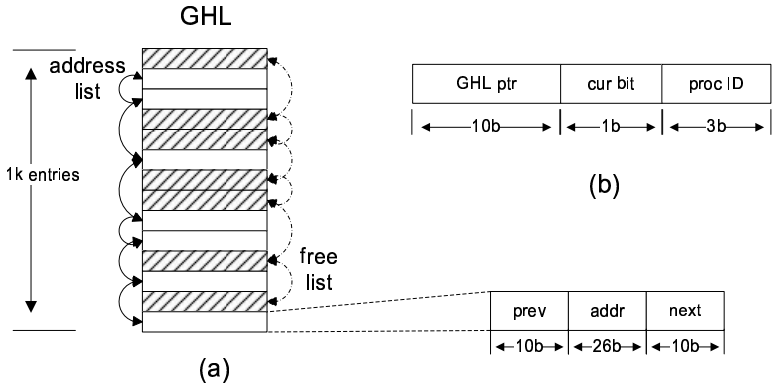


Figure 3.2: (a) The Global History List, shown here with 1K entries. Blank entries are containing valid block address and linked together to form the address list. Shaded entries are unused entries and form the free list. As shown in the bottom, each GHL entry has three fields. Prev and next point to the previous and next entry in its own list respectively. (b) Additional information kept in each cache line. "GHL ptr" is a 10-bit pointer pointing to the corresponding GHL entry. "cur bit" indicates whether the cache is brought in by the current process. "proc ID" indicates by which process the block is brought in.

information kept in each cache line. The prefetch buffer is just a FIFO that stores the addresses to be prefetched and the rest of the components will be explained in detail.

The GHL is a buffer organized into two doubly-linked lists: the *Address List* and the *Free List*, which are shown in Figure 3.2 (a). Each GHL entry has three fields: an L2 cache physical block address and two pointers. The pointers point to the previous entry and next entry respectively, either in the address list or the free list. The address list records all L2 read accesses, while the free list holds the unused entries. To record an L2 access, one entry is taken from the free list and inserted at the tail of the address list. To reclaim an entry, we remove it from the address list and insert it back into the free list. The pointers in the affected entries are changed accordingly in these operations.

The address list is maintained in LRU order meaning new block addresses are inserted at the tail. We record the accesses on a per-process basis. When the process is swapped out, the block addresses in the address list are saved in a dedicated region in main memory. The next time this process is swapped in, the block addresses (physical) will be loaded into the GHL and the prefetch buffer to guide prefetching. Prefetches are issued starting from the MRU entry to bring in the most recently used data items into the L2 cache first.

A pointer, as shown in the "GHL ptr" field in Figure 3.2 (b), is kept in each cache line to help reduce the number of duplicates in the address list. The pointer points to the GHL entry that corresponds to the address of the block contained in the cache line. As shown in Figure 3.3, if a new access hits at this line, it must have the same block address as the entry identified by the pointer. Thus, this duplicate entry can be reclaimed. The doubly-linked structure of the GHL facilitates easy removal of duplicate entries.

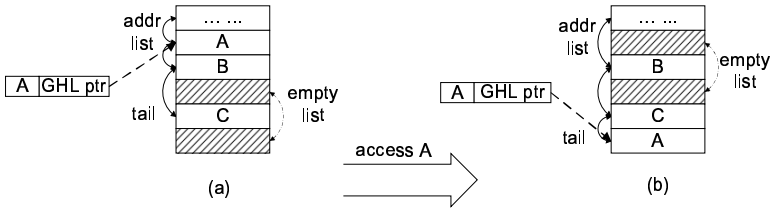


Figure 3.3: Removing duplicates in the GHL. (a) The three entries closest to the tail of the address list contain C, B and A respectively. The GHL pointer in the cache line that contains block A is pointing (the dashed line) to the corresponding entry in the address list. (b) After an access to block A, the old entry that contains A is reclaimed (becomes shaded) and a new entry is allocated for A at the tail of the address list. The GHL pointer then points to the new entry.

To reduce the on-chip area overhead, we adopt a two-level GHL scheme with a small on-chip component and a larger off-chip part. The GHL described above will be the *on-chip GHL*, which caches the most recently accessed addresses. The off-chip part, referred to as the *off-chip GHL*, is a circular buffer in a dedicated region of the main memory. When the on-chip GHL is full, the oldest entry will be moved to the off-chip one and the GHL pointer in the corresponding L2 cache line will be invalidated. If the off-chip GHL is full, the oldest entry is overwritten. Compared to the all on-chip approach, the two-level design reduces the effective overall GHL size since duplicates stored in the off-chip GHL can not be removed. However, our experiments show that this has a negligible impact on performance.

Our experiments show that a 16K-entry GHL where 1K entries are on chip and 15K are off chip is a good trade-off between performance and area. For a block size of 64 bytes and a 32-bit memory address space, 26 bits are required to store a block address. With a capacity of 1K on-chip entries, each pointer in the GHL takes up 10 bits. All together, the on-chip size is less than 100KB.

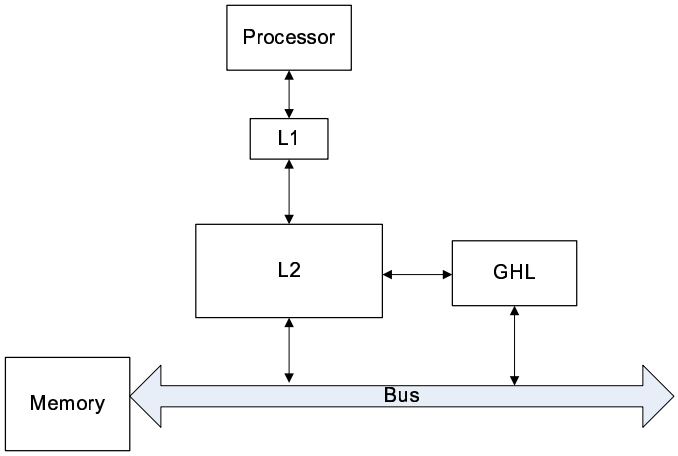


Figure 3.4: Location of GHL. Block 'GHL' represents all components related to GHL except those in the L2 cache.

3.3 Prefetch Placement Policy

Since thousands of blocks are being prefetched, it is very important to not replace useful blocks, which include two categories: (a) blocks brought during the current interval, including demand and prefetched blocks; (b) blocks brought in previously by the current process. The first category is referred to as *current blocks* and the second category is referred to as *survivor blocks*. Identifying survivor blocks from prior intervals requires distinguishing blocks brought in by all interfering processes. Since this requires tagging each cache line with a large process ID field but does not bring enough benefit, we chose not to implement survivor block identification.

To identify current blocks, we keep a *current bit* field in each cache line, as shown in Figure 3.2 (b). This bit is cleared when a new process is swapped in and set when a block is placed in the line by a prefetch or a demand access. When a block is prefetched and there are no empty lines in a set, we first try to replace a line whose current bit is zero. If no such line can be found, we replace the LRU line.

The GHL and related components are not on the critical path. They only communicate with the L2 cache and the main memory bus, as shown in Figure 3.4. Since most operations (in Section 3.4) related to GHL are only for performance and does not affect correctness, they can be delayed if necessary.

3.4 Operations

GHL-prefetching includes maintaining the access history during execution, saving the address list upon a context switch, and after being swapped in, loading the address list and issuing prefetches.

Recording accesses: Whenever there is an L2 read access, an entry is allocated in the on-chip GHL and inserted at the tail of the address list. If this insertion results in duplication, the older entry will be removed as described in Section 3.2. Note that when a cache line is replaced, we do not reclaim the address list entry corresponding to the replaced line. Therefore, it is possible that an address is still in the on-chip GHL but the corresponding block has been replaced in the cache. In this case, the pointer to the on-chip GHL entry is lost and there is no way to reclaim that entry even if it is a duplicate. Our experiments show that this does not result in a noticeable waste of entries. When there is an L2 access but no entries in the free list, we move the oldest entry in the address list to the off-chip GHL. As we described in Section 3.2, the oldest entry in the off-chip GHL will be overwritten when it becomes full.

Saving the address list: When a process is swapped out, besides saving its context, its address list in the on-chip GHL is merged with the off-chip GHL. As each entry is saved, we add it to the free list in preparation for the next process. Note that only the block address field in each address list entry is saved to memory. The linked-list pointers are

installed when the list is repopulated upon swapping the process in.

Loading the address list and issuing prefetches: When a process is swapped in later, the addresses saved in the off-chip GHL will be loaded into the prefetch buffer to guide prefetching. Since the buffer has only 1K entries, prefetching will begin after it has been filled or all saved addresses have been loaded. A single prefetch is issued every cycle from the prefetch buffer. The next 1K will be loaded after prefetches have been issued for the current ones. The first 1K addresses will also be inserted at the head (LRU entry) of the address list of the on-chip GHL, until it becomes full. When a prefetch is issued for any of these 1K entries, the "GHL ptr" in the destination cache line is changed to point to this entry. This enables duplicate removal for the 1K MRU addresses. Unlike a demand access, issuing a prefetch does not move the entry to the tail of the address list. We consider the time it takes to save and repopulate the first 1K entries of the address list to be part of the context switch overhead. The latter parts of the off-chip GHL are brought in as bandwidth permits while the process is running.

The Operating System (OS) needs to have a few modifications in order to support these operations: (1) The OS needs to activate and deactivate GHL for a process. It also needs to tell GHL when a context switch happens and wait for GHL to finish saving and loading addresses. (2) The off-chip GHL resides in a dedicated region in the main memory. The addresses in the on-chip GHL also need to be saved in this region when the process is swapped out. Hence, the OS needs to set aside a region in the kernel space for each process. When the OS activates the GHL or a context switch happens, it also updates a pair of dedicated registers with the beginning and ending addresses of the region for the current process. GHL will only access the region indicated by these two registers. (3) GHL

prefetches physical memory blocks into the L2 cache. However, some types of memory pages are uncacheable and cannot be placed in the L2 cache. To make sure blocks in these pages are not prefetched, GHL records an access only if the accessed block can be placed in the L2 cache. This can be easily known at the time of the access. Another problem caused by uncacheable pages is that this attribute can be changed dynamically and blocks already in the caches and GHLs could become uncacheable. When this happens, the OS needs to flush the caches and the on-chip/off-chip GHLs. Since such changes are rare in real systems except in system initialization stage, they do not cause any noticeable performance degradation to GHL-prefetching.

3.5 Feedback Mechanism

We found that depending on the behavior of the program, the percentage of useful prefetches varies and sometimes is low enough to cause significant waste of memory bandwidth. Our experiments reveal that a certain cluster of prefetches would be used, followed by a series of wasteful prefetches. Furthermore, this pattern exhibits temporal locality across different executions of the same process. Hence, we use a *reuse bit array* to record which prefetched blocks are used by the program. Each position in this reuse bit array corresponds to a GHL entry allocated a certain number accesses ago (e.g. the LRU order). Since the address list is also in LRU order, the reuse bit array can be mapped to the entries in the address list and used to track which addresses in the address list are used by the program. If the access pattern of a program remains fairly stable across context switches, prefetching only the addresses that are used eliminates most of the useless prefetches. Our experiments show that this assumption holds for most of the benchmarks we studied.

Our experiments also reveal that tracking the utilization of each prefetched block

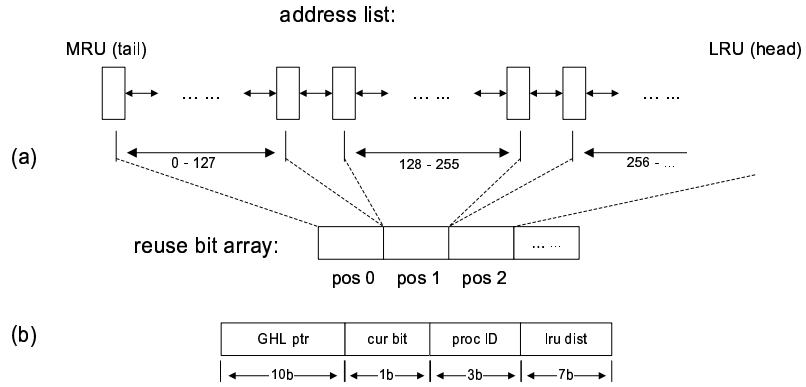


Figure 3.5: (a) Mapping between the reuse bit array and the address list, e.g. position 0 in the reuse bit array corresponds to entry 0 through 127 in the address list. (b) In addition to the fields in Figure 3.2 (b), a "lru dist" field is added to each line, which contains the corresponding position in the reuse bit array.

is too fine-grained. Thus, a single bit is used to identify the reuse of blocks in a region of GHL entries. More precisely, a single bit is mapped to 128 consecutive entries in the address list of the GHL in our design, as shown in Figure 3.5 (a). For a GHL with 16K entries, the reuse bit array needs to have 128 bits. To record which blocks are reused, we store the reuse bit array position in each cache line, as shown in the "lru dist" field in Figure 3.5 (b). A GHL-prefetch can only be issued if the corresponding bit in the reuse bit array is set. When a prefetch issued, the position of the corresponding bit is kept in the "lru dist" field in the destination cache line. When a demand access hits in this cache line, the bit indicated by "lru dist" is set to indicate that this region has been shown to be useful. To eliminate the ambiguity between the lookup and update operations in the reuse bit array, two reuse bit arrays are kept. One records the reuse of prefetches during the current interval. The other one contains the reuse pattern of the previous execution and is used to issue prefetches for the current run. If a block in the cache is not brought in by a GHL-prefetch, its "lru dist" field will be set to zero.

One issue with the feedback mechanism is that once we stop prefetching blocks in a certain address list region, we cannot detect whether they become useful in the future. Thus, we issue a prefetch for the first block in each region regardless of its reuse bit value. As a result, if the prefetched block is reused, the corresponding bit in the reuse bit array would be set and cause the entire region to be prefetched. This would result in a minor waste of memory bandwidth, but ensures that we do not ban a certain range in the address list forever.

3.6 CMP Extension

Our original scheme is designed for a uni-processor. In this section, we discuss how it can be applied on CMPs. GHL-related components can be duplicated with each core. Prefetch requests from multiple GHLs are treated on a First Come First Serve (FCFS) basis. No extra communication hardware is needed for GHLs on different cores.

The only case that requires extra hardware is when several GHLs are prefetching into the same shared cache. Since there are pointers in each block that points to entries in the GHLs, the hardware must distinguish which core's GHL a pointer is pointing to. This requires two-bit core IDs per cache line to track which core the block belongs to. The current bits also need to be reset at this time. In a private cache, this could be done by just sending out signal to the entire cache. In a shared cache, extra circuitry is needed to search for lines with matching core ID. However, this is not too complicated since there are already circuits that search for tags, which are much wider than 2-bit core IDs. Moreover, no data needs to be read out in this operation. All sets could be searched in parallel to reset the pointers. An alternative is to search a small number of sets at a time to reduce power and temperature impact. This operation can be performed in parallel with loading

the GHL and thus does not introduce extra delay. Further design of the circuity is out of the scope of this dissertation and will not be further discussed.

3.7 GHL-NLP Hybrid Scheme

Recognizing that GHL-prefetching and NLP are complementary, we designed a hybrid scheme that tries to take advantage of the strengths of both schemes by adaptively alternating between the two prefetchers. Since GHL-prefetching already has a feedback mechanism in place, we will use this feedback information to choose which scheme we will use after a context switch. When more than 50% of the bits of the reuse bit array are set (defined in Section 3.5), only GHL will be used for prefetching. When fewer than 50% of the bits are set, this is a good indication that GHL is not being effectively utilized. In this case, we activate NLP and not use GHL-prefetching. However, GHL-prefetching will continue to record accesses and one prefetch will still be issued for each region indicated by the reuse bit array, as mentioned in Section 3.5. Experiment results show that by taking advantage of the reuse bit array, the better prefetcher can be selected in most cases.

3.8 Phase-Guided Prefetching

As mentioned previously, GHL-prefetching without a feedback control mechanism can incur significant memory traffic overhead. Since the feedback mechanism is essentially predicting which blocks will be reused at a fine granularity, we extend our work by proposing a phase-guided scheme to more accurately predict reuse patterns by observing large scale access behavior. Once the phase transition pattern is captured, all blocks needed by a particular phase are prefetched in advance. The phase-guided approach complements the LRU or address-based tracking schemes which assume the program's behavior does not

change. Combining the two schemes will lead to higher performance.

The phase-detection techniques we adopt are based on those of Sherwood et al. [49]. A program's execution is divided into intervals, referred to as the *sample intervals*. Two intervals sharing a large number of addresses are considered in the same phase. Signatures are computed from the accessed addresses in each interval. Signatures need to be compact and tell how many addresses two sample intervals share. We found that signatures that can accurately identify different phases require at least 512 bits. Different from Sherwood's scheme [49], it is calculated through a combination of bitwise and modulus operations on load addresses. For each block address X and a prime number Y close to 512, X modulo Y locates a bit in the signature and this bit is set.

The signatures of all phases are saved in the *phase-table* in Figure 3.6 (a) and its position in the table is its *phase ID*. The *Markov table* in Figure 3.6 (b) saves possible phase *transitions*. Each entry has one phase ID and a "Next Phase" field, where the phase ID of possible transitions are saved, together with their confidence counters. The confidence counters are 2-bit saturating counters. Our simulations show saving 4 transitions are sufficient for each phase.

Figure 3.6 (c) shows how predictions are made. At the beginning of a sample interval (interval y in the figure), phase classification is performed on the previous interval (interval x in the figure). Its signature is compared against the signatures in the phase-table. If there exists a signature whose Manhattan distance [49] from the signature of x is below a certain threshold (8% in our experiments), x is considered belong to the same phase. If not, a new entry will be allocated in the phase-table for the new signature. If there are no available entries, the LRU one will be replaced and all corresponding entries in the Markov

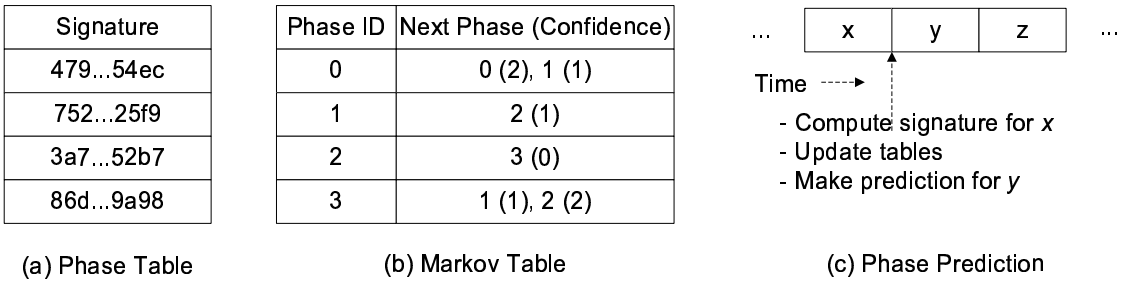


Figure 3.6: (a) Phase table. (b) Markov table. (c) Phase prediction. Prediction is made at the beginning of sample interval y and tables are updated at the end.

table will be invalidated.

Phase classification is a time-consuming operation since a significant number of phase-table entries need to be searched before a phase can be determined. To speed up the operation, the signature of the predicted phase x will be compared first. If it is a hit (Manhattan distance below a certain threshold), no more entries need to be searched. If it is a miss, the remaining entries will be compared against the signature of interval x. However, it does not delay prefetching since prefetches are not issued in this case anyway (explained in the next paragraph). This is also how the previous prediction is verified. The corresponding confidence counter in the Markov table will be incremented if the previous phase was predicted correctly and reset if predicted incorrectly. Transitions with non-zero confidence counter values are deemed *confident transitions*.

After the previous interval has been classified, prediction for the current interval (y in the figure) is made based on the previous phase and the transitions in the Markov table. The transition with the highest confidence will be predicted. Ties are broken by predicting the MRU one. At the end of a sample interval, its signature will be calculated.

Prefetches are issued based on the prediction. Though a prediction is made at the beginning of each interval, not every prediction triggers prefetches. Only when (1) there have been at least three consecutive correct predictions and (2) the current prediction is a confident transition would prefetches be triggered. As a result, searching the phase-table does not delay prefetching because it only takes place when the last prediction is incorrect and no prefetches are issued in this case.

In order to obtain the L2 accesses for a certain phase, we take advantage of the on-chip GHL. Whenever the end of a sample interval has been reached, a special entry is inserted into the GHL to separate accesses between different phases. The accesses of the current phase will be saved to the a dedicated region in main memory if the transition is deemed confident. In our design, we allow at most 100K addresses to be saved in the main memory. Since memory is cheap, this is not too expensive.

GHL-prefetching and phase-guided prefetching work together to form a hybrid scheme. If phase-guided prefetching is triggered, GHL-prefetching will be delayed until it finishes. If a certain number of confident phase transitions have been predicted, no GHL-prefetches will be issued except for one per region indicated by the reuse bit array, which is mentioned in Section 3.5.

Unlike Dhodapkar and Sherwood's [11, 49] techinques, which track instructions, our technique needs to track both instruction and data accesses. For most SPEC'2K benchmarks, data accesses are dominant. The number of unique data accesses is much greater than the number of unique instruction accesses. As a result, sample intervals can not be too long or the number of distinct L2 accesses would be unmanageable. In our experiments, a sample interval length between 50K and 100K instructions yielded adequate results. We

also limit the number of accesses in each interval to be less than 1K. For the same reason, we did not use Run Length Encoding (RLE) in the Markov table since the same phase seldom appears back to back.

Chapter 4

Methodology

4.1 Overview

Our simulator is based on the timing simulator in SimpleScalar 3.0 toolset [5] for the Alpha AXP ISA. Its configuration is shown in Table 4.1. We extended the baseline simulator to model queuing at the various levels of the memory hierarchy. However the detailed structure inside the main memory is not modeled. It is viewed as a whole and accesses to different rows or columns are not distinguished.

We evaluate our scheme on all benchmarks in the SPEC'2K benchmark suite. For each benchmark we use the reference input set and simulate a single simulation point obtained with SimPoint [18]. Each simulation alternates between the simulated benchmark and an interfering benchmark until the main application is executed for 500 million instructions. Unless noted otherwise, the results presented in the next section are the average across all benchmarks and are collected with the parameters shown in Table 4.2.

4.2 CMP Extension

We also improved the approach by Donald [12] to add CMP support to SimpleScalar. Since most of SimpleScalar's variables are static globals, not class members, it is impossible to implement CMP by just instantiating multiple instances, each of which simulates a single core. One alternative is to spawn multiple threads, each of which simulates a core. To make it possible, all static global variables need to be examined to determine whether it should be private to the thread (core) or shared. For private variables, we make it a TLS (Thread Local Storage) variable, which is less error-prone than arrays. For the shared variables, we need to apply an appropriate synchronization mechanism depending

Table 4.1: Architectural configuration.

Parameter	Value
Fetch/Decode/Retire width	4 instructions
RUU size	128
Load/store queue size	64
Function units	4 intALU, 1 int mul/div
Branch Pred.	256-1K 2-level predictor
L1 D-cache/I-cache	16KB, 2-way set assoc., 64 byte lines, 2 Cycle hit latency
L2 cache	2MB, 16-way set assoc., 64 byte lines, 18 Cycle hit latency, 350 Cycle miss latency
CMP Mode	Private or shared L2, other configurations are same as above
L2/MEM bus bandwidth	8 Bytes/cycle (shared among cores)
Max outstanding memory requests	64 (shared among cores)

Table 4.2: Simulation parameters.

Parameter	Value
GHL size	16K entries, 1K on-chip, max 15K off-chip
Reuse bit array	128 entries
Interfering benchmark	art
Context switch period	1M cycles
Interfering benchmark duration	1M instructions
Nextline Prefetcher	Prefetch degree is 4
Stride Prefetcher	PC-indexed, 4k entries, prefetch degree is 4

on how they are used. For instance, a shared L2 cache needs to be protected by a mutex, which guarantees exclusive accesses. To ensure correct timing, there is a barrier at the end of each cycle that synchronizes all threads.

We examine both private and shared L2 cache schemes. The private L2 scheme manifests the bandwidth problem for a CMP machine but does not involve managing the shared caches, which was already studied extensively in literature [6, 25, 53]. The shared L2 scheme is more common in modern CMPs and shows interesting interaction among the cores. With either scheme, we focus on prefetching and no cache management mechanism is applied.

4.3 Context Switch Emulation

We used the SimpleScalar cache simulator with similar configuration to our timing simulator to collect L2 read access traces for several of the SPEC'2K benchmarks. We use these traces to emulate the effects of a context switch on the cache hierarchy. At each context switch, for each address in the trace, we clear the valid bit of the corresponding L2 cache line. Following a context switch, we assume the interfering benchmark runs for 1M instructions (only part of which are loads). After this, the main application resumes execution. Since we do not have a scheduler, we trigger context switches at a fixed period. We also tried random context switch intervals and found that there was not a noticeable difference as long as the mean of the random intervals was the same as the fixed period. This scheme consumes less simulation time than real execution and is accurate enough.

For experiments on the CMP machine, we actually execute the interfering benchmarks instead of flushing the L2 cache with traces. On a uni-processor, these two schemes

give similar results. However, they are different when applied on a CMP. This is because context switches could happen at different times on different cores. As a result, the interfering benchmark on one core could compete for resources with the interfered benchmarks on other cores. Using only traces is not able to simulate the correct timing and resource consumption. In both schemes, the triggering of context switches is the same.

4.4 Selecting Simulation Point

4.4.1 Introduction

Since architectural timing simulation takes a long time, researchers usually choose to simulate a small portion of the entire execution of a benchmark. They typically skip the initialization stage of the program and run timing simulation for a certain number of instructions. However, there is no guarantee that the selected portion would give the same, or even similar results as the full run. We need a more systematic methodology to approach this problem instead of blindly selecting the portions right after the initialization stage.

On the other hand, the large scale behavior of programs has been shown to be cyclic. As loops and function calls dominate program execution, the inherent repetition in these constructs culminate in this phase behavior. Recent research [3, 9, 10, 47, 48, 49, 42, 14, 28], has shown that it is possible to accurately identify and predict these phases in program execution. In addition, they have found that a few phases account for the majority of program execution. The top 20 phases account for 99% of execution for most SPEC programs [49].

This regular and predictable behavior of phases enables accurate architecture tim-

ing simulation [47, 48] without running to the end. One or more portions of the full run, which belong to several major phases, could be selected to simulate. Since they belong to several major phases (phases that account for most of the execution), they are representative of the full run and can yield similar results.

4.4.2 SimPoint

SimPoint [18, 48] is a tool that can automatically classify analyze a program's profile off-line and choose one or a few simulation points that can be representative of of the full run.

The entire execution of a program is divided into fixed length intervals. The length of the intervals could be from several million to hundreds of millions of dynamic instructions. It is a fixed-length approach because the length will be the same for all intervals once it has been chosen. SimPoint will try to classify these intervals into phases, and select on interval from each phase, which is the most representative of that phase.

SimPoint takes a file containing *Basic Block Vectors* (BBVs) as input, one BBV for each interval. A BBV is a vector whose elements are counters. The file is obtained by running a function simulator. PCs corresponding to all static basic blocks will collected. Since there are too many distinct static basic blocks, they will be hashed into very small numbers, usually 10 to 20. Each of this number corresponds to an element in a BBV. Whenever a single PC is encountered, the corresponding BBV element will be incremented. As a result, a BBV tells which portion of the code is being executed in an interval. Since the code being executed is directly related to the program's behavior, two intervals with similar BBVs should have similar behavior and be classified into the same phase.

After reading the input file, SimPoint performs k-means [33] clustering on the BBVs and similar BBVs will be grouped together and form a phase. Finally, one interval will be selected from each phase as a simulation point. This interval is the one with a BBV that is at the centroid of all BBVs in the phase (consider an n-element BBV is a point in an n dimension space). SimPoint also supports selecting a single simulation point, which greatly facilitates simulation, and keeps the error rate almost the same in most cases.

4.4.3 Variable-Length Phases

Most techniques that identify program phases, including SimPoint, first divide the execution into fixed-size contiguous *intervals*. In the next step, a footprint or signature of each interval is generated. Then they compare the signature of an interval to those of other intervals and group "similar" ones into a phase. Thus, intervals in the same phase can belong to different sections of execution (i.e. they are not necessarily temporally adjacent).

One shortcoming of this approach is its reliance on fixed-sized intervals. If the beginning of an interval does not overlap with the actual phase change, we cannot identify the phase transition point accurately. Even worse, we create a slew of transitionary phases, whose intervals do not exhibit homogeneous behavior. That means optimizations such as resizing the cache for this phase can produce suboptimal results because theoretically this transitionary phase includes intervals of two different natural program phases.

Recognizing this deficiency, Lau et al. have recently proposed using variable length intervals and described the changes to the SimPoint [48] phase classification framework needed to support this [27]. The goal of this paper is also creating an automated analysis technique for capturing natural program phases by recursively converging at a hierarchical

phase behavior. We first start by profiling program basic blocks in terms of the instruction mix (e.g. number of integer ALU ops, number of branches etc.). Based on this profile, we group basic blocks that are similar in their composition into a single basic block type. We believe grouping in terms of instruction types is a more accurate approach in reducing the dimensionality of the basic block search space when compared to random projection as is customary in prior work. In addition, instead of using basic block vectors and string matching (described in more detail in Section 2), our approach relies on the reuse distance between program basic blocks to determine natural phase boundaries. And finally, rather than using the k-means clustering algorithm, we employ a hierarchical clustering algorithm that is more adept at finding similarities among intervals.

Variable-length interval generation is challenging. Relatively long intervals are required to make them representative of whole program execution and manageable for the clustering algorithms. However, even if the same section of code is executing, there are minor differences in the instruction stream due to control flow changes. And at the intersection of two repeated sections, there is usually "noise" which exhibits rather random behavior.

Lau et al. use string matching to find patterns [27]. But, in order to find intervals that are long enough, they have to relax the strict matching rules of the Sequitur algorithm [41]. We also found in our experiments that exact string matching is too strict for program phase analysis. Instead, we propose an LRU stack based multiple-pass algorithm to generate variable-length intervals that align with the program's natural phase boundaries. Then, we perform hierarchical clustering on the resulting intervals to find phases. In the remainder of this section, we describe each step in our phase analysis framework in detail.

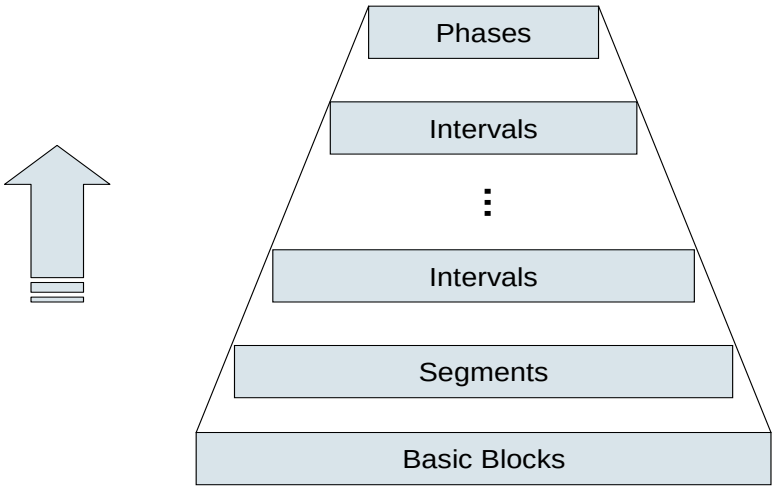


Figure 4.1: The big picture.

Figure 4.1 shows all the steps we take to generate phases. We form segments from basic block. From segments, we form multiple levels of intervals. An interval in the higher level consists of one or more from the lower level. Finally, we perform hierarchical clustering on the highes level intervals to genearte phases.

4.4.3.1 Classifying Basic Blocks

Like prior research on phase analysis [10, 48], we believe that the sections of code that are executing are responsible for program behavior. Therefore, we also use basic blocks as the atomic unit for classification in our approach. Previous research shows that by tracking the executed basic blocks during a program's execution, one can accurately identify program phases [47]. However, some programs have tens of thousands of basic blocks. If we consider each basic block a dimension, that would result in an unmanageably high dimensional space for comparison purposes. Since the number of basic blocks is directly

proportional to the complexity of a phase classification mechanism, it is critical to reduce the number of dimensions. To efficiently track basic block execution and extract useful information from it, we need to reduce the dimensions to a manageable size (e.g. less than 16). Sherwood et al. perform random projection on the beginning PCs of basic blocks to reduce the dimensionality of the basic block space [47]. This approach has some shortcomings however. First, the PC of a basic block does not reveal any information other than its location in the memory. Secondly, random projection maps PCs randomly to a much smaller range without taking advantage of any information about the basic blocks. As a result, two basic blocks with completely different behavior could be mapped into the same dimension.

We address these issues by performing k-means clustering [33] on the basic blocks to classify them into the desired number of classes (16 in our case). We categorize the instruction stream into several classes (e.g. floating point, integer, load, branch, etc.). Then, we count the number of each different instruction class in every single basic block and form this into a vector where each dimension is an instruction class. Finally, we perform k-means on the vectors to combine basic blocks with a similar composition into the same basic block type.

4.4.3.2 Tracking Basic Block Reuse with an LRU Stack

Phases are created by repeating program segments. Therefore, an LRU stack, which is known to be adept at capturing recurring patterns in a stream, can be used to extract natural phase boundaries. An LRU stack is a special stack where access is not restricted to the top entry. Suppose the stream to be scanned consists of a number of

elements. When scanning the stream, we search every stack entry to find out whether the current element is in the stack. If not, it is a miss and the element is pushed onto the top of the stack. If the current element is found at the top, the stack hit distance is 1; if it is the second one from the top, the stack hit depth would be two, and so on. On a stack hit, the current element is moved to the top of the stack. As a result, a smaller stack hit depth indicates that an element reappears soon after its previous occurrence. An example is shown in Figure 4.2. The sequence is shown on the top and how the LRU stack generates hit depth is shown below it. We can see that, after the "warm up" stage, the hit depth keeps to be 3, which means we have a loop consisting of three basic blocks.

To capture repeated execution of basic blocks, we use an LRU stack that is accessed with the type IDs of executed basic blocks. Each time a basic block is executed, we update the LRU stack with its basic block type ID. It's the same thing we did in Figure 4.2 except that the original sequence is replaced by the sequence of type IDs. We follow the two rules below to merge basic blocks into segments:

1. Consecutive basic blocks with the same hit depth will be merged into a single segment.

2. Basic blocks with a hit depth of 1 will be merged with the previous block.

Consider the example shown in Figure 4.3. We are currently executing the basic blocks in the control flow graph shown on the left in Figure 4.3. There are two loops executed back to back. "T" denotes the type ID and "#I" denotes the number of instructions in that basic block. In the upper right portion are the traces. The first row are the basic blocks being executed, and their corresponding type IDs are shown in the second row. The third row are the LRU stack hit depth of the basic blocks. Segments have already been generated

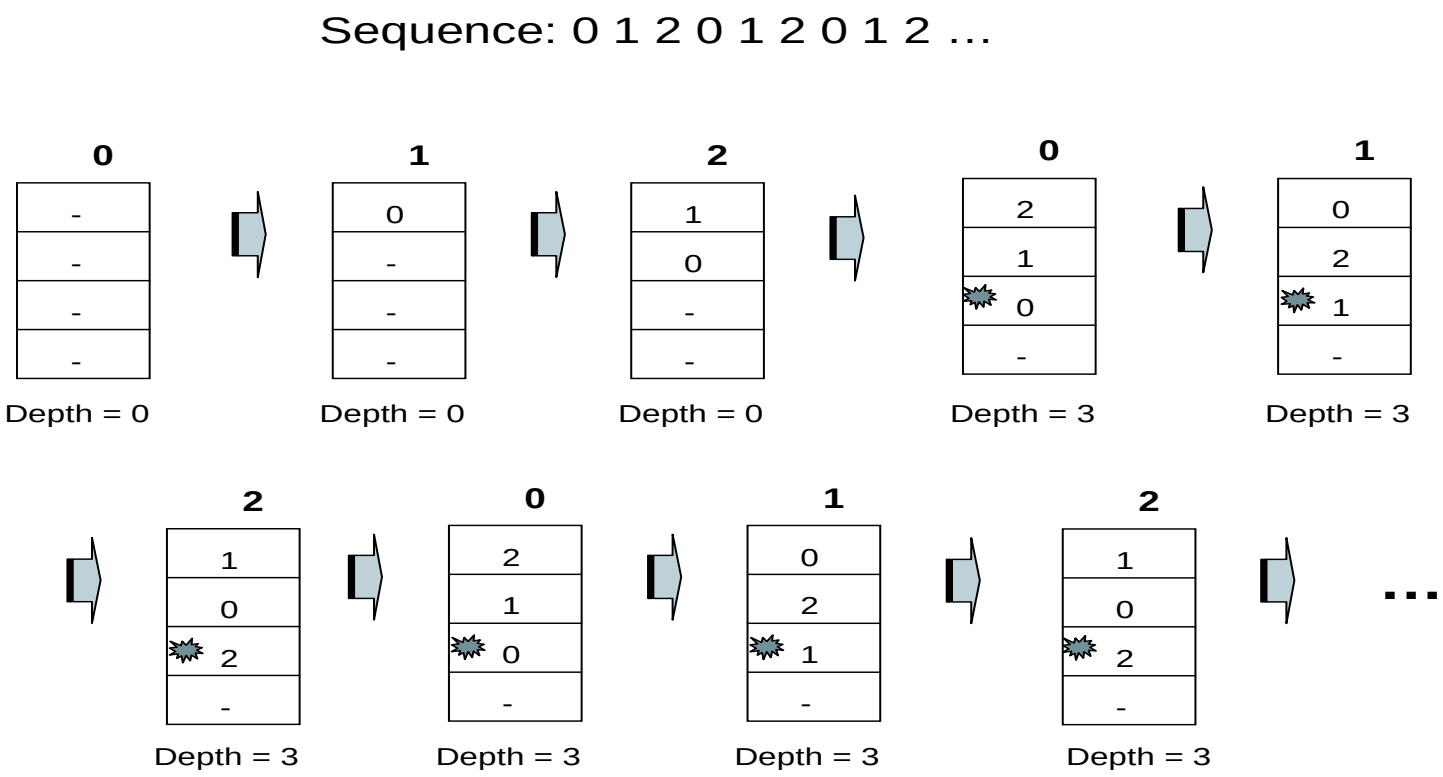


Figure 4.2: Example illustrating how the LRU stack is updated.

following the rules above. Two different segments are separated by a space.

We use segment composition vectors to represent the resulting segments. The number of dimensions in a segment composition vector is the number of basic block types. Each dimension stores how many times the corresponding basic block executed, weighted with the basic block's instruction count. The resulting segment vectors are shown in the lower right portion, which will be explained and used in next section.

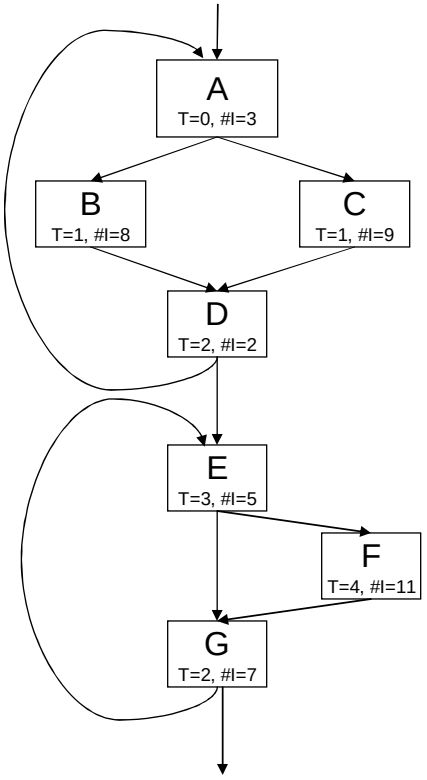
We have two observations from the segments generated:

1. Our technique inevitably involves a "warm-up" problem. At the beginning of a new loop, there are always several misses when the stack is cold and thus the first iteration of a loop will always be separated the rest. This will generate more short segments than necessary. We tried to combine it with the rest but found that would result in a lot of false hit and involved tedious and complex algorithms. According to our experiment results, this "warm-up" problem didn't have a noticable impact on the number of intervals generated.

2. The two loops result in quite different segment patterns. The first loop only results in two segments, the first iteration of which is the result of a "cold" LRU stack. The second loop results in a lot of short segments, which is caused by the alternating stack hit depth. However, by applying LRU stack recursively, which is shown later, most of these short segments will be merged together due to their repeated pattern.

4.4.3.3 Interval Generation

Even though the LRU stack of basic block type IDs enables us to capture repeating behavior, the resulting segments are relatively short. Thus, before starting the phase classification process, we would like to combine adjacent segments that are similar in nature.




```
ABD ACDABDACDABDACD
012 012012012012012
000 333333333333333

EG E F GE GE FGE GE
32 3 4 23 23 423 23
00 2 0 33 22 333 22

FGE GE FGE GE FGE GE
423 23 423 23 423 23
333 22 333 22 333 22

  ...

Resulting Composition Vectors:
(3, 8, 2, 0, 0 )    (0, 0, 7, 5, 11)
(15,44,10,0, 0 )    (0, 0, 7, 5, 0 )
                    (0, 0, 7, 5, 11)
(0, 0, 7, 5, 0 )    (0, 0, 7, 5, 0 )
(0, 0, 0, 5, 0 )    (0, 0, 7, 5, 11)
(0, 0, 0, 0, 11)    (0, 0, 7, 5, 0 )
(0, 0, 7, 5, 0 )    ...
(0, 0, 7, 5, 11)
(0, 0, 7, 5, 0 )
```

Figure 4.3: Example illustrating how the LRU stack is used to form segments. Basic blocks in the control flow graph shown on the left are being executed. There are two loops executed back to back. "T" denotes the type ID and "#I" denotes the number of instructions in that basic block. In the upper right portion are the traces. The first row are the basic blocks being executed, and their corresponding type IDs are shown in the second row. The third row are the LRU stack hit depth of the basic blocks. Segments have already been generated following the rules above. Two different segments are separated by a space.

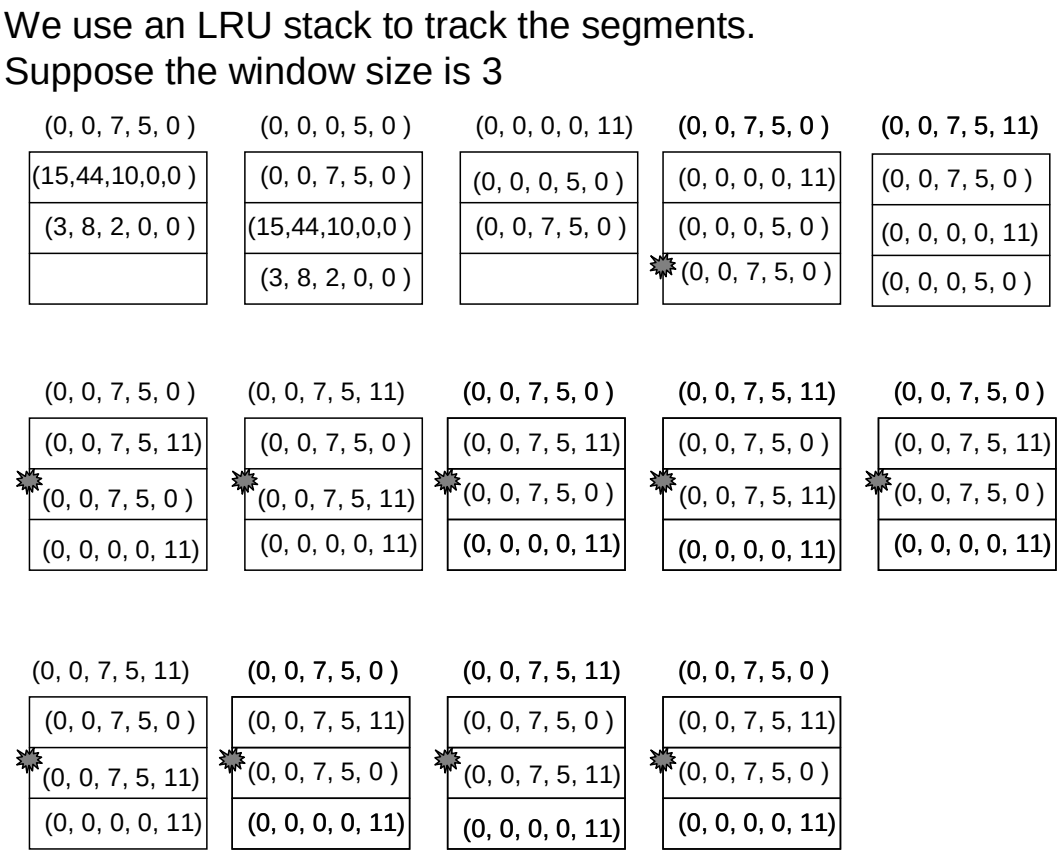


Figure 4.4: Merging segments into intervals using an LRU stack.

Since the patterns we are looking for are inherently similar to those of basic blocks but at a larger granularity, we create a new LRU stack and update it with segment composition vectors to facilitate combining segments. We enforce a maximum size for the LRU stack and the oldest entry will be discarded when the stack size exceeds this limit. Hence, vectors that recur with a stack hit depth larger than the maximum stack depth (referred to as window size) will be treated as if this is the first time they are observed. If a composition vector hits in the stack, this implies it is one of the small group of segments that is currently being executed. If a composition vector misses in the LRU stack, this is an indication of a possible phase boundary and we mark it as the beginning of a new *interval*. An interval is a set of several consecutive segments. When formed this way, the resulting intervals are unlikely to cross any natural phase boundaries. Figure 4.4 shows how the LRU stack is updated and segments are merged. The first stack shows the state when the first two segments have been pushed onto the stack and the third one is being accessed.

Our experiments show that a single pass is far from optimal in generating intervals of desired length (over 1 million instructions). Hence, we form interval composition vectors in the same way that we form segment composition vectors, and feed these vectors into another LRU stack to look for high-level patterns recursively. This is an iterative process and at each iteration, we use a decreasing stack size (referred to as the window size). By doing this, we are able to find larger-scale patterns hierarchically. The process of combining segments into intervals continues until most of the intervals have reached the desired length. Intervals shorter than a minimum length will not be fed into the clustering algorithm introduced in the next section. To capture even higher-level patterns, we can increase the number of passes. We found that 40 passes is enough for most of the benchmarks we

studied in order to generate intervals longer than 1 million instructions. When combined, these intervals account for over 99% of the entire program execution. `gcc` is the only exception, where we needed 60 iterations and the resulting intervals longer than 0.1 million instructions only accounts for 96.4% of its execution. Our experiments show that ignoring such a small part of the program does not introduce any appreciable error.

Note that while we were seeking exact basic block type ID matches in the first level of our pattern searching algorithm, this requirement needs to be relaxed when we are searching for composition vector matches. To allow for some tolerance in our LRU stack based algorithm, we normalize the composition vectors with the sum of all dimensions before comparing them. In this way, if two different segments is just the same section of code that have different trip counts, they can be considered as part of the same segment. When comparing two normalized compositions, they are considered to be "similar" if their Manhattan distance [48] is less than 5% of the maximum possible value. The Manhattan distance between two composition vectors is the sum of the differences of their corresponding components. In Figure 4.5, the upper portion shows how composition vectors before and after normalization and the lower portion shows we calcuate their Manhattan distance. Notice that the first and second segments, which wouldn't have been merged using exact match, will be merged into a single interval since their Manhattan distance is 0.04 (within 2%).

4.4.3.4 Hierarchical Clustering

While intervals represent contiguous sections of execution that are separated at natural phase boundaries, the same (or similar) intervals can be occurring in different places

```
1. ABD                  : (3, 8, 2, 0, 0 ) -> (0.23,0.62,0.15,0,   0   )
2. ACDABDACDABDACD      : (15,44,10,0, 0 ) -> (0.22,0.64,0.14,0,   0   )
3. EG                   : (0, 0, 7, 5, 0 ) -> (0,   0,   0.58,0.42,0   )
4. E                    : (0, 0, 0, 5, 0 ) -> (0,   0,   0,   1,   0   )
5. F                    : (0, 0, 0, 0, 11) -> (0,   0,   0,   0,   1   )
6. GE                   : (0, 0, 7, 5, 0 ) -> (0,   0,   0.58,0.42,0   )
7. FGE                  : (0, 0, 56,50,44) -> (0,   0,   0.37,0.33,0.29)

  For example, we list some of the Manhattan distances between two
  segments/intervals:
  1. And 2.: diff vec: (0.01,0.02,0.01,0,   0  ), Manhattan Dist=0.04;
  1. And 3.: diff vec: (0.23,0.62,0.43,0.42,0  ), Manhattan Dist=1.80;
  1. And 4.: diff vec: (0.23,0.62,0.15,1   ,0  ), Manhattan Dist=2;
```

Figure 4.5: Calculating Manhattan Distance between segments/intervals.

of the execution. In the final step of our phase analysis framework, we perform clustering on the intervals we generated from LRU stack hit patterns to form phases.

K-means is one of the most popular clustering algorithms since it is fast and easy to implement [33]. However, it has two potential drawbacks. First, it requires the number of clusters as an input parameter. However, in phase analysis, there usually is no easy way to determine the necessary number of clusters before clustering. One way to deal with this is to try multiple k values. Nevertheless, there is no guarantee that the range chosen includes the best k and the iteration over different values takes time. Secondly, k-means requires the set of k initial means before clustering. From previous research in machine learning and our experiments, we found that the quality of the resulting clusters heavily depends on the initial means. Yet, it is impractical to try all possible combinations. And this problem is exacerbated by the large number of dimensions we commonly use in phase analysis (e.g. SimPoint uses BBVs with 16 dimensions).

We propose to use hierarchical clustering [13] to address these two issues. In addition, with hierarchical clustering, it is easy to visualize and control the tradeoff between the number of clusters and the variation within a single cluster.

Hierarchical clustering is an iterative algorithm. Each interval is assumed to be its own cluster before clustering starts. In each iteration, we calculate the Manhattan distance between every cluster pair and weight it with the total length of the cluster. When the cluster contains more than one intervals, we use the centroid of the cluster to represent it. Then, we merge the pair of clusters that have the minimum distance. As such, the number of clusters is reduced by one in each iteration. Figure 4.6 depicts the hierarchical clustering process. In this example, we use a two-dimensional space for clarity. Each point represents an interval. Each oval indicates a cluster where a pair of intervals are merged in every iteration. The number on an oval is the iteration number in which the clusters were merged. The interval at the bottom right corner is not merged with any other intervals because it is far away from the others. We repeat this process until the desired number (3 in this example) of clusters is reached.

This algorithm produces optimal clusters in terms of minimum intra-cluster distance. Another advantage is that it maintains small intra-class variation even when more clusters are allowed. Hence, if cluster quality is more important, it is better to use a large number of clusters. If having fewer clusters is a higher priority, the quality of clusters can be sacrificed. In addition to specifying the total number of clusters, one can use other termination criteria. For example, one can stop merging clusters when the standard deviation of a cluster exceeds a certain threshold. Unlike k-means, hierarchical clustering does not depend on any initial parameters, like k or the initial means.

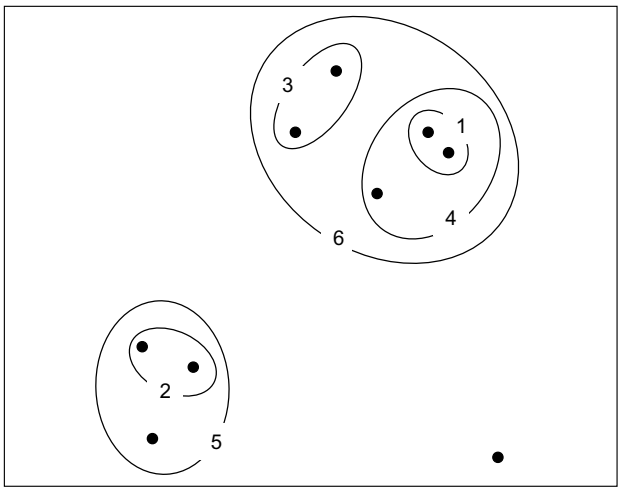


Figure 4.6: Hierarchical clustering example in a two dimensional space. Each point represents an interval. Each oval indicates a cluster where a pair of intervals are merged in every iteration. The number on an oval is the iteration number in which the clusters were merged.

The brute force hierarchical clustering algorithm has a complexity on the order of $O(N^3)$, where N is the number of intervals. In each iteration we search for the pair of clusters that have the minimum distance between them to combine into a single cluster ($O(N^2)$). During the first iteration, for each interval, we can record the interval that is a minimum distance away from this interval. In essence this forms a directed graph. In every subsequent iteration, we just search through the list of intervals to find the one that has the minimum distance from the others ($O(N)$). When we combine those two intervals (say intervals A and B), we need to update the list of intervals that were pointing to A (i.e. intervals from whom A was the minimum distant) and the list of intervals that were pointing to B. To facilitate this, in each node, we keep the list of nodes that are pointing to this interval. Memoizing the closest one of every cluster (e.g. dynamic programming)

reduces the overall complexity to O(N^2). In comparison, each iteration of the k-means algorithm is O(N).

4.4.4 Conclusion

From experiments, our scheme achieves an CoV (Coefficient of Variance) [27] of 0.04 while SimPoint attains 0.05. Since both are pretty low already, it does not make a significant difference. Hence, we chose to use SimPoint because of its ease of use.

Chapter 5

Evaluation

5.1 Evaluation of GHL-Prefetching

5.1.1 Evaluation on Uni-Processors

In this section we evaluate the performance of GHL-prefetching methods outlined in Section 3. We compare GHL-prefetching to a baseline with no prefetching, as well as assuming a Nextline (NLP) and a Stride prefetcher in the memory hierarchy. The best case scenario compared against is the case with no context switches.

Figure 5.1 (a) depicts the case where the baseline architecture does not prefetch. In this scenario, GHL-prefetching results in a 36% average speedup across the entire SPEC'2K suite in the presence of frequent context switches. Most benchmarks benefit from GHL-prefetching and more than half the benchmarks have significant speedup over no prefetching. It is also important that there is no slow down in any benchmarks. For some benchmarks, like `gzip`, our technique restores almost the entire performance loss caused by context switches. Some benchmarks, like `twolf` and `galgel`, suffer seriously from frequent context switches. Our scheme boosts their performance significantly, but not to the level of no context switches. After we further investigated their behavior, we found that these benchmarks have a large working set and require a longer GHL. For example, with a GHL size of 128K, `twolf`'s performance can be restored to within 5% of the performance without context switches. We did not consider such a large GHL in our scheme because the performance it brings does not justify its area overhead. As expected, for benchmarks that do not suffer from context switches, e.g. `equake` and `lucas`, our technique does not help. It is also shown in the figure that our feedback mechanism only causes a 3% IPC reduction, but later results in Figure 5.3 (a) will show that it is worthwhile for its ability to reduce

memory bandwidth usage significantly.

Figure 5.1 (b) and Figure 5.1 (c) show the performance in the presence of a baseline NLP and a Stride Prefetcher respectively. Our scheme still brings 11%-24% respective speedup in these cases. This demonstrates that our scheme is orthogonal to these prefetching schemes and captures access patterns that other techniques might miss. It is interesting to notice that some benchmarks, e.g. `equake` and `swim`, perform better when both context switches and NLP/Stride prefetching are present than when there is no context switching. We notice these benchmarks do not suffer from context switches, but are helped by an NLP or a Stride prefetcher. After further investigation, we discovered that this is a result of the interaction between NLP/Stride prefetching, our placement policy, and context switches. According to our prefetch placement policy, if there are no lines with the valid bit or the current bit cleared, we will replace the LRU line to store a prefetch. When there are no context switches, most lines are valid or current, and the majority of the prefetched lines can only go to the LRU line. If the prefetched block is not used soon, it is replaced by a subsequent demand access or another prefetch. However, when there are context switches, the current bit in conflicting caches line is cleared. Thus, after a context switch, most prefetches survive for a longer period in the cache bringing more benefit. For most benchmarks, the benefit gained in this way is much smaller than the impact incurred by a context switch. Another surprising observation is that NLP works better than Stride prefetcher for most benchmarks at the L2 level. This is because the PC-indexed Stride predictor does not start issuing prefetches until it detects a stride and it cannot capture global address regularities that NLP can exploit.

In Figure 5.2, GHL-prefetches are broken down into used and unused prefetches.

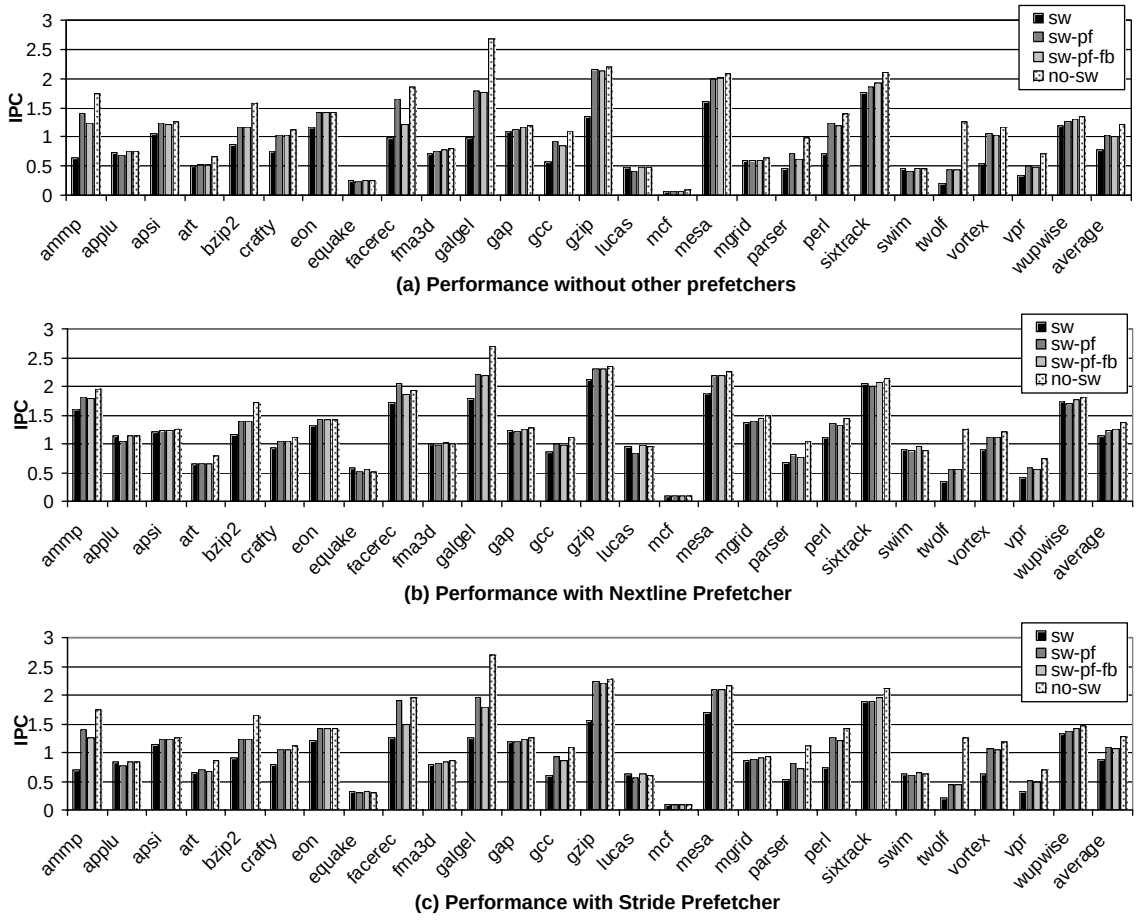


Figure 5.1: Performance with and without GHL-prefetching. Four cases are compared in each graph: context switch is present (*sw*), context switch with GHL-prefetching (*sw-pf*), context switch with GHL-prefetching and feedback (*sw-pf-fb*), no context switch (*no-sw*).

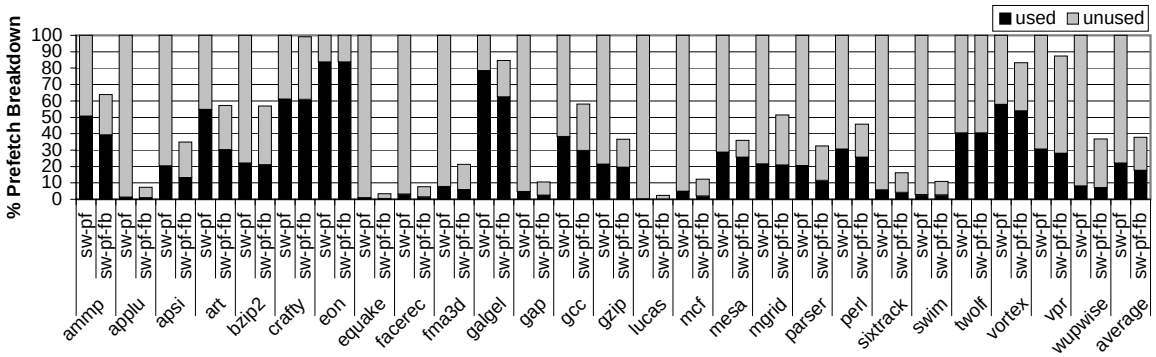


Figure 5.2: Used and unused GHL-prefetches. Two cases (*sw-pf* and *sw-pf-fb*) are presented for each benchmark. The y-axis shows the percentage relative to the number of GHL-prefetches without feedback (*sw-pf*), i.e. prefetching with feedback issues fewer prefetches.

The y-axis in the graph shows the percentage of total prefetches relative to the number of GHL-prefetches without feedback (*sw-pf*). It is shown that more than 70% of the issued GHL-prefetches are not used without the feedback mechanism. And it can also be clearly seen that our feedback mechanism significantly reduces the number of useless prefetches at the cost of a small reduction in the number of useful prefetches. The results also explain why some benchmarks, e.g., `applu`, `equake` and `lucas`, do not benefit from our scheme. The prefetches issued for these benchmarks are almost never used. The useless prefetches for some benchmarks, like `galgel` and `twolf`, are not reduced by the feedback mechanism at all. What is even worse is that it reduces only the useful prefetches for `galgel`. When we examined the reuse patterns of prefetches for these benchmarks, we found that they do not cluster as most other benchmarks do, thus rendering the grouping of prefetches ineffective. Furthermore, they do not exhibit the reuse pattern learned during the previous execution. These could have been captured by more sophisticated schemes, but the added complexity does not justify the meager gains they provide.

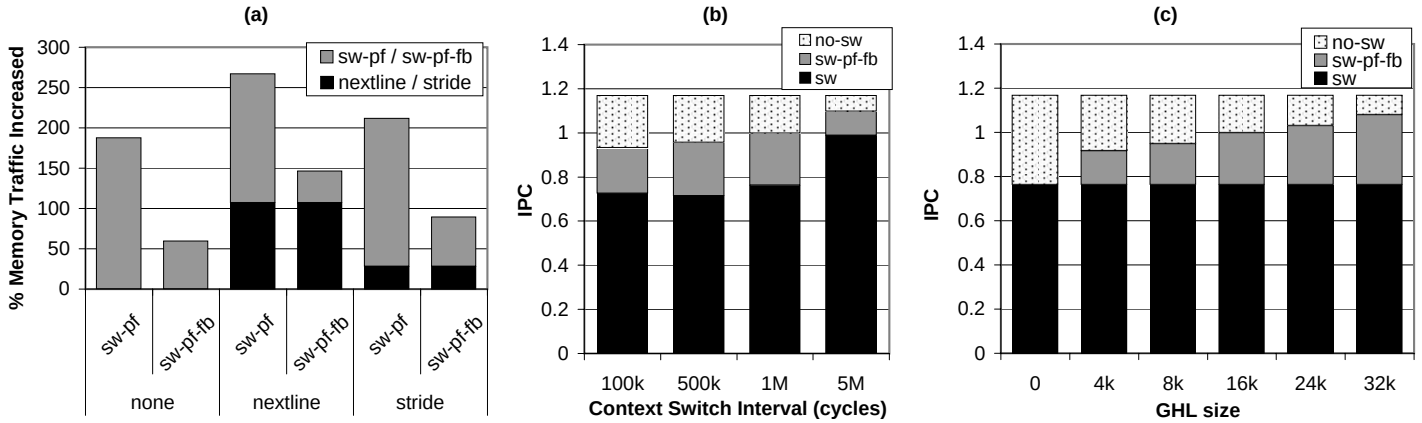


Figure 5.3: (a) Percentage memory traffic increased in the presence of different prefetching schemes. Results are presented for GHL-prefetching only (*none*), with a NLP (*nextline*) and with a Stride prefetcher (*stride*). For each of these cases, results with and without feedback are shown. (b) Performance with different context switch intervals. (b) Performance with different GHL sizes.

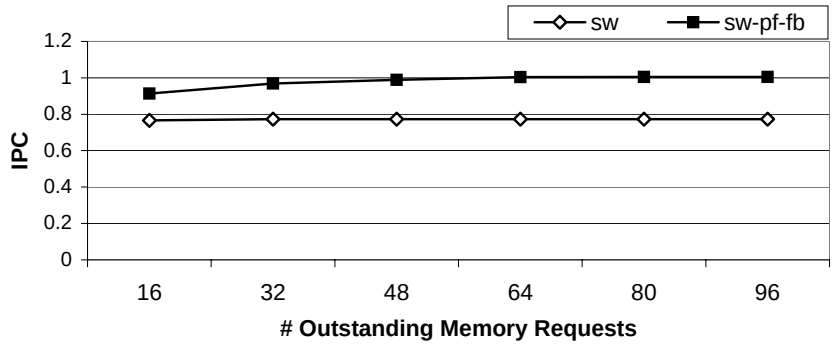


Figure 5.4: Performance when varying the number of outstanding memory requests allowed. *Sw* shows the average IPC without any prefetching schemes while *sw-pf-fb* shows the any IPC of GHL-prefetching.

Figure 5.3 (a) shows the extra memory traffic incurred by different prefetching schemes relative to the no prefetching case where context switches are present (*sw*). The gray segment of each bar is the memory traffic caused by GHL-prefetching. The dark segment of each bar is the traffic resulting from NLP or Stride prefetcher. Although our scheme results in a 36% speedup (Figure 5.1), it also incurs 185% more memory traffic. When we use the proposed feedback mechanism, it brings the bandwidth overhead down to 60% with a slightly lower speedup of 31% (Figure 5.1). While discussing Figure 5.1, we had mentioned that NLP outperforms Stride prefetcher. Figure 5.3 (a) depicts the cost at which this speedup is obtained. The Stride prefetcher generates a quarter of the extra memory traffic when compared to NLP. This figure shows the feedback mechanism effectively reduces the memory bandwidth consumption of GHL-prefetching.

Figure 5.3 (b) shows the impact of context switch intervals. We can see that longer intervals (5M cycles) tend to have a smaller impact on a program's performance. A 1M-cycle context switch interval tends to degrade performance a lot while 100K-cycle and 500K-cycle intervals are similar to the 1M interval. In contrast, the impact of context switch intervals

in the presence of our GHL-prefetching scheme is much less. The figure demonstrates that our technique removes at least half the impact of context switches on average and as shown earlier, we maintain the performance close to when no context switches are present across a wide range of intervals.

Figure 5.3 (c) shows the effect of changing the length of the GHL. In the chart, it is shown that a 4K GHL already does quite well. On average, it results in a 20% speedup. On average, a 16K GHL only brings 11% more speedup when compared to the 4K GHL. We retain a 16K GHL because it does make a clear difference for some of the benchmarks.

Since GHL-prefetching needs to bring all blocks needed by the process right after a context switch, it requires the memory bus to service multiple simultaneous outstanding memory requests. Figure 5.4 shows the impact of limiting the number of allowed outstanding memory requests on GHL-prefetching. It can be seen that 48 or more allowed outstanding requests no longer affects GHL-prefetching's performance. It shows that GHL-prefetching is affected by the number of allowed pending memory request to a limited degree.

Figure 5.5 shows the percentage of context switch misses remaining for different GHL sizes when feedback is on. To collect context switch misses, we simulate two identical L2 caches side by side. One is time-shared with the interfering benchmark while the other is private to the interfered benchmark. All L2 accesses are fed to both caches. If an L2 access hits in the normal cache but misses in the private cache, that miss is considered a context switch miss. We can see that a 4K GHL already eliminates more than 30% of the context switch misses, which agrees with our previous result which shows that a 4K GHL accounts for almost half of the 16K-entry GHL's speedup. On the other hand, the figure also justifies our choice of a 16K GHL because it removes almost another 30% of the context

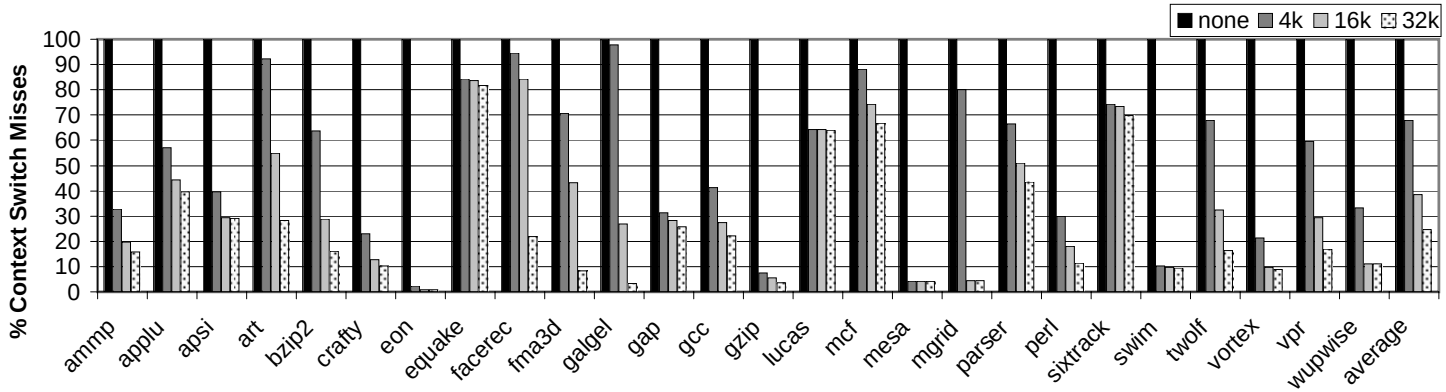


Figure 5.5: Context switch misses remaining with our prefetching scheme using various size GHLs.

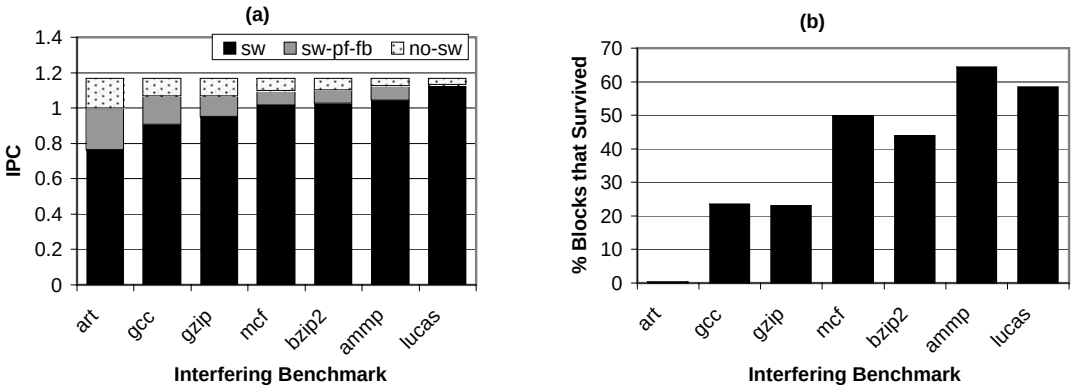


Figure 5.6: (a) Performance with different interfering benchmarks. (b) Percentage of the original application's blocks that survive a context switch under various interfering benchmarks.

misses, leaving only a small portion of the context switch misses.

Figure 5.6 (a) shows the average performance when different benchmarks interfere with the main application. We try a wide range of interfering benchmarks, from the most thrashing `art` to `lucas`, which has the smallest L2 access trace in our experiments. It is shown that the interfering benchmarks make a difference depending on their L2 cache behavior. `Art` brings the performance down by 35% while `lucas` has almost no effect on performance. It also shows that our GHL-prefetching scheme greatly reduces this impact and restores most of the performance. Our experiments also show that, if there is more than one interfering benchmark, the combined impact is as severe or worse than that of `art`. Figure 5.6 (b) shows the percentage of blocks in the cache that survive a context switch when different applications interfere. This figure corroborates the results in Figure 5.6 (a). `Art` has such huge impact because almost no blocks from the original application survive after `art` executes. `Ammp` and `lucas` do not affect performance too much because about 70% of the blocks can survive their interference.

5.1.2 Evaluation on CMPs

After examining GHL-prefetching performance on uni-processors, we also evaluate its performance on CMPs, where several cores are active simultaneously and compete for limited resources.

We first examine how GHL-prefetching on one core (the *GHL core*) interacts with the other cores that do not have GHL-prefetching (*non-GHL cores*). We compare their performance on a single core processor to that on a 4-core CMP in a series of experiments, where each machine has one GHL core and three non-GHL cores.

Two benchmarks, `bzip2` and `perl`, which benefit from GHL-prefetching and have medium bandwidth requirement, were selected to run on the GHL core. **Two groups** of benchmarks, three in each group, were run on the non-GHL cores. The first group includes `apsi`, `crafty` and `eon`, which consume low memory bandwidth. The second group comprises of `applu`, `art` and `swim`, which consume high bandwidth. This setup yields four combinations totally. We use these benchmarks as representatives of the rest of the suite and try to provide some insight into the interaction between GHL-perfetching and other processes in a CMP environment.

Figure 5.7 shows the results for the first set of experiments in this series. There are no context switches on the non-GHL cores. In *single*, all benchmarks were actually run alone on uni-processors, either with or without GHL-prefetching enabled. However, the results are grouped in the way mentioned above. It serves as a reference point for following experiments. In CMP-2Mp, benchmarks are no long separate but run on 4-core CMPs with a 2M private L2 cache per core. The aggregate memory bandwidth stays the same as the uni-processor case in *single*. This configuration examines how GHL-prefetching interacts

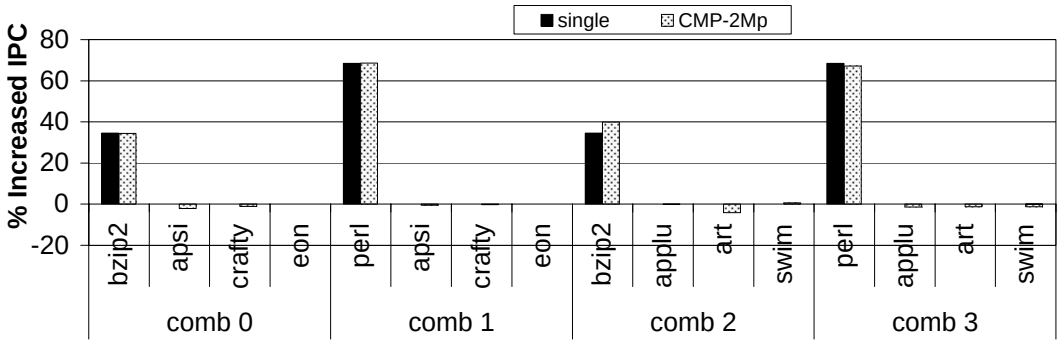


Figure 5.7: GHL-prefetching interacts with non-GHL cores. No context switching on non-GHL cores. Results for 4 combinations *comb 0* - *comb 3* are shown. There are 2 configurations for each benchmark: *single*, where each benchmark is run on a uni-processor; *CMP-2Mp*, where each group are run on a 4-core CMP with 2M private L2 caches.

with other cores that do not have context switches when sharing the memory bus. Results show that there is little impact either to the GHL core or the non-GHL cores. We also did experiments when they share the L2 in addition to the memory bus, and have examined more benchmarks. We found that when there are no context switches on the non-GHL cores, the impact to other cores is usually less than when there are context switches, and even negligible in some cases. Hence, we will focus on the cases where there is also context switching on all cores in later experiments.

Figure 5.8 shows results for the second set of experiments in this series. Configuration *single* is similar to its counterpart in Figure 5.7, where all benchmark were actually run alone on uni-processors and serve as a reference point for other configurations. The only difference is that there is context switching on all cores. And as we mentioned above, there will always be context switching on all cores in later experiments in the series.

CMP-2Mp shows results on 4-core CMPs with a 2M private cache per core. The aggregate memory bandwidth stays the same as the uni-processor case in *single*. This configuration examines how GHL-prefetching interacts with other cores that have context

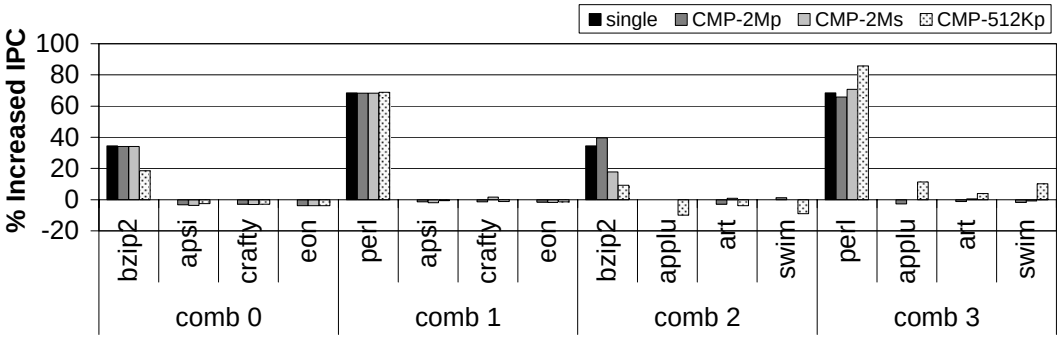


Figure 5.8: GHL-prefetching interacts with non-GHL cores. There is context switching on all cores. Results for 4 combinations *comb 0 - comb 3* are shown. There are 4 configurations for each benchmark: *single*, where each benchmark is run on a uni-processor; *CMP-2Mp*, where each group are run on a 4-core CMP with 2M private caches; *CMP-2Ms*, where each group are run on a 4-core CMP with a 2M shared cache; *CMP-512Kp*, where each group are run on a 4-core CMP with 512K private caches.

switches when sharing the memory bus. Results show that there is little impact either to the GHL core or the non-GHL cores compared to *single*. `Bzip2` even has higher speedup in *comb 2* because the baseline IPC is lower. However, when compared to *CMP-2Mp* in Figure 5.7 , benchmarks on non-GHL cores have more slowdown since context switches causes higher memory bandwidth consumption on the non-GHL cores.

CMP-2Ms is on a 4-core CMP with a 2M shared L2 cache. Compared with the setup in *CMP-2Mp*, the cores need to compete for a single shared cache. And the resulting extra cache misses also add to the burden on the memory bus. This change has significant impact on the performance of GHL, which can be seen from *comb 2*, where `bzip2`'s speedup drops from around 40% (in *CMP-2Mp*) to 18% in *CMP-2Ms* and benchmarks on the non-GHL cores have more slowdown. However, GHL's speedup on `perl` has increased since the baseline IPC (GHL is disabled) is impacted more severely. Benchmarks running on non-GHL cores are almost not affected by GHL. And contrary to one's expectation, GHL

actually helps benchmarks running on non-GHL cores in some cases. For example, `crafty` in *comb 1*, `art` in *comb 2* and *comb 3*, even has non-negative speedup when GHL-prefetching is on. This is because the cache and memory bandwidth contention is so high that they become the bottle neck. By running faster and finishing earlier, the GHL-core frees up the shared cache and memory bus for other cores sooner and improves their performance. This phenomenon is more apparent in *CMP-512Kp*.

In *CMP-512Kp*, the shared 2M L2 cache is statically divided into four 512KB private caches. Results show that performance is worse than the shared cache case because of the unbalanced cache partition. Thus, the system is stressed further but a similar conclusion can be drawn, except where L2-intensive benchmarks suffer lower IPC because of the reduced L2 cache capacity. GHL's positive impact of speedup other cores is more significant in this case due to more limited shared resources, as shown in *comb 3*.

From the experiments, it can be concluded that GHL's impact on non-GHL cores is mixed. There are two major factors. The first one is that GHL consumes more memory bandwidth and a little extra cache capacity, which could slow down other cores. The second factor is that GHL could actually improves the performance on other cores by finishing earlier and frees up the shared resources. The actual impact would depend on which factor dominates for the specific group of benchmark interacting.

We next examine the interaction among cores with GHL-prefetching enabled on all cores. We selected 16 benchmarks from the SPEC'2K suite, all of which benefit from GHL-prefetching. For the rest of the benchmarks, GHL rarely issues prefetches and thus the results would be very similar as those shown in Figure 5.8. As shown in Figure 5.9, they were divided into 4 groups and each group were run on a 4-core CMP. Results are

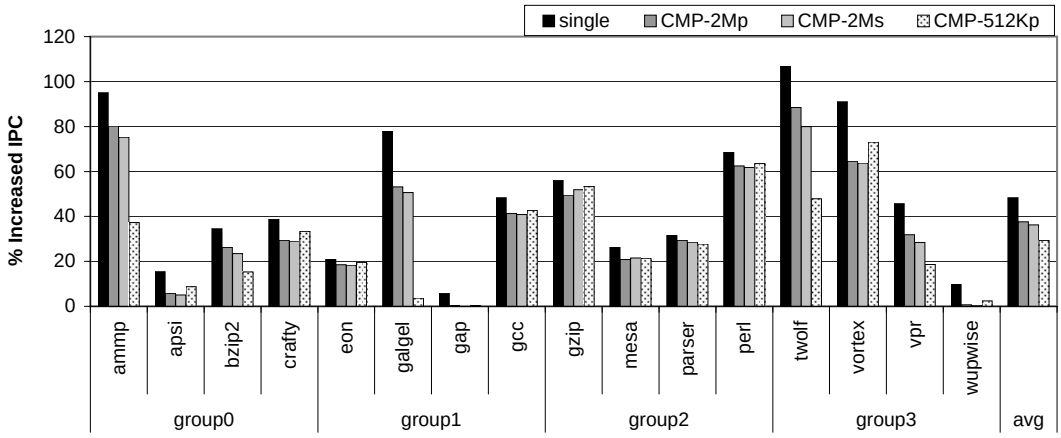


Figure 5.9: Speedup of 16 benchmarks grouped into 4 groups. 4 configurations are shown for each benchmark: *single*, where each benchmark is run on a uni-processor; *CMP-2Mp*, where each group are run on a 4-core CMP with 2M private caches; *CMP-2Ms*, where each group are run on a 4-core CMP with a 2M shared cache; *CMP-512Kp*, where each group are run on a 4-core CMP with 512K private caches.

also compared to those obtained on a uni-processor machine. All of the cores have frequent context switches and GHL-prefetching enabled. Speedup is presented for each benchmark in four different configurations, which are uni-processor (*single*), CMP with 2M private caches per core (*CMP-2Mp*), CMP with a 2M shared cache (*CMP-2Ms*) and CMP with 512KB private caches per core (*CMP-512Kp*). From left to right, the four configurations have more limited resources and lead to lower GHL speedup on average, which is expected. In the worst case, GHL still retains 2/3 of the speedup.

It is quite interesting that not every benchmark follows the same trend with the average. Take `apsi` for instance, where the speedup goes up slightly when the configuration changes from *CMP-2Mp* to *CMP-512Kp*. This is because the baseline's performance is impacted more severely by limiting the shared resources. As a result, its IPC does go down but the speedup rises slightly. This explains the similar phenomenon on `gcc`. However, there are also a number of benchmarks, whose actual IPC (not speedup) goes up. For

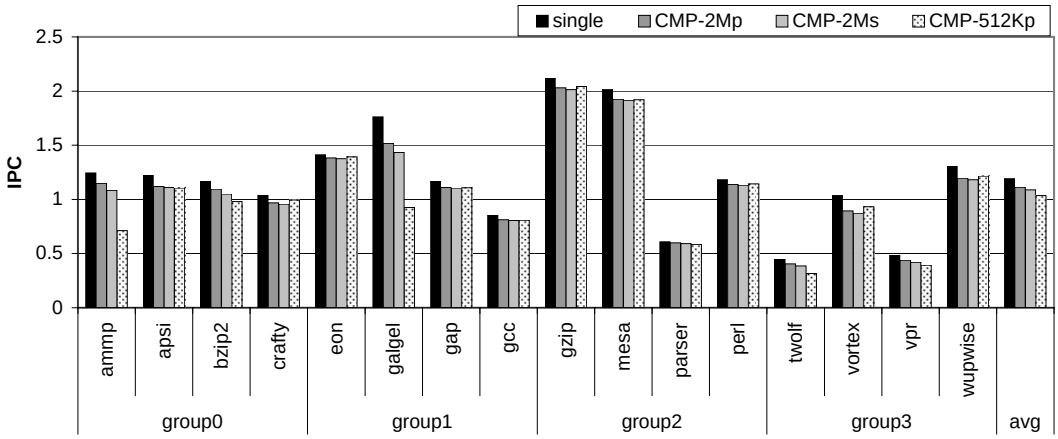


Figure 5.10: IPC of 16 benchmarks with GHL-prefetching grouped into 4 groups. 4 configurations are shown for each benchmark: *single*, where each benchmark is run on a uniprocessor; *CMP-2Mp*, where each group are run on a 4-core CMP with 2M private caches; *CMP-2Ms*, where each group are run on a 4-core CMP with a 2M shared cache; *CMP-512Kp*, where each group are run on a 4-core CMP with 512K private caches.

example, in Figure 5.10, the IPC of crafty changes from 0.96 to 0.99 when the shared 2M cache was replaced with 512K private caches. Further research reveals the cause: `crafty` does not require a large cache and a larger shared cache does not help. At the same time, since it is run with `ammp`, which has a large working set, a lot of its data is replaced by `ammp` and the effective cache is even smaller than 512K. A few other benchmarks, e.g. `eon`, `gzip` and `vortex`, also have similar trends.

Figure 5.11 shows results with configurations similar to those of Figure 5.9 but the number of allowed outstanding memory requests is increased to 128. Speedups are higher in this case and the trend is similar to that of Figure 5.9. It is to be noticed that configurations *CMP-2Mp*, *CMP-2Ms* and *CMP-512Kp* all benefit from more ample memory bandwidth. This is expected since these are more limited than the uni-processor case.

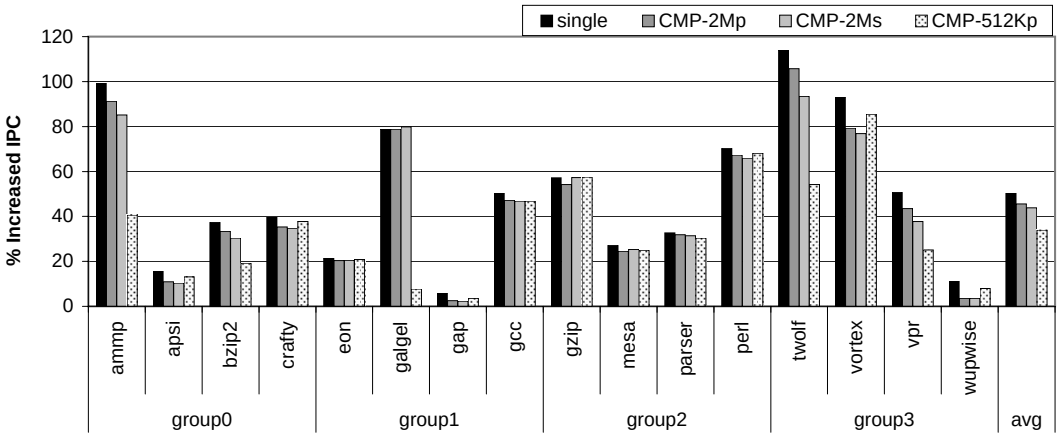


Figure 5.11: IPC of 16 benchmarks with GHL-prefetching grouped into 4 groups. Number of max outstanding memory requests is 128. 4 configurations are shown for each benchmark: *single*, where each benchmark is run on a uni-processor; *CMP-2Mp*, where each group are run on a 4-core CMP with 2M private caches; *CMP-2Ms*, where each group are run on a 4-core CMP with a 2M shared cache; *CMP-512Kp*, where each group are run on a 4-core CMP with 512K private caches.

5.2 Comparison with Other Schemes

We first introduce two metrics used to compare different perfetchers:

Efficiency: We define efficiency of the prefetchers as the ratio of the percentage of increased IPC to the percentage of increased memory bandwidth in Equation 5.1:

$$Efficiency = \frac{\%Increased\_IPC}{\%Incrased\_bandwidth} \tag{5.1}$$

Consecutively-accessed memory region (CAMR): To characterize L2 accesses and explain the performance of different prefetchers, we calculated the average size of CAMR. A CAMR is defined as a set of consecutive memory blocks, all of which are accessed in a given interval. In our simulations, each interval is of 500K dynamic instructions. Benchmarks with small CAMR has scattered accesses and GHL

could perform better while a large CAMR indicate the application is accessing a relatively small and consecutive memory region, for which NLP performs better. Stride prefetcher tracks misses based on PCs and usually captures different patterns than GHL-prefetching and NLP.

Figure 5.12 (a) compares the speedup of GHL-prefetcher to Stride prefetcher and NLP. We can see that they are suited for different benchmarks. Some benchmarks, like `art` and `facerec`, have strided access patterns, and Stride prefetcher is a more compact form to capture and store these patterns. Interestingly, despite its simplicity, NLP also results in significant speedup for more than half of the benchmarks. After further investigation, we discovered this is because: (1) for these benchmarks, most of their blocks have been replaced after a context switch and thus there are a lot of blocks that can accommodate prefetches according to our replacement policy. Hence, prefetched lines can stay longer in the cache. (2) Their accesses are less scattered. Hence, even though NLP is not as accurate as the Stride prefetcher and the blocks it prefetches might belong to different streams, those blocks still have a great chance to be used later on. For the same reason, NLP also shows decent speedup for benchmarks with small working sets. However, for the benchmarks that do not have strided patterns and have larger working sets, like `mesa` and `twolf`, GHL-prefetching does better. For these benchmarks, Stride prefetcher could not capture any patterns and NLP cannot improve performance by just prefetching consecutive blocks. We also examine the L2 access trace for all the benchmarks, which also supports our reasoning: the benchmarks for which NLP performs well all have many consecutive or near accesses, even though they are not strictly strided. Figure 5.12 (b) compares the memory bandwidth increase for the prefetchers. As expected, the Stride prefetcher has the lowest

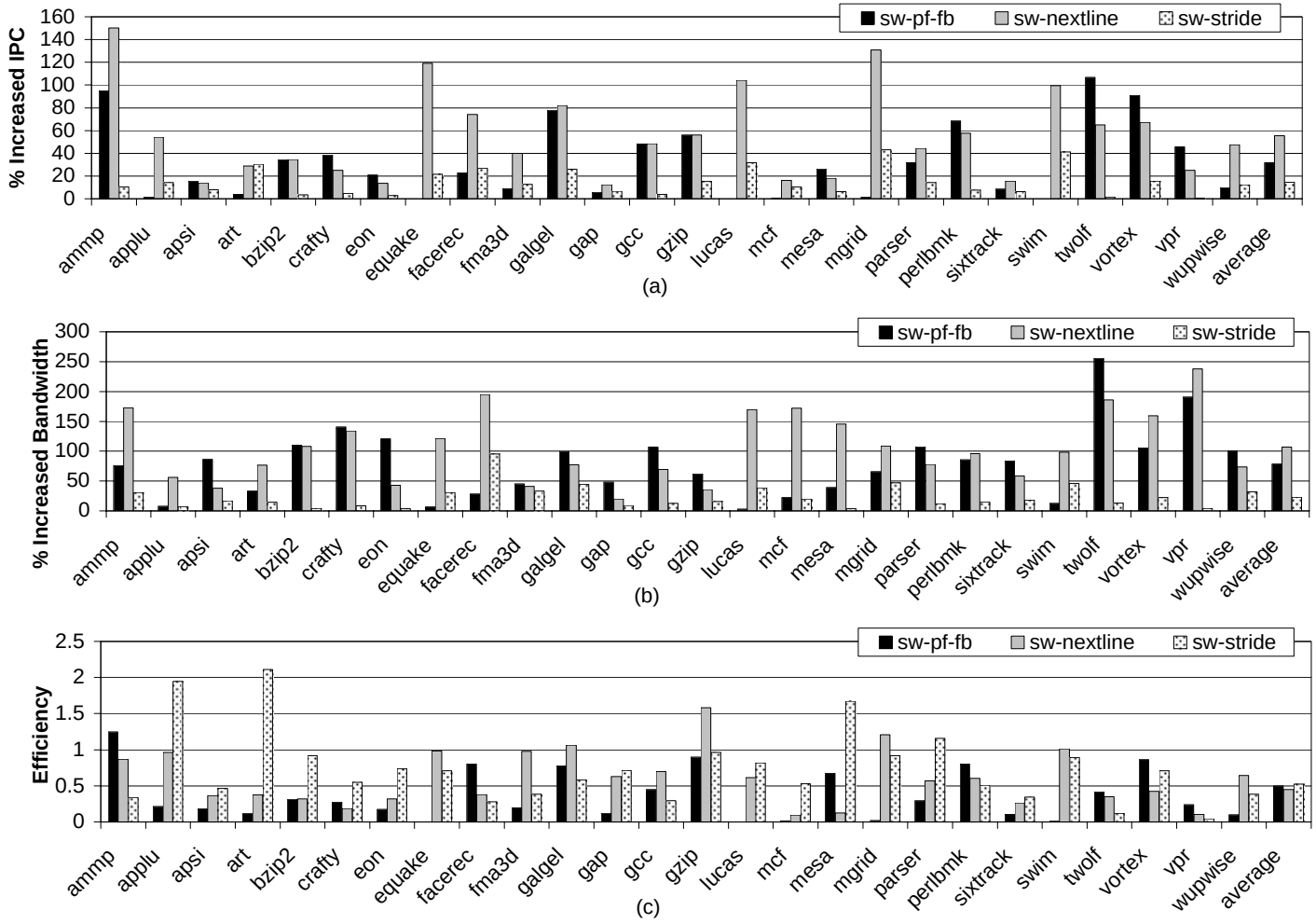


Figure 5.12: (a) Speedup of GHL-prefetching (*sw-pf-fb*), Stride prefetcher (*sw-stride*) and NLP (*sw-nextline*). (b) Memory bandwidth of the three prefetchers. (c) Efficiency of the three prefetchers.

Table 5.1: Baseline bandwidth utilization.

Benchmark	Utilization(%)	Benchmark	Utilization(%)
ammp	8.85	gzip	8.11
applu	23.50	lucas	13.63
apsi	2.96	mcf	13.32
art	29.01	mesa	4.15
bzip2	8.88	mgrid	11.70
crafty	2.87	parser	4.77
eon	1.59	perl	3.95
equake	11.53	sixtrack	2.09
facerec	5.71	swim	17.33
fma3d	6.81	twolf	6.63
galgel	14.54	vortex	4.11
gap	3.15	vpr	7.93
gcc	4.18	wupwise	7.41
Average (%): 7.79			

bandwidth and NLP has the highest bandwidth. Figure 5.12 (c) shows the efficiency of different prefetchers. Stride prefetcher is slightly better than other prefetchers. The three schemes perform differently for different benchmarks. However, their efficiency is similar on average across the entire SPEC'2K suite. This is consistent with the results shown in Figure 5.12 (a) and (b).

Table 5.1 shows the baseline memory bus utilization of all the benchmarks shown in Figure 5.12. It can be seen that all benchmarks except `applu` and `art` use less than 20% of the available bandwidth. This leaves ample idle bandwidth for the prefetchers to consume.

Figure 5.13 (a) depicts how speedup changes along with the increase of the Stride prefetcher's prefetch degree. In the graph, both curves saturate after a degree of 32. However, the curve that shows the IPC with GHL-prefetching on is consistently above the one

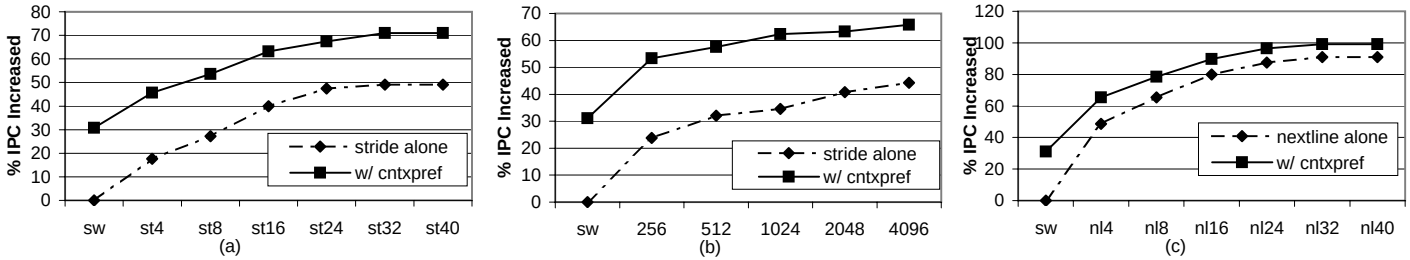


Figure 5.13: (a) Average speedup when increasing the prefetching degree of a Stride prefetcher. (b) Average speedup when increasing the table size of a Stride prefetcher. (c) Average speedup when increasing the prefetching degree of NLP.

showing Stride prefetcher only. This demonstrates the two schemes can work together to achieve cumulative speedup. Figure 5.13 (b) shows the trend of speedup when the prefetcher table size changes with a prefetch degree of 24, which leads to the same conclusion. Figure 5.13 (c) shows the speedup when the NLP's prefetch degree varies. It shows GHL-prefetching constantly adds speedup to NLP, especially in the low prefetch degree range. It suggests GHL-prefetching and NLP can attain cumulative speedup, even though not as much as GHL and Stride. To see the potential of each prefetching scheme, the number of allowed pending memory requests is unlimited in these experiments.

We further studied what caused the difference between GHL and NLP. Figure 5.14 shows the average CAMR size of different benchmarks. For benchmarks with large CAMR sizes, NLP is more suited. For those with small CAMRs, GHL does pretty well for some of them, like `perl`, `twolf`, `vortex`, `vpr`; and has better efficiency for `ammp`, `crafty` and `mesa`. However, NLP has higher efficiency for `apsi`, `eon`, `galgel`, `gcc`, `sixtrack` and `wupwise`, whose CAMRs are also small. Further investigation reveals that for these benchmarks, most of their misses are in the center of a CAMR. In such a case, the nextline prefetches have a high possibility to hit at some useful addresses; in contrast, for the former group, many misses are towards the end of a CAMR and the triggered prefetches are unlikely to hit any useful addresses. Figure 5.15 presents the percentage of misses that are within 4 blocks from the end of a CAMR, for benchmarks with CAMRs that are smaller than 200 blocks. It can be observed that GHL has higher speedup for the benchmarks that have a higher percentage, as shown in the left half of Figure 5.15 indicated by the brace. The only exceptions are `facerec` and `sixtrack`. Even though their CAMR sizes are small and most of the misses are towards the end of a CAMR, they benefit more from NLP. The

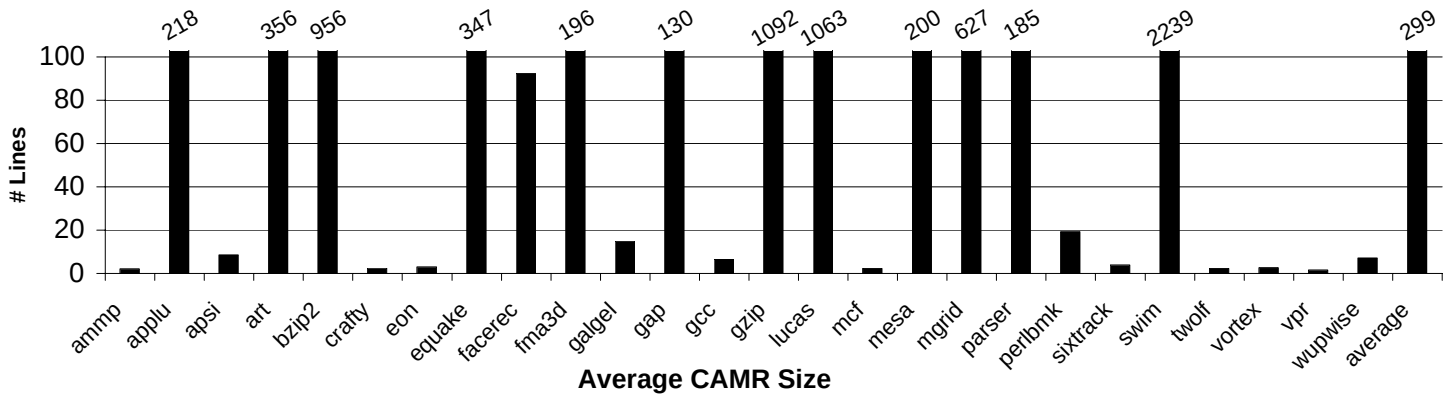


Figure 5.14: Average CAMR sizes with 500K instruction intervals.

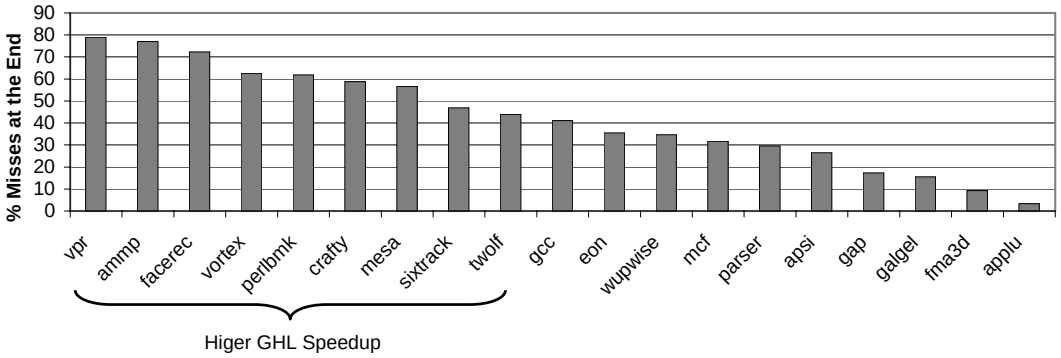


Figure 5.15: Percentage of misses that are with in 4 blocks from the end of a CAMR.

cause is that the distances between their CAMRs are extremely small, e.g. 4 on average for `sixtrack`, which suggests most of its CAMRs are very close to each other. As a result, they can be considered almost consecutive.

Figure 5.16 shows the coverage of GHL-prefetching (*sw-pf-fb*), NLP (*sw-nextline*) and Stride prefetcher (*sw-stride*). Misses are broken down into 4 types: eliminated context switch misses (*cntxmiss*), eliminated normal misses (*miss*), residual context switch misses (*res-cntxmiss*), and residual normal misses (*res-miss*). Coverage is calculated by adding the eliminated context switch misses and eliminated normal misses bars. The results for eliminated context switch misses are consistent with their speedups shown in Figure 5.12: Stride prefetcher does pretty well for strided benchmarks like `art` and `facerec`, but is less effective than NLP or GHL-prefetching on average. NLP eliminates the most context switch misses for a similar group of benchmarks, most of which have large CAMRs, e.g. `applu`, `art`, `equake`, etc. GHL removes the most context switch misses for benchmarks with small CAMR sizes, e.g. `crafty`, `eon`, `perl`, etc. It is interesting to note that a few benchmarks with small CAMRs, like `apsi`, `gcc` and `wupwise`, have a higher speedup with NLP. This

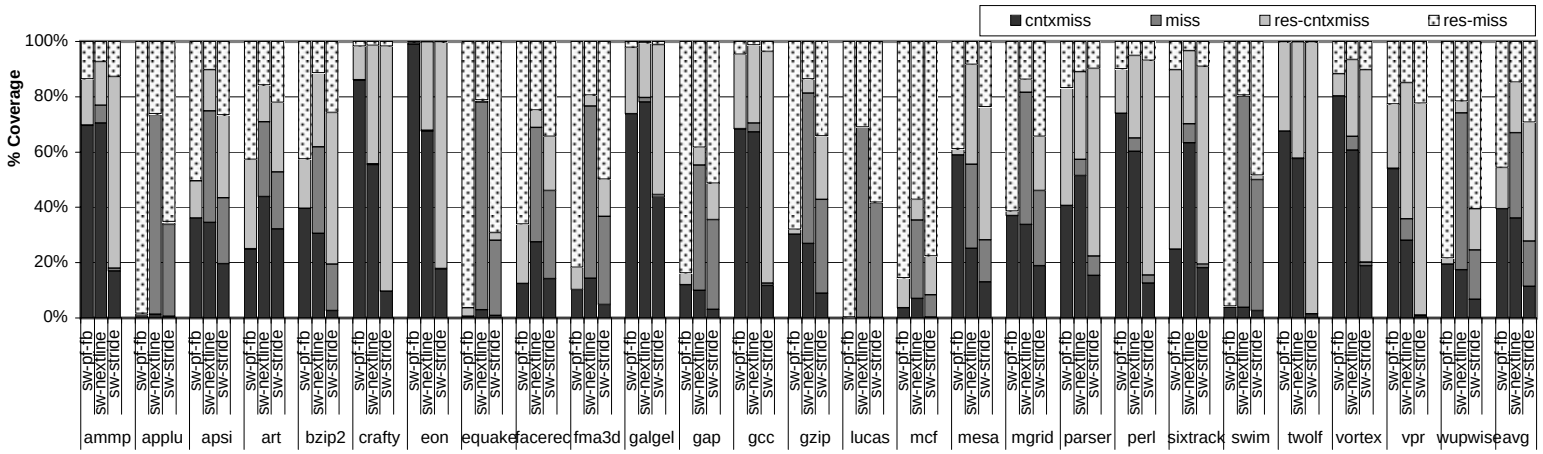


Figure 5.16: Coverage of GHL-prefetching (*sw-pf-fb*), NLP (*sw-nextline*) and Stride prefetcher (*sw-stride*). Misses are broken down into 4 types: eliminated context switch misses (*cntxmiss*), eliminated normal misses (*miss*), residual context switch misses (*res-cntxmiss*), and residual normal misses (*res-miss*).

is because NLP removed more normal misses besides context switch misses. On average, GHL eliminates more context switch misses while NLP removes the most total misses. Stride eliminates the fewest due to its incapability of capturing context switch misses.

Figure 5.17 shows the timeliness of GHL-prefetching (*sw-pf-fb*), NLP (*sw-nextline*) and Stride prefetcher (*sw-stride*). Prefetches are broken down into 4 types: (1) prefetch hits, which are prefetches that fully hide the latency of eliminated misses (*hit*); (2) delayed hits, which hide only part of the latency of an eliminated miss (*delayed-hit*); (3) replaced prefetches, which are brought in the cache too early and are replaced before the same address is accessed (*replaced*); (4) mis-prefetches, which are prefetch addresses that are never used (*miss*). Accuracy is calculated by adding the hits and delayed hits bars. The figure shows that Stride prefetcher has the highest accuracy, however, it also has the lowest speedup because of its low converage. GHL-prefetching has higher accuracy than NLP for a similar group of benchmarks who have small CAMRs, e.g. `crafty`, `eon`, `perl`, etc. And NLP performs better for benchmarks that have large CAMRs. On average, NLP has higher accuracy than GHL-prefetching because more benchmarks have clustered access patterns (large CAMRs). However, it cannot substitute GHL-prefetching for benchmarks with scattered accesses (small CAMRs). The results are consistent with those in Figure 5.12. It is also interesting that GHL-prefetching has very few delayed hits because it is able to prefetch far enough ahead.

Since GHL-prefetching and NLP are complementary, we also examine the hybrid scheme that adaptively selects between these two prefetchers, as described in Section 3.8. Figure 5.18 (a) examines the speedup of GHL-prefetching, NLP and GHL-NLP hybrid. It can be seen that GHL-NLP hybrid has equal or higher speedup than GHL-prefetching

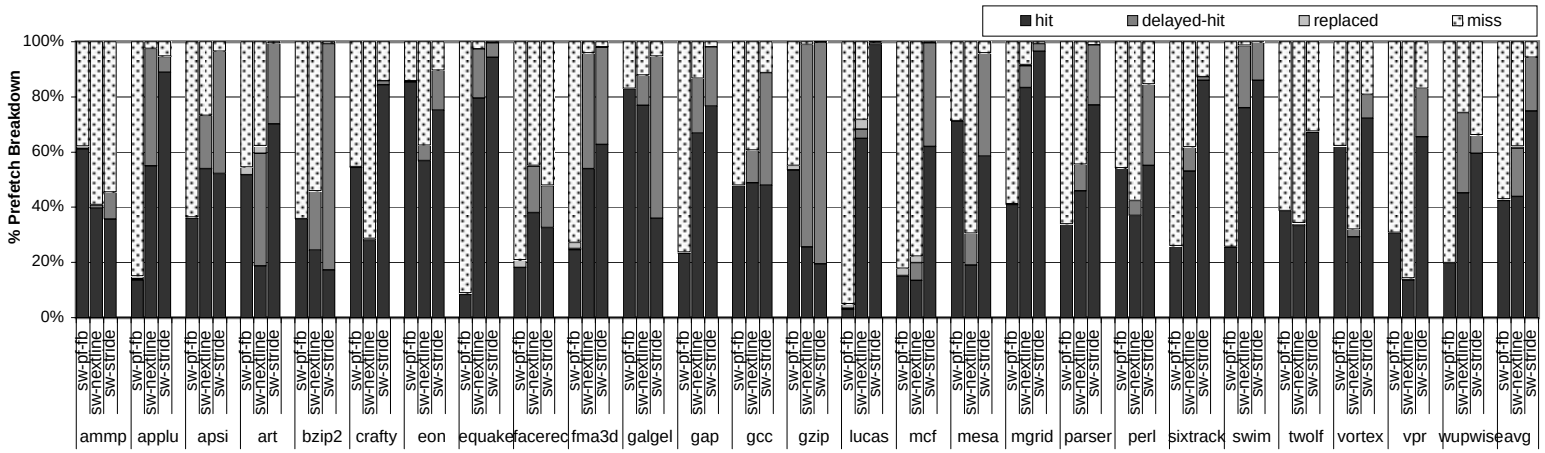


Figure 5.17: Prefetch timeliness of GHL-prefetching (*sw-pf-fb*), NLP (*sw-nextline*) and Stride prefetcher (*sw-stride*). Prefetches are broken down into 4 types: prefetch hits (*hit*), delayed prefetch hits (*delayed-hit*), replaced prefetches (*replaced*), and misprefetches (*miss*).

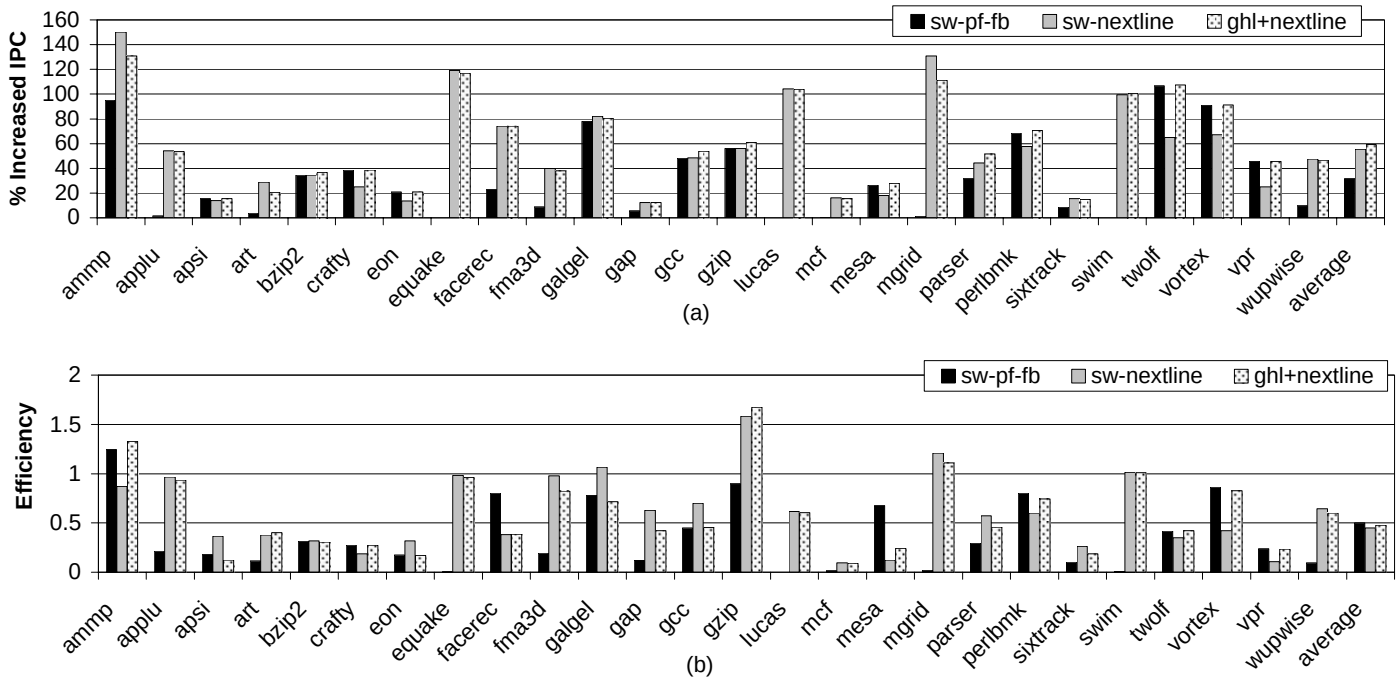


Figure 5.18: Comparing GHL-prefetching (*sw-pf-fb*), NLP (*sw-nextline*) and the GHL-NLP hybrid scheme (*ghl+nextline*): (a) Speedup. (b) Prefetch efficiency.

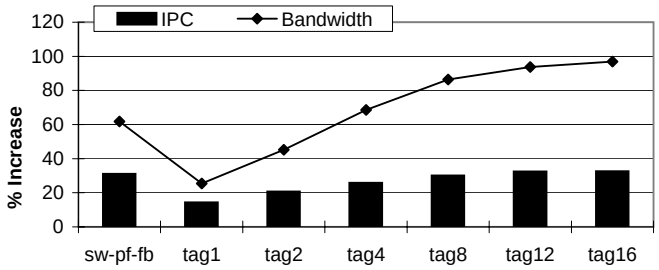


Figure 5.19: Average speedup across all benchmarks for GHL-prefetching (*sw-pf-fb*) and saving tags (*tag**). The blocks show the speedup while the line shows the percentage of memory bandwidth increase.

and NLP for most benchmarks, and its speedup is even higher than both for `parser`. For `ammp`, `art` and `mgrid`, the hybrid scheme has slightly lower speedup than NLP. This is because the feedback mechanism could not detect all the cases where GHL-prefetching is less effective than NLP. On average, GHL-NLP hybrid has the highest speedup, which is 5% higher than NLP's. Besides a hybrid scheme, which is either GHL-prefetching or NLP, GHL-prefetching and NLP can also have cumulative speedup by running together, which is depicted in Figure 5.13 (c). Figure 5.18 (b) shows the efficiency of the three schemes and they are fairly close on average. GHL-NLP hybrid has slightly higher efficiency than NLP and enjoys the highest speedup because it is able to select the better prefetching scheme in most cases.

Besides using a GHL, we also explored other approaches that reduce context switch misses, such as saving the L2 cache tags. We investigated saving all the tags, or a certain number of MRU tags from each cache set. Figure 5.19 compares GHL-prefetching to saving the L2 tags. The L2 cache is 16-way set-associative. The figure shows the speedup and bandwidth increase when saving the tags of different number of MRU lines in each set. It indicates GHL-provides similar speedup compared to saving 4 or more tags with less

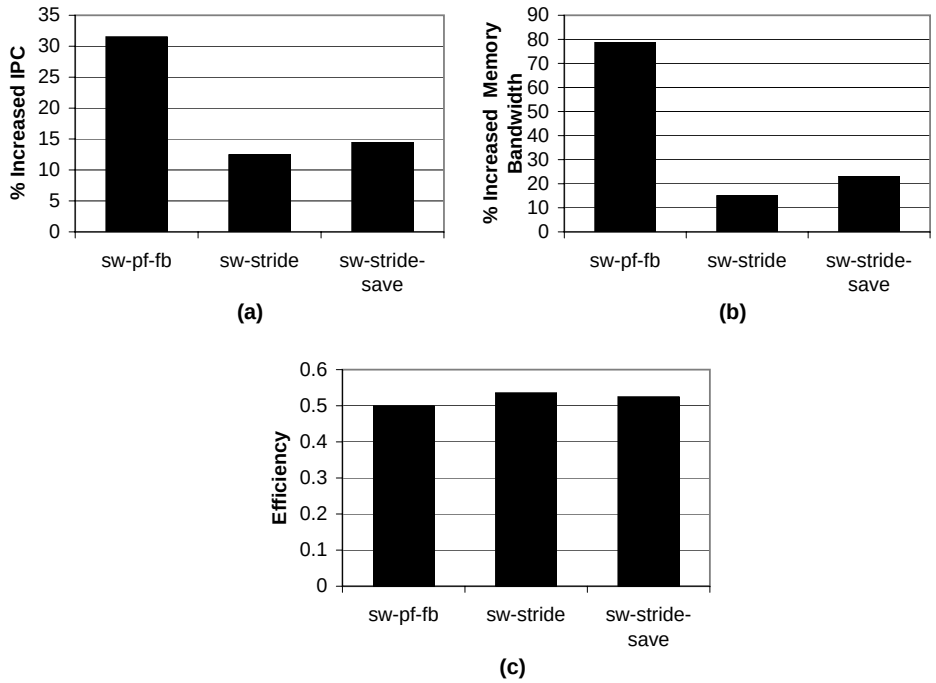


Figure 5.20: Compares GHL-prefetching (*sw-pf-fb*) to Stride prefetcher (*sw-stride*) and Stride prefetcher with saving its tables across context switches (*sw-stride-save*): (a) Increased IPC. (b) Increased bandwidth. (c) Efficiency of the three schemes.

bandwidth consumed. It is true that GHL-prefetching has advantages due to the feedback mechanism. However, a cache line is not an appropriate location for keeping prediction or feedback information since it could often be replaced by other cache accesses and is less likely to survive context switches. We also explored saving the tags of the blocks touched before a context switch but the resulting performance is significantly worse than GHL prefetching. This is expected since the GHL scheme retains information across several context switches.

We also examined saving the tables of a Stride prefetcher. Stride prefetcher's tables are small and storing them across context switches brings little speedup (around 2%) as shown in Figure 5.20.

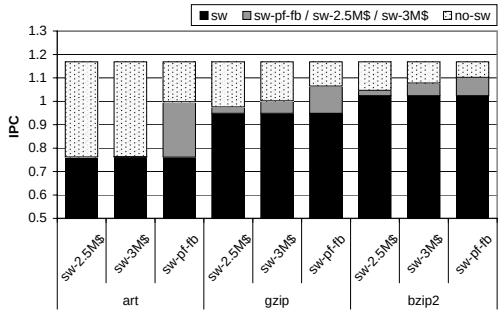


Figure 5.21: Performance of our GHL-prefetching mechanism with a 2MB L2 cache compared to the performance of larger caches.

We also compare a 2.5MB L2 cache (*sw-2.5M$*) and a 3MB L2 cache (*sw-3M$*) with our feedback controlled GHL-prefetching scheme with a 2M L2 cache (*sw-pf-fb*). The results are shown for different interfering benchmarks in Figure 5.21. Our scheme uses less than 100KBytes for a 2MB cache with 64-byte block size but outperforms even a 3MB L2 cache for interfering benchmarks with intermediate to high impact. Unlike caches, which only try to keep a process' working set on chip for the duration when a process occupies the processor, our scheme tries to maintain the process' working set across context switches. This is achieved by spending some area on saving which data is needed rather than saving the data itself. Since this information uses much less space than the data itself, it is possible to save and load it in much less time and use it to guide prefetching. As shown earlier, when there are multiple interfering benchmarks, more than half of the original application's blocks will not survive. Thus, the area spent on caches could be used much more efficiently by using it for GHL-prefetching.

To better understand the advantages of different schemes, we also compare their average IPC and speedup across the entire SPEC'2K suite on 4-core CMPs, as shown in Figure 5.22. The benchmarks are sorted alphabetically and every four are grouped together

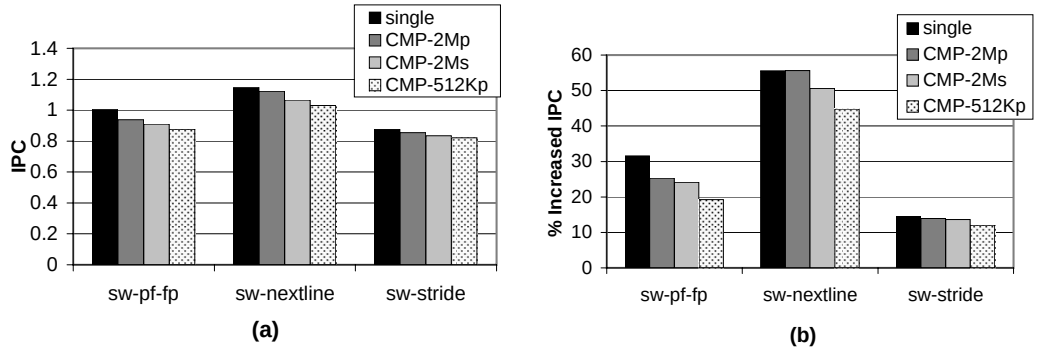


Figure 5.22: Comparing average (a) IPC and (b) Speedup of GHL-prefetching (*sw-pf-fb*), NLP (*sw-nextline*) and Stride (*sw-stride*), each with 4 different configurations: uni-processor (*single*), private 2M L2 caches (*CMP-2Mp*), shared 2M L2 (*CMP-2Ms*), and private 512K L2 caches (*CMP-512Kp*).

to run on a 4-core CMP.

Single is the uni-processor case. *CMP-2Mp* is using 2M private L2 caches while *CMP-2Ms* is using a single shared L2 cache. In *CMP-512Kp*, the shared L2 cache is statically divided into four 512K private caches. The shared resources are progressingly limited going from *single* to *CMP-512Kp* and should lead to lower performance for all schemes, including the baseline. This is easily confirmed from the results in Figure 5.22 (a). However, it is not necessarily true that the speedup will go down since the baseline is also impacted. From Figure 5.22 (b), it can be seen that NLP actually has slightly higher speedup for the 2M private L2 case. However, the overall trend is still going down. On average, GHL-prefetching is more sensitive to memory bandwidth than NLP and Stride prefetcher. This is expected because it issues prefetches right after each context switch rather than being triggered by misses. However, GHL-prefetching is better than other schemes when bandwidth permits.

It would be more interesting to look at the schemes' performance for each individual benchmark, as shown in Figure 5.23 and Figure 5.24. The benchmarks are sorted

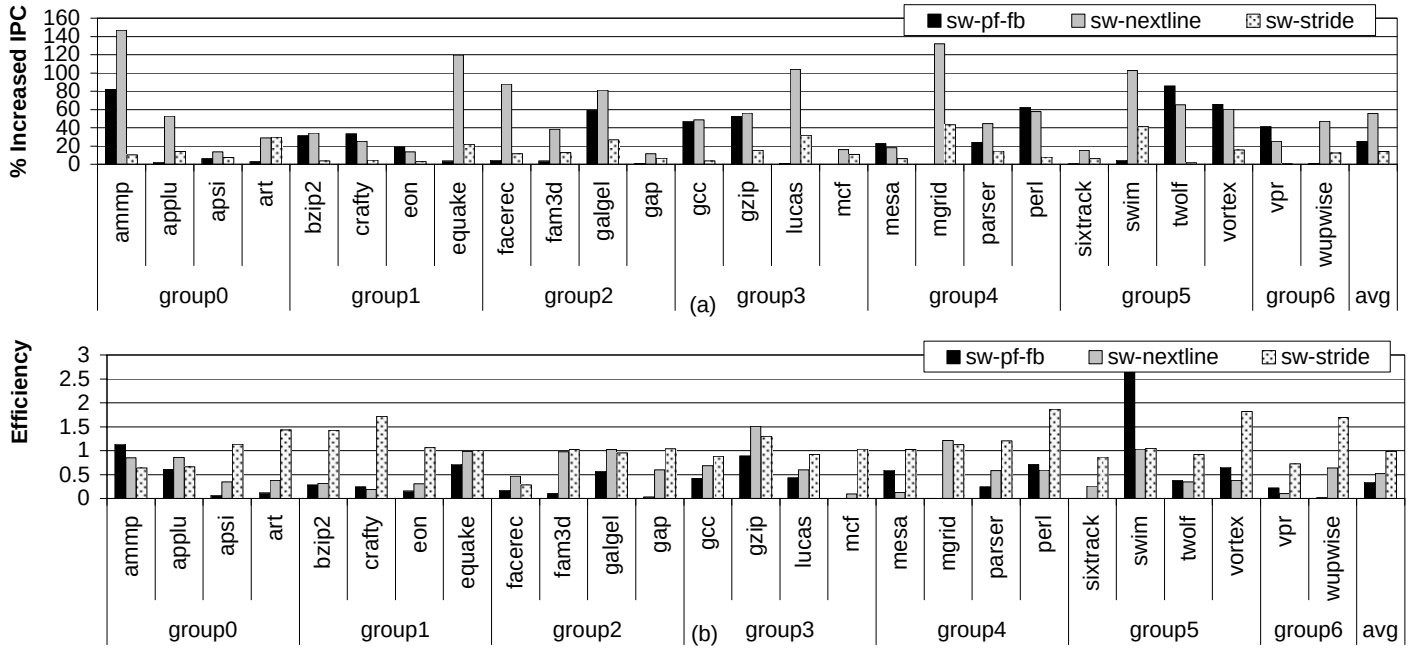


Figure 5.23: Comparing the 3 prefetchers with 2M private L2 caches. (a) Speedup of GHL-prefetching (*sw-pf-fb*), NLP (*sw-nextline*) and Stride (*sw-stride*). (b) Prefetch efficiency of the 3 prefetchers.

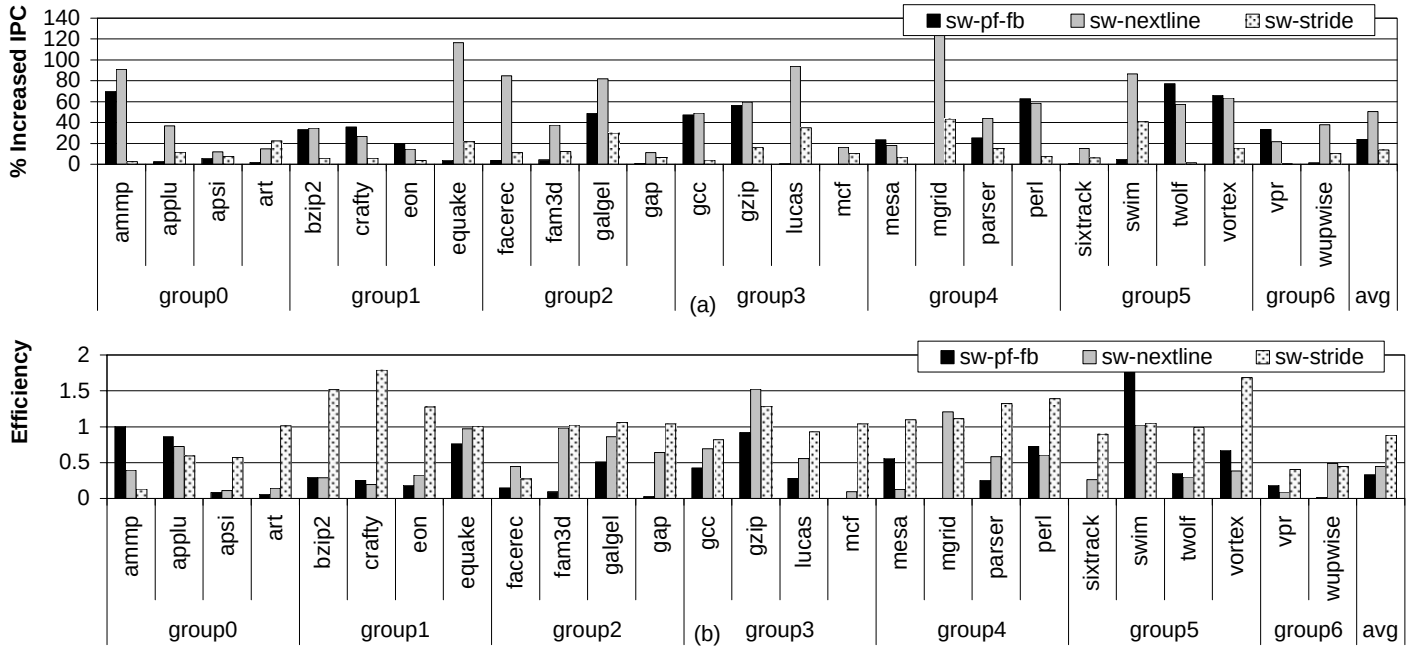


Figure 5.24: Comparing the 3 prefetchers with 2M shared L2 cache. (a) Speedup of GHL-prefetching (*sw-pf-fb*), NLP (*sw-nextline*) and Stride (*sw-stride*). (b) Prefetch efficiency of the 3 prefetchers.

alphabetically and every four are grouped together to run on a 4-core CMP. The first figure is the private L2 case while the second figure shows results for a shared L2. Compared with Figure 5.12, all prefetching schemes have lower performance due to higher contention on shared resources. But the trend is similar. GHL-prefetching, NLP and stride are still suited for the same types of benchmarks. The differences between different schemes for each benchmark are not exactly proportional. For instance, in Figure 5.24 (a), NLP's lead over GHL-prefetching is much smaller for `ammp`, while GHL-prefetching outperforms NLP to a lesser degree on `twolf` and `vortex`. This is due to their interaction through the shared cache and the memory bus, which results in different behavior compared to when they are run alone. In both figures, GHL-prefetching efficiency is extremely high for `swim`, which is very different from Figure 5.12. This is because GHL-prefetching is rarely activated for `swim`. Thus, both speedup and increased memory traffic is very small and interference from other benchmarks running on the same core can change the computed efficiency dramatically.

5.3 Evaluation of Phase Guided-Prefetching

Table 5.2 shows the accuracy of the phase predictor for all benchmarks in the SPEC'2K suite. The predictor has very good accuracy for `facerec`, `fma3d` and `galgel`. It also shows moderate accuracy for `art` and `gzip`. We examined the L2 access patterns for these benchmarks and discovered that these applications are relatively regular at a coarser granularity compared to the rest of the suite. Take `galgel` for example, whose accuracy is 31.5% with 50k intervals and 60% with 200k intervals. Its access patterns are quite irregular at a per-access granularity, which cannot be captured by Stride or Markov prefetchers. However, at a coarser granularity, say, every 1000 accesses, it follows a very

Table 5.2: Phase predictor accuracy with 50K instruction intervals.

Benchmark	Accuracy(%)	Benchmark	Accuracy(%)
ammp	2.99	gzip	15.3
applu	0	lucas	0
apsi	4.09	mcf	0
art	17.3	mesa	0.42
bzip2	0.01	mgrid	0
crafty	0	parser	0.07
eon	0.82	perlbmk	3.49
equake	0	sixtrack	0
facerec	67.6	swim	0
fma3d	93.4	twolf	0
galgel	31.5	vortex	0.43
gap	0	vpr	0
gcc	0.31	wupwise	0

regular pattern and can be captured by our phase predictor. Among these 5 benchmarks, `fma3d` and `gzip`'s behavior can be captured by other prefetching techniques (NLP, stride or GHL) and phase-guided prefetching does not perform better. Thus further discussion will focus on the other three benchmarks.

Figure 5.25 (a) depicts the trend when changing the maximum number of phases allowed (including non-confident phases). The maximum number of phases is limited by the size of the phase table. When all entries have been used, the LRU one will be replaced by the new phase. Allowing more phases increases the chance of finding a match but it also increases die area. Most of the phases in the phase table are non-confident phases and no addresses are saved for them.

Figure 5.25 (b) examines the impact of the length of sample intervals. The length of the sample intervals determines the targeted granularity. Appropriate granularity is benchmark-dependent. In the graph, we can see that `galgel` benefits from longer intervals

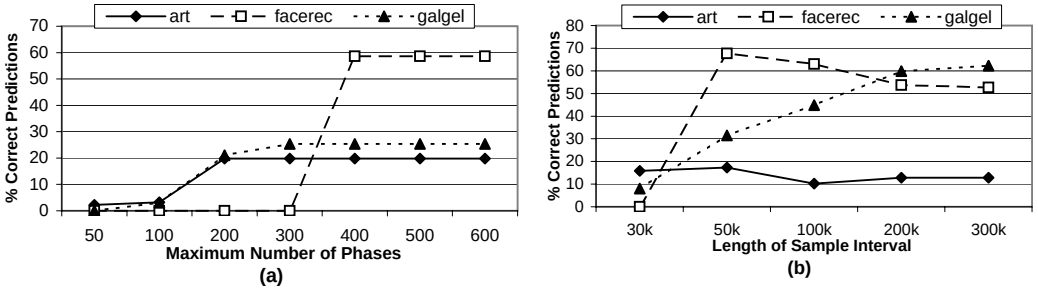


Figure 5.25: Accuracy of phase predictor.

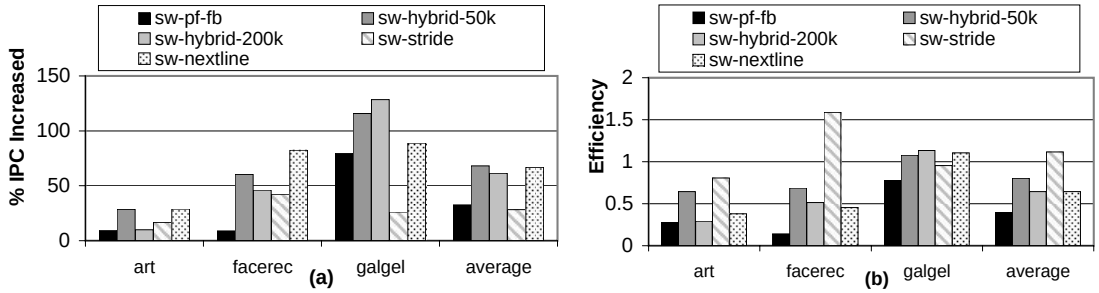


Figure 5.26: (a) Speedup of GHL-prefetching (*sw-pf-fb*), GHL-phase hybrid approach with 50k and 200k sample intervals (*sw-hybrid-50k* and *sw-hybrid-200k*), Stride prefetcher (*sw-stride*) and NLP (*sw-nextline*). (b) Prefetching efficiency of the two approaches

while `art` favors shorter ones. Thus, we choose 50K-instruction intervals which is a good compromise across all benchmarks. We also tried variable-length intervals. However, all existing techniques [11, 49] focus on coarser granularity (greater than 10M instructions). When we apply variable-length phase detection techniques on the short intervals we used, experiments show it requires long learning periods and adds difficulty to phase classification because the number of phases (including both confident and non-confident phases) is large and the length of intervals could be significantly different.

Figure 5.26 (a) shows the speedup of GHL-prefetching and the GHL-phase hybrid approach for a group of benchmarks and compares them against the speedup of Stride

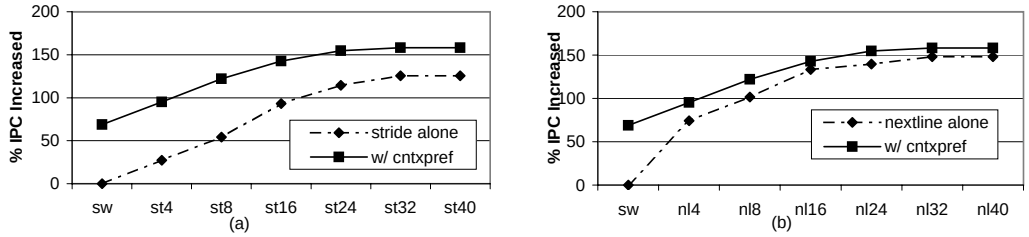


Figure 5.27: (a) Average speedup when increasing prefetching degree of Stride prefetcher. (b) Average speedup when varying prefetching degree of NLP.

prefetcher and NLP. In the graph, the hybrid approach with 50k sample intervals constantly outperforms the original GHL-prefetching. This results in 40% higher speedup for the group and 5% higher across the entire SPEC'2K suite. When compared against the other two prefetching schemes, it is significantly better than Stride and slightly worse than NLP. However, it has a higher efficiency than NLP. We also try 200k sample intervals. However, only `galgel` has higher speedup while the other two benchmarks suffer worse performance. Figure 5.26 (b) compares the prefetching efficiency of the two approaches. The 50k-interval hybrid approach has even higher efficiency than the original GHL. Further investigation indicates these benchmarks have relatively regular L2 access patterns at this granularity. The hybrid approach can accurately capture these patterns. The original GHL-prefetching performs poorly for these benchmarks and the new hybrid approach improves it by capturing the phase behavior. Unfortunately, this is not true for the rest of the benchmarks in the SPEC'2K suite. However, we expect to see more of these patterns exist for programs that do not have regular patterns at a fine granularity, but are regular at a coarser one. One example would be scientific programs that perform complex operations and data accesses in a limited region of code but exhibit regular behavior on higher levels.

Figure 5.27 (a) shows the speedup when the prefetching degree of stride prefetcher

is increased, with or without GHL hybrid prefetching. Both curves stop going up at a degree of 32. The figure demonstrates GHL hybrid prefetching consistently adds performance to Stride prefetcher. It is clear that they capture different types of patterns and combining the two will result in accumulative speedup. Figure 5.27 (b) shows the speedup of running NLP, with or without GHL hybrid prefetching. A similar conclusion can be drawn for NLP, even though the difference between NLP and GHL hybrid has a smaller difference.

Chapter 6

Conclusions

Among the various costs of a context switch, its impact on the performance of L2 caches is the most significant because of the resulting high miss penalty. In this dissertation, we propose mitigating this impact by prefetching into the L2 cache the data a program was using before it was swapped out. This is effectively restoring a program's locality destroyed by the context switch.

We use a Global History List to track which blocks are accessed when a program is running. When the OS decides to swap it out, these blocks' addresses are saved along with its context into memory. The next time the application runs, these addresses are loaded into a prefetching queue to guide prefetching. We carefully devise a placement policy to increase the lifetime of prefetches in the cache while minimizing the chance of replacing a line brought in by the process' demand accesses. To reduce the memory traffic incurred by prefetching, we design a feedback mechanism which eliminates useless prefetches by tracking the reuse patterns of prefetches, resulting in a significant memory bandwidth usage reduction. Finally, we extend GHL-prefetching with phase-guided prefetching, which accurately captures more access patterns.

Experimental results show that our scheme achieves 36% average speedup over no prefetching and 11%, and 24% average speedup in the presence of NLP and Stride prefetchers respectively. In addition, the proposed feedback mechanism brings the extra memory traffic overhead down to 60% with a slightly lower speedup of 31% over no prefetching. Context prefetching with feedback not only outperforms traditional prefetching mechanisms, it does so while generating considerably lower extra memory traffic. These results show that context prefetching greatly reduces the impact of context switches on L2 cache performance, even outperforming a significantly larger L2 cache. The phase-guided prefetching scheme

extends the prefetching capability of GHL-prefetching on three benchmarks, which results in a 40% higher speedup on average for the three benchmarks and 5% higher speedup across all SPEC'2K benchmarks.

Since we only examined benchmarks in SPEC'2K, one of our future directions would be to look at more benchmarks, e.g. SPEC 2006 and commercial benchmarks. In our design, we only explored the case where one application occupies only a single core. It is also interesting to look at multi-threaded programs, which use multiple cores simultaneously. This includes a lot of scientific and commercial benchmarks. In such a case, we can assign application IDs to cache blocks, which identify which application the blocks belong to. Potentially, a single application can use GHLs on several cores, which provides more storage space to track accesses. As we mentioned in section 5.3, some scientific benchmark could also benefit from phase-guided prefetching due to their high-level regularity.

Bibliography

[1] M. Ajtai, J. Aspnes, M. Naor, Y. Rabani, L. J. Schulman, and O. Waarts. Fairness in scheduling. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 477–485, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system, 1990.

[3] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *33rd International Symposium on Microarchitecture*, pages 245–257, 2000.

[4] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2005.

[5] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, June 1997.

[6] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In

ICS '07: Proceedings of the 21st annual international conference on Supercomputing, pages 242–252, New York, NY, USA, 2007. ACM.

[7] T. F. Chen and J. L. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, 5(44):609–623, May 1995.

[8] A. Dhodapkar and J. Smith. Comparing program phase detection techniques. In *36th International Symposium on Microarchitecture*, Dec. 2003.

[9] A. Dhodapkar and J. E. Smith. Dynamic microarchitecture adaptation via co-designed virtual machines. In *International Solid State Circuits Conference*, Feb. 2002.

[10] A. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[11] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *36th International Symposium on Microarchitecture*, Dec. 2003.

[12] J. Donald and M. Martonosi. An efficient, practical parallelization methodology for multicore architecture simulation. *Computer Architecture Letters*, 5(2), August 2006.

[13] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley-Interscience, New York, NY, 2001.

[14] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *12th Annual International Conference on Parallel Architectures and Compilation Techniques*, Oct. 2003.

[15] R. Fromm and N. Treuhaft. Revisiting the cache interference costs of context switching. Technical report, Computer Science Division, University of California, Berkeley, 1996.

[16] W. Grnewald and T. Ungerer. Towards extremely fast context switching in a block-multithreaded processor. In *Proc. of the 22 nd Euromicro Conf*, pages 592–599, 1996.

[17] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–355, Washington, DC, USA, 2007. IEEE Computer Society.

[18] G. Hamerly, E. Perelman, and B. Calder. How to use simpoint to pick simulation points. *ACM SIGMETRICS Performance Evaluation Review*, 31(4), Mar. 2004.

[19] S. I. Hong, S. A. McKee, M. H. Salinas, R. H. Klenke, J. H. Aylor, and W. A. Wulf. Access order and effective bandwidth for streams on a direct rambus memory. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 80, Washington, DC, USA, 1999. IEEE Computer Society.

[20] M. Huang, J. Renau, and J.Torrellas. Positional Adaptation of Processors: Application to Energy Reduction. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[21] C. Isci and M. Martonosi. Identifying program power phase behavior using power vectors. In *Workshop on Workload Characterization*, Sept. 2003.

[22] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Method-

ology and empirical data. In *36th International Symposium on Microarchitecture*, Dec. 2003.

[23] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[24] G. B. Kandiraju and A. Sivasubramaniam. Going the distance for tlb prefetching: an application-driven study. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 195–206, Washington, DC, USA, 2002. IEEE Computer Society.

[25] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society.

[26] H. Kwak, B. Lee, A. R. Hurson, S.-H. Yoon, and W.-J. Hahn. Effects of multithreading on cache performance. *IEEE Trans. Comput.*, 48(2):176–184, 1999.

[27] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. In *International Symposium on Performance Analysis of Systems and Software*, pages 135–146, Mar. 2005.

[28] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *11th International Symposium on High Performance Computer Architecture*, Feb. 2005.

[29] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *ExpCS '07:*

Proceedings of the 2007 workshop on Experimental computer science, page 2, New York, NY, USA, 2007. ACM.

[30] W.-F. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, Feb. 2001.

[31] F. Liu, F. Guo, Y. Solihin, S. Kim, and A. Eker. Characterizing and modeling the behavior of context switch misses. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 91–101, New York, NY, USA, 2008. ACM.

[32] W. Liu and M. Huang. Expert: Expedited simulation exploiting program behavior repetition. In *International Conference on Supercomputing*, June 2004.

[33] J. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. LeCam and J. Neyman, editors, *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkeley, CA, 1967. University of California Press.

[34] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 104–113, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[35] S. A. Mckee and W. A. Wulf. Access ordering and memory-conscious cache utilization.

In *HPCA '95: Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, page 253, Washington, DC, USA, 1995. IEEE Computer Society.

[36] B. Millar and P. Gillingham. Two high-bandwidth memory bus structures. *IEEE Des. Test*, 16(1):42–52, 1999.

[37] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 75–84, New York, NY, USA, 1991. ACM.

[38] J. Mukundan. Instruction cache checkpoints using phase tracking and prediction. *Master's Thesis, North Carolina State University*, 2005.

[39] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 96, Washington, DC, USA, 2004. IEEE Computer Society.

[40] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. *IEEE Micro*, 25(1):90–97, 2005.

[41] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, August 1997.

[42] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2003.

[43] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[44] RedHat, IBM, Intel, and Hitachi. http://sourceware.org/systemtap/.

[45] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 3–13, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[46] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 128–138, New York, NY, USA, 2000. ACM.

[47] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[48] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[49] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[50] B. Smith. The architecture of hep. In *on Parallel MIMD computation: HEP supercomputer and its applications*, pages 41–55, Cambridge, MA, USA, 1985. Massachusetts Institute of Technology.

[51] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Proceedings of Supercomputing*, Nov. 1992.

[52] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 171–182, Washington, DC, USA, 2002. IEEE Computer Society.

[53] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 1–12, New York, NY, USA, 2001. ACM.

[54] M. R. Thistle and B. J. Smith. A processor architecture for horizon. In *Supercomputing '88: Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, pages 35–41, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[55] J. Torrellas, A. Tucker, and A. Gupta. Benefits of cache-affinity scheduling in shared-memory multiprocessors: a summary. In *SIGMETRICS '93: Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 272–274, New York, NY, USA, 1993. ACM.

[56] D. Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the

enigma of do-nothing loops). In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, page 4, New York, NY, USA, 2007. ACM.

[57] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee. The impulse memory controller. *IEEE Trans. Comput.*, 50(11):1117–1132, 2001.