IMPROVING MEMORY HIERARCHY PERFORMANCE WITH ADDRESSLESS PRELOAD, ORDER-FREE LSQ, AND RUNAHEAD SCHEDULING

By

ZHEN YANG

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2007

To my family

ACKNOWLEDGMENTS

Thanks to all for their help and guidance.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

IMPROVING MEMORY HIERARCHY PERFORMANCE WITH ADDRESSLESS PRELOAD, ORDER-FREE LSQ, AND RUNAHEAD SCHEDULING

By

Zhen Yang

December 2007

Chair: Jih-Kwon Peir
Major: Computer Engineering

The average memory access latency is determined by three primary factors, cache hit latency, miss rate, and miss penalty. It is well known that cache miss penalty in processor cycles continues to grow. For those memory-bound workloads, a promising alternative is to exploit memory-level parallelism by overlapping multiple memory accesses. We study P-load scheme (P-load stands for Preload), an efficient solution to reduce the cache miss penalty by overlapping cache misses. To reduce cache misses, we also introduce a cache organization with an efficient replacement policy to specifically reduce conflict misses.

A recent trend is to fetch and issue multiple instructions from multiple threads at the same time on one processor. This design benefits much from resource sharing among multiple threads. However, contentions of shared resources including caches, instruction issue window and instruction window may hamper the performance improvement from multi-threading schemes. In the third proposed research, we evaluate a technique to solve the resource contention problem in multi-threading environment.

Store-load forwarding is a critical aspect of dynamically scheduled execution in modern processors. Conventional processors implement store-load forwarding by buffering the addresses

and data values of all in-flight stores in an age-ordered store queue. A load accesses the data cache and in parallel associatively searches the store queue for older stores with matching addresses. Associative structures can be made fast, but often at the cost of substantial additional energy, area, and/or design effort. We introduce a new order-free store queue that decouples the matching of the store/load address and its corresponding age-based priority encoding logic from the original store queue and largely decreases the hardware complexity.

CHAPTER 1
INTRODUCTION

Computer pioneers correctly predicted that programmers would want unlimited amounts of fast memory. An economical solution to that desire is a *memory hierarchy*, which takes advantage of locality and cost-performance of memory techniques. The principle of locality, that most programs do not access all instructions or data uniformly, combined with the guideline that smaller hardware is faster, led to hierarchies based on memories of different speeds and sizes.

Cache is used to reduce the *average memory access latency*. It is a smaller, faster memory which stores copies of the instruction or data from the most frequently used main memory locations. As long as most memory accesses are to cached memory locations, the average latency of memory accesses will be closer to the cache access latency than to the latency of main memory.

The average memory access latency can be determined by three primary factors, the time needed to access the cache (*hit latency*), the fraction of memory references that can not be satisfied by the cache (*miss rate*) and the time needed to fetch data and instructions from the next memory hierarchy (*miss penalty*).

$$\textit{Average Memory Access Latency} = \textit{(Hit Latency)} + \textit{(Miss Rate)} \times \textit{(Miss Penalty)}$$

It is well known that cache miss penalty in processor cycles continues to grow because the rapid improvements in processor clock frequencies have outpaced the improvements in memory and interconnect speeds, which is the so called "*memory wall*" problem. For those memory-bound workloads, a promising alternative is to exploit *memory-level parallelism* (*MLP*) by overlapping multiple memory accesses. In the first proposed research, we will study P-load scheme (P-load stands for Preload), an efficient solution to reduce the cache miss penalty by overlapping cache misses.

A cache array is broken into fixed-size blocks. Typically, a cache can be organized in three ways. If each block has only one place it can appear in the cache, the cache is said to be *direct mapped*. If a block can be placed anywhere in the cache, the cache is said to be *fully associative*. If a block can be placed in a restricted set of places in the cache, the cache is *set associative*. A block is first mapped onto a set, and then the block can be placed anywhere within that set. The set is usually chosen by decoding a set of index bit from the block address.

In order to lower cache miss rate, a great deal of analysis has been done on cache behavior in an attempt to find the best combination of size, associativity, block size, and so on. Cache misses can be divided into three categories (known as the Three Cs): *Compulsory misses* are those misses caused by the first reference to a datum. Cache size and associativity make no difference to the number of compulsory misses. *Capacity misses* are those misses that a cache of a given size will have, regardless of its associativity or block size. *Conflict misses* are those misses that could have been avoided, had the cache not evicted an entry earlier. In the second proposed research, a cache organization with an efficient replacement policy is introduced to specifically reduce conflict misses.

A recent trend is to fetch and issue multiple instructions from multiple threads at the same time on one processor. This design benefits much from resource sharing among multiple threads. It outperforms previous models of hardware multithreading primarily because it hides short latencies much more effectively, which can often dominate performance on a uniprocessor. However, contentions of shared resources including caches, instruction issue window and instruction window may hamper the performance improvement from multi-threading schemes. In the third proposed research, we will evaluate a technique to solve the resource contention problem in multi-threading environment.

Store-load forwarding is a critical aspect of dynamically scheduled execution. Conventional processors implement store-load forwarding by buffering the addresses and data values of all in-flight stores in an age-ordered *store queue* (*SQ*). A load accesses the data cache and in parallel associatively searches the SQ for older stores with matching addresses. Associative structures can be made fast, but often at the cost of substantial additional energy, area, and/or design effort. We introduce a new order-free load-store queue that decouples the matching of the store/load address and its corresponding age-based priority encoding logic from the original content-addressable memory SQ and largely decreases the hardware complexity. The performance evaluation shows a significant improvement in the execution time is achievable comparing with other existing scalable load-store queue proposals.

1.1 Exploitation of Memory Level Parallelism with P-load

Modern out-of-order processors with non-blocking caches exploit MLP by overlapping cache misses in a wide instruction window. The exploitation of MLP, however, can be limited due to long-latency operations in producing the base address of a cache miss load. Under this circumstance, the child load cannot be issued and start to execute until its base register is produced by the parent instruction. When the parent instruction is also a cache miss load, a serialization of the two loads must be enforced to satisfy the load-load data dependence. One typical example is the pointer-chasing problem in many applications with *linked data structures* (*LDS*), where accessing the successor node cannot start until the pointer is available, possibly from memory. Similarly, indirect accesses to large array structures may face the same problem when both address and data accesses encounter cache misses. With limited numbers of instruction window entries and issue window entries in an out-of-order execution and in-order commit processor, these non-overlapped long-latency memory accesses can congest the instruction and issue windows and stall the processor.

Each cache miss encounters delays in sending the request to memory controller, accessing the DRAM array, and receiving the data. The sum of delays in sending request and receiving data is called interconnect delay. With load-load data dependences, normally, resolution of the dependent load's address and triggering of the dependent load's execution are done at processor side after the parent's data is returned from memory to processor. In fact, the resolution can be done at memory controller as soon as the parent load finishes DRAM access. In this way, the dependent load can start to access DRAM array upon the parent's data is fetched from DRAM array. The issuing of the dependent load does not have to experience the receiving parent's data and sending dependent load's request. Hence, the interconnect delay of the two loads can be overlapped.

Based on the above observation, we propose a mechanism that dynamically captures the load-load data dependences at runtime. A special P-load is issued in place of the dependent load without waiting for the parent load, thus effectively overlapping the two loads. The P-load provides necessary information for the memory controller to calculate the correct memory address upon the availability of the parent's data to eliminate any interconnect delay between the two loads. Performance evaluations based on SPEC2000 and Olden benchmarks show that significant speedups up to 40% with an average of 16% are achievable using the P-load.

1.2 Least Frequently Used Replacement Policy in Elbow Cache

As the performance gap between processor and memory continues to widen, cache hierarchy designs and performance become even more critical. Applications with regular patterns of memory access can experience severe cache conflict misses in set-associative cache, in which the number of cache frames that a memory block can be mapped into is fixed as the set associativity. When all of the frames in a set are occupied, a newly missed block replaces an old block according to the principle of memory reference locality. Furthermore, the same block

address bits are used to index every cache partition. If two blocks conflict for a single location in one partition, they also conflict for the same location in the other partitions. Under these constraints, heavy conflicts may occur in a few sets due to uneven distribution of memory addresses across the entire cache sets that cause severe performance degradation.

To alleviate conflict misses, the skewed-associative cache [Seznec 1993a; Seznec and Bodin 1993b; Bodin and Seznec 1997] employs multiple hashing functions for members in a set. Each set also consists of one frame from each of the *n* cache partitions. But the location of the frame in each partition can be different based on a different hashing function. The insight behind the skewed-associative cache is that whenever two blocks conflict for a single location in cache partition *i*, they have low probability to conflict for a location in cache partition *j*.

The elbow cache [Spjuth et al. 2005] extends skewed-associative cache organization by carefully selecting its victim and, in the case of a conflict, move the conflicting cache block to its alternative location in the other partition. In a sense, the new data block “uses its elbows” to make space for conflicting data instead of evicting it. The enlarged replacement set provides better opportunity to find a suitable victim for evicting.

It is imperative to design effective replacement policy to identify suitable victim for evicting in the elbow cache, which featured with enlarged replacement set. Recency-based replacement policy like the *least recently used* (*LRU*) replacement is generally thought to be the most efficient policy for processor cache. However, the traditional LRU replacement policy based on the *most frequently used-most recently used* (*MRU-LRU*) sequence is difficult to implement with multiple hashing functions. Since the number of sets grows exponentially with multiple hashing, it is prohibitively expensive to maintain the needed MRU-LRU sequences.

The *least frequently used* (*LFU*) replacement policy considers the frequency of block references, such that the least frequently used block will be replaced when needed. In fully-associative cache and set-associative cache, the performance of LRU and LFU are mixed. That's because fully-associative cache and set-associative cache with LRU replacement policy suffers the worst cache pollution when a "never-reuse" block is moved into the cache. It takes *c* more misses to replace a never-reused block, where *c* is the set associativity. Such a block can be replaced much faster with LFU replacement once the block has the smallest frequency counter. We propose a low-cost and effective LFU replacement policy for the elbow cache, which has cache performance comparable to the recency-based replacement policy.

1.3 Tolerating Resource Contentions with Runahead on Multithreading Processors

Simultaneous Multithreading (SMT) processors exploit both *instruction-level parallelisms (ILP)* and *thread-level parallelisms (TLP)* by fetching and issuing multiple instructions from multiple threads to the function units of a superscalar architecture each cycle to utilize wide-issue slots [Tullsen et al. 1995; Tullsen et al. 1996]. SMT outperforms previous models of hardware multithreading primarily because it hides short latencies much more effectively, which can often dominate performance on a uniprocessor. For example, neither fine-grain multithreaded architectures [Alverson et al. 1990; Laudon et al. 1994], which context switch every cycle, nor coarse-grain multithreaded architectures [Agarwal et al. 1993b; Saavedra-Barrera et al. 1990], which context switch only on long-latency operations, can hide the latency of a single-cycle integer add if there is not sufficient parallelism in the same thread.

In SMT, multiple threads share resources such as caches, functional units, instruction queue, instruction issue window, and instruction window [Tullsen et al. 1995; Tullsen et al. 1996]. SMT typically benefits from giving threads complete access to all resources every cycle. But contentions of these resources may significantly hamper the performance of individual

threads and hinder the benefit of exploiting more parallelism from multiple threads. First, disruptive cache contentions lead to more cache misses and hurt overall performance. Serious cache contention problems on SMT processors were reported [Tullsen and Brown 2001]. The optimal allocation of cache memory between two competing threads was studied [Stone et al. 1992]. Dynamic partitioning of shared caches among concurrent threads based on "marginal gains" was reported in [Suh et al. 2001]. The results showed that significantly higher hit ratios over the global LRU replacement could be achieved.

Second, threads can hold critical resources while they are not making progress due to long-latency operations and block other threads from making normal execution. For example, if the stalled thread fills the issue window and instruction window with waiting instructions, it shrinks the window available for the other threads to find instructions to issue and bring in new instructions to the pipeline. Thus, when parallelism is most needed when one or more threads are no longer contributing to the instruction flow, fewer resources are available to expose that parallelism. Previously proposed methods [El-Moursy and Albonesi 2003; Cazorla et al. 2004a; Cazorla et al. 2004b] attempt to identify threads that will encounter long-latency operations. The thread with long-latency operation may be delayed to prevent it from occupying more resources. A balance scheme was proposed [Cazorla et al. 2003] to dynamically switch between flushing and keeping long-latency threads to avoid overhead of flushing.

We propose a valuable solution to this problem, runahead execution on SMTs. Runahead execution was first proposed to improve MLP on single-thread processors [Dundas and Mudge 1997; Mutlu et al. 2003]. Effectively, runahead execution can achieve the same performance level as that with a much bigger instruction window. We investigate and evaluate runahead execution on SMT processors with multiple threads running simultaneously. Besides the inherent

advantage of memory prefetching, runahead execution can also prevent a thread with long latency loads from occupying shared resources and impeding other threads from making forward progress. Performance evaluation based on a mixture of SPEC2000 benchmarks demonstrates the performance improvement of runahead execution on SMTs.

1.4 Order-Free Store Queue Using Decoupled Address Matching and Age-Priority Logic

Store-load forwarding is a critical aspect of dynamically scheduled execution. Conventional processors implement store-load forwarding by buffering the addresses and data values of all in-flight stores in an age-ordered SQ. A load accesses the data cache and in parallel associatively searches the SQ for older stores with matching addresses. The load obtains its value from the youngest such store (if any) or from the data cache. Associative structures can be made fast, but often at the cost of substantial additional energy, area, and/or design effort. Furthermore, these implementation disadvantages compound super-linearly especially for ordered associative structures like the SQ as structure size or bandwidth scales up. As SQ access is on the load execution critical path, fully-associative search of a large SQ can result in load latency that is longer than data cache access latency, which in turn complicates scheduling and introduces replay overheads.

There have been many recent proposals to design a scalable SQ by getting rid of the expensive and time-consuming full *content-addressable memory* (*CAM*) design. One category of solution is to adopt a two-level SQ where the small first-level SQ enables fast and energy efficient forwarding and a much larger second-level SQ corrects and complements the first-level SQ [Buyuktosunoglu et al. 2002; Gandhi et al. 2005; Sethumadhavan et al. 2006]. A store-forwarding cache is implemented in [Castro et al. 2006] for store-load forwarding. It relies on a separate memory disambiguation table to resolve any dependence violation. In [Akkary et al.

2003], a much larger L0 cache is used to replace the first-level SQ for caching the latest store data. A load, upon a hit, can fetch the data directly from the L0. In this approach, instead of maintaining speculative load information, the load is also executed in the in-order pipeline fashion. An inconsistent between the data in L0 and L1 can identify memory dependence violations. Besides the complexity and extra space, a fundamental issue in this category of approaches is the heavy mismatch between the latest store and the correct last store for the dependent load. Such mismatches produce costly re-executions.

We introduce a new order-free SQ that decouples the store address matching and its corresponding age-order priority logic from the original Content-Addressable Memory (CAM) SQ where the outstanding store addresses and data are buffered. A separate SQ directory is maintained for searching the stores in the SQ. Unlike a conventional SQ, a single address is recorded in the SQ directory for multiple outstanding stores with the same address. Each entry in the directory is augmented with a new age-order vector to indicate the correct program order of the stores with the same address. The decoupled SQ directory allows stores to enter the directory when they are issued which can be different from their program order. It relies on the age-order vector to recover the correct program order of the stores. The relaxation of the program-order requirement helps to reduce the directory size as well as to abandon the fully-associative CAM-based directory that is the key obstacle for a scalable SQ.

1.5 Benchmarks, Evaluation Methods, and Dissertation Outline

SimpleScalar tool set is used to evaluate performance for the first three works in this dissertation. It consists of compiler, assembler, linker, simulation, and visualization tools for modern processor architecture and provides researchers with an easily extensible, portable, high-performance test bed for systems design. It can simulate an out-of-order issue processor that supports non-blocking caches, speculative execution, and state-of-the-art branch prediction.

Table 1-1. SimpleScalar simulation parameters

Processor	
Fetch/Decode/Issue/Commit Width: 8	
Instruction Fetch Queue: 8	
Branch Predictor: 64K-entry G-share, 4K-entry Branch Target Buffer (BTB)	
Mis-Prediction Penalty: 10 cycles	
Instruction Window/Load-Store Queue size: 512/512	
Instruction Issue Window: 32	
Processor Translation Lookaside Buffer (TLB): 2K-entry, 8-way	
Integer Arithmetic Logic Unit (ALU):	6 ALU (1 cycle); 2 Mult/Div: Mult (3 cycles), Div (20 cycles)
Floating Point (FP) ALU:	4 ALU (2 cycles); 2 Mult/Div/Sqrt: Mult (4 cycles), Div (12 cycles), Sqrt (24 cycles)
Memory System	
Level 1 (L1) Instruction/Data Cache: 64KB, 4-way, 64B Line, 2 cycles	
L1 Data Cache Port: 4 read/write port	
Level 2 (L2) Cache: 1MB, 8-way, 64B Line, 15 cycles	
L1/L2 Memory Status Holding Registers (MSHR): 16/16	
Request/Dynamic Random Access Memory (DRAM)/Data Latency: 80/160/80	
Memory Channel: 4 with line-based interleaved	
Memory Request Window: 32	
Channel/Return Queue: 8/8	

We modified the SimpleScalar simulator to model an 8-wide superscalar, out-of-order processor with Alpha 21264-like pipeline stages [Kessler 1999]. We made two enhancements. First, the original *Issue/Execute* stage is extended to the *Issue*, *Register* read, and *Execute* stages to reflect the delay in instruction scheduling, operands read, and instruction execution. Second, instead of waking up dependent instructions at the *Writeback* stage of the parent instruction, the dependent instructions are pre-issued after the parent instruction is issued and the delay of obtaining the result is known. Important simulation parameters are summarized in Table 1-1.

To evaluate the order-free SQ, we modified the PTLsim simulator [Youst 2007] to model a cycle accurate full system x86-64 microprocessor. We followed the basic PTLsim pipeline design, which has 13 stages (1 fetch, 1 rename, 5 frontend, 1 dispatch, 1 issue, 1 execution, 1

transfer, 1 writeback, 1 commit). Important simulation parameters for PTLSim are summarized in Table 1-2.

Table 1-2. PTLSim simulation parameters

Fetch/Dispatch/Issue/Commit Width: 32/32/16/16
Instruction Fetch Queue: 128
Branch Predictor: 64K-entry G-share, 4K-entry 4-way BTB
Branch Mis-Prediction Penalty: 7 cycles
RUU/LQ/SQ size: 512/128/64
Instruction Issue Window int0/int1/ld/fp: 64/64/64/64
ALU/STU/LDU/FPU: 8/8/8/8
L1 I-Cache: 16KB, 4-way, 64B line, 2 cycles
L1 D-Cache: 32KB 4-way, 64B line, 2 cycles, 8 read/write port
L2 U-Cache: 256KB, 4-way, 64B Line, 6 cycles
L3 U-Cache: 2MB, 16-way, 128B Line, 16 cycles
L1/L2 MSHRs: 16/16
Memory Latency: 100 cycles
I-TLB: 64-entry fully-associative
D-TLB: 64-entry fully-associative

Benchmark programs are used to provide a measure to compare performance. The SPEC2000 [SPEC2000 Benchmarks] from the Standard Performance Evaluation Corporation is one of the most widely used benchmark programs in our research community. It consists of two types of benchmarks, one is the SPECint2000, a set of integer-intensive benchmarks, and the other is the SPECfp2000, a set of floating-point intensive benchmarks. Another benchmark suite we evaluated is the Olden benchmarks [Olden Benchmark], which are pointer-intensive programs built by Princeton University. We follow the studies done in [Sair and Charney 2000] to skip certain instructions, warm up caches and other system components with 100 million instructions, and then collect statistics from the next 500 million instructions.

The outline of this dissertation is as followed. In chapter 2, we first study the missing memory-level parallelism opportunities because of data dependences and then describe P-load scheme. In chapter 3, the severity of cache conflict misses is demonstrated and a cache

organization with a frequency-based replacement policy is introduced to specifically reduce conflict misses. In chapter 4, we will evaluate a technique to solve the resource contention problem in multi-threading environment. In chapter 5, we introduce the order-free SQ that decouples the matching of the store/load address from its corresponding age-based priority encoding logic. The dissertation is concluded in chapter 6.

CHAPTER 2
EXPLOITATION OF MEMORY LEVEL PARALLELISM WITH P-LOAD

2.1 Introduction

Over the past two decades, ILP has been a primary focus of computer architecture research and a variety of microarchitecture techniques to exploit ILP such as pipelining, *very long instruction word* (*VLIW*), superscalar issue, branch prediction, and data speculation have been developed and refined. These techniques make current processors to effectively utilize deep multiple-issue pipelines in applications such as media processing and scientific floating-point intensive applications.

However, the performance of commercial applications such as databases is dominated by the frequency and cost of memory accesses. Typically, they have large instruction and data footprints that do not fit in caches, hence, requiring frequent accesses to memory. Furthermore, these applications exhibit data-dependent irregular patterns in their memory accesses that are not amenable to conventional prefetching schemes. For those memory-bound workloads, a promising alternative is to exploit MLP by overlapping multiple memory accesses.

MLP is the number of outstanding cache misses that can be generated and executed in an overlapped manner. It is essential to exploit MLP by overlapping multiple cache misses in a wide instruction window [Chou et al. 2004]. The exploitation of MLP, however, can be limited due to a load that depends on another load to produce the base address (referred as *load-load dependences*). If the parent load misses the cache, sequential execution of these two loads must be enforced. One typical example is the pointer-chasing problem in many applications with LDS, where accessing the successor node cannot start until the pointer is available, possibly from memory. Similarly, indirect accesses to large array structures may face the same problem when both address and data accesses encounter cache misses.

There have been several prefetching techniques to reduce penalties on consecutive cache misses of tight load-load dependences [Luk and Mowry 1996; Roth et al. 1998; Yang and Lebeck 2000; Vanderwiel and Lilja 2000; Cahoon and McKinley 2001; Collins et al. 2002; Cooksey et al. 2002; Mutlu et al. 2003; Yang and Lebeck 2004; Hughes and Adve 2005]. Luk et al. [1996] proposed using jump-pointers, which were further developed by Roth and Sohi [Roth et al. 1998]. A LDS node is augmented with jump-pointers that point to nodes that will be accessed in multiple iterations or recursive calls in the future. When a LDS node is visited, prefetches are issued for the locations pointed by its jump-pointers. They focused on a software implementation of the four jump-pointer idioms proposed by Roth and Sohi. They also proposed hardware and cooperative hardware/software implementations that use significant additional hardware support at the processor to overcome some of the software scheme's limitations. The hardware automatically creates and updates jump-pointers and generates address for and issues prefetches. The hardware can eliminate the instruction overhead of jump pointers and reduce the steady state stall time for root and chain jumping, but it does not affect the startup stall time for any case and does not eliminate the steady state stall time for root and chain jumping.

The push-pull scheme [Yang and Lebeck 2000; Yang and Lebeck 2004] proposed a prefetch engine at each level of memory hierarchy to handle LDS. A kernel of load instructions, which encompass the LDS traversals, is generated by software. The processor downloads this kernel to the prefetch engine, then executes the load instructions repeatedly traverse the LDS. The lack of address ordering hardware and comparison hardware restricts their scheme's traversals to LDS and excludes some data dependences. The kernels and prefetch engine would require significant changes to allow more general traversals. A similar approach with compiler help has been presented in [Hughes and Adve 2005].

The content-aware data prefetcher [Cooksey et al. 2002] identifies potential pointers by examining word-aligned content of cache-miss data blocks. The identified pointers are used to initiate prefetching of the successor nodes. Using the same mechanism to identify pointer loads, the pointer-cache approach [Collins et al. 2002] builds a correlation history between heap pointers and the addresses of the heap objects they point to. A prefetch is issued when a pointer load misses the data cache, but hits the pointer cache.

We introduce a new approach to overlap cache misses involved in load-load dependences. After dispatch, if the base register of a load is not ready due to an early cache miss load, a special *P-load* is issued in place of the dependent load. The P-load instructs the memory controller to calculate the needed address once the parent load's data is available from the *dynamic random access memory* (*DRAM*). The inherent interconnect delay between processor and memory can thus be overlapped regardless the location of the memory controller [Opteron Processors]. When executing pointer-chasing loads, a sequence of P-loads can be initiated according to the dispatching speed of these loads.

The proposed P-load makes three unique contributions. First of all, in contrast to the existing methods, it does not require any special predictors and/or any software-inserted prefetching hints. Instead, the P-load scheme issues the dependent load early following the instruction stream. Secondly, the P-load exploits more MLP from a larger instruction window without the need to enlarge the critical issue window [Akkary et al. 2003]. Thirdly, an enhanced memory controller with proper processing power is introduced that can share certain computations with the processor.

2.2 Missing Memory Level Parallelism Opportunities

Overlapping cache misses can reduce the performance loss due to long-latency memory operations. However, data dependence between a load and an early instruction may stall the load

from issuing. In this section, we will show the performance loss due to such data dependences in real applications by comparing a baseline model with an idealized MLP exploitation model.

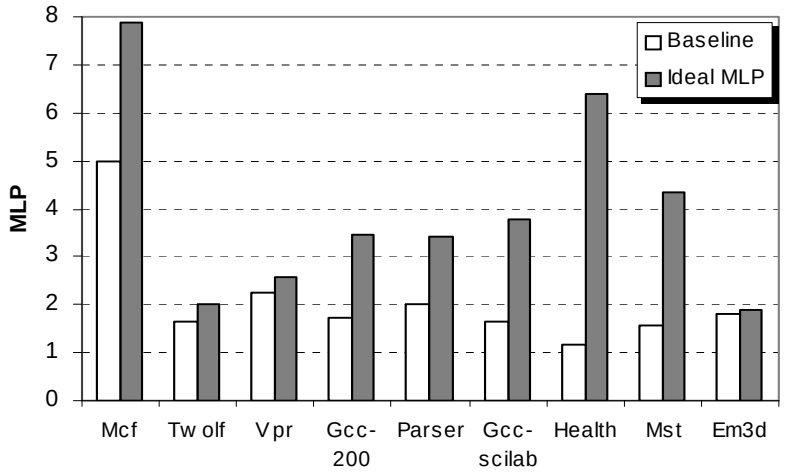


Figure 2-1. Gaps between base and ideal memory level parallelism exploitations.

MLP can be quantified as the average number of memory requests during the period when there is at least one outstanding memory request [Chou et al. 2004]. We compare the MLPs of the baseline model and the ideal model. In the baseline model, all data dependences are strictly enforced. On the contrary, in the ideal model, a cache miss load is issued right after the load is dispatched regardless of whether the base register is ready or not.

Nine workloads, *Mcf, Twolf, Vpr, Gcc-200, Parser,* and *Gcc-scilab* from SPEC2000 integer benchmarks, and *Health*, *Mst,* and *Em3d* from Olden benchmarks are selected for this experiment because of their high L2 miss rates. An Alpha 21264-like processor with 1MB L2 cache is simulated.

Figure 2-1 illustrates the measured MLPs of the baseline and the ideal models. It shows that there are huge gaps between them, especially for *Mcf, Gcc-200, Parser*, *Gcc-scilab*, *Health,* and *Mst*. The results reveal that significant MLP improvement can be achieved if the delay of issuing cache misses due to data dependences is reduced.

2.3 Overlapping Cache Misses with P-loads

We describe the P-load scheme using function r*efresh_potential* from *Mcf* as shown in Figure 2-2. *Refresh_potential* is invoked frequently to refresh a huge tree structure that exceeds 4MB. The tree is initialized with a regular stride pattern among adjacent nodes on the traversal path such that the address pattern can be accurately predicted. However, the tree structure is slightly modified with insertions and deletions between two consecutive visits. After a period of time, the address pattern on the traversal path becomes irregular and is hard to predict accurately. Heavy misses are encountered when caches cannot accommodate the huge working set.

```
Long refresh_potential (network_t *net)
{
    ……
    tmp = node = root->child;
    while (node != root) {
        while (node) {
            if (node->orientation == UP)
                node->potential = node->basic_arc->cost + node->pred->potential;
            else {
                node->potential = node->pred->potential – node->basic_arc->cost;
                checksum++; }
             tmp = node;
             node = node->child;
        }
        node = tmp;
        while (node->pred) {
            tmp = node->sibling;
            if (tmp) {
                node = tmp;
                break; }
            else node = node->pred;
        }
    }
    return checksum;
}
```

Figure 2-2. Example tree-traversal function from *Mcf*

This function traverses a data structure with three traversal links: *child*, *pred*, and *sibling* (highlighted in *italics*), and accesses basic records with a data link, *basic_arc*. In the first inner *while* loop, the execution traverses down the path through the link: *node→child*.

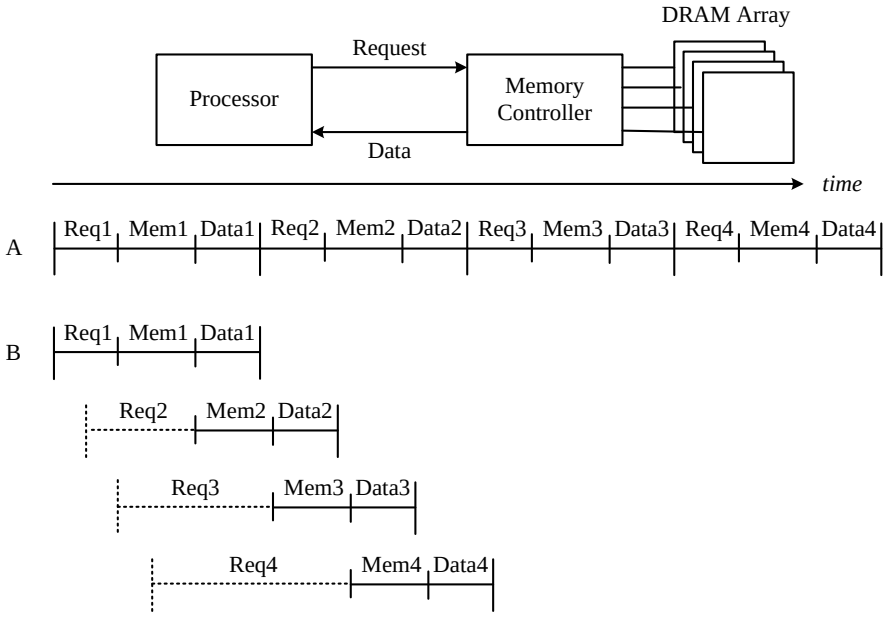


Figure 2-3. Pointer Chasing: A) Sequential accesses; B) Pipeline using P-load

With accurate branch predictions, several iterations of the *while* loop can be initiated in a wide instruction window. The recurrent instruction, *node = node→child* that advances the pointer to the next node, becomes a potential bottleneck since accesses of the records in the next node must wait until the pointer (base address) of the node is available. As shown in Figure 2-3 A), four consecutive *node = node→child* must be executed sequentially. In the case of a cache miss, each of them encounters delays in sending the request, accessing the DRAM array, and receiving the data. These non-overlapped long-latency memory accesses can congest the instruction and issue windows and stall the processor. On the other hand, the proposed P-load can effectively overlap the interconnect delay in sending/receiving data as shown in Figure 2-3 B). In the following subsections, detailed descriptions of identifying and issuing P-loads are

given first, followed by the design of the memory controller. Several issues and enhancements about P-load will also be discussed.

2.3.1 Issuing P-Loads

We will describe P-load issuing and execution within the *instruction window* and the *memory request window* (Figure 2-4) by walking through the first inner *while* loop of *refresh-potential* from *Mcf* (Figure 2-2).

Instruction Window

ID	Instr.	Request		
		Type	ID	Disp
101	lw $v0,28($a0)	Load [28($a0)]	—	—
102	bne $v0,$a3,L1			
103	lw $v0,32($a0)	(partial hit)		
104	lw $v1,8($a0)	(partial hit)		
105	lw $v0,16($v0)	P-load [addr(103)]	105	16
106	lw $v1,44($v1)	P-load [addr(104)]	106	44
107	addu $v0,$v0,$v1			
108	J L2			
109	sw $v0,44($a0)			
110	addu $v0,$0,$a0			
111	lw $a0,12($a0)	(partial hit)		
112	bne $a0,$0,L0			
113	lw $v0,28($a0)	P-load [addr(111)]	113	28
114	lw $v0,32($a0)	P-load [addr(111)]	114	32
115	lw $v1,8($a0)	P-load [addr(111)]	115	8
116	lw $v0,16($v0)	P-load [p-id(114)]	116	16
117	lw $v1,44($v1)	P-load [p-id(115)]	117	44
118	lw $a0,12($a0)	P-load [addr(111)]	118	12

Memory Request Window

ID	Link	Offset	Address	Disp
New	—	—	[28($a0)]	—
105	New	32($a0)		16
106	New	8($a0)		44
113	New	12($a0)		28
114	New	12($a0)		32
115	New	12($a0)		8
116	114	—		16
117	115	—		44
118	—	—	[12($a0)]*	—

* Assume New removed, 118 uses address [12($a0)] to fetch DRAM or P-load Buffer

Note, thick lines divide iterations

Figure 2-4. Example of issuing P-loads seamlessly without load address

Assume the first load, *lw $v0,28($a0)*, is a cache miss and is issued normally. The second and third loads encounter partial hits to the same block as the first load, thus no memory request is issued. After the fourth load, *lw $v0,16($v0)*, is dispatched, a search through the current instruction window finds it depends on the second load, *lw $v0,32($a0)*. Normally, the fourth load must be stalled. In the proposed scheme, however, a special P-load will be inserted into a small *P-load issue window* at this time. When the cache hit/miss of the parent load is known,

associative search for dependent loads in the *P-load issue window* is performed. All dependent P-loads are either ready to be issued (if the parent load is a miss), or canceled (if the parent load is a hit). The P-load consists of the address of the parent load, the displacement, and a unique instruction ID to instruct the memory controller to calculate the address and fetch the correct block. Details of the memory controller will be given in Section 2.3.2. The fifth load is similar to the fourth. The sixth load, *lw $a0,12($a0)*, advances the pointer and is also a partial hit to the first load.

With correct branch prediction, instructions of the second iteration are placed in the instruction window. The first three loads in the second iteration all depend on *lw $a0,12($a0)* in the previous iteration. Three corresponding P-loads of them are issued accordingly with the parent load's address. The fourth and fifth loads, however, depend on early loads that are themselves also identified as P-loads. In this case, instead of the parent's addresses, the parent load IDs (*p-id*), 114 and 115 for the fourth and fifth loads respectively, are encoded in the address fields to instruct the memory controller to obtain correct base addresses. This process continues to issue a sequence of P-loads within the entire instruction window seamlessly.

A P-load does not occupy a separate location in the instruction window, nor does it keep a record in the *memory status holding registers (MSHRs)*. Similar to other memory-side prefetching methods [Solihin et al. 2002], the returned data block of a P-load must come back with address. Upon receiving a P-load returned block from memory, the processor searches and satisfies any existing memory requests located in the MSHRs. The block is then placed into cache if it is not there. Searching in the MSHRs is necessary, since a P-load cannot prevent other requests that target the same block from issuing. The load, from which a P-load was initiated, will be issued normally when the base register is ready.

In general, the P-load can be viewed as an accurate data prefetching method. It should not interfere with normal store-load forwarding. A P-load can be issued even there are unresolved previous stores in the load-store queue. Upon the completion of the parent miss-load, the address of the dependent load can be calculated that will trigger any necessary store-load forwarding.

2.3.2 Memory Controller Design

Figure 2-5 illustrates the basic design of the memory controller. Normal cache misses and P-loads are processed and issued in the *memory request window* similar to the out-of-order execution in processor's instruction window. The memory *address*, the *offset* for the base address, the *displacement* for computing the target block address, and the dependence *link*, are recorded for each request in arriving order. For a normal cache miss, its address and a unique ID assigned by the *request sequencer* are recorded. Such cache miss requests will access the DRAM without delay as soon as the target DRAM channel is open. A normal cache miss may be merged with an early active P-load that targets the same block to achieve reduced penalties.

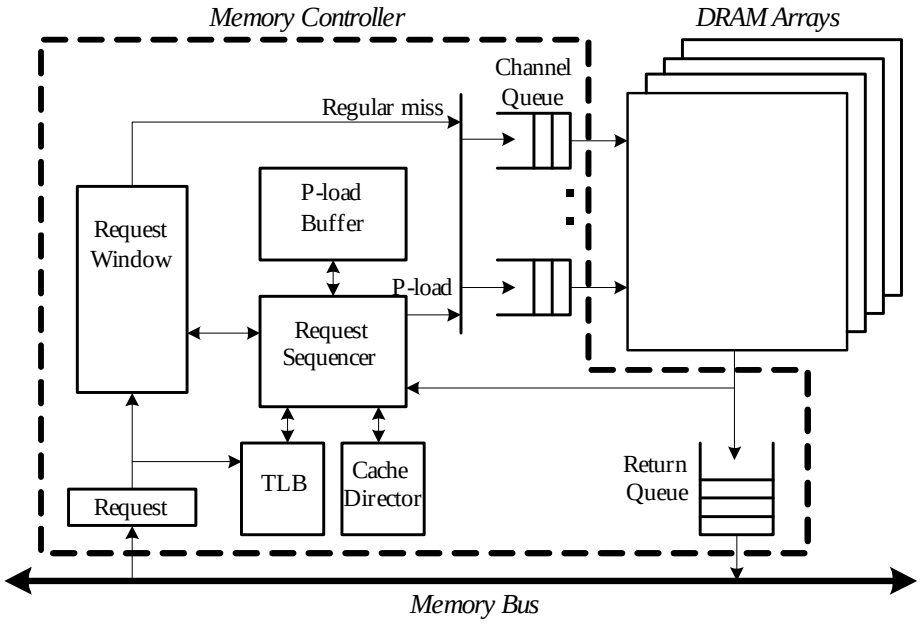


Figure 2-5. Basic design of the memory controller

Two different procedures are applied when a P-load arrives. Firstly, if a P-load comes with valid address, the block address is used to search for any existing memory requests. Upon a match, a dependence link is established between them; the *offset* within the block is used to access the correct word from the parent's data block without the need to access the DRAM. In the case of no match, the address that comes with the P-load is used to access the DRAM as illustrated by request 118 assuming that the first request has been removed from the memory request window. Secondly, if a P-load comes without a valid address, the dependence link encoded in the address field is extracted and saved in the corresponding entry as shown by requests 116 and 117 (Figure 2-4). In this case, the correct base addresses can be obtained from 116's and 117's parent requests, 114 and 115, respectively. The P-load is dropped if its parent P-load is no longer in the memory request window.

Once a data block is fetched, all dependent P-loads will be woken up. For example, the availability of the *New* block will trigger P-loads 105, 106, 113, 114, and 115 as shown in Figure 2-4. The target word in the block can be retrieved and forwarded to the dependent P-loads. The memory address of the dependent P-load is then calculated by adding the target word (base address) with the displacement value. The P-load's block is fetched if its address does not match any early active P-load. The fetched P-load's block in turn triggers its dependent P-loads. A memory request will be removed from the memory request window after its data block is sent back.

2.3.3 Issues and Enhancements

There are many essential issues that need to be resolved to implement the P-load scheme efficiently.

Maintaining Base Register Identity: The base register of a qualified P-load may experience renaming or constant increment/decrement after the parent load is dispatched. These indirect

dependences can be identified and established by proper adjustment to the displacement value of the P-load. There are different implementation options. In our simulation model, we used a separate register renaming table to provide association of the current dispatched load with the parent load, if exist. This direct association can be established whenever a “simple” register update instruction is encountered and its parent (could be multiple levels) is a miss load. The association is dropped when the register is modified again.

Address Translation at Memory Controller: The memory controller must perform virtual to physical address translation for a P-load in order to access the physical memory. A shadowed TLB needs to be maintained at the memory controller for this purpose (Figure 2-5). The processor issues a TLB update to the memory controller whenever a TLB miss occurs and the new address translation is available. The TLB consistency can be handled similarly to that in a multiprocessor environment. A P-load is simply dropped upon a TLB miss.

Reducing Excessive Memory Requests: Since a P-load is issued without memory address, it may generate unnecessary memory traffic if the target block is already in cache or multiple requests address the same data block. Three approaches are considered here. Firstly, when a normal cache miss request arrives, all outstanding P-loads are searched. In the case of a match, the P-load is changed to a normal cache miss for saving variable delays. Secondly, a small P-load buffer (Figure 2-5) buffers the data blocks of recent P-loads and normal cache miss requests. A fast access to the buffer occurs when the requested block is located in the buffer. Thirdly, a topologically equivalent cache directory of the lowest level cache is maintained to predict cache hit/miss for filtering the returned blocks. By capturing normal cache misses, P-loads, and dirty block writebacks, the memory-side cache directory can predict cache hits accurately.

Inconsistent Data Blocks between Caches and Memory: Similar to other memory-side prefetching techniques, the P-load scheme fetches data blocks without knowing whether they are already located in cache. It is possible to fetch a stale copy if the block has been modified. In general, the stale copy is likely to be dropped either by cache-hit prediction or by searching through the directory before updating the cache. However, in a rather rare case when a modified block is written back to the memory, this modified block must be detected against outstanding P-loads to avoid fetching the stale data.

Complexity, Overhead, and Need for Associative Search Structure: There are two new structures: P-load issue window and memory request window (with 8 and 32 entries in our simulations) that require associative searches. Others do not require expensive associative searches. We carefully model the delays and access conflicts. For instance, although multiple P-loads can be waked up simultaneously, it takes two memory controller cycles (10 processor cycles) conservatively to initiate each DRAM access sequentially. The delay is charged due to the associative wakeup as well as the need for TLB and directory accesses. Our current simulation does not consider TLB shootdown overhead. Our results showed that it has ignorable impact due to small TLB misses and the flexibility of dropping overflow P-loads during TLB updates.

2.4 Performance Evaluation

To handle P-loads, the processor includes an 8-entry P-load issue window along with a 512-entry instruction window and a 32-entry issue window. Several new components are added to the memory controller. A 32-entry memory request window with a 16-entry fully associative P-load buffer is added to process both normal cache misses and P-loads. An 8-way 16K-entry cache directory of the second level cache to predict cache hit/miss is simulated. A shadowed

TLB with the same configuration as processor side TLB is simulated for address translation on the memory controller.

Nine workloads, *Mcf, Twolf, Vpr, Gcc-200, Parser,* and *Gcc-scilab* from SPEC2000 integer benchmarks, and *Health, Mst,* and *Em3d* from Olden benchmarks are selected because of high L2 miss rates as ordered according to their appearances.

A processor-side *stride* prefetcher is included in all simulated models [Fu et al. 1992]. To demonstrate the performance advantage of the P-load scheme, the historyless *content-aware* data prefetcher [Cooksey et al. 2002] is also simulated. We search exhaustively to determine the *width* (number of adjacent blocks) and the *depth* (level of prefetching) of the prefetcher for best performance improvement. Two configurations are selected. In the limited option (*Content-limit; width=1, depth=1*), a single block is prefetched for each identified pointer from a missed data block, i.e. both width and depth are equal to 1. In the best-performance option (*Content-best; width=3, depth=4*), three adjacent blocks starting from the target block of each identified pointer are fetched. The prefetched block initiates content-aware prefetching up to the fourth level. Other prefetchers are excluded due to the need of huge history information and/or software prefetching help.

2.4.1 Instructions Per Cycle Comparison

Figure 2-6 summarizes the *Instructions Per Cycle* (*IPC*) and the normalized memory access time for the *baseline* model, the content-aware prefetching (*Content-limit* and *Content-best*) and the P-load schemes without (*Pload-no*) and with (*Pload-16*) a 16-entry P-load buffer. Generally, the P-load scheme shows better performance.

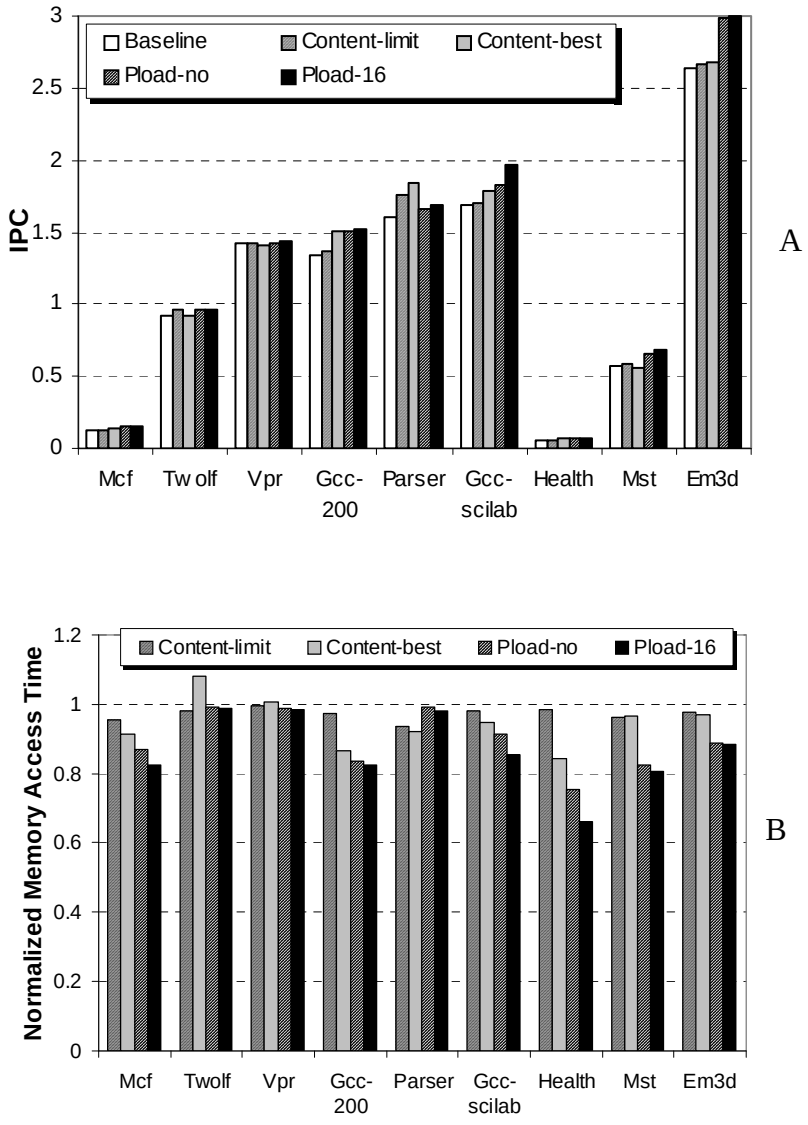


Figure 2-6. Performance comparisons: A) Instructions Per Cycle; B) Normalized memory access time

Compared with the *baseline* model, the *Pload-16* shows speedups of 28%, 5%, 2%, 14%, 5%, 17%, 39%, 18% and 14% for the respective workloads. In comparison with the *Content-best*, the *Pload-16* performs better by 11%, 4%, 2%, 2%, -8%, 11%, 16%, 22%, and 12%. The P-load is most effective on the workloads that traverse linked data structures with tight load-load dependences such as *Mcf, Gcc-200, Gcc-scilab, Health, Mst*, and *Em3d*. The content-aware scheme, on the other hand, can prefetch more load-load dependent blocks beyond the instruction

window. For example, the traversal lists in *Parser* are very short, and thus provide limited room for issuing P-loads. But the *Content-best* shows better improvement on *Parser*. Lastly, the results show that a 16-entry P-load buffer provides about 1-10% performance improvement with an average of 4%.

To further understand the P-load effect, we compare the memory access time of various schemes normalized to the memory access time without prefetching (Figure 2-6 B). The improvement of the memory access time matches the IPC improvement very well. In general, the P-load reduces the memory access delay significantly. We observe 10-30% reduction of memory access delay for *Mcf, Gcc-200, Gcc-scilab, Health, Mst,* and *Em3d.*

2.4.2 Miss Coverage and Extra Traffic

In Figure 2-7, the miss coverage and total traffic are plotted. The total traffic is classified into five categories: misses, partial hits, miss reductions (i.e. successful P-load or prefetches), extra prefetches, and wasted prefetches. The sum of the misses, partial hits and miss reductions is equal to the baseline misses without prefetching, which is normalized to 1. The partial hits represent normal misses that catch early P-loads or prefetches at the memory controller, so that the memory access delays are reduced. The extra prefetch represents the prefetched blocks that are replaced before any use. The wasted prefetches are referred to the prefetched blocks that are presented in cache already.

Except for *Twolf* and *Vpr*, the P-load reduces 20-80% overall misses. These miss reductions are accomplished with little extra data traffic because the P-load is issued according to the instruction stream. Among the workloads, *Health* has the highest miss reduction. It simulates health-care systems using a 4-way B-tree structure. Each node in the B-tree consists of a link-list with patient records. At the memory controller, each pointer-advance P-load usually wakes up a large number of dependent P-loads ready to access DRAM. At the processor side, the return of a

parent load normally triggers dependent loads after their respective blocks are available from early P-loads. *Mcf*, on the other hand, has much simpler operations on each node visit. The return of a parent load may initiate the dependent loads before the blocks are ready from early P-loads. Therefore, about 20% of the misses have reduced penalties due to the early P-loads. *Twolf* and *Vpr* show insignificant miss reductions because of very small amount of tight load-load dependences.

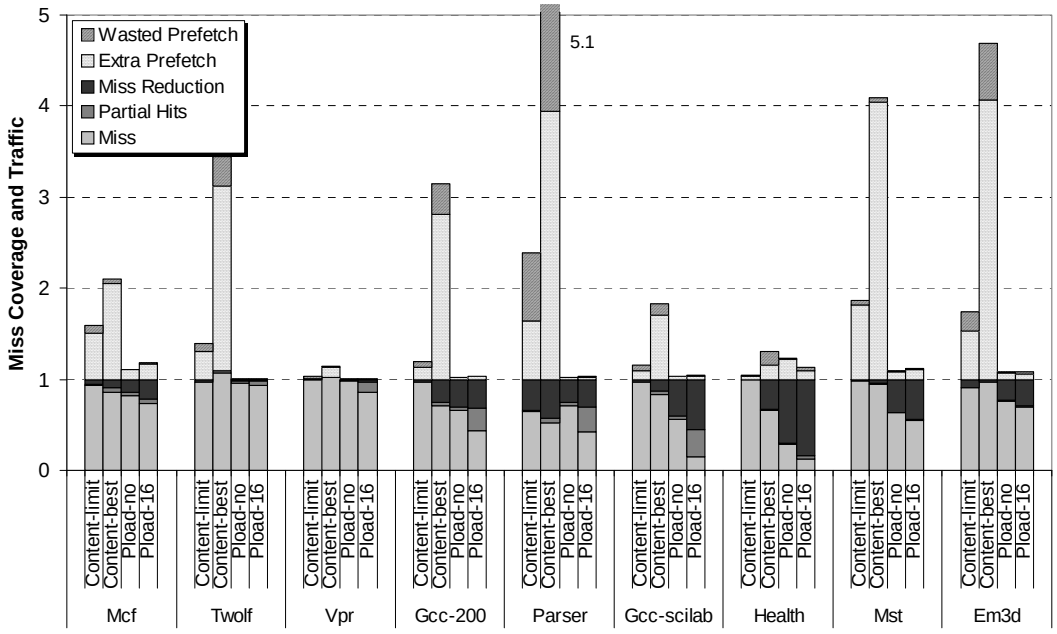


Figure 2-7. Miss coverage and extra traffic

The content-aware prefetcher generates a large amount of extra traffic for aggressive data prefetching. For *Twolf* and *Vpr*, such aggressive and incorrect prefetching actually increases the overall misses due to cache pollution. For *Parser*, the *Content-best* out-performs the *Pload-16* that is accomplished with 5 times memory traffic. In many workloads, the *Contest-best* generates high percentages of wasted prefetches. For example for *Parser*, the cache prediction at the memory controller is very accurate with only 0.6% false-negative prediction (predicted hit,

actual miss) and 3.2% false-positive prediction (predicted miss, actual hit). However, the total predicted misses are only 10%, which makes 30% of the return P-load blocks wasted.

2.4.3 Large Window and Runahead

The scope of the MLP exploitation with P-load is confined within the instruction window. In Figure 2-8, the IPC speedups of the P-load with five window sizes: 128, 256, 384, 512 and 640 in comparison with the baseline model of the same window size are plotted.

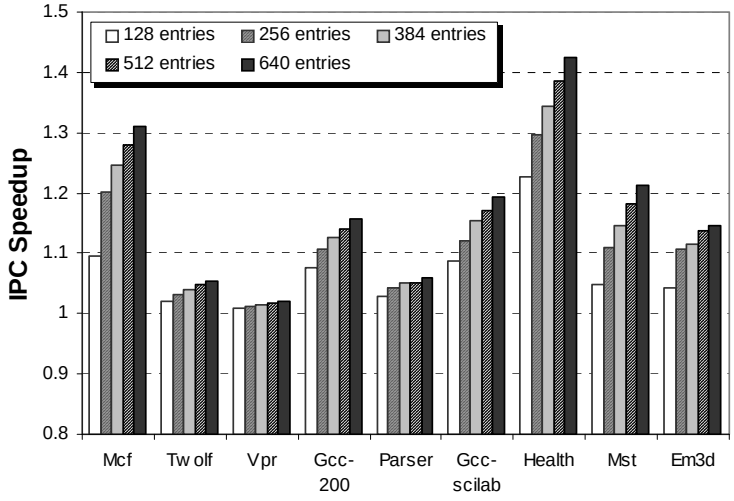


Figure 2-8. Sensitivity of P-load with respect to instruction window size

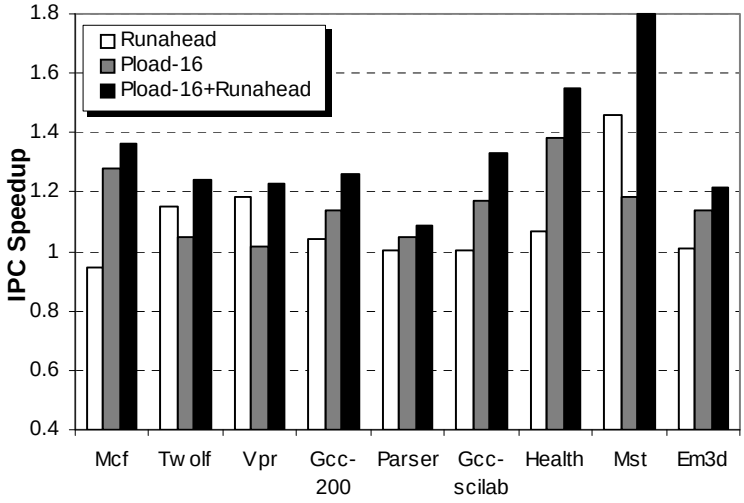


Figure 2-9. Performance impact from combining P-load with runahead

The advantage of larger window is obvious, since the bigger the instruction window, the more the P-loads can be discovered and issued. It is important to point out that issuing P-loads is independent of the issue window size. In our simulation, the issue window size remains 32 for all five instruction windows.

The speculative runahead execution effectively enlarges the instruction window by removing cache miss instructions from the top of the instruction window. More instructions and potential P-loads can thus be processed on the runahead path. Figure 2-9 shows the IPC speedups of *runahead*, *pload-16*, and the combined *pload-16 + Runahead*. All three schemes use a 512-entry instruction window and a 32-entry issue window. *Runahead* execution is very effective on *Twolf, Vpr*, and *Mst*. It out-performs *Pload-16* due to the ability to enlarge both the instruction and the issue windows. On the other hand, *Mcf, Gcc-200, Gcc-scilab, Health*, and *em3d* show little benefits from *runahead* because of intensive load-load dependences. The performance of *Mcf* is actually degraded because of the overhead associated with canceling instructions on the runahead path.

The benefit of issuing P-loads on the runahead path is very significant for all workloads as shown in the figure. Basically, these two schemes are complementary to each other and show an additive speedup benefit. The average IPC speedups of *runahead, P-load*, and *P-load+runahead* relative to the baseline model are 10%, 16% and 34% respectively. Combining P-load with runahead provides on average of 22% speedup over using only runahead execution, and 16% average speedup over using P-load alone.

2.4.4 Interconnect Delay

To reduce memory latency, a recent trend is to integrate the memory controller into the processor die with reduced interconnect delay [Opteron Processors]. However, in a multiple processor-die system, significant interconnect delay is still encountered in accessing another

memory controller located off-die. In Figure 2-10, the IPC speedups of the P-load with different interconnect delays relative to the baseline model with the same interconnect delay are plotted. The delay indeed impacts the overall IPC significantly. But the P-load still demonstrates performance improvement even with fast interconnect. The average IPC improvements of the nine workloads are 18%, 16%, 12%, 8% and 5% with 100-, 80-, 60-, 40-, and 20-cycle one-way delays respectively.

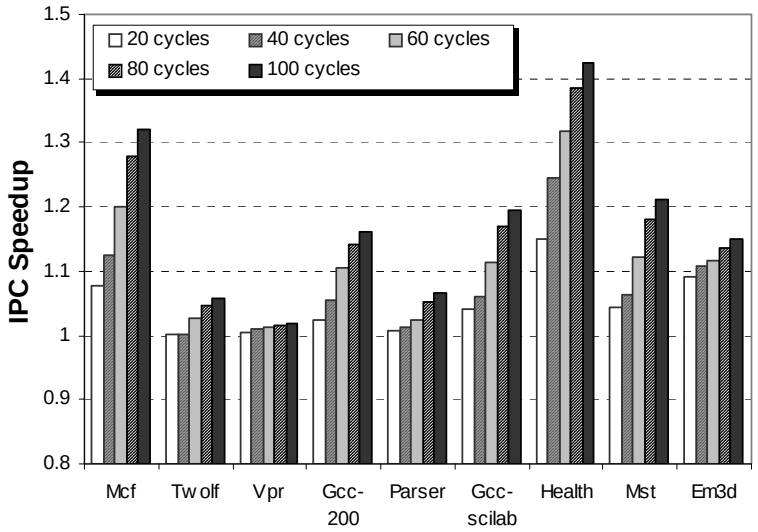


Figure 2-10. Sensitivity of P-load with respect to interconnect delay

2.4.5 Memory Request Window and P-load Buffer

Recall that the memory request window records normal cache misses and P-loads. The size of this window determines the total number of outstanding memory requests can be handled on the memory controller. The issuing and execution of requests in the memory request window are similar to the out-of-order execution in processor's instruction window. In Figure 2-11, the IPC speedups of the P-load with four memory request window sizes: 16, 32, 64, and 128 relative to the baseline model without P-load are plotted. A 32-entry window size is enough to hold almost all of the requests at the memory controller for all workloads except health.

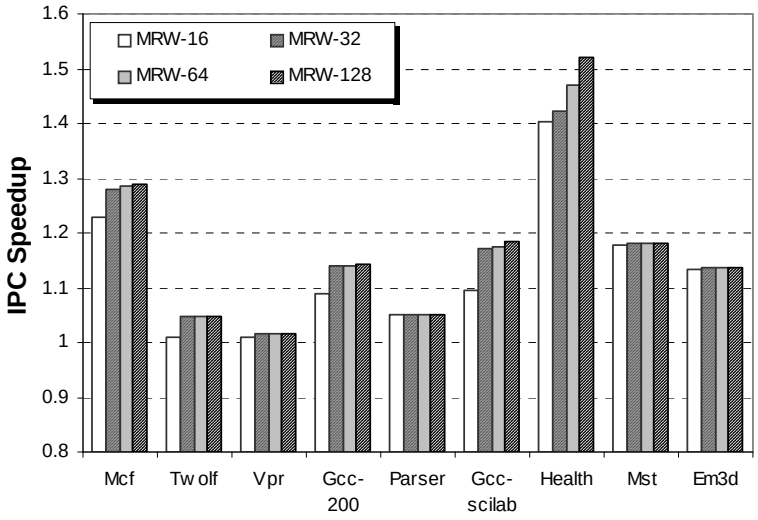


Figure 2-11. Sensitivity of P-load with respect to memory request window size

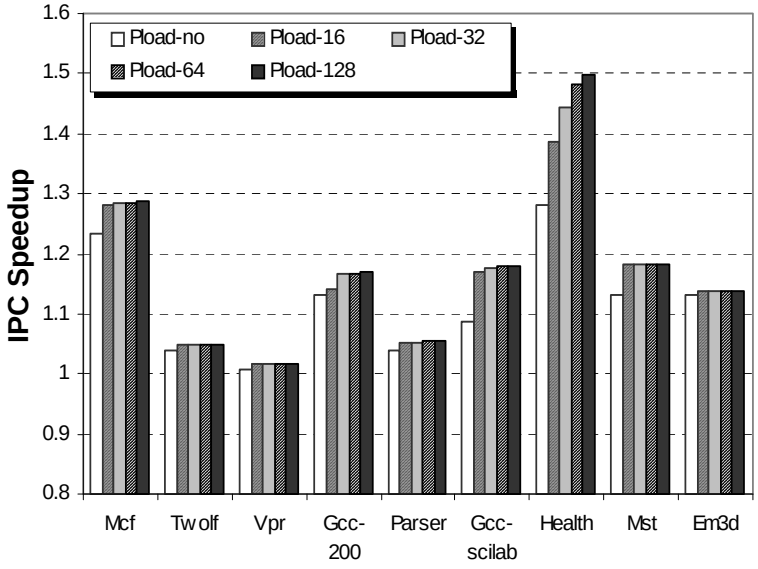


Figure 2-12. Sensitivity of P-load with respect to P-load buffer size

The performance impacts of the P-load buffer with 0, 16, 32, 64 and 128 entries are simulated. Figure 2-12 shows the IPC speedups of the five P-load buffer sizes relative to the baseline model. In all of the workloads, adding the P-load buffer increases the performance gain. For most of the workloads, a 16-entry buffer can capture the majority of the benefit.

2.5 Related Work

There have been many software and hardware oriented prefetching proposals for alleviating performance penalties on cache misses [Jouppi 1990; Chen and Baer 1992; Luk and Mowry 1996; Joseph and Grunwald 1997; Yang and Lebeck 2000; Vanderwiel and Lilja 2000; Cahoon and McKinley 2001; Solihin et al. 2002; Cooksey et al. 2002; Collins et al. 2002; Wang et al. 2003; Yang and Lebeck 2004; Hughes and Adve 2005]. Traditional hardware-oriented sequential or stride-based prefetchers work well for applications with regular memory access patterns [Chen and Baer 1992; Jouppi 1990]. However, in many modern applications and runtime environments, dynamic memory allocations and linked data structure accesses are very common. It is difficult to accurately prefetch due to their irregular address patterns. Correlated and Markov prefetchers [Charney and Reeves 1995; Joseph and Grunwald 1997] record patterns of miss addresses and use the past miss correlations to predict future cache misses. These approaches require a huge history table to record the past miss correlations. Besides, these prefetchers also face challenges in providing accurate and timely prefetches.

A memory-side correlation-based prefetcher moves the prefetcher to the memory controller [Solihin et al. 2002]. To handle timely prefetches, a chain of prefetches based on a pair-wise correlation history can be pushed from memory. Accuracy and memory traffic, however, remain difficult issues. To overlap load-load dependent misses, a cooperative hardware-software approach called push-pull uses a hardware prefetch engine to execute software-inserted pointer-based instructions ahead of the actual computation to supply the needed data [Yang and Lebeck 2000; Yang and Lebeck 2004]. A similar approach has been presented in [Hughes and Adve 2005].

A stateless, content-aware data prefetcher identifies potential pointers by examining word-based content of a missed data block and eliminates the need to maintain a huge miss history

[Cooksey et al. 2002]. After the prefetching of the target memory block by a hardware-identified pointer, a match of the block address with the content of the block can recognize any other pointers in the block. The newly identified pointer can trigger a chain of prefetches. However, to overlap long latency in sending the request and receiving the pointer data for a chain of dependent load-loads, the stateless prefetcher needs to be implemented at the memory side. Both virtual and physical addresses are required in order to identify pointers in a block. Furthermore, by prefetching all identified pointers continuously, the accuracy issue still exists. Using the same mechanism to identify pointer loads, the pointer-cache approach [Collins et al. 2002] builds a correlation history between heap pointers and the addresses of the heap objects they point to. A prefetch is issued when a pointer load misses the data cache, but hits the pointer cache. Additional complications occur when the pointer values are updated.

The proposed P-load abandons the traditional approach of predicting prefetches with huge miss histories. It also gives up the idea of using hardware and/or software to discover special pointer instructions. With deep instruction windows in future out-of-order processors, the proposed approach identifies existing load-load dependences in the instruction stream that may delay the dependent loads. By issuing a P-load in place of the dependent load, any pointer-chasing or indirect addressing that causes serialized memory access, can be overlapped to effectively exploit memory-level parallelism. The execution-driven P-load can precisely preload the needed block without involving any prediction.

2.6 Conclusion

Processor performance is significantly hampered by limited MLP exploitation due to the serialization of loads that are dependent on one another and miss the cache. The proposed special P-load has demonstrated its ability to effectively overlap these loads. Instead of relying on miss predictions of the requested blocks, the execution-driven P-load precisely instructs the memory

controller in fetching the needed data block non-speculatively. The simulation results demonstrate high accuracy and significant speedups using the P-load. The proposed P-load scheme can be integrated with other aggressive MLP exploitation methods for even greater performance benefit.

CHAPTER 3
LEAST FREQUENTLY USED REPLACEMENT POLICY IN ELBOW CACHE

3.1 Introduction

In cache designs, a *set* includes a number of cache frames that a memory block can be mapped into. When all of the frames in a set are occupied, a newly missed block replaces an old block according to the principle of memory reference locality. In classical set-associative caches, both the *lookup* for identifying cache hit/miss and the *replacement* are within the same set, normally based on hashing of a few index bits from the block address. For fast cache access time, the set size (also referred as *associativity*) is usually small. In addition, all of the sets have identical size and are disjoint to simplify the cache design. Under these constraints, heavy conflicts may occur in a few sets (referred as *hot sets*) due to uneven distribution of memory addresses across the entire cache sets that cause severe performance degradation.

There have been several efforts to alleviate conflicts in heavily accessed sets. The hash-rehash cache [Agarwal et al. 1988] and the column-associative cache [Agarwal and Pudar 1993a] establish a secondary set for each block using a different hashing function from the primary set. Cache replacement is extended across both sets to reduce conflicts. An additional cache lookup is required for blocks that are not located in the primary set. The group-associative cache [Peir et al. 1998] maintains a separate cache directory for more flexible secondary set. A different hashing function is used to lookup blocks in the secondary set. Similar to the hash-rehash, lookups in both of the directories are necessary. In addition, a link is added for each entry of the secondary directory to locate the data block. Recently, the V-way cache [Qureshi et al. 2005] eliminates multiple lookups by doubling the cache directory size with respect to the actual number of data blocks. In the V-way cache, any unused directory entry in the lookup set can be used to record a newly missed block without replacing an existing block in the set. The existence

of unused entries allows searching for a replacement block across the entire cache, and thus decouples the replacement set from the lookup set. Although flexible, the V-way cache requires a bi-directional link between each directory entry and its corresponding data block in the data array. Data accesses must go through the link indirectly that lengthens the cache access time. Also, even with extra directory space, the V-way cache cannot eliminate the hot sets and must replace a block within the lookup set if all directory frames in the set are occupied.

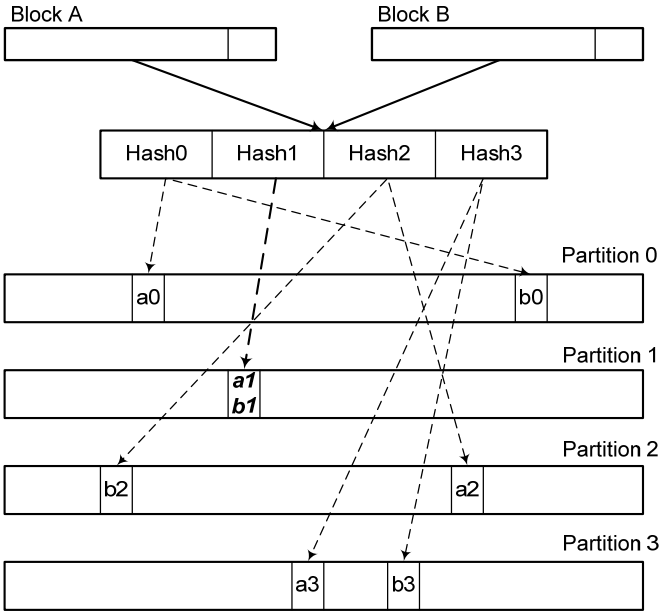


Figure 3-1. Connected cache sets with multiple hashing functions

The skewed-associative cache [Seznec1993a; Seznec and Bodin 1993b; Bodin and Seznec 1997] is another cache organization to alleviate conflict misses. In contrast to the conventional set-associative caches, the skewed-associative cache employs multiple hashing functions for members in a set. In an *n*-way cache, the cache is partitioned equally into *n* banks. For set-associative caches, each set consists of one frame from each partition in the same position addressed by the index bits. In caches with multiple-hashing, each set also consists of one frame from each of the *n* cache partitions. But the location of the frame in each partition can be different based on a different hashing function. To lookup a cache block, the *n* independent

hashing functions address the *n* frames where the *n* existing cache blocks can be matched against the requested block to determine a cache hit or a miss. The insight behind the skewed-associative cache is that whenever two blocks conflict for a single location in partition *i*, they have low probability to conflict for a location in partition *j*.

The elbow cache [Spjuth et al. 2005], an extension to the skewed-associative cache, can expand the replacement set without any extra cache tag directory. In set-associative caches, two blocks are either mapped into the same set, or they belong to two disjoint sets. In contrast, the multiple-hashing cache presents an interesting property that two blocks can be mapped into two sets which share a common frame in one or more partitions. Let us assume a 4-way partitioned cache as illustrated in Figure 3-1, through four hashing functions, blocks A and B can be mapped to different locations in the four cache partitions. In this example, A and B share the same frame, *a1/b1* in Partition 1, but are disjoint in the others. When two sets share the same frame in one or more cache partitions, the two sets are *connected*. The common frame provides a *link* to expand the *replacement set* beyond the original *lookup set*. Instead of replacing a block from the original lookup set, the new block can take over the shared frame, then the block located in the shared frame can be moved to and replace a block in the connected set. For example, assume that when block A is requested, A is not present in any of the four allocated frames, *a0, a1, a2,* and *a3* and *a1* is occupied by block B. Instead of forcing out a block in *a0*, *a1*, *a2*, or *a3*, block A can take over frame *a1*, and push block B to other frames *b0, b2,* or *b3* in the connected set. It is essential that relocating block B does not change the lookup mechanism. Furthermore, instead of replacing blocks in *b0, b2,* or *b3,* the recursive interconnection allows those blocks to be moved to the other frames in their own connected sets. The elbow cache extends skewed-associative cache organization by carefully selecting its victim and, in the case of a conflict, move the

conflicting cache block to its alternative location in the other partition. In a sense, the new data block “uses its elbows” to make space for conflicting data instead of evicting it. The enlarged replacement set provides better opportunity to find a suitable victim for evicting.

It is imperative to design effective replacement policy to identify suitable victim for evicting in the elbow cache, which featured with enlarged replacement set. Recency-based replacement policy like the LRU replacement is generally thought to be the most efficient policy for processor cache, but it can be expensive to be implemented in the elbow cache. In this dissertation, we introduce a frequency-based replacement cache replacement policy based on the concept that the least frequently used blocked is more suitable for replacement.

3.2 Conflict Misses

The severity of set conflicts is demonstrated using SPEC2000 benchmarks. Twelve applications: *Twolf, Bzip, Gzip, Parser, Equake, Vpr, Gcc, Vortex, Perlbmk, Crafty, Apsi* and *Eon* were chosen for our study because of their high conflict misses. Their appearance from the left to the right shows the severity of conflicts from the least to the most. In this study, we simulate a 32KB L1 data cache with 64-byte block size. The severity of conflicts is measured by the miss ratio reductions from a fully-associative to a 4-way set-associative design. Figure 3-2 shows cache miss ratios of 2-way, 4-way, 16-way, and fully-associative caches.

As expected, both 2-way and 4-way set-associative caches suffer significant conflict misses for all selected workloads. It is interesting to see that even with a 16-way set-associative cache, *Bzip, Gzip, Gcc, Perlbmk, crafty,* and *Apsi* still suffer significant performance degradation due to conflicts. For *Apsi*, more than 60% of the misses can be saved using a fully associative cache comparing to that of a 16-way cache.

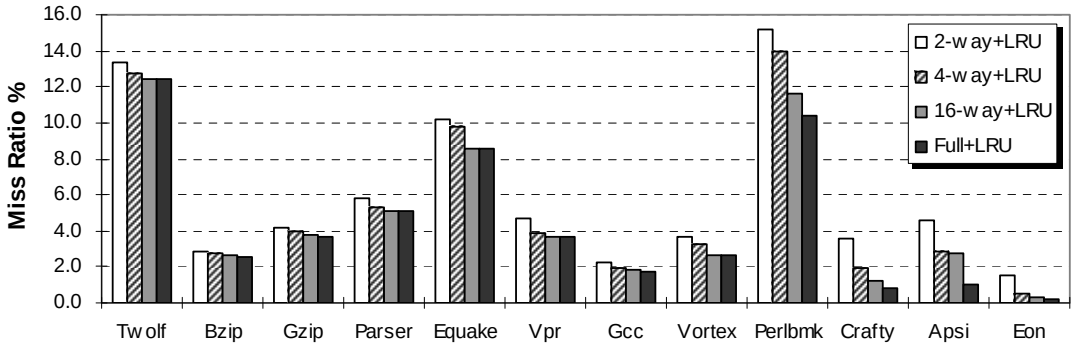


Figure 3-2. Cache miss ratios with different degrees of associativity

3.3 Cache Replacement Policies for Elbow Cache

The fundamental idea behind the elbow cache is to think of an *n*-way partitioned cache as an *n*-dimensional Cartesian coordinate system. The coordinates of a cache block are the components of a tuple of nature numbers $(index_0, index_1, ..., index_{n-1})$, which are generated by applying the corresponding hashing function on the cache block address for each partition. Since the theme of this dissertation is not to invent new hashing functions, the multiple hashing functions described in [Seznec1993a; Seznec and Bodin 1993b; Bodin and Seznec 1997] are borrowed. The definitions of these skewed mapping functions are given as follows.

In the original skewed-associative cache [Seznec1993a; Seznec and Bodin 1993b], two functions *H*, *G* are defined, where *G* is the inverse function of *H* and *n* is the width of the index bits:

$$H: \ \{0,...,2^n-1\} \rightarrow \{0,...,2^n-1\}$$
$$(y_n, y_{n-1}, ..., y_1) \rightarrow (y_n \oplus y_1, y_n, y_{n-1}, ..., y_3, y_2)$$
$$G: \ \{0,...,2^n-1\} \rightarrow \{0,...,2^n-1\}$$
$$(y_n, y_{n-1}, ..., y_1) \rightarrow (y_{n-1}, ..., y_1, y_n \oplus y_{n-1})$$

For a 4-way partitioned cache, four hashing functions are defined (referred as *Mapping Function 93*). A data block at memory address $A = A_3 2^{c+2n} + A_2 2^{c+n} + A_1 2^c + A_0$, where *c* is the width of the offset, is mapped to:

1. cache frame $f_0(A) = H(A_1) \oplus G(A_2) \oplus A_2$ in cache partition 0,

   where $\oplus$ represents an *exclusive-or* operator;

2. cache frame $f_1(A) = H(A_1) \oplus G(A_2) \oplus A_1$ in cache partition 1;

3. cache frame $f_2(A) = G(A_1) \oplus H(A_2) \oplus A_2$ in cache partition 2;

4. cache frame $f_3(A) = G(A_1) \oplus H(A_2) \oplus A_1$ in cache partition 3.

In an alternative skewed function family reported later [Bodin and Seznec 1997] (referred as *Mapping Function 97*), let σ be the one-position circular shift on *n* bits [Stone 1971], a data block at memory address $A = A_3 2^{c+2n} + A_2 2^{c+n} + A_1 2^c + A_0$ can be mapped to:

1. cache frame $f_0(A) = A_1 \oplus A_2$ in cache partition 0;

2. cache frame $f_1(A) = \sigma(A_1) \oplus A_2$ in cache partition 1;

3. cache frame $f_2(A) = \sigma^2(A_1) \oplus A_2$ in cache partition 2;

4. cache frame $f_3(A) = \sigma^3(A_1) \oplus A_2$ in cache partition 3.

3.3.1 Scope of Replacement

To expand the replacement set beyond the lookup set boundary, the coordinates of the blocks in the lookup set provide links for reaching other connected sets. Each block has *n* coordinates and can thus reach *n-1* new frames in the other partitions as long as those frames have not been reached before. The coordinates of each block in the connected sets can in turn link to other connected sets recursively until all of the new sets have been reached. We call the union of all connected sets as the *scope* for a replacement set.

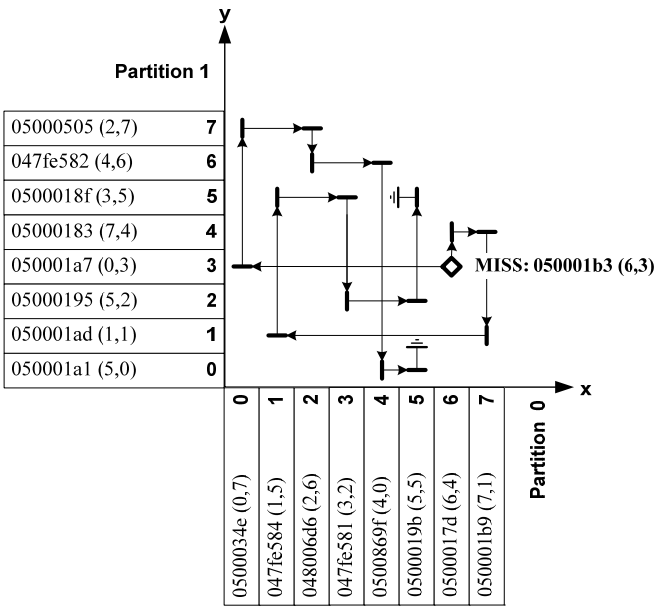


Figure 3-3. Example of search for replacement

In Figure 3-3, we use a simple example of a 2-way partitioned cache to illustrate that the replacement scope can cover the whole cache with the elbow cache mechanism. The cache has 8 frames in each partition, where *x* and *y* coordinates represent partition *0* and partition *1* respectively. This snapshot was taken from *Vortex* of SPEC2000 running on the 2-way partitioned cache. All frames are occupied as indicated by the corresponding hexadecimal block addresses. The two integer numbers in the parenthesis next to the block address represent the coordinate values of each block obtained from the two hashing functions. When a request *050001b3 (6,3)* arrives, a miss occurs since the block is not located in the lookup set of frame *6* on coordinate *x* and frame *3* on coordinate *y*. The search for a replacement begins from the lookup set. Block *0500017d (6,4)* located in frame *6* on coordinate *x* connects to the frame *4* on coordinate *y*. Similarly, block *050001a7 (0,3)* located in frame *3* on coordinate *y* connects to frame *0* on coordinate *x*. The blocks located in the connected frames: *05000183 (7,4)* and *050034e (0,7)* can make a further connection to *050001b9 (7,1)* and *05000505 (2,7)* respectively. The search continues until block *0500019b (5,5)* in frame *5* on coordinate *x*, and *0500018f (3,5)*

in frame *5* on coordinate *y* are revisited as illustrated by the arrows in the figure. In this example, the replacement scope covers the entire cache frames.

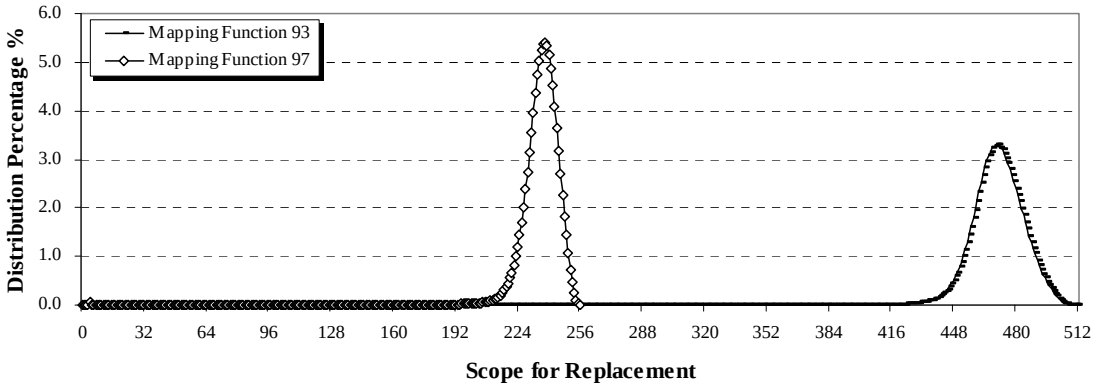


Figure 3-4. Distribution of scopes using two sets of hashing functions

Although the interconnections of multiple sets for replacement is recursive, the scope for each requested block can be limited once all newly expanded frames have already been reached. With the selected integer and floating-point applications from SPEC2000, we can demonstrate the scope of replacement. Figure 3-4 shows the accumulated scope distributions for the selected applications using the two skewed mapping function families described before. It is interesting to observe that the scope of the elbow cache using *Mapping Function 93* almost covers the entire cache. But when using *Mapping Function 97*, the scope is limited within half of the cache frames. This is due to the fact of certain constraints imposed on the selected randomization functions. Further discussions on the mathematical property of these hashing functions are out of the scope of this dissertation. It is important to emphasize that for all practical purposes, the scopes of both skew-based hashing schemes are sufficient to find a proper victim for replacement.

3.3.2 Previous Replacement Algorithms

The study of cache block replacement policies is, in essence, a study of the characteristics or behavior of workloads to a system. Specifically, it is a study of access patterns to blocks within the cache. Based on the recognition of access patterns through acquisition and analysis of past behavior or history, replacement policies resolve to identify the block that will be used furthest down in the future, so that that block may be replaced when needed. The LRU policy does this by attaining the recency of block references, such that the least recently used block will be replaced when needed. The LFU policy considers the frequency of block references, such that the least frequently used block will be replaced when needed. These respective policies are inherently assuming that future behavior of the workload will be dominated by the recency or frequency factors of past behavior.

The ability of the elbow cache to reduce conflict misses depends primarily on the intelligence of the cache replacement policy. Different replacement policies may be used. The random replacement policy is the simplest to implement but it increases the miss rate compared to the baseline configuration (4-way set associative cache).

LRU replacement policy is more effective than random. The traditional LRU replacement policy based on the MRU-LRU sequence is difficult to implement with multiple hashing functions. It is well-known that the complexity of maintaining a MRU-LRU sequence is $s!$, where s is the set associativity. The LRU replacement can be applied to set-associative caches due to their limited sets. Furthermore, pseudo-LRU schemes can be used to reduce complexity for highly associative caches. Since the number of sets grows exponentially with multiple hashing, it is prohibitively expensive to maintain the needed MRU-LRU sequences.

Instead of maintaining the precise MRU-LRU sequence for replacement, a scheme based on the time stamp can be considered. The *time-stamp* (*TS*) scheme emulates the LRU

replacement by maintaining a global memory request ID for each cache block. When a miss occurs, the block with the oldest time-stamp in the set (or the connected set) is replaced [Seznec1993a; Seznec and Bodin 1993b; Bodin and Seznec 1997]. To save the directory space as well as to simplify calculations, a smaller time-stamp is desirable. A more practical scheme, that uses a small number of bits both in the counter and the time-stamp, would work by shifting the counter and all the time-stamps one bit to the right whenever the reference counter overflowed [Gonzalez et al. 1997].

Not Recently Used Not Recently Written (*NRUNRW*) replacement policy is an effective replacement implemented on skew-associative cache [Stone 1971; Seznec 1993a]. The bit tag *Recently Used* (*RU*) is set when the cache block is accessed. Periodically the bit tags RU of all the cache blocks are zeroed. When a cache access misses in the cache, the replaced block is chosen among replacement set in the following priority order. First, randomly pick among the blocks for which the RU tag is not set. Second, randomly pick among the blocks for which the RU tag is set, but which have not been modified since they have been loaded in the cache. Last, randomly pick among the blocks for which the RU tag is set and which have been modified.

Another key issue in implementing cache replacement in the elbow cache is that a linear search for the replacement block among all the connected sets is necessary. It is also prohibitively expensive to traverse the entire scope to find a suitable victim for replacement. Restriction must be added to confine the search within a small set of cache frames.

3.3.3 Elbow Cache Replacement Example

In Figure 3-5, a sequence of memory requests is used to illustrate how a time-stamp based 2-way elbow cache replacement works. Each request is denoted as: *Tag-*f_0*,*f_1*-(ID),* where *Tag* is the block address tag; f_0,and f_1 represent the location of the block in the two coordinates based on two different hashing functions; and *(ID)* represents the request ID for using as a time stamp. For

simplicity, we assume that f_0, f_1, are taken directly from address bits and may be needed as part of the tag to determine a cache hit or miss. Within each partition, there are only four cache frames addressed by the two hashing bits.

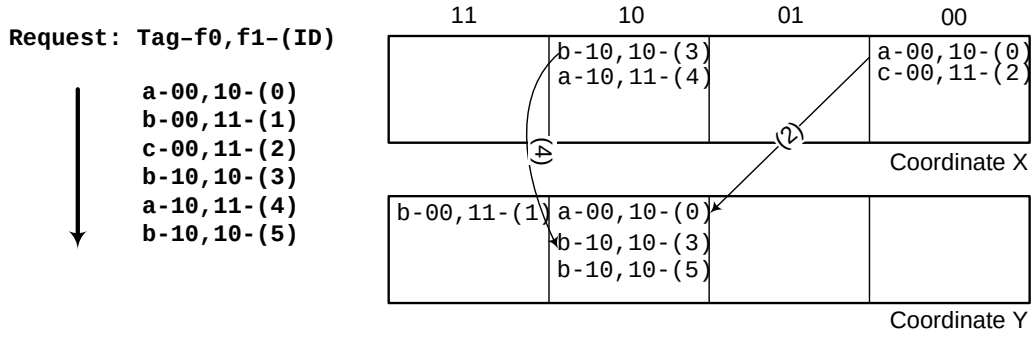


Figure 3-5. Replacement based on time-stamp in elbow caches

When the first request *a-00,10-(0)* is issued, both frames *00* on coordinate *x* and *10* on coordinate *y* are empty. A miss occurs and *a-00,10-(0)* is allocated to frames *00* on coordinate *x*. The second request *b-00,11-(1)* is also a miss and is allocated to frame *11* on coordinate *y* since it is empty in the lookup set. For the third request, *c-00,11-(2)*, both frames in the lookup set are now occupied by the first two requests. However, frame *10* on coordinate *y* is empty, which is in the connected set of the current lookup set through the shared frame *00* on coordinate *x*. Therefore, block *a-00,10-(0)* can be moved to filled frame *10* on coordinate *y* as indicated by the arrow with the request ID *(2)* that leaves the shared frame *00* on coordinate *x* for the newly missed request *c-00,11-(2)*. The fourth request *b-10,10-(3)* finds an empty frame *10* on coordinate *x*. The fifth request, *a-10,11-(4)*, again, misses the cache and both frames in the lookup set are occupied. Assume in this case that both existing blocks are not “old” enough to be replaced. Through the block *b-10,10-(3)* in the shared frame, an “older” block *a-00,10-(0)* in the connected set of frame *10* on coordinate *y* is found and can be replaced as indicated by the arrow

with the request ID *(4)*. Finally, the last request *b-10,10-(5)* can easily be located as a hit even though the block has been relocated.

3.3.4 Least Frequently Used Replacement Policy

The cost and the performance associated with LRU replacement depends on the number of bits devoted to the time-stamp. The implementation of wide bit comparison for multiple parallel time-stamp comparison is very expensive and time consuming. Furthermore, time-stamp also requires extra storage for each block in cache. The more number of bits devoted for time-stamp, the more accurate LRU sequences can be maintained, and the higher implementation cost. Even using the most optimized time-stamp [Gonzalez et al. 1997], the counter for each cache block has at least 8 bits. To reduce the implementation complexity and maintain equal cache performance, we introduce a new cache replacement policy for the elbow cache.

LFU replacement policy maintains the most frequently used blocks in the cache. The least frequently used blocks are evicted from the cache when new blocks are put into it. It is based on the observation that a block tends to be reused if the block has been used more frequently after it was moved into the cache [Qureshi et al. 2005]. We propose *reuse-count scheme* (*RC*), which is a kind of implementation of LFU replacement on the elbow cache. A reuse counter is maintained for each cache block. Upon a miss, the block with the least reuse frequency is replaced. The reuse count is given an initial value when the block is moved into cache. The value is incremented when the block is accessed. These results in the following kind of problem: certain blocks are relative infrequently referenced overall, and yet when they are referenced, due to locality there are short intervals of repeated re-references, thus building up high reuse counts. After such an interval is over, the high reuse count is misleading: it is due to locality, and cannot be used to estimate the probability that such a block will be reused following the end of this interval. Here, this problem is addressed by “factoring out locality” from reuse counts, as

follows. The reuse count is decremented when the cache block is searched, but is mismatched with the requested block. A block can be replaced when the count reaches zero. In this way, the recency information is also counted in the frequency-based replacement policy. Performance evaluation shows that on an elbow cache, a reuse-count replacement policy with 3-bit reuse-counter can perform as good as an LRU replacement policy, with very small storage and low hardware complexity.

To avoid searching for a victim for replacement through all of the connected replacement sets, a block is considered to be *replaceable* when the recorded reference count reaches certain threshold (zero). The search stops when a replaceable block is found. Furthermore, the replacement search can be confined within a limited search domain. For instance, the elbow cache search can be confined within the original lookup set plus single-level interconnected sets to it. In case that no replaceable block is found, a block in the lookup set is replaced. Since the search and replacement are only encountered on a cache miss, they are not on the critical path in cache access. In addition, our simulation results show that about 40% to 70% of the replacement is still located in the lookup set, thus no extra overhead for searching and replacement is incurred.

Vacating the shared frame to make room for the newly missed block involves a data block movement. To limit this data movement, the breadth-first traversal is used to search all possible first-level connected sets through the blocks located in the lookup set. In case no replaceable block is found, the oldest block in the lookup set can be picked for replacement, and thus limits the block movements to at most one per cache miss. The search can be extended to further levels with the cost of an additional block movement per interconnection level. A more dramatic approach to avoid the data movement is to establish a pointer from each directory entry to its

corresponding block in the data array similar to those in [Peir et al. 1998; Qureshi et al. 2005]. However, this indirect pointer approach lengthens the cache access time.

3.4 Performance Evaluation

3.4.1 Miss Ratio Reduction

In this dissertation, we use miss ratios as the primary performance metric to demonstrate the effectiveness of the reuse-count replacement policy for the elbow cache. Various caching schemes and replacement policies were implemented on the L1 data cache, which is 32KB, 4-way partition with 64B line. For comparison purposes, we consider a 32KB, 4-way set-associative L1 cache with LRU replacement as the *baseline* cache.

To evaluate the elbow cache, we excluded workloads that have less than 3% miss ratio gap between a 32KB fully-associative cache and the baseline cache. Based on these criteria, twelve workloads from SPEC2000, *Twolf, Bzip, Gzip, Parser, Equake, Vpr, Gcc, Vortex, Perlbmk, Crafty, Apsi*, and *Eon* were selected.

Three categories of existing caching schemes are evaluated and compared with the elbow cache. The first category is conventional caches with high associativity, including 16-way and fully-associative using the LRU replacement policy, denoted as *16-way+LRU* and *Full+LRU*. The second category is the skewed-associative caches. Three improved replacement policies, NRUNRW [Seznec and Bodin 1993b], time-stamp [Gonzalez et al. 1997], and reuse-count [Qureshi et al. 2005] are considered, denoted as *Skew+NRUNRW, Skew+TS*, and *Skew+RC* respectively. The third category is the V-way cache. Only reuse-count replacement policy is applied due to the nature of the V-way replacement, denoted as *V-way+RC*. Finally, for the elbow caches, the same three replacement policies as the skewed-associative caches are implemented, denoted as *Elbow+NRUNRW*, *Elbow+TS*, and *Elbow+RC*.

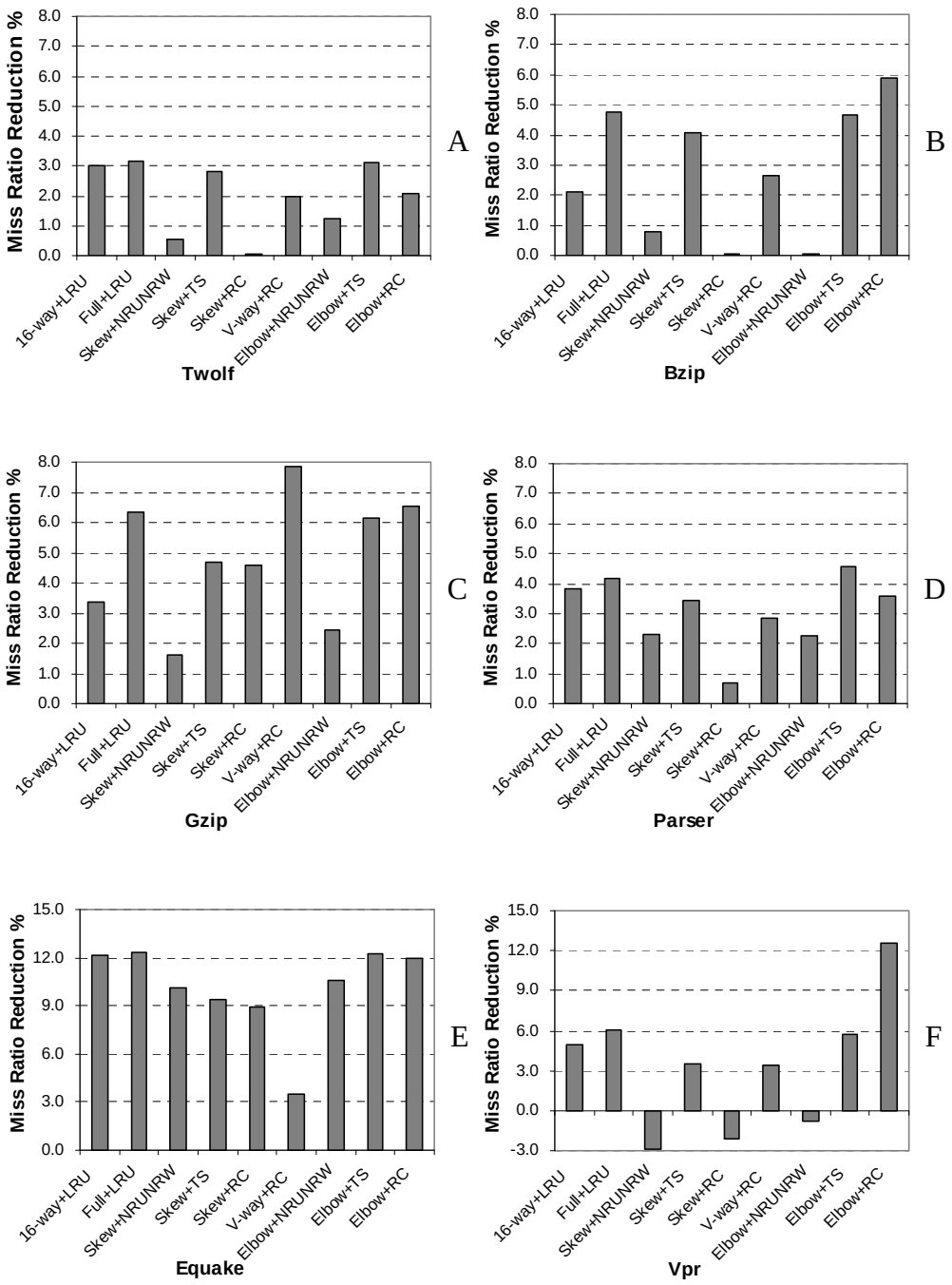
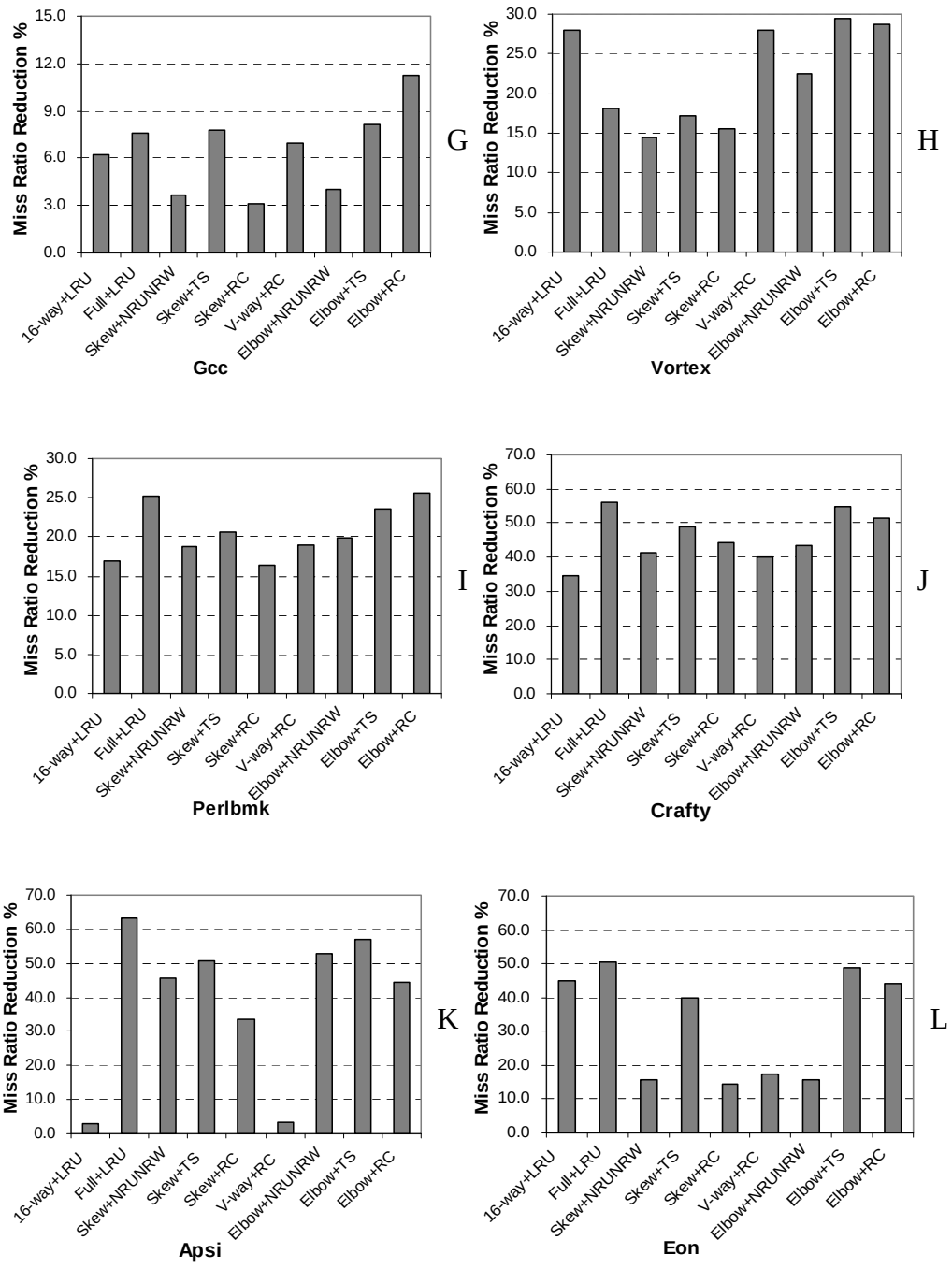


Figure 3-6. Miss ratio reduction with caching schemes and replacement policies



A practical time-stamp scheme is evaluated, which uses an 8-bit time-stamp as reported in [Gonzalez et al. 1997]. For the reuse-count scheme, our evaluation suggested a 3-bit counter with initial value of 3 and victim value of zero for the best performance. If no zero reuse count is found, a victim is picked randomly within the lookup set. For skewed-associative caches and

elbow caches, we borrow *Mapping Function 97* as the hashing functions [Bodin and Seznec 1997]. For V-way cache, we simulate a directory with twice as many entries as the total cache frames. Due to the overhead in sequential search for the replacement and the extra data movement involved in moving the block from the connected frame, we limit the replacement scope within two levels (the lookup set, plus one level of connected set) in elbow caches. As a result, during the replacement search, up to 16 frames can be reached, and at most four directory searches and one block movement may be needed. In case that a replaceable block is not found, the best candidate in the lookup set is replaced.

We compare the miss ratio reduction for various caching mechanisms with different replacement policies. Figure 3-6 summarizes the relative miss ratio reductions for the nine caching schemes compared with the *baseline* cache (32KB, 4-way set-associative cache). Due to a wide range of reductions, the twelve workloads are divided into four groups as can be identified from the figures with four different y-axis scales.

Several interesting observations can be made. First, the elbow caches show more miss ratio reduction than that of the skewed-associative caches. Obviously, it is due to the advantage that the connected sets extended the searching domain from 4 frames to 16 frames. The *Elbow-RC* has miss ratio reduction ranging from 2% to as high as 52% with an average reduction of 21%, while the *Elbow-TS* has miss ratio reduction ranging from 3% to as high as 57% with an average of 22%. The *Skew-RC* has miss ratio reduction ranging from less than 1% to 45% with an average of 11%. On the other hand, the *Skew-TS*'s reduction ranges from 3% to 50% with an average about 17%. For elbow cache, the time-stamp based and the reuse-count based replacement show mixed results. Both methods work effectively with a slight edge to the time-stamp scheme. In contrast, the time-stamp works much better than the reuse-count on skewed-

associative caches. Apparently, LRU replacement performs better when the replacement is confined within the lookup set. Although the time-stamp scheme performs slightly better, its cost is also higher comparing with the reuse-count scheme.

Second, in general the elbow cache out-performs the V-way cache by a significant margin. The average miss ratio reduction for the V-way is about 11%. The relative V-way performance fluctuates a lot among different applications. For example, the V-way cache shows the most reduction compared with other schemes in *Gzip*. It also performs nearly the best in *Vortex*. However, for *Equake* and *Apsi*, V-way's performance is at the very bottom. The main reason for this discrepancy is due to the inability to handle hot sets. For *Gzip* and *Vortex*, 92% and 75% of the time, an unused directory entry in the lookup set can be allocated for the missed block, while for *Equake* and *Apsi*, only 27% and 30% of the chance that a search for replacement outside the lookup set is permitted. This confirms that even doubling the directory size, the V-way cache is constraint in solving the hot set problem.

Third, it is interesting to observe that the elbow cache can out-perform a fully-associative cache by a significant margin in many applications. The average miss-ratio reductions for *Full-LRU*, *Elbow-TS*, and *Elbow-RC* are 21.4%, 21.6%, and 20.9% respectively. These interesting results are due to two reasons. First, fully-associative cache suffers the worst cache pollution when a "never-reuse" block is moved into the cache. It takes *c* more misses to replace a never-reused block, where *c* is the number of frames in the cache. Such a block can be replaced much faster in elbow caches once the block becomes *replaceable*. *Vortex* is a good example to demonstrate the cache pollution problem, in which *Full+LRU* performs much worse than 16-*way+LRU*. Second, the skew hashing functions provide a good randomization for alleviating set

conflicts. By searching through the connected sets in elbow caches, the hot set issue is pretty much diminished.

3.4.2 Searching Length and Cost

In this section, we evaluate extra cost associated with elbow caches. A normal cache access involves a single tag directory access to determine a hit/miss. During the replacement in the elbow cache, extra tag accesses are required when traversing through the replacement set. Moreover, an additional block movement may happen between search levels when the replaced block is not located in the lookup set. We simulated replacement policy with 2-level searching (lookup set plus one level of connected sets). If a victim is found at the first level, there is no extra tag access and data movement. Otherwise, an extra block movement along with up to three additional tag accesses is needed if the replaced block is found in the second level. In case that no replaceable block can be found within 2 levels, a victim will be chosen from the lookup set and no extra block movement is required. However, it does incur 3 additional tag accesses.

Table 3-1. Searching levels, extra tag access, and block movement

Workload	Replacement search			Overhead/Access	
	1st level	2nd level	Not found	Extra tag accesses	Block movement
Bzip	61.8%	35.0%	3.2%	1.5%	1.0%
Vpr	47.0%	45.0%	8.0%	2.7%	1.5%
Perlbmk	39.0%	45.4%	15.6%	3.3%	1.6%
Apsi	45.6%	45.0%	9.4%	2.5%	1.4%

Table 3-1 summarizes the cost of the elbow cache with four workloads, *Bzip*, *Vpr*, *Perlbmk,* and *Apsi*. Note that we selected these four workloads, one from each miss reduction range as described previously to simplify the presentation. The percentage of chance in finding a replaceable block at respective searching levels is shown. About 40%-60% of the replaceable blocks are located in the lookup set and about 35%-45% are found at the connected sets. The

percentages that no replaceable block is found in the first two levels are varied from 3% to 15%. We also count the extra tag accesses and block movements. As shown in the table, extra 1.5% to 3.3% tag accesses are encountered in the elbow cache. Also, on the average, an extra block movement is needed for 1% to 1.6% of the memory accesses. It is important to notice that extra tag accesses and block movements are not on the critical cache access path, because they are only encountered on cache misses. These extra tag accesses and block movements can be delayed in case of a conflict with normal cache accesses.

3.4.3 Impact of Cache Partitions

So far, our evaluations of the elbow cache are based on a 4-way partitioned structure. In this section, we show the results of 2-, 4-, and 8-way partitions. Again, the four workloads, *Bzip*, *Vpr, Perlbmk* and *Apsi,* one from each miss reduction range, are selected. The miss ratio, instead of miss ratio reduction is used for the comparison.

As shown in Figure 3-7, increasing the degree of partition improves the miss ratios for all four workloads. These results are obtained using *Elbow+RC*. Similar results are also observed using *Elbow+TS*. From 2-way to 8-way, the miss ratios are reduced accordingly: 2.8%, 2.6%, 2.5% for *Bzip*; 4.2%, 3.4%, 3.2% for *Vpr*; 12.2%, 10.3%, 8.8% for *Perlbmk*; and 2.0%, 1.6%, 1.4% for *Apsi,* respectively. These reduction rates are much faster than the miss reduction rates for set-associative caches when the associativity increases from 2-way to 8-way.

In a 2-level elbow cache, the search domain is equal to p^2 where p is the number of partitions. For elbow caches from 2-way to 8-way, the replacement scope can reach from 4, 16, to 64 frames. This power-of-2 increase in replacement set out-performs the linear increase in replacement set for the set-associative caches. In term of costs, however, the extra directory tag

access only increases linearly with the number of partitions. Moreover, the increase of partitions requires no extra block movement when the replacement is confined within two levels.

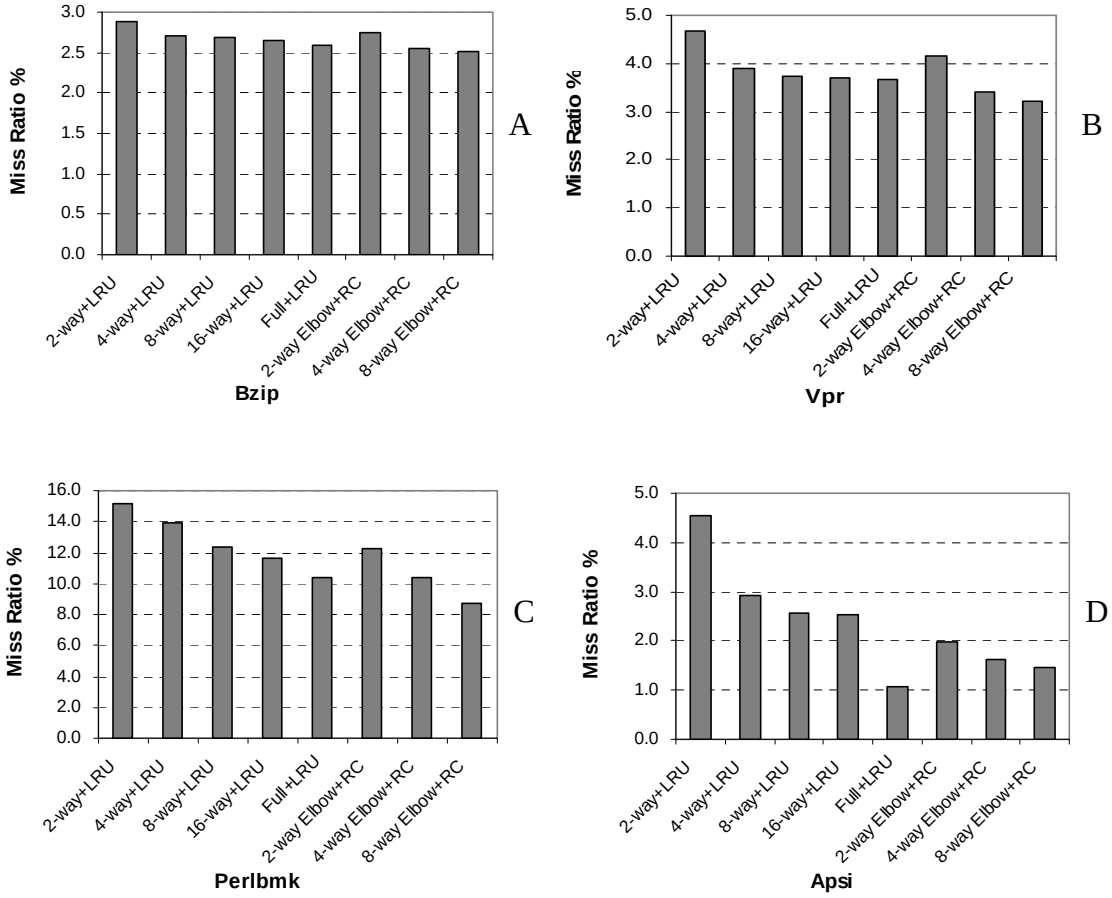


Figure 3-7. Miss ratio for different cache associativities

In comparison with fully-associative cache, further miss reductions with 8-way elbow caches make it surpluses fully-associative cache performance for three of the four workloads. For *Bzip*, *Vpr* and *Perlbmk*, the miss ratios reduce by 3.2%, 12.3%, and 16.6%, respectively.

3.4.4 Impact of Varying Cache Sizes

We analyze the performance impact of cache sizes on elbow caches. Four L1 data cache sizes 8KB, 16KB, 32KB, and 64KB are simulated using the same four workloads as those in Section 3.4.3. The miss ratios are plotted in Figure 3-8 for three caching schemes: *4-way+LRU*,

Full+LRU, and *4-way Elbow+RC.* As observed, the cache size does make a huge impact on the miss ratios of the three caching schemes. It is straightforward that bigger caches reduce the miss ratios for all three caching schemes.

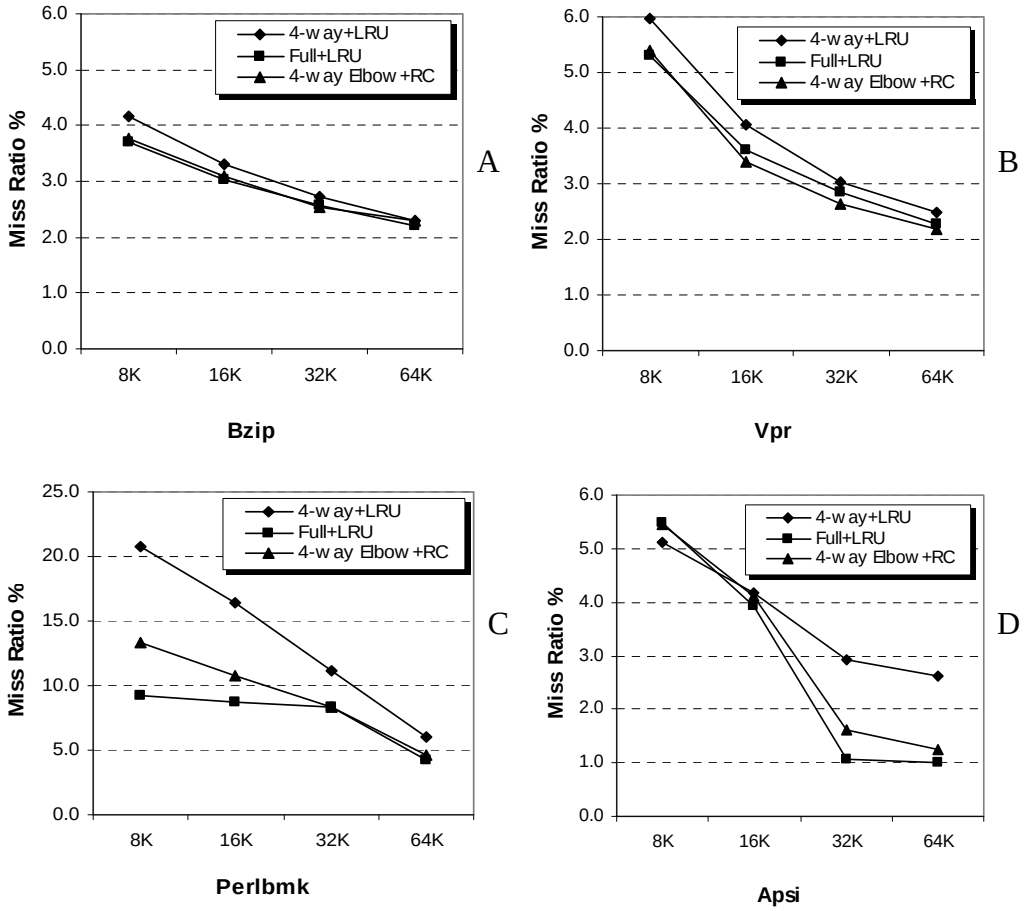


Figure 3-8. Miss rate for different cache sizes

However, the relative gaps among the three schemes vary widely among different workloads with different cache sizes. Generally speaking, conflict misses are reduced with bigger caches that make the elbow cache less effective. This is true for *Bzip* and *Perlbmk.* However, for *Vpr*, the gap between *4-way+LRU* and *Full+LRU* stay relatively the same with 32KB and 64KB caches. Therefore, the elbow cache is equally effective with all four cache sizes. *Apsi* acts oppositely. The elbow cache is much more effective for 32KB and 64KB caches.

Detailed study of *Apsi* indicates that the 32KB *Full+LRU* can hold the working set, but the 32KB *4-way+LRU* cannot due to conflicts in hot sets. Consequently, *Elbow+RC* shows a huge gap against *4-way+LRU* due to better replacement. At 16KB, none of caching scheme can hold the working set that creates heavy capacity misses. As a result, all three caching schemes show similar performance.

3.5 Related Work

Applications with regular patterns of memory access can experience severe cache conflict misses in set-associative cache. There are few works on finding better mapping functions for cache memories. Most of the prior hashing functions permute the accesses using some form of Exclusive OR (*XOR*) operations. The elbow cache is not limited to any specific randomization/hashing method. Other possible functions could be found in [Yang and Adina 1994; Kharbutli et al. 2004]. The skewed-associative cache apples different mapping functions on different partitions. Although various replacement policies [Seznec1993a; Seznec and Bodin 1993b; Bodin and Seznec 1997; Gonzalez et al. 1997] have been proposed for the skewed-associative cache, it is still an open issue to find an efficient and effective one.

The Hash-rehash [Agarwal et al. 1988], the column-associative [Agarwal and Pudar 1993a], and the group-associative [Peir et al. 1998] are using extra directories to increase associativity. In the contrast, the elbow cache uses links to connect blocks in the cache without any extra directory storage. The V-way cache [Qureshi et al. 2005] can be viewed as a new way to increase associativity by doubling the cache directory size. However, it requires indirect links between each entry in the directory and its corresponding block in the data array. Furthermore, even with extra directory space, it can not solve the hot set problem since the directory entries in the hot sets are always occupied.

3.6 Conclusion

The efficiency of the traditional set-associative cache is degraded because of severe conflict misses. The elbow cache has demonstrated its ability to expand the replacement set beyond the lookup set boundary without adding any complexity on the lookup path. Because of the characteristics of elbow cache, it is difficult to implement recency-based replacement. The proposed reuse-count replacement policy with low-cost can achieve cache performance comparable to the recency-based replacement policy.

CHAPTER 4
TOLERATING RESOURCE CONTENTIONS WITH RUNAHEAD ON MULTITHREADING PROCESSORS

4.1 Introduction

SMT processors exploit both ILP and TLP by fetching and issuing multiple instructions from multiple threads at the same cycle to utilize wide-issue slots. In SMT, multiple threads share resources such as caches, functional units, instruction queue, instruction issue window, and instruction window [Tullsen et al. 1995; Tullsen et al. 1996]. SMT typically benefits from giving threads complete access to all resources every cycle. But contentions of these resources may significantly hamper the performance of individual threads and hinder the benefit of exploiting more parallelism from multiple threads.

First, disruptive cache contentions lead to more cache misses and hurt overall performance. Second, threads can hold critical resources while they are not making progress due to long-latency operations and block other threads from making normal execution. For example, if the stalled thread fills the issue window and instruction window with waiting instructions, it shrinks the window available for the other threads to find instructions to issue and bring in new instructions to the pipeline. Thus, when parallelism is most needed when one or more threads are no longer contributing to the instruction flow, fewer resources are available to expose that parallelism.

We investigate and evaluate a valuable solution to this problem, runahead execution on SMTs. Runahead execution was first proposed to improve MLP on single-thread processors [Dundas and Mudge 1997; Mutlu et al. 2003]. Effectively, runahead execution can achieve the same performance level as that with a much bigger instruction window. With heavier cache contentions on SMTs, runahead execution is more effective in exploiting MLP. Besides the inherent advantage of memory prefetching, by removing long-latency memory operations from

the instruction window, runahead execution can ease resource blocking among multiple threads on SMTs, thus make other sophisticated thread-scheduling mechanisms unnecessary [Tullsen and Brown 2001; Cazorla et al. 2003].

4.2 Resource Contentions on Multithreading Processors

This section demonstrates the resource contention problem in SMT processors. Several SPEC2000 benchmarks are chosen for this study based on their L2 cache performance [Hu et al. 2003]. The *weighted speedup* [Snavely and Tullsen 2000] is used to compare the IPC of multithreaded execution against the IPC when each thread is executed independently. IPC_{SMT} represents individual thread's IPC in the SMT mode.

Figure 4-1 shows the weighted speedups of eight combinations of two threads on SMTs using the ICOUNT2.8 scheduling strategy [Tullsen et al. 1996]. These results show that running two threads on SMTs may present worse IPC improvement than running two threads separately. We can categorize workloads into three groups. The first group includes *Twolf/Art, Twolf/Mcf,* and *Art/Mcf*, which are composed of programs with relatively high L2 miss penalties [Hu et al. 2003]. The second group includes *Parser/Vpr, Vpr/Gcc*, and *Twolf/Gcc*, which consist of programs with median L2 miss penalties. Finally, the third group has two workloads *Gap/Bzip* and *Gap/Mcf* in which either one or both programs have low L2 miss penalties. In general, except for the third group, other workloads have poor performance with median-size L2 caches. For the first group, the median size is about 2MB to 8MB, while for the second group, the median size is about 512KB to 1MB. The weighted speedups in the SMT mode in these cache sizes can be significantly lower than 1.

$$Weighted\ Speedup = \sum_{threads} \frac{IPC_{SMT}}{IPC_{Single-thread}}$$

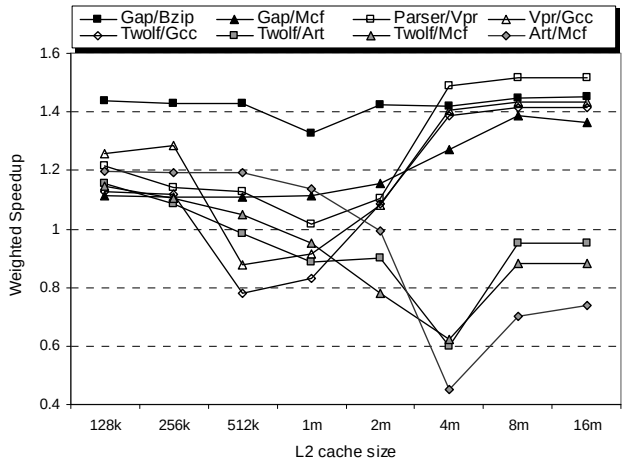


Figure 4-1. Weighted instruction per cycle speedups for multiple threads vs. single thread on simultaneous multithreading

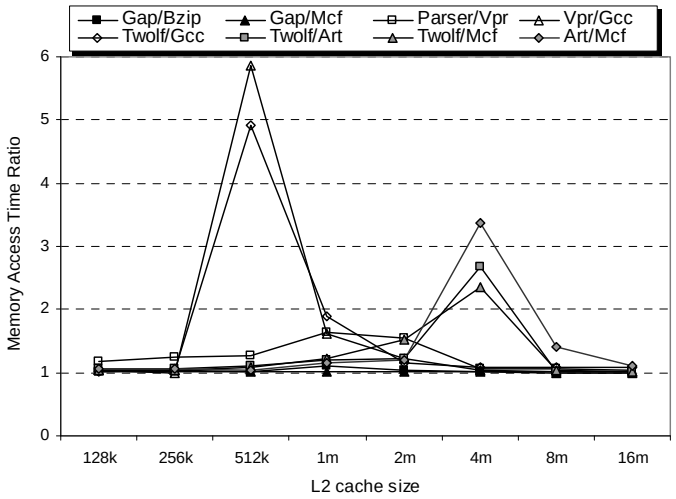


Figure 4-2. Average memory access time ratio for multiple threads vs. single thread on simultaneous multithreading

Generally speaking, with small caches, heavy cache misses are encountered regardless the number of threads. Thus, no significant performance degradation is observed in the SMT mode. With large caches, both threads may incur very few cache misses even when two threads are running together in the SMT mode. This U-shape speedup curve is evident with workloads in the second group. It is generally true for workloads in the first group too. However, negative IPC speedups can still result from workloads in the first group even with very large L2 caches due to

degradations on other resource contentions out-weight the benefit of exploiting TLP. Workloads in the third group benefit from TLP consistently.

To prove the impact of cache contentions on SMT performance, we plot the *average memory access time* ratio between two threads running in the SMT mode and running sequentially in a single-thread mode (Figure 4-2). Except for *Gap/Bzip* and *Gap/Mcf*, other workloads experience increases in the average memory access time with median-size caches ranging from 512KB to 4MB. The degree of increases of the average memory access time matches well against the IPC losses in Figure 4-1. Nevertheless, although little increases of average memory access times can be observed with 8 MB or 16MB L2 caches, workloads in the first group still show huge IPC degradations. This is again due to contentions on other resources.

4.3 Runahead Execution on Multithreading Processors

Runahead execution on SMT processors follows the same general principle as in single-thread processors [Mutlu et al. 2003]. It prevents the instruction window from stalling on long-latency memory operations by executing speculative instructions. Runahead execution of a thread starts once a long latency load reaches the top of the instruction window. An *invalid* value is assigned to the long-latency load to allow the load to be pseudo-committed without blocking the instruction window. A checkpoint of all the architecture states must be made before entering the runahead execution mode. During runahead mode, the processor speculatively executes instructions relying on the invalid value. All the instructions that operate over the invalid value will also produce invalid results. However, the instructions that do not depend on the invalid value will be pre-executed. When the memory operation that started runahead mode is resolved, the processor rolls back to the initial checkpoint and resumes normal execution. As a consequence, all the speculative work done by the processor is discarded. Nevertheless, this previous execution is not completely useless. The main advantage of runahead is that the

speculative execution would have generated useful data prefetches, improving the behavior of the memory hierarchy during the real execution. In some sense, runahead execution has the same effect as physically enlarging the instruction window.

We adapted and modified a SimpleScalar-based SMT model from the Vortex Project [Dorai and Yeung 2002]. The out-of-order SimpleScalar processor separates in-order execution at the functional level from the detailed pipelined timing simulation. At the function level, instructions are executed one at a time without any overlap. The results from the function execution drive the pipelined timing model. One important implementation issue is that the checkpoint for runahead execution must be made at the functional level. The actual *invalid* value from runahead execution will not be simulated. Instead, registers or memory locations are marked when their content is invalid. In the runahead mode, only those L2 misses with correct memory addresses will be issued.

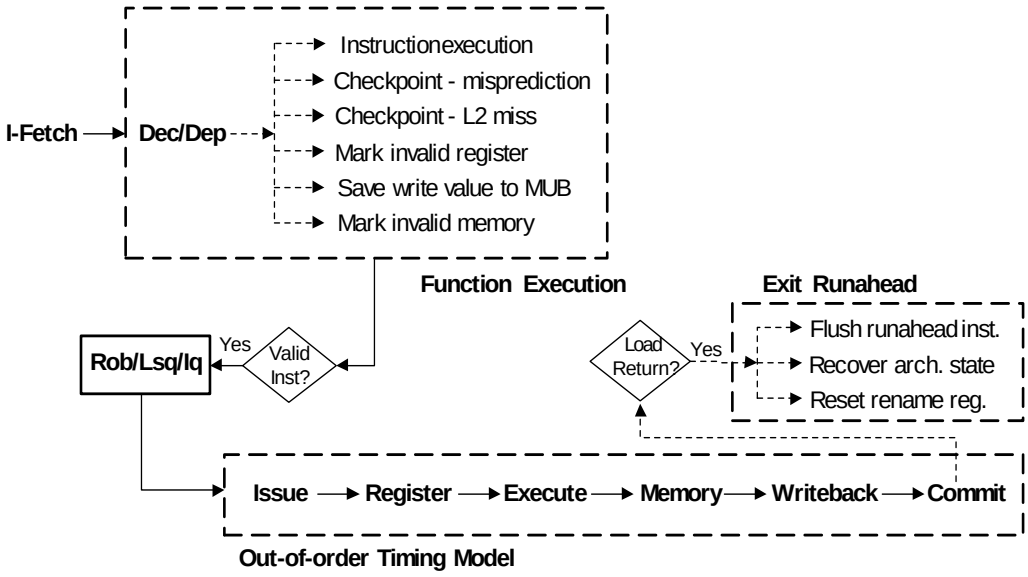


Figure 4-3. Basic function-driven pipeline model with runahead execution

Figure 4-3 illustrates the pipeline microarchitecture. The checkpoint is made at the *Dec/Dep* state in the function mode when a load misses the L2 cache. During runahead

executions, all destination registers and memory locations are marked invalid for the L2 miss and all its descendent instructions. All memory writes are buffered in MUB (Memory Update Buffer) to allow correct executions while maintaining memory states at the checkpoint.

4.4 Performance Evaluation

Performance evaluations of runahead execution are carried out on modified out-of-order SimpleScalar-based SMT model [Dorai and Yeung 2002]. ICOUNT2.8 scheduling policy [Tullsen et al. 1996] is used. Except for replicated program counters, register files, and completion logic, other resources are shared among multiple threads. Threads share a 256-entry instruction window and a 256-entry load-store queue. Issue window is assumed to be the same size of the instruction window. We fix the size of L1 caches as 64KB and vary the size of L2 cache from 128KB to 16MB. Memory access latency is set as 400 cycles. There can be at most 60 outstanding memory request at the same time.

Eight mixed workload combinations from SPEC2000, *Twolf/Art*, *Twolf/Mcf*, *Art/Mcf*, *Parser/Vpr*, *Vpr/Gcc*, *Twolf/Gcc*, *Gap/Bzip*, and *Gap/Mcf* are selected. For measuring the weighted speedup, total simulated instructions of individual threads are kept the same between the multiple-thread execution mode and the single-thread execution mode.

4.4.1 Instructions Per Cycle Improvement

Figure 4-4 summarizes the IPC improvement of runahead on SMTs with 1MB L2 caches, where the IPCs of individual threads as well as the IPCs of the mixed threads are plotted. The three bars on the left of each workload represent the IPCs without runahead execution, while the right three bars are IPCs with runahead.

In general, runahead improves IPCs in both single-thread and multithread modes. There is more significant improvement on the SMT mode than that on the single-thread mode. Among the three groups, workloads in the second group benefit the most. With runahead, the combined IPCs

on SMTs are consistently higher than the IPC of each individual thread. This is consistent with the results in Figure 4-2, where workloads in the second group show the most increases in the average memory access time. Although runahead on SMTs also shows much higher improvement, the resulting IPCs in other two groups of workloads still fall between the IPCs of the two individual threads. Therefore, we decided to compare IPC improvements using the weighted speedup as suggested in [Tullsen and Brown 2001] with various cache sizes.

$$Weighted\ Speedup = \sum_{threads} \frac{IPC_{new}}{IPC_{baseline}}$$

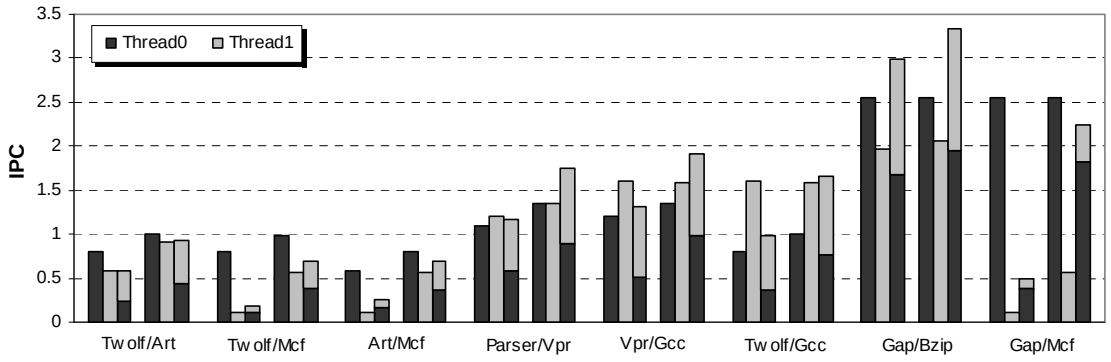


Figure 4-4. Instructions per cycle with/without runahead on simultaneous multithreading

4.4.2 Weighted Speedups on Multithreading Processors

Figure 4-5 shows the weighted IPC speedups of runahead execution on SMTs. In general, significant performance improvement can be observed as long as the cache size is not very large. As a result of very few misses on 8 MB or 16 MB caches, runahead execution is not effective in overlapping scatter cache misses. Similarly, since *Gap/Bzip* has very small combined working set, runahead execution is ineffective for all cache sizes. Among eight workloads, it is unexpected that *Gap/Mcf* displays the highest speedup. Cache contentions of *Gap/Mcf* should not be as severe as workloads in the first group since *Gap* has the lowest L2 miss penalty among selected programs. However, runahead execution not only exploits MLP for *Mcf*, it also releases

resources to unblock *Gap* from frequent L2 misses of *Mcf*. Other workloads show performance benefits of various degrees from runahead execution with small/median caches. Because of heavier cache misses for workloads in the first group, the weighted speedup is generally higher than that of workloads in the second group.

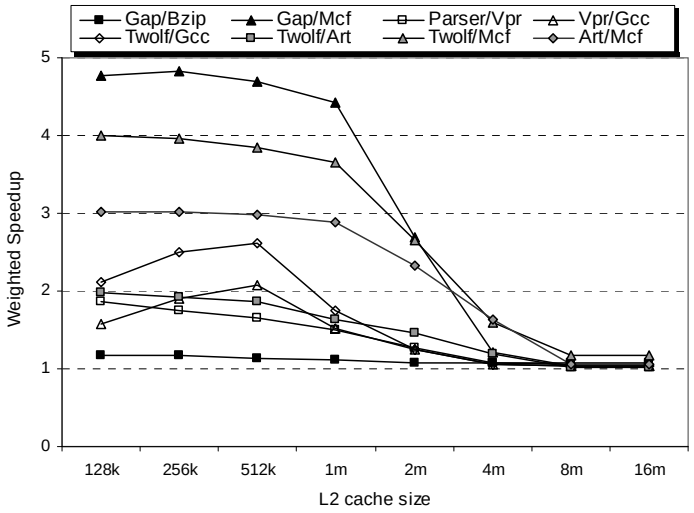


Figure 4-5. Weighted speedup of runahead execution on simultaneous multithreading

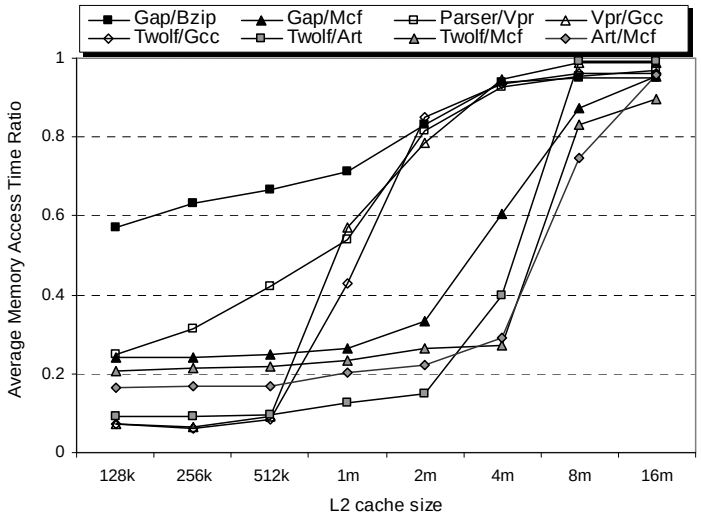


Figure 4-6. Average memory access time ratio of runahead execution on simultaneous multithreading

The improvement of the average memory access time of runahead execution on SMTs is shown in Figure 4-6, where the ratio is the average memory access time with runahead over that without runahead. Significant drops on the average memory access time are very evident for all workloads except for 16MB caches. It is interesting to observe that because of differences in working set, significant jumps in memory access time ratios are from 2MB to 8MB for workloads in the first group, but from 512KB to 2MB for workloads in the second group. As expected, *Gap/Bzip* is not as beneficial with runahead due to small working set.

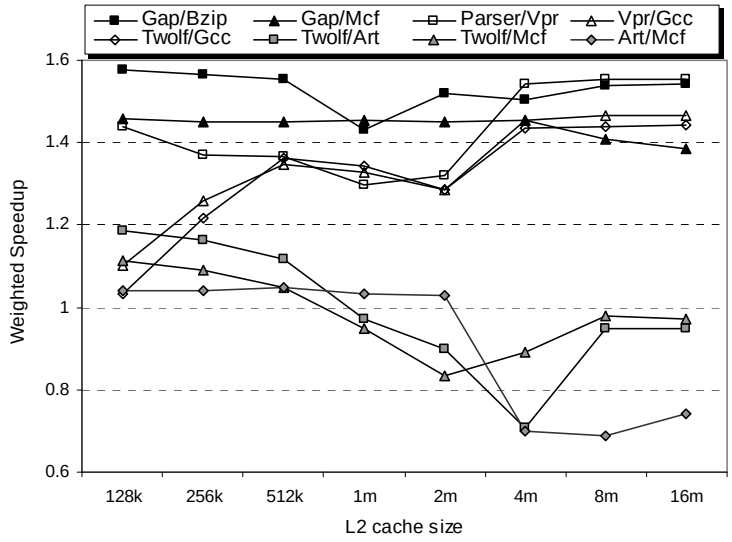


Figure 4-7. Weighted speedup of runahead execution between two threads running in the simultaneous multithreading mode and running separately in a single-thread mode

Recall that the overall IPC improvement with runahead execution comes both from exploiting MLP and from better sharing of other resources. Therefore, minor discrepancies between the average memory access time improvement and the overall IPC improvement can be expected. For example, the average memory access time improvement of *Gap/Mcf* is less than that of workloads in the first group. However, *Gap/Mcf* displays significantly more IPC improvement (Figure 4-5). Similarly, *Twolf/Mcf* has worse improvement in memory access time

than the other two workloads in the first group, but its IPC improvement is the highest among the three workloads.

Figure 4-7 shows the weighted speedups of SMTs with runahead over single thread execution with runahead. The advantages of runahead execution on the SMT mode are clearly displayed for workloads in both the second and the third groups. However, workloads in the first group are still experiencing negative speedups when cache sizes are 1MB or bigger. In comparison with negative speedups without runahead execution (Figure 4-1), runahead execution helps to pull negative speedups in the positive direction. For 4MB caches especially, weighted speedups are improved from 0.61, 0.60, 0.44 without runahead to 0.90, 0.71, 0.70 with runahead for *Twolf/Mcf, Twolf/Art* and *Art/Mcf* respectively. As a result of very poor SMT performance due to cache and other resource contentions, runahead execution can alleviate but cannot overcome the huge loss from running two threads in the SMT mode.

The weighted speedup in Figure 4-7 is a combination of two factors: the benefit of runahead execution when two threads run together vs. run separately, and the impact of SMT itself. In order to separate effects of runahead execution from effects of SMT execution, we define a new *Weighted Speedup Ratio*. The basic idea is to calculate speedup ratios between individual thread's runahead speedup in the SMT mode and its runahead speedup in the single-thread mode.

$$Weighted\ \ Speedup\ \ Ratio\ \ =$$
$$\frac{1}{Number\ of\ thread} \times \sum_{Thread} \left(\frac{IPC_{SMT-runahead}}{IPC_{SMT-norunahead}} \Big/ \frac{IPC_{Single-runahead}}{IPC_{Single-norunahead}} \right)$$

As shown in Figure 4-8, the speedup of runahead execution in SMT mode is generally better than the speedup of runahead execution in single-thread execution mode. For example, *Gap/Mcf* shows huge benefit for runahead execution on the SMT mode. This proves why

Gap/Mcf has the highest overall IPC speedup (Figure 4-5). Similarly, *Twolf/Mcf* shows higher overall speedups comparing with other workloads in the first group due to more benefit of runahead execution in the SMT mode. Two workloads *Vpr/Gcc* and *Twolf/Gcc* from the second group exhibit negative improvement with tiny caches. Recall that programs in this group have moderate L2 cache penalties. With unrealistically small caches, cache misses can increase to the point that runahead execution becomes very effective in the single-thread mode.

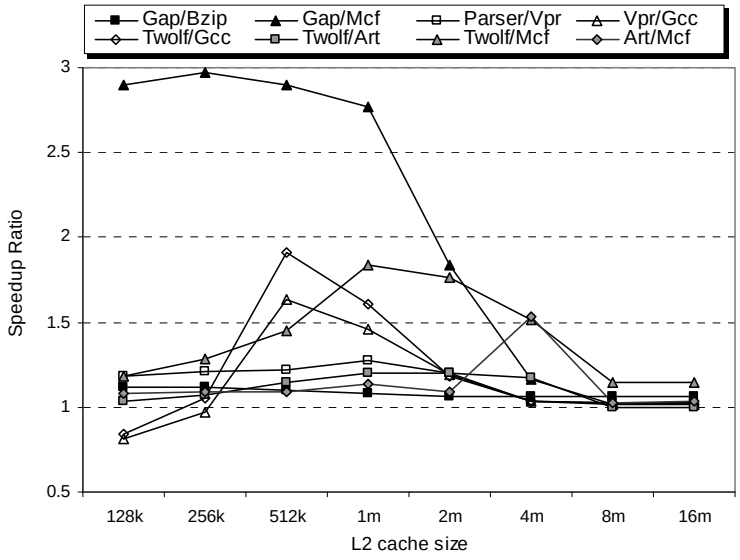


Figure 4-8. Ratios of runahead speedup in simultaneous multithreading mode vs. runahead speedup in single-thread mode

4.5 Related Work

SMT permits the processor to issue multiple instructions from multiple threads at the same cycle [Tullsen et al. 1995; Tullsen et al. 1996]. The scheduling strategy based on the instruction count (ICOUNT) of each active thread regulates the fetching policy to prevent any thread from taking more resources than its fair share [Tullsen et al. 1996]. Other proposed methods [El-Moursy and Albonesi 2003; Cazorla et al. 2004a; Cazorla et al. 2004b] attempt to identify

threads that will encounter long-latency operations (L2 cache miss). The thread with long-latency operation may be delayed to prevent it from occupying more resources.

Serious cache contention problems on SMT processors were reported [Tullsen and Brown 2001]. Instead of keeping the thread that involves long-latency load ready to immediately begin execution upon return of the loaded data, they proposed methods to identify threads that are likely stalled and to free all resources associated with those threads. A balance scheme was proposed [Cazorla et al. 2003] to dynamically switch between flushing and keeping long-latency threads to avoid overhead of flushing.

The optimal allocation of cache memory between two competing threads was studied [Stone et al. 1992]. Dynamic partitioning of shared caches among concurrent threads based on "marginal gains" was reported in [Suh et al. 2001]. The results showed that significantly higher hit ratios over the global LRU replacement could be achieved.

Runahead execution was first proposed to improve MLP on single-thread processors [Dundas and Mudge 1997; Mutlu et al. 2003]. Effectively, runahead execution can achieve the same performance level as that with a much bigger instruction window. We investigate and evaluate runahead execution on SMT processors with multiple threads running simultaneously. It is our understanding that this is the first work to apply runahead execution on SMT processors to tolerate shared resource contentions. Besides the inherent advantage of memory prefetching, runahead execution can also prevent a thread with long latency loads from occupying shared resources and impeding other threads from making forward progress.

4.6 Conclusion

Simultaneous Multithreading technique has been moved from laboratory ideas into real and commercially successful processors. However, studies have shown that without proper

mechanisms to regulate the shared resources, especially shared caches and the instruction window, multiple threads show lower overall performance when running simultaneously.

Runahead execution, proposed initially for achieving better performance for single-thread applications, works very well in the multiple-thread environment. In runahead execution, multiple long-latency memory operations can be discovered and overlapped to exploit the memory-level parallelism; meanwhile, shared critical recourses held by the stalling thread can be released to keep the other thread running smoothly to exploit the thread-level parallelism. Performance evaluations have demonstrated that up to 4-5 times the speedups are achievable with runahead executions on SMT environments.

CHAPTER 5
ORDER-FREE STORE QUEUE USING DECOUPLED ADDRESS MATCHING AND AGE-PRIORITY LOGIC

5.1 Introduction

Timely handling correct memory dependences in a dynamically scheduled, out-of-order execution processor has posted a long-standing challenge, especially when the instruction window is scaled up to hundreds or even thousands of instructions [Sankaralingam et al. 2003; Sankaralingam et al. 2006; Sethumadhavan et al. 2007]. Two types of queues are usually implemented in resolving memory dependences. A *Store Queue (SQ)* records all in-flight stores for determining store-load forwarding and a *Load Queue (LQ)* records all in-flight loads for detecting any memory dependence violation [Kessler 1999; Hinton et al. 2001]. There are two fundamental challenges in enforcing correct memory dependences. The first one is to forward values from the youngest older in-flight store with matched address to a dependent load. In a conventional processor, this is implemented by forcing stores enter the SQ in program order and finding the parent store using expensive fully-associative search. The second challenge is to maintain correct memory dependence when a load is issued but early store addresses have not been resolved. Speculation based on memory dependence prediction or other aggressive methods [Adams et al. 1997; Hesson et al. 1997; Chrysos and Emer 1998; Kessler 1999; Yoaz et al. 1999; Hinton et al. 2001; Subramaniam and Loh 2006] enables the load to proceed without waiting for the early store addresses. Any offending load that violates the dependence must be identified later by searching the LQ and causes a pipeline flush. In a conventional processor, the program order and fully-associative search are also required in the LQ for identifying any memory dependence violation by early executed loads when an older store is executed [Sha et al. 2005].

There have been many proposals for improving the scalability of the SQ and LQ [Moshovos et al 1997; Park et al. 2003; Sethumadhavan et al. 2003; Roth 2004; Srinivasan et al.

2004; Cristal et al. 2005; Sethumadhavan et al. 2006; Sha et al. 2005; Stone et al. 2005; Torres et al. 2005; Castro et al. 2006; Garg et al. 2006; Sha et al. 2006; Subramaniam and Loh 2006]. In this work, we focus on an efficient SQ design for store-load forwarding. Since store addresses can be generated out of program order, the SQ cannot be partitioned into smaller address-based banks while maintaining program order in each bank for avoiding fully-associative searches in the entire SQ. Among many proposed solutions for scalable SQ [Akkary et al. 2003; Sethumadhavan et al. 2003; Gandi et al. 2005; Sha et al. 2005; Torres et al. 2005; Baugh and Zilles 2006; Garg et al. 2006; Sethumadhavan et al. 2007], two recent approaches are of great significance and related to our proposal. The first approach is to accept potential unordered stores and loads by speculatively forwarding the matched *latest* store based on the execution sequence, instead of the correct *last* store in program order [Gandi et al. 2005; Garg et al. 2006]. The second approach is to allow an unordered banked SQ indexed by store address, but record the correct *age* along with the store address [Sethumadhavan et al. 2007]. Sophisticated hardware can match the *last* store according to the age of the load without requiring the stores to be ordered by their ages in the SQ. Our simulation results show that a significant amount of mismatches exist between the latest and the last stores for dependent loads that causes expensive re-executions. Furthermore, there is at most one matching parent store in the SQ even though multiple stores could have the same address. Recording each store and age pair complicates the age priority logic and may become the source of conflicts with limited capacity in each SQ bank.

In this work, we introduce an innovative SQ design that *decouples* the store/load address matching unit and its corresponding age-order priority encoding logic from the original SQ. In our design, renamed/dispatched stores' address and data enter a *SQ RAM* array in program order. Instead of relying on fully-associative searches in the entire SQ, a separate *SQ directory* is

maintained for matching the store addresses in the SQ. A store enters the SQ directory when its address is available. Only a single entry is allocated in the SQ directory for multiple outstanding stores that have the same address. Each entry in the SQ directory is augmented with an *age-order vector* to indicate the correct locations (*ages*) of multiple stores with the same address in the SQ RAM. The width of the age-order vector is equal to the size of the SQ RAM. When a store is issued, a directory entry is created if it does not already exist. The corresponding bit in the age-order vector is turned on based on the location of the store. When a load is issued, an address match to an entry in the SQ directory triggers the age-order priority logic on the associated age-order vector. Based on the age of the load, a simple leading-one detector can locate the youngest store that is older than the load for data forwarding. With the age-order vector, the store address is free from imposing any order in the SQ directory. Consequently, the decoupled SQ directory can be organized as a set-associative structure to avoid fully-associative searches. Besides the basic decoupled SQ without considering the data alignment, we further extend the design to handle partial stores and loads by using byte masks to identify which bytes within an 8-byte range are read or written. We also include the memory dependence resolution for the misaligned stores and loads that cross the 8-byte boundary.

The decoupled address matching and age-order priority logic presents significant contributions over the existing full-CAM SQ or other scalable SQ designs. First, because the size and configuration of the decoupled SQ directory are independent of the program-ordered SQ RAM, it provides new opportunities to optimize the SQ directory design for locating the parent store. Second, the relaxation of the program-order requirement in the SQ directory using a detached age-order vector helps to abandon the fully-associative search that is the key obstacle for a scalable SQ. Third, a store needs not be present in the SQ directory until its address

becomes available. As reported in [Sethumadhavan et al. 2007], a significant amount of renamed stores do not have their address available and hence need not occupy any SQ directory space. Fourth, our evaluation shows that on average, close to 30% of the executed stores have duplicated store addresses in SQ. The SQ directory only needs to cover the unique addresses of the issued stores, and hence the SQ directory size can be further reduced. Moreover, by getting rid of duplicated addresses in the SQ directory, the potential set (bank) conflict can also be alleviated since the duplicated store addresses must be located in the same set causing more conflicts. Fifth, a full-CAM directory is inflexible for parallel searches that are often necessary in handling memory dependences for stores and loads misaligned across the 8-byte entry boundary. The set-associative (banked) SQ directory, on the other hand, permits concurrent searches on different sets and hence can eliminate the need to duplicate the SQ directory. Lastly, we believe this is the first proposal that correctly accounts and handles frequently occurring partial and misaligned stores and loads, such as in x86 architecture, while previous proposals have not dealt with this important issue. The performance evaluation results show that the decoupled SQ outperforms the latest-store forwarding scheme by 20-24%, while outperforms the late-binding SQ by 8-12%. In comparison with an expensive full-CAM SQ, the decoupled SQ only loses less than 0.5% of the IPC.

5.2 Motivation and Opportunity

In this section, we demonstrate that multiple in-flight stores with the same address constantly exist in the instruction window and they present significant impact on the SQ design. We also show the severity of mismatches between the *latest* and the *last* store to dependent load. The simulation is carried out on PTLsim [Yourst 2007] running SPEC workloads. We simulated a 512-entry instruction window with unlimited LQ/SQ.

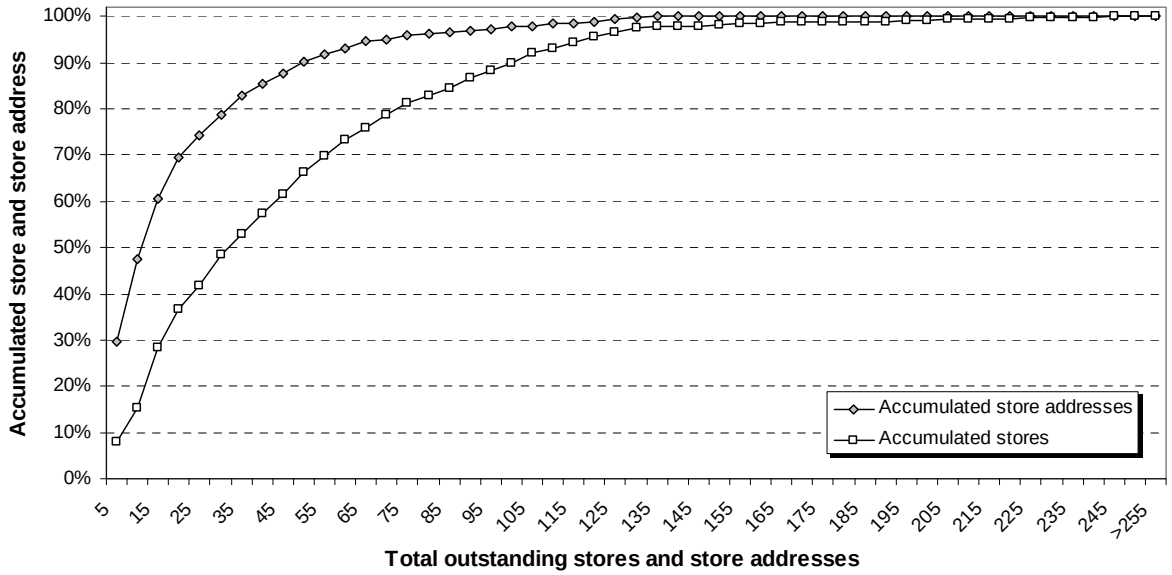


Figure 5-1. Accumulated percentage of stores and store addresses for SPEC applications

Figure 5-1 plots the accumulated percentages of stores and unique store addresses with respect to the total number of outstanding stores and store addresses for the twenty simulated SPEC2000 applications. The statistics is collected after each store is issued and its address is available. An infinite SQ is maintained to records all outstanding store addresses. Each address has an associated counter to track the number of stores that have the same store address. After the store is issued, a search through the SQ is carried out. Upon a hit, the counter of the matched store address is incremented by 1. In case of a miss, the new address is inserted into the SQ with the counter initiated to 1. Meanwhile, the total unique store addresses and the total outstanding stores are counted after each issued store. The total number of unique store addresses is equal to the size of the SQ, while the outstanding stores are the summation of the counter for each store address in the SQ. These two numbers indicate the SQ size required for recording only the unique store addresses or all individual stores in the SQ. Once a store is committed, the counter associated with the store address in the SQ is decremented by 1. The address is removed from the

SQ when the counter reaches to 0. When a branch mis-prediction occurs, all stores younger than the branch are removed from the SQ.

The resulting curves reveal two important messages. First, there is a substantial difference between the two accumulated curves. For example, given a SQ size of 64, 95% of the stores can insert their addresses into the SQ if no duplicated store address is allowed in the SQ. On the other hand, only 75% of the issued stores can find an empty slot in the SQ if all stores regardless their addresses must be recorded in the SQ. Insufficient SQ space causes pipeline flush and degrades the overall IPC. Second, with 512-entry instruction window, the SQ must be large enough to avoid constant overflow. For example, to cover 99% of the stores, the per-store based SQ needs 195 entries, while the per-address based SQ requires 120 entries. The required large CAM-based SQ increases the design complexity, access delay, and power dissipation.

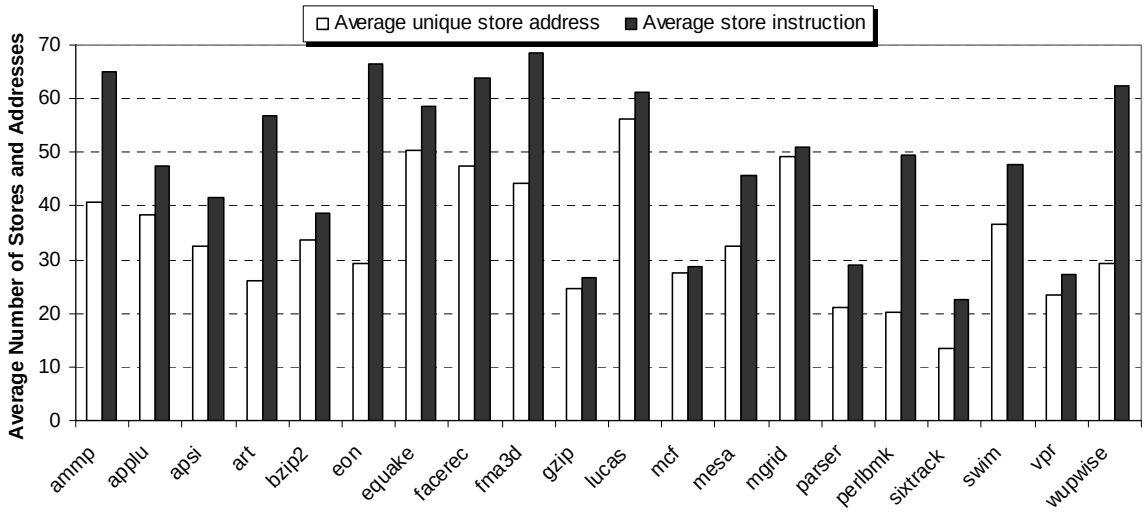


Figure 5-2. The average number of stores and unique store addresses

Figure 5-2 shows the average number of stores and unique store addresses for each application throughout the simulation. On average, the number of stores is considerably larger than the number of store addresses by 30%. Among the applications, only *Gzip*, *Mcf* and *Mgrid*

have rather small difference. The figures imply there are better SQ solutions which records unique store addresses for store-load forwarding.

Figure 5-3 shows the mismatches between the latest store in execution order and the last store in program order when dependent load is issued. To collect the mismatch information, we simulated two 64-entry CAM-based SQs. One is ordered by the execution order for detecting the latest matched store, and the other is ordered by program order for detecting the last matched store. Note that we consider the latest matches the last if the parent store address is unavailable. The mismatch is very significant. On average, about 25% of the forwarded latest stores are incorrect and cause costly re-executions.

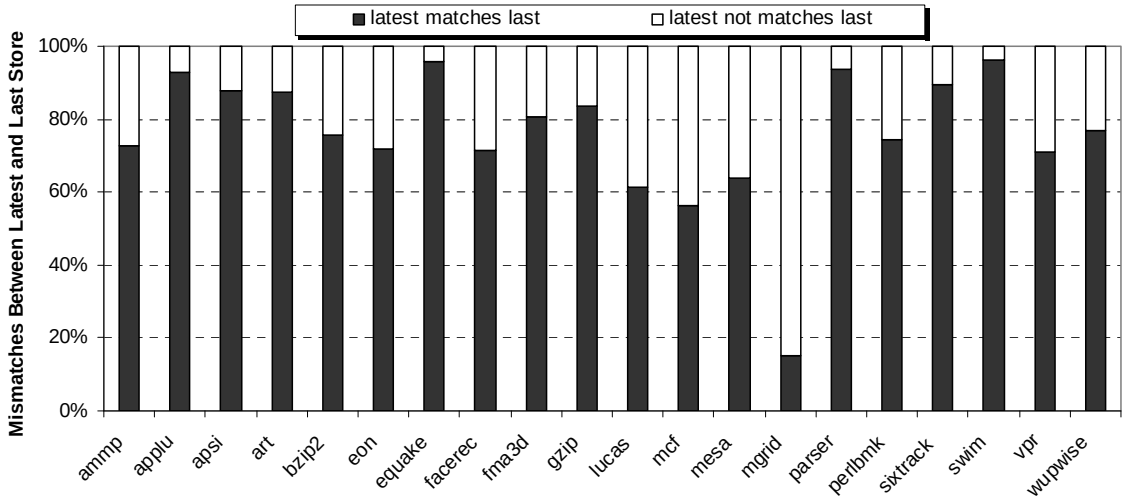


Figure 5-3. Mismatches between the latest and the last store for dependent load

5.3 Order-Free SQ Directory with Age-Order Vectors

In this section, we first describe the basic design of the decoupled SQ without considering data alignment. The basic scheme is then extended to handle partial stores and loads that are commonly defined in many instruction set architectures. Handling misaligned stores and loads that cross the 8-byte boundary is presented at the end.

5.3.1 Basic Design without Data Alignment

The decoupled SQ consists of three key components, the *SQ RAM* array, the *SQ directory*, and the corresponding *age-order vector* as shown in Figure 5-4. The *SQ RAM* buffers the address and data for each store until the store commit. Similar to the conventional SQ, a store enters the SQ RAM when it is renamed and is removed after commit in program order. The store data in each SQ RAM entry is 8-byte wide with eight associated ready bits to indicate the readiness of the respective byte for forwarding. The SQ RAM is organized as a circular queue with two pointers where the *head* points to the oldest store and the *next* points to the next available location. When a store is renamed, the next location in the SQ RAM is reserved.

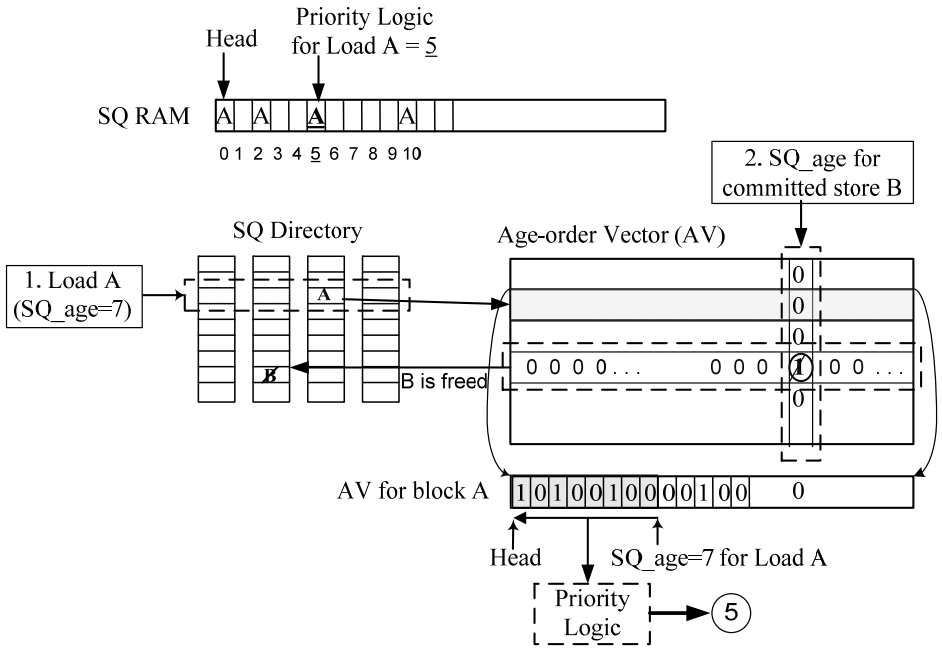


Figure 5-4. SQ with decoupled address matching and age priority

The *age* of the store is defined by its position in the SQ RAM (denoted as *SQ_age*), which is saved in the ROB. Upon a store commit, the *SQ_age* can locate the store from the SQ RAM for putting away the data into the cache. The pipeline stalls when the SQ RAM is full. Since the SQ directory is *decoupled* from the SQ RAM, the SQ RAM size is independent of the SQ directory size and imposes minimum impact on searching for the parent store. When a load is

renamed, the *SQ_age* of the last store in the SQ is recorded along with the load in the LQ. When the load is issued later, the *SQ_age* is used for searching the parent store

Organized similar to the conventional cache directory, the *SQ directory* records the store addresses for matching dependent load address in store-load forwarding. Instead of keeping a directory entry for every individual store in the SQ RAM, the SQ directory records a single address for all outstanding stores with the same address. Since a store is recorded in the SQ directory after the store has been issued and its address is available, the SQ directory can be partitioned into multiple banks (sets) based on the store address to avoid fully-associative searches.

The *age-order vector* is a new structure that provides the ordered SQ RAM locations for all stores with the same address. Each address entry in the SQ directory has an associated age-order vector. The width of the vector is equal to the size of the SQ RAM. When a store address is available, the SQ directory is searched for recording the new store. If there is a match, the bit in the corresponding age-order vector indexed by the *SQ_age* of the current store is set to indicate its location in the SQ RAM. If the store address is not already in the SQ directory, a new entry is created. The corresponding bit in the age-order vector is turned on in the same way as when a match is found. In case there is no empty entry in the SQ directory, the store is simply dropped and cannot be recorded in the SQ directory. Consequently, the dependent load may not see the store and causes re-execution. Since stores are always recorded in program order in the SQ RAM, imprecision in recording the in-flight store addresses in the directory is allowed.

When a load is issued, a search through the SQ directory determines proper store-load forwarding. If there is no hit, the load proceeds to access the cache. If there is a matched store address in the SQ directory, the corresponding age-order vector is scanned to locate the youngest

older store for the load. The search starts from the first bit defined by the *SQ_age* of the load and ends with the "head" of the SQ RAM. A simple leading-one detector is used to find the closest (youngest) location where the bit is set indicating the location of the parent store.

Two enhancements are considered in shortening the critical timing in searching for the parent store. First, the well-known way-prediction technique enables the SQ directory lookup and the targeted age-order vector scanning in parallel. By establishing a small but accurate way history table, the targeted age-order vector can be predicted and scanned before the directory lookup result comes out. Second, the delay of the leading-one detector is logarithmically proportional to the width of the age-order vector. Given the fact that a majority parents can be located without searching the entire SQ RAM, only searching within a partial age-order vector starting from the *SG_age* may be enough to catch the correct parent store. The accuracy of these enhancements will be evaluated in Section 5.4.

When a store commits, its *SQ_age* is used to update the SQ directory and the age-order vector. The bit position of all age vectors pointed by the *SQ_age* is reset. When all bits in a vector are reset, the entry in the SQ directory is freed. The store also retires from the SQ RAM. When a mis-predicted branch occurs, all stores after the branch can be removed from the SQ in a similar way. The last *SQ_age* is saved in the ROB with a branch when the branch is renamed. When a mis-prediction is detected, all entries younger than *SQ_age* in the SQ RAM are emptied. Meanwhile, all "columns" in all age-order vectors that correspond to the removed entries from the SQ RAM are reset. When all bits in any age-order vector are reset, the corresponding entry in the SQ directory is freed.

A simple example is illustrated in Figure 5-4. Assume that a sequence of memory stores are dispatched and recorded in the SQ RAM. Among them, the first, the third, the sixth, and the

eleventh stores have the same address *A*. These four requests may be issued out of order, but eventually are recorded in the SQ directory and the associated age-order vector. Since all four stores have the same address *A*, they only occupy a single entry in the SQ directory indexed by certain lower-order bits of *A*. The corresponding age-order vector records the locations of the four stores in the SQ RAM by setting the corresponding bits as shown in the figure. Assuming a *load A* is finally issued with the *SQ_age* of 7 as indicated in box 1, it finds an entry with matched address in the SQ directory. The priority logic uses the *SQ_age* of the load and the age-ordered vector associated with A to locate the parent store at location 5 in the SQ RAM. In this figure, we also illustrate an example when store B commits as indicated in box 2. The *SQ_age* of B from the ROB is used to reset the corresponding position in all age-order vectors. The address of B can be freed from the SQ directory when the age-order vector contains all zeros. Given the fact that each store in the SQ RAM cannot have two addresses, at most one vector can have '1' in the *SQ_age* position. Therefore, at most one directory entry can be freed for each committed store.

5.3.2 Handling Partial Store/Load with Mask

In the basic design, we assume loads and stores are always aligned within the 8-byte boundary and access the entire 8 bytes every time. Realistically, partial loads/stores are commonly encountered. The address of partial loads and stores is always aligned in the 8-byte boundary. An 8-bit mask is used to indicate the precise accessed bytes. The decoupled order-free SQ can be extended to handle memory dependence detection and forwarding for partial stores/loads. The age-order vector for each store address in the SQ directory is expanded to 8 vectors; each covers one mask bit in the 8-bit mask. If a load address matches a store address in the SQ directory, the leading-1 detector finds the youngest stores older than the load for each individual valid mask bit of the load. In other words, each age-order vector identifies the parent store for each byte of the load. If the found youngest older store covers all of the bytes of the

load, store-load forwarding is detected. Otherwise, unless forwarding from multiple stores are permitted, the load cannot proceed until the youngest store which updates a subset of the load bytes commits and puts the store data away into the cache.

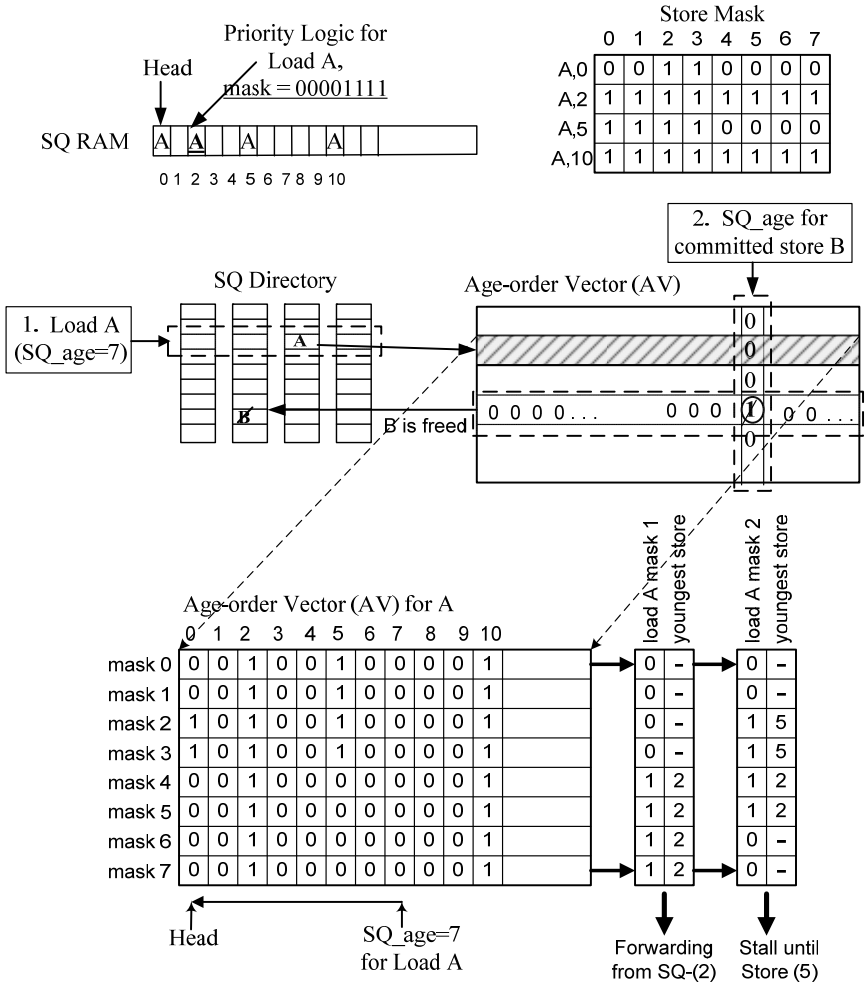


Figure 5-5. Decoupled SQ with Partial Store/load using Mask

In Figure 5-5, the example from Figure 5-4 is modified to illustrate how the decoupled SQ works with partial loads and stores. Now, the first (A,0) and the sixth (A,5) stores are partial stores, where (A,0) stores bytes 2 and 3, and (A,5) stores bytes 0, 1, 2 and 3 as indicated by the 8-bit masks. Assume that a partial Load A with SQ_age=7 is issued. To illustrate forwarding detection, we assume the load issued twice; one with a mask of '00001111' and the other with '00111100' for loading bytes 4, 5, 6, and 7, and bytes 2, 3, 4, and 5, respectively. A matched

address A is found in the SQ directory for the load. The subsequent searches in the corresponding age-order vectors find the youngest older store is (A,2) that covers all the bytes for the first load. As a result, the load gets the forwarded data from the SQ RAM. For the second load, however, it has two parent stores, where (A,2) produces bytes 4 and 5, and (A,5) produces bytes 2 and 3 for the load. Unless a merge from multiple stores is permitted for forwarding, the second load will be stalled until (A,5) is retired.

When a store is retired, the SQ directory and the age-order vectors are updated the same way as that without the mask. However, given the fact that each store may store only partial bytes, instead of detecting zeros on a single age-order vector, all eight age-order vectors that are associated with the 8 mask bits must be all zeroed before the corresponding directory entry can be freed.

5.3.3 Memory Dependence Detection for Stores/Loads Across 8-byte Boundary

Intel's x86 ISA permits individual store or load to cross an 8-byte aligned boundary, referred as a *misaligned* store or load. Misaligned loads and stores complicate the memory dependence detection and data forwarding. When a misaligned load is issued, a simple but costly solution is to stall until all stores ahead of it are drained from the SQ. To avoid this costly wait for those misaligned loads that are independent of any outstanding store, duplicated full CAMs for SQ are implemented for detecting the dependences as described in [Abramson et al. 1997]. When a misaligned store is issued, its starting aligned address and the byte mask are stored in the SQ along with an additional *overflow* bit to indicate that the store spills out of range. In this case, a load falling into the next 8-byte range of the misaligned store will miss this store during the search if there is only one CAM. Therefore, two SQ CAMs are needed for searching both the address of the load *(load)* and the adjacent lower 8-byte address *(load-8)* in parallel. The decremented load-address CAM will match the store address, and the overflow bit will indicate

to the forwarding logic that there is a misaligned hit. The load is stalled until the misaligned store is retired and the data is stored into the cache. For handling misaligned loads, a third SQ CAM is needed in order to provide parallel searches of the next higher 8-byte address *(load+8)* for potential misaligned hits.

The banked (set-associative) order-free SQ directory provides another distinct advantage in detecting misaligned store/load dependences. By using the lower-order bits from the aligned address to select the bank (set), all three addresses *(load-8), (load),* and *(load+8)* of a misaligned load are likely to be located in a different bank. Hence, the costly duplications of the CAM for parallel searches can be avoided.

When Load A is issued, there are several cases in dependence detections and data forwarding. If Load A does not cross the 8-byte boundary, two searches for store A and store A-8 from the SQ directory are carried out. If A hits but A-8 misses, or both A and A-8 hit but the store A-8 does not cross the 8-byte boundary, a search of the youngest parent of store A for Load A can follow the algorithm described in Section 5.3.2. If A-8 hits and the store A-8 crosses the 8-byte boundary, a misaligned hit is detected. Load A must stall until the store A-8 ahead of the load is retired and put the data away into the cache. In case that A also hits, Load A is stalled until both of the stores ahead of the load retire and their data is stored in to cache. If none of the above conditions is true, load A proceeds to access the cache.

If Load A crosses the 8-byte boundary, three searches for store A, store A-8, and store A+8 from the SQ directory must be performed. Any hit of A, A+8, or a hit of A-8 with overflow bit set in the SQ directory indicates a misaligned hit. Load A is stalled until all of these stores ahead it retire and their data is stored into the cache. If the above condition is not true, load A proceeds to access the cache.

5.4 Performance Results

We modified the PTLsim simulator to model a cycle accurate full system x86-64 microprocessor. We followed the basic PTLsim pipeline design, which has 13 stages (1 fetch, 1 rename, 5 frontend, 1 dispatch, 1 issue, 1 execution, 1 transfer, 1 writeback, 1 commit). Note that the 5-cycle "frontend" stages are functionless and were inserted for more closely model realistic pipeline delays in the front-end stages. In this pipeline design, there are 7 cycles between a store entering SQ RAM in the rename stage and entering SQ directory in the issue stage. When any memory dependence violation is detected for an early load, the penalty of re-dispatching the replayed load is 2 cycles (from dispatch to execution). Any uops after this load in program order are also re-dispatched. To reduce the memory dependence violations, a Load Store Alias Predictor (LSAP) is added [Chrysos and Emmer 1998]. This fully-associative 16-entry LSAP records the loads that were mispredicted in the recent history. In case there is any unresolved store address ahead of a load in the SQ when the load is issued, the LSAP is looked up. If a match is found, the load is delayed until the unresolved store address is resolved. Similar method of memory aliasing prediction is used by Alpha 21264 [Kessler 1999].

The x86 architecture is known for its relatively widespread use of unaligned memory operations. In PTLsim, once a given load or store is known to have a misaligned address, it is preemptively split into two aligned loads or stores at the decode time. PTLsim does this by initially causing all misaligned loads and stores to raise an internal exception that forces a pipeline flush. At this point, the special misaligned bit is set for the problem load or store in PTLsim's internal translated basic block representation. When the offending load or store is encountered again, it will be split into two aligned loads or stores early in the pipeline. The split loads and stores will be merged later in the commit stage. In our simulation, we followed the PTLsim in handling misaligned loads and stores. The simulation results of all applications show

that pipeline flushes due to misaligned stores and loads happen very infrequently with only about 0.5% of the total stores and loads.

To gain inside on the impact of various store queue implementations, we stretch other out-of-order pipeline parameters [Akkary et al. 2003] as summarized in Table 1-2. Other detailed parameter setting can be found in the original PTLsim source code. SPEC 2000 integer and floating-point applications are used to drive the performance evaluation. We skip the initialization part of the workloads and collect statistics from the next 200 million instructions.

Four SQ designs are evaluated and compared including the traditional full CAM, the latest-store forwarding, the late-binding SQ, and the decoupled SQ. We evaluated a 32- and a 64-entry full CAMs denoted as *Conventional 32-CAM* and *Conventional 64-CAM*; latest-store forwarding using a 64-entry full CAM denoted as *LS 64-CAM*; late-binding SQ with a 4×8 directory and a 8×8 directory recording address/age pair together with a 64-entry SQ RAM denoted as *LB 4×8 64-RAM* and *LB 8×8 64-RAM*; and lastly the decoupled SQ with a 4×8 directory and a 8×8 directory also with a 64-entry SQ RAM denoted as *Decoupled 4×8 64-RAM* and *Decoupled 8×8 64-RAM*. The notation $a \times b$ represents the configuration of the SQ directory, where a is the number of sets and b is the set associativity. In the decoupled SQ, we simulated a 48-bit leading-1 detector in finding the youngest parent. We also implemented a way predictor with a 256-entry prediction table using a total of 96 bytes. When a load is issued, the way is predicted. If a misprediction is detected from the address comparison, the way prediction table is updated and the load is re-issued with a 2-cycle delay. We do not consider forwarding from multiple stores. We simplified the latest-store and late-binding forwarding schemes. A full CAM is used for the latest-store forwarding. Instead of maintaining stores in program order as in a conventional full CAM, stores are maintained in execution order for searching the latest store. We do not

implement the Bloom Filter and other techniques in evaluating the late-binding scheme. If the SQ directory is full, a store is simply dropped without recording it in the directory.

5.4.1 IPC Comparison

Figure 5-6 shows the IPC comparison of the seven SQ designs running the SPEC2000 programs. We can make a few observations. First, the decoupled SQ outperforms both the latest-store forwarding and the late-binding SQ by a sizeable margin for most applications. On average, *Decoupled 4×8 64-RAM* and *Decoupled 8×8 64-RAM* outperform *LS 64-CAM* by 20.8% and 23.8% respectively. With an equal directory size of *4×8* and *8×8*, they outperform the late-binding counterparts by 12.4% and 7.6%. Given the fact that the decoupled SQ records one address for all outstanding stores with the same address, it tolerates a smaller *4×8* SQ directory much better than the late-binding scheme. In fact, the decoupled SQ with a bigger *8×8* directory only improves the IPC by about 2.4% comparing with that using a *4×8* directory.

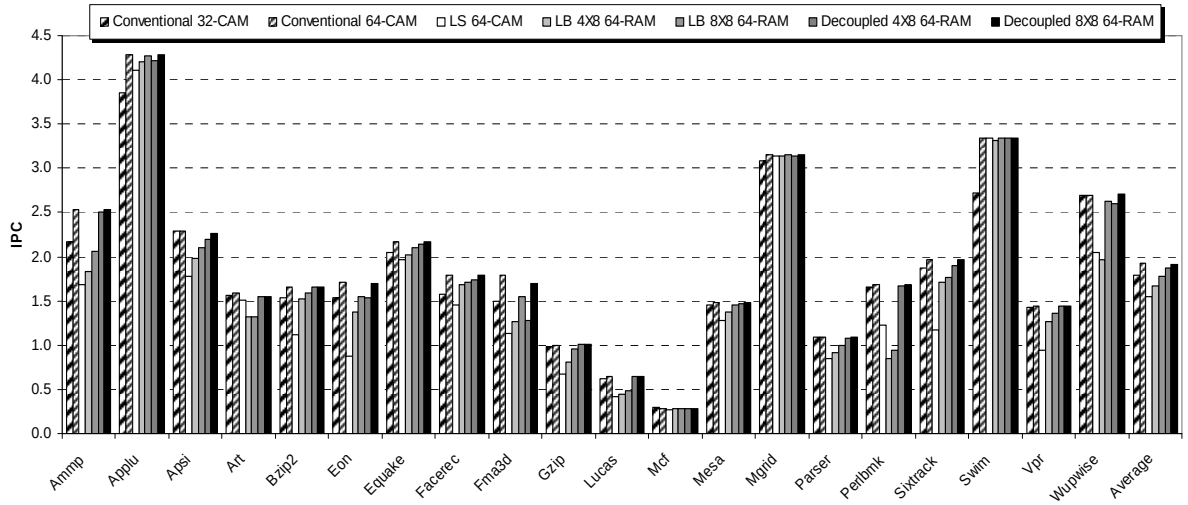


Figure 5-6. IPC comparison

Second, *Decoupled 4×8 64-RAM* shows better performance than *Conventional 32-CAM* by an average of 4.2%. Since the SQ directory is decoupled from the SQ RAM, a 64-entry SQ RAM allows more outstanding stores than that of a 32-entry CAM without requiring any bigger

directory for matching the store addresses. Besides the same number of directory entries as in *Conventional 32-CAM*, *Decoupled 4×8 64-RAM* achieves better IPCs using only an 8-way comparator for faster speed and lower power consumption.

Third, even if the SQ RAM size is no larger than full-CAM size, the performance of *Decoupled 8×8 64-RAM* with banked directory is close to the expensive *Conventional 64-CAM* performance. On average, *Decoupled 8×8 64-RAM* degrades less than 0.5% of the IPC compared with *Conventional 64-CAM*.

Fourth, *Applu*, *Mcf*, *Mgrid*, and *Swim* are less sensitive to different SQ designs except for *Conventional 32-CAM*, because only a very small number of store-load forwarding exists in these applications regardless the SQ designs. Table 5-1 summarizes the percentage of loads get forwarded data from stores in an ideal SQ for all the simulated applications. An ideal SQ has infinite size and all stores addresses before a load are known when the load is issued. *Conventional 32-CAM* shows worse performance in these applications because its smaller CAM size hinders stores from being renamed. Among all the loads that can be forwarded in an ideal SQ, the seven simulated SQ designs can correctly capture 83.8%, 97.2%, 77.6%, 87.1%, 93.0%, 95.0%, and 96.4% of the ideal forwarded loads respectively.

Table 5-1. Percentage of forwarded load using an ideal SQ

Workload	Ammp	Applu	Apsi	Art	Bzip2	Eon	Equake	Facerec	Fma3d	Gzip
Forward %	12.7%	0.5%	8.4%	4.7%	13.0%	18.4%	5.4%	6.8%	23.2%	4.3%
Workload	Lucas	Mcf	Mesa	Mgrid	Parser	Perlbmk	Sixtrack	Swim	Vpr	Wupwise
Forward %	7.5%	0.3%	4.7%	0.1%	8.8%	14.8%	8.9%	0.0%	15.6%	7.4%

By getting rid of the duplicated stores with the same address in the SQ directory, the decoupled SQ requires smaller directory than that of the late-binding SQ. Figure 5-7 plots the total number of dropped stores from entering the SQ directory per 10K instructions due to a full SQ directory. The huge gaps in terms of dropped stores between the late-binding and the

decoupled SQ are very evident. Within each method, 8×8 directory causes much less dropped stores than 4×8 directory.

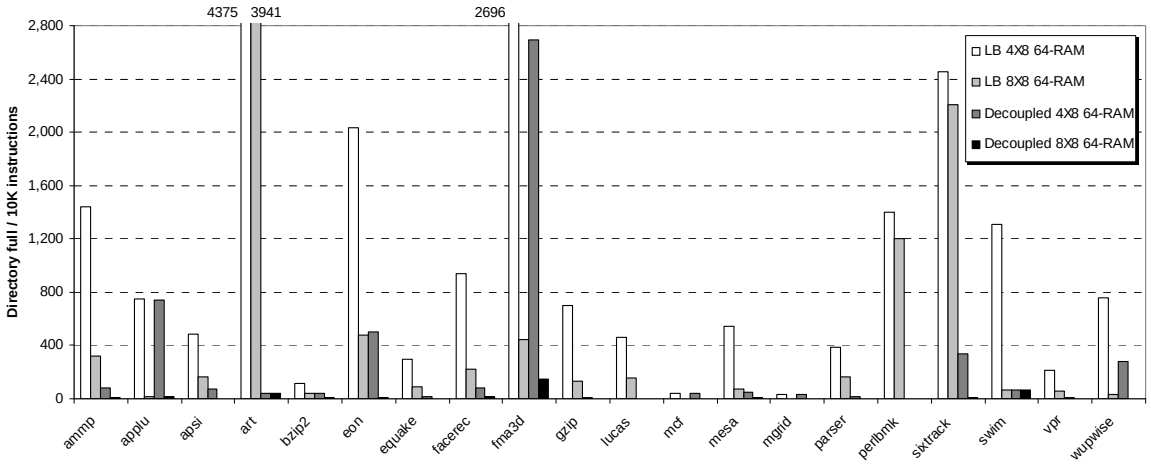


Figure 5-7. Comparison of directory full in decoupled SQ and late-binding SQ

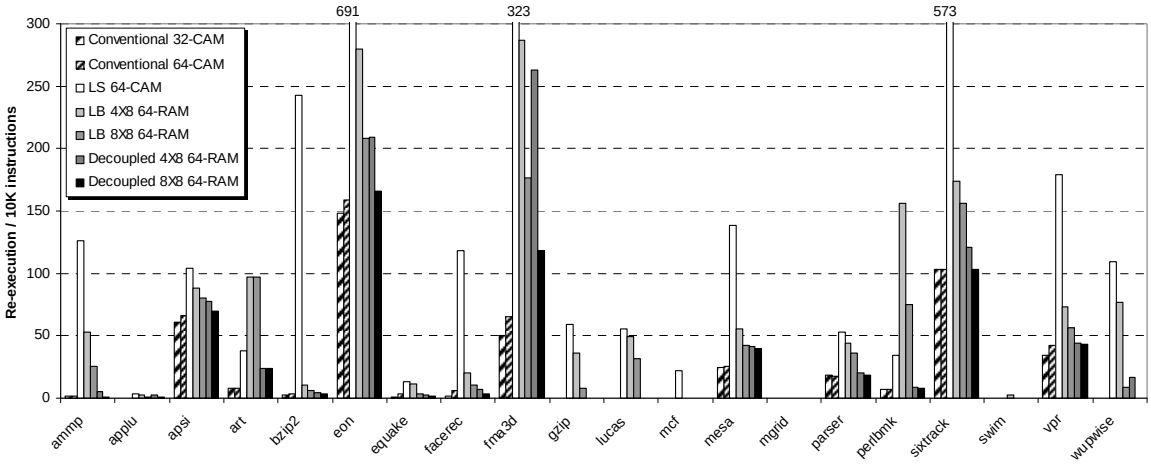


Figure 5-8. Comparison of load re-execution

There are various memory dependence mis-speculations which require loads to be re-executed: lack of the parent store address, dropped stores due to full SQ directory, incorrectly identifying the youngest older store, etc. Figure 5-8 summarizes the total number of re-executed loads per 10K instructions due to mis-speculations of store-load dependences. As expected, the latest store SQ scheme causes the most number of re-execution, while conventional CAM schemes cause the least. Note that *Conventional 32-CAM* causes less re-executions than that of

Conventional 64-CAM because smaller *32-CAM* causes more stalls at the rename stage in such a way that it can be considered as putting a limit in out-of-order execution of stores and loads.

5.4.2 Sensitivity of the SQ Directory

The size and configuration of the decoupled SQ directory is flexible. In Figure 5-9, we compare four SQ directory sizes with *2×8*, *4×8*, *4×12*, and 8×*8* configurations that are decoupled from a 64-entry SQ RAM using the average IPC of all twenty applications. Note that we increase the set associativity to 12 for the directory with 48 entries to simplify the set selection. We also run full-CAM with 16, 32, 48, and 64 entries.

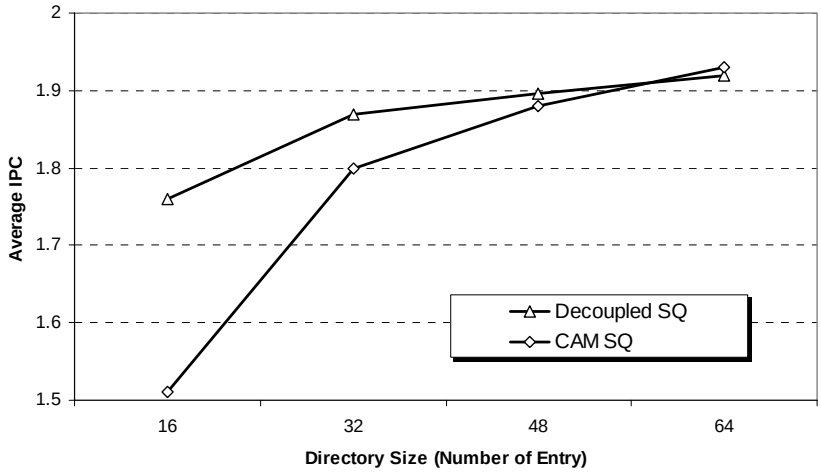


Figure 5-9. Sensitivity on the SQ directory size

As shown in the figure, the smaller directories of *2×8* and *4×8*, with a 64-entry SQ RAM perform much better than the full-CAM counterparts of equal directory sizes by 17% and 4% respectively. The advantage of the decoupled SQ is obvious since it is much cheaper to enlarge the circular SQ RAM while keeping the associative directory small. When the directory increases to 48, the performance gap becomes very narrow due to the smaller size difference between the CAM and the RAM. Even with a full 64-entry CAM that matches the SQ RAM size, the decoupled SQ only loses by less than 0.5% of the overall IPC.

The age distance from the load to its parent affects the timing of the lead-1 detector. Figure 5-10 shows the accumulated distribution of the searching distances. With the distance of 32 that is half of the total distance of 64, 15 out of 20 applications can locate almost 100% of their parents. 3 of the remaining 5 also achieve 97-99% of the accuracy. When the search distance increases to 48, all but Fma3d can find 100% of their parents. Fma3d can also find 98.3% of its parents at this distance. In all our simulations, the search distance is set to 48.

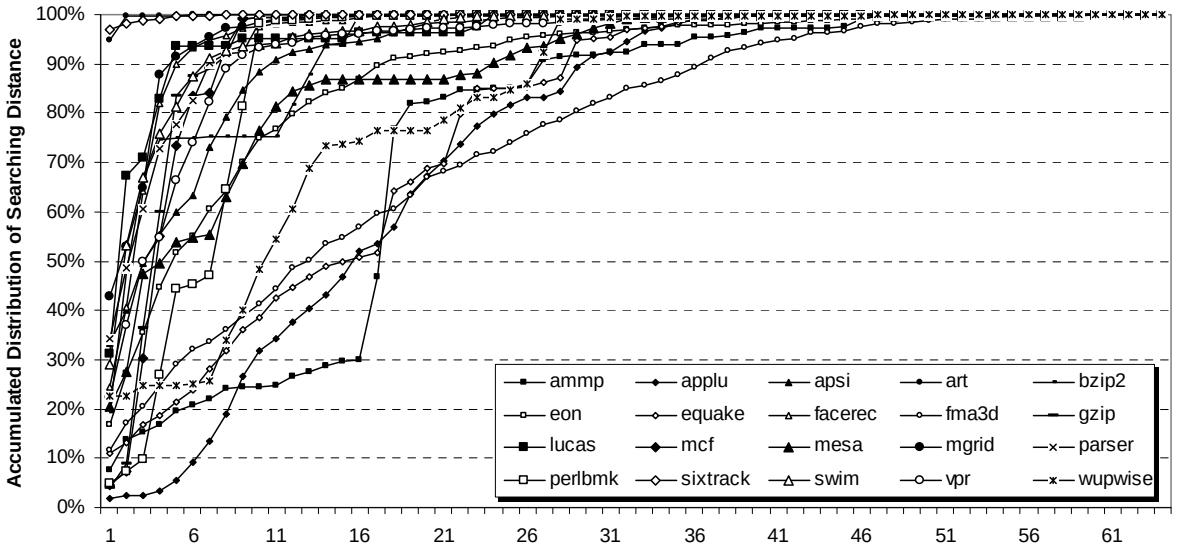


Figure 5-10. Sensitivity on the leading-1 detector width

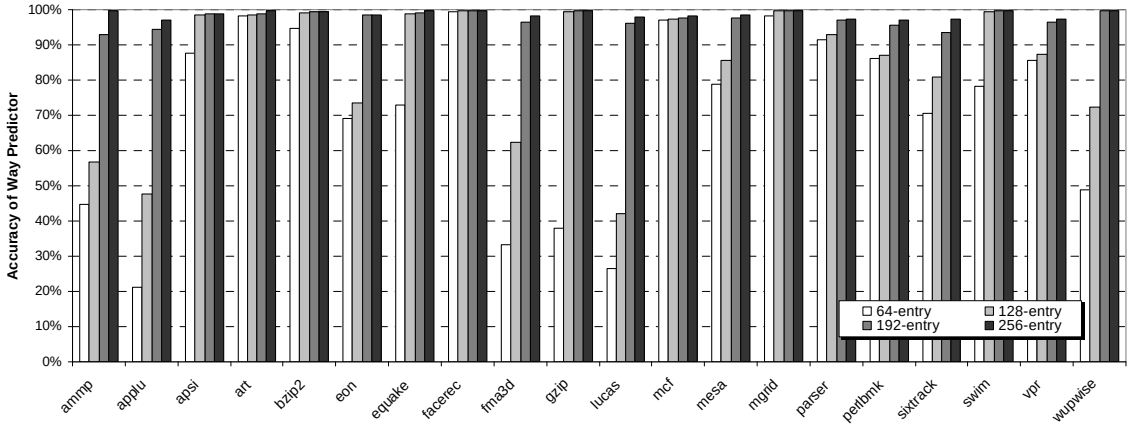


Figure 5-11. Sensitivity on way prediction accuracy

To predict the way in a SQ directory set where the parent store is located, a 256-entry way-prediction table is maintained that reaches close to 100% accuracy for a majority of the applications as shown in Figure 5-11. The prediction table is indexed by the lower-order 8 bits of load and store addresses aligned in the 8-byte boundary. Each entry has 3-bit to record which way is touched by store/load most recently. The total size of the table is only 96 bytes. When a load is issued, the way ID is fetched out of the way prediction table to access the age-order vectors for searching the parent store. When a mis-prediction is identified, the correct way is updated and the load is recycled with a 2-cycle delay. From Figure 5-11, the way prediction accuracy is proportional to the table size. The tables with 64 or 128 entries show very poor prediction accuracy. Significant improvement is observed with 192 and 256 entries. All simulations are based on an 8×8 decoupled SQ directory.

5.5 Related Work

There have been many recent proposals in designing a scalable SQ by getting rid of the expensive and time-consuming full-CAM design. One category is to adopt a two-level SQ which has a smaller first-level SQ to enable fast and energy efficient forwarding and a much larger second-level SQ to correct and complement the first-level SQ [Roth 2004; Srinivasan et al. 2004; Torres et al. 2005]. In general, the first-level SQ records the latest executed stores. A store to load forwarding occurs when the load address matches a store address in the first-level store queue. Accessed after a first-level miss, the bigger and slower second-level SQ records those stores that cannot fit into the first-level SQ based on FIFO or some prediction mechanisms. Instead of a first-level SQ, a store-forwarding cache is implemented in [Stone et al. 2005] for store-load forwarding. It relies on a separate memory disambiguation table to resolve any dependence violation. In [Garg et al. 2006], a much larger L0 cache is used to replace the first-level SQ for caching the latest store data. A load, upon a hit, can fetch the data directly from the

L0. In this approach, instead of maintaining speculative load information, the load is also executed in the in-order pipeline fashion. An inconsistent between the data in L0 and L1 can identify memory dependence violations. Besides the complexity and extra space, a fundamental issue in this category of approaches is the heavy mismatch between the *latest* store and the correct *last* store for dependent load as reported in Section 5.2. Such mismatches produce costly re-executions.

The late-binding technique enables an un-ordered SQ as reported in [Sethumadhavan et al. 2007]. A load or a store enters the SQ when the instruction is issued, instead of renamed. In order to get correct store-load forwarding, the age of each load/store is also recorded along with the address. In case when there are multiple hits to the SQ for an issued load, complicated decoding logic can re-create the order of the stores based on the recorded age. Afterwards, the search for the youngest matched store that is older than the load can locate the correct parent store for forwarding. The late-binding avoids full-CAM search and allows small bank implementation for the SQ. However, it records the address/age pair for every memory instruction unnecessarily that requires more directory entries and intensifies the banking conflicts. Furthermore, it relies on complicated age-based priority logic to locate the parent that may lengthen the access time for store-load forwarding.

Another category of solution is also prediction based for efficiently implement or even to get rid of the SQ entirely. A bloom filter [Sethumadhavan et al. 2003] or a predicted forwarding technique [Park et al. 2003] can filter out a majority of SQ searches. A sizeable memory dependence predictor is used in [Sha et al. 2005] to match loads with the precise store buffer slots for the forwarded data so as to abandon the associative SQ structure. Proposals in [Sha et al. 2006; Subramaniam and Loh 2006] can completely eliminate the SQ by bypassing store data

through the register file or through the LQ. However, these prediction based approaches are always accompanied with the expense of an extra sizeable prediction table.

5.6 Conclusion

When the instruction window scales up to hundreds or even thousands of instructions, it requires an efficient SQ design to timely detect memory dependences and forward the store data to dependent loads. Although there have been many proposed scalable SQ solutions, they generally suffer certain inaccuracy and inefficiency, require complicated hardware logic along with additional buffers or tables, and compromise the performance. In this work, we propose a new scheme that decouples the address matching unit and the age-based priority logic from the SQ RAM array. The proposed solution enables an efficient set-associative SQ directory for searching the parent store using a detached age-order vector. Moreover, by recording a single address for multiple stores with the same address in the SQ directory, the decoupled SQ further reduces the directory size and alleviates potential bank conflicts. We also provide solutions in handling the commonly used partial and misaligned stores and loads in designing a scalable SQ. The performance evaluation shows that the new scheme outperforms other scalable SQ proposals based on latest-store forwarding and late SQ binding techniques and is comparable with full-CAM SQ. By removing the costly fully-associative CAM structure, the new scheme is both power-efficient and scaling to large window design.

CHAPTER 6
CONCLUSIONS

In this dissertation, we propose three works related to cache performance improvement and resource contention resolution, and one work related to LSQ design.

The proposed special P-load has demonstrated its ability to effectively overlap load- load data dependences. Instead of relying on miss predictions of the requested blocks, the execution-driven P-load precisely instructs the memory controller in fetching the needed data block non-speculatively. The simulation results demonstrate high accuracy and significant speedups using the P-load.

The elbow cache has demonstrated its ability to expand the replacement set beyond the lookup set boundary without adding any complexity on the lookup path. Because of the characteristics of elbow cache, it is difficult to implement recency-based replacement. The proposed reuse-count replacement policy with low-cost can achieve cache performance comparable to the recency-based replacement policy.

Simultaneous Multithreading techniques have been moved from laboratory ideas into real and commercially successful processors. However, studies have shown that without proper mechanisms to regulate the shared resources, especially shared caches and the instruction window, multiple threads show lower overall performance when running simultaneously. Runahead execution, proposed initially for achieving better performance for single-thread applications, works very well in the multiple-thread environment. In runahead execution, multiple long-latency memory operations can be discovered and overlapped to exploit the memory-level parallelism; meanwhile, shared critical recourses held by the stalling thread can be released to keep the other thread running smoothly to exploit the thread-level parallelism.

The order-free SQ design decouples the address matching unit and the age-based priority logic from the original store queue. The proposed solution enables an efficient set-associative SQ directory for searching the parent store using a detached age-order vector. Moreover, by recording a single address for multiple stores with the same address in the SQ directory, the decoupled SQ further reduces the directory size and alleviates potential bank conflicts. We also provide solutions in handling the commonly used partial and misaligned stores and loads in designing a scalable SQ. The performance evaluation shows that the new scheme outperforms other scalable SQ proposals based on latest-store forwarding and late SQ binding techniques and is comparable with full-CAM SQ. By removing the costly fully-associative CAM structure, the new scheme is both power-efficient and scaling to large-window design.

LIST OF REFERENCES

Abramson, J. M., Akkary, H., Glew, A. F., Hinton, G. J., Konigsfeld, K. G., Madland, P. D., and Papworth, D. B. 1997. Method and Apparatus for Signalling a Store Buffer to Output Buffered Store Data for a Load Operation on an Out-of-Order Execution Computer System, Intel, *US Patent 5606670*.

Adams, D., Allen, A., Bergkvist, R., Hesson, J., and LeBlanc, J. 1997. A 5ns Store Barrier Cache with Dynamic Prediction of Load/Store Conflicts in Superscalar Processors, *Proceedings of the 1997 International Solid-State Circuits Conference*, 414–415, 496.

Agarwal, A., Hennessy, J., and Horwitz, M. 1988. Cache Performance of Operating System and Multiprogramming Workloads, *ACM Transactions on Computer Systems* 6, 4, 393–431.

Agarwal, A., Kubiatowicz, J., Kranz, D., Lim, B.-H, Yeung, D., D'Souza, G., and Parkin, M. 1993. Sparcle: An Evolutionary Processor Design for Large-Scale Multiprocessors, *IEEE Micro 13, 3, 48*.

Agarwal, A., and Pudar, S.D. 1993a. Column-Associative Caches: a Technique for Reducing the Miss Rate of Direct-Mapped Caches, *Proceedings of the 29th Annual International Symposium on Computer Architecture,* 179–190.

Akkary, H., Pajwar, R. and Srinivasan, S. T. 2003. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors, *Proceedings of the 36th International Conference on Microarchitecture,* 423–434.

Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A., and smith, B. 1990. The Tera Computer System, *Proceedings of the 4th International Conference on Supercomputing,* 1–6.

Baugh, L., and Zilles, C. B. 2006. Decomposing the Load-Store Queue by Function for Power Reduction and Scalability, *IBM Journal of Research and Development* 50, 2-3, 287-298.

Bodin, F., and Seznec, A. 1997. Skewed Associativity Improves Performance and Enhances Predictability, *IEEE Transactions on Computers* 46, 5, 530–544.

Burger, D., and Austin, T. 1997. The SimpleScalar Tool Set, Version 2.0, *Technical Report #1342*, Computer Science Department, University of Wisconsin-Madison.

Buyuktosunoglu, A., Albonesi, D.H., Bose, P., Cook, P., and Schuster, S. 2002. Tradeoffs in Power-Efficient Issue Queue Design, *Proceedings of the 2002 International Symposium on Low Power Electronics and Design,* 184–189.

Cahoon, B., and McKinley, K.S. 2001. Data Flow Analysis for Software Prefetching Linked Data Structures in Java, *Proceedings of the 10th International Conference On Parallel Architectures and Compilation Techniques,* 280–291.

Castro, F., Pinuel, L., Chaver, D., Prieto, M., Huang, M., and Tirado, F. 2006. DMDC: Delayed Memory Dependence Checking through Age-Based Filtering, *Proceedings of the 38th International Symposium on Microarchitecture*, 297–306.

Cazorla, F. J., Fernandez, E., Ramirez, A., and Valero, M. 2003. Improving Memory Latency Aware Fetch Policies for SMT Processors, *Proceedings of the 5th International Symposium on High Performance Computing*, 70–85.

Cazorla, F. J., Ramirez, A., Valero, M., and Fernandez, E. 2004a. DCache Warn: An I-Fetch Policy to Increase SMT Efficiency, *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 74a .

Cazorla, F. J., Ramirez, A., Valero, M., and Fernandez, E. 2004b. Dynamically Controlled Resource Allocation in SMT Processors, *Proceedings of the 37th International Symposium on Microarchitecture*, 171–182.

Charney, M., and Reeves, A. 1995. Generalized Correlation Based Hardware Prefetching, *Technical Report EE-CEG-95-1*, Cornell University.

Chen, T., and Baer, J. 1992. Reducing Memory Latency Via Non-Blocking and Prefetching Caches, *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 51–61.

Chou, Y., Fahs, B., and Abraham, S. 2004. Microarchitecture optimizations for exploiting memory-level parallelism, *Proceedings of the 31st International Symposium on Computer Architecture*, 76–87.

Chrysos, G. Z., and Emer, J. S. 1998. Memory Dependence Prediction Using Store Sets, *Proceedings of the 25th International Symposium on Computer Architecture*, 142-153.

Collins, J., Sair, S., Calder, B. and Tullsen, D. M. 2002. Pointer Cache Assisted Prefetching, *Proceedings of the 35th International Symposium on Microarchitecture*, 62–73.

Cooksey, R., Jourdan, S., and Grunwald, D. 2002. A Stateless, Content-Directed Data Prefetching Mechanism, *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 279–290.

Cristal, A., Santana, O. J., Cazorla, F., Galluzzi, M., Ramirez, T., Pericas, M., and Valero, M. 2005. Kilo-Instruction Processors: Overcoming the Memory Wall, *IEEE Micro* 25, 3, 48–57.

Dorai, G., and Yeung, D. 2002. Transparent Threads: Resource Sharing in SMT Processors for High Single-Thread Performance, *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, 30–41.

Dundas, J. and Mudge, T. 1997. Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss, *Proceedings of the 11th International Conference on Supercomputing*, 68–75.

El-Moursy, A., and Albonesi, D. H. 2003. Front-End Policies for Improved Issue Efficiency in SMT Processors, *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 31–42.

Fu, J., Patel, J.H., and Janssens, B.L. 1992. Stride directed prefetching in scalar processors, *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 102–110.

Gandhi, A., Akkary, H., Rajwar, R., Srinivasan, S. T., and Lai, K. K. 2005. Scalable Load and Store Processing in Latency Tolerant Processors, *Proceedings of the 32nd International Symposium on Computer Architecture*, 446–457.

Garg, A., Rashid, M. W., and Huang, M. C. 2006. Slackened Memory Dependence Enforcement: Combining Opportunistic Forwarding with Decoupled Verification, *Proceedings of the 33rd International Symposium on Computer Architecture*, 142–154.

Gonzalez, A., Valero, M., Topham, N., and Parcerisa, J. 1997. Eliminating Cache Conflict Misses through XOR-Based Placement Functions, *Proceedings of the 11th International Conference of Supercomputing*, 76–83.

Hammond, L., Hubbert, B. A., Siu, M., Prabhu, M. K., Chen, M., and Olukotun, K. 2000. The Stanford Hydra CMP, *IEEE Micro* 20, 2, 71–84.

Hesson, J., LeBlanc, J., and Ciavaglia, S. 1997. Aparatus to dynamically control the out-of-order execution of load-store instructions in a processor capable of dispatching, issuing and executing multiple instructions in a single processor cycle, IBM, *US Patent 5615350*.

Hinton, G., Sager, D., Upton, M., Boggs, D. Kyker, A. and Roussel, P. 2001. The Microarchitecture of the Pentium 4 Processor, *Intel Technology Journal*, 2001.

Hu, Z., Martonosi, M., and Kaxiras, S. 2003. TCP: Tag Correlating Prefetchers, *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 317–328.

Hughes, H. J. and Adve, S. V. 2005. Memory-Side Prefetching for Linked Data Structures for Processor-in-Memory Systems, *Journal of Parallel and Distributed Computing* 65, 4, 448–463.

Joseph, D., and Grunwald, D. 1997. Prefetching Using Markov Predictors, *Proceedings of the 26th International Symposium on Computer Architecture*, 252–263.

Jouppi, N. P. 1990. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers, *Proceedings of the 17th International Symposium on Computer Architecture*, 364–373.

Kessler, R. E. 1999. The Alpha 21264 microprocessor, *IEEE Micro* 19, 2, 24–36.

Kharbutli, M., Irwin, K., Solihin, Y., and Lee, J. 2004. Using Prime Numbers for Cache Indexing to Eliminate Conflict Misses, *Proceedings of the 10th International Symposium on High Performance Computer Architecture,* 288–299.

Kirman, N., Kirman, M., Chaudhuri, M. and Martinez J. F. 2005. Checkpointed Early Load Retirement, *Proceedings of the 11th International Symposium on High Performance Computer Architecture,* 16–27.

Laudon, J., Gupta, A., and Horowitz, M. 1994. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations, *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems,* 308–318.

Lin, W.-F., Reinhardt, S. K., and Burger, D. 2001. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design, *Proceedings of the 7th International Symposium on High Performance Computer Architecture,* 301–312.

Luk C., and Mowry, T. C. 1996. Compiler-Based Prefetching for Recursive Data Structures, *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems,* 222–233.

Moshovos, A., Breach, S. E., Vijaykumar, T. N., Sohi, G. S. 1997. Dynamic Speculation and Synchronization of Data Dependence, *Proceedings of the 24th International Symposium on Computer Architecture,* 181-193.

Mowry, T. C., Lam, M. S., and Gupta, A. 1992. Design and Evaluation of a Compiler Algorithm for Prefetching, *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems,* 62–73.

Mutlu, O., Stark, J., Wilkerson, C., and Patt, Y. 2003. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors, *Proceedings of the 9th International Symposium on High Performance Computer Architecture,* 129–140.

Olden Benchmark, http://www.cs.princeton.edu/~mcc/olden.html.

Opteron Processors, http://www.amd.com.

Park, I., Ooi, C.-L., and Vijaykumar, T.N. 2003. Reducing Design Complexity of the LoadStore Queue, *Proceedings of the 36th International Symposium Microarchitecture,* 411–422.

Peir, J. K., Lee, Y., and Hsu, W. W. 1998. Capturing Dynamic Memory Reference Behavior with Adaptive Cache Topology, *Proceedings of the 8th International Conference on Architectural Support for Programming Language and Operating Systems,* 240–250.

Qureshi, M. K., Thompson, D., and Patt Y. N. 2005. The V-Way Cache: Demand Based Associativity via Global Replacement, *Proceedings of the 32nd Annual International Symposium on Computer Architecture,* 544–555.

Roth, A. 2004. A High-Bandwidth Load-Store Unit for Single- and Multi-Threaded Processors, *Technical Report MS-CIS-04-09*, University of Pennsylvania.

Roth, A., Moshovos, A. and Sohi, G. 1998. Dependence Based Prefetching for Linked Data Structure, *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 115–126.

Saavedra-Barrera, R., Culler, D. and von Eicken, T. 1990. Analysis of Multithreaded Architectures for Parallel Computing, *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, 169–178.

Sair, S. and Charney, M. 2000. Memory Behavior of the SPEC2000 Benchmark Suite, *Technical Report*, IBM Corp.

Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Keckler, S. W., Burger, D., and Moore, C. R. 2003. Exploiting ILP, TLP and DLP with the Polymorphous TRIPS Architecture, *Proceedings of the 30th International Symposium on Computer Architecture*, 422-433.

Sankaralingam, K., Nagarajan, R., McDonald, R., Desikan, R. Drolia, S., Govindan, M., Gratz, P., Gulati, D., Hanson, H., Kim, C., Liu, H., Ranganathan, N., Sethumadhavan, S., Sharif, S., Shivakumar, P., Keckler, S. W., and Burger, D. 2006. Distributed Microarchitectural Protocols in the TRIPS Prototype Processor, *Proceedings of the 39th International Symposium on Microarchitecture*, 480–491.

Sethumadhavan, S., Desikan, R., Burger, D., Moore, C. R., and Keckler, S.W. 2003. Scalable Hardware Memory Disambiguation for High ILP Processors, *Proceedings of the 36th International Symposium on Microarchitecture*, 339-410.

Sethumadhavan, S., McDonald, R., Desikan, R., Burger, D., and Keckler, S.W. 2006. Design and Implementation of the TRIPS Primary Memory System, *Proceedings of the 24th International Conference on Computer Design*, 470–476.

Sethumadhavan, S., Roesner, F., Emer, J. S., Burger, D., and Keckler S. W. 2007. Late-Binding: Enabling Unordered Load-Store Queues, *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 347–357.

Seznec, A. 1993. A Case for Two-Way Skewed-Associative Cache, *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 169–178.

Seznec, A., and Bodin, F. 1993. Skewed-Associative Caches, *Proceedings of the 5th International Conference on Parallel Architectures and Languages Europe*, 304–316.

Sha, T., Martin, M. M. K., and Roth, A. 2005. Scalable Store-Load Forwarding via Store Queue Index Prediction, *Proceedings of the 38th International Symposium on Microarchitecture*, 159–170.

Sha, T., Martin, M. M. K., and Roth, A. 2006. NoSQ: Store-Load Communication without a Store Queue, *Proceedings of the 39th International Symposium on Microarchitecture*, 285–296.

Snavely, A. and Tullsen, D. M. 2000. Symbiotic Job Scheduling for a Simultaneous Multithreading Processor, *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 234–244.

Solihin, Y., Lee, J., and Torrellas, J. 2002. Using a User-Level Memory Thread for Correlation Prefetching, *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 171–182.

SPEC2000 Alpha Binaries from SimpleScalar website, http://www.eecs.umich.edu/~chriswea/benchmarks/spec2000.html.

SPEC2000 Benchmarks, http://www.spec.org/osg/cpu2000/.

Spjuth, M., Karlsson, M., and Hagersten, E. 2005. Skewed Caches from a Low-Power Perspective, *Proceedings of the 2nd Conference on Computing Frontiers*, 152–160.

Spracklen L., and Abraham, S. 2005. Chip Multithreading: Opportunities and Challenges, *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, 248–252.

Srinivasan, S. T., Rajwar, R., Akkary, H., Gandhi, A., and Upton, M. 2004. Continual Flow Pipelines, *Proceedings of the 11th International Symposium on Architectural Support for Programming Languages and Operating Systems*, 107-119.

Stone, H. S. 1971. Parallel Processing with the Perfect Shuffle, *IEEE Trans on Computers* 20, 6, 153–161.

Stone, H. S., Turek, J., and Wolf, J. L. 1992. Optimal Partitioning of Cache Memory, *IEEE Transactions on Computers* 41, 9.

Stone, S. S., Woley, K. M., and Frank, M. I. 2005. Address-Indexed Memory Disambiguation and Store-to-Load Forwarding, *Proceedings of the 38th International Symposium on Microarchitecture*, 171–182.

Subramaniam S. and Loh G. 2006. Store Vectors for Scalable Memory Dependence Prediction and Scheduling, *Proceedings of the12th International Symposium on High Performance Computer Architecture*, 64-75.

Suh, G. E., Rudolph, L., and Devadas, S. 2001. Dynamic Cache Partitioning for Simultaneous Multithreading Systems, *Proceedings of the 13th International Conference on Parallel and Distributed Computing Systems*, 116–127.

Tendler, J. M., Dodson, J. S., Field, J. S. Jr., Le, H., and Sinharoy, B. 2002. POWER4 System Microarchitecture, *IBM Journal of Research and Development* 26, 1, 5–26.

Topham, N. P., and González, A. 1997. Randomized Cache Placement for Eliminating Conflicts, *IEEE Transactions on Computers* 48, 2, 185–192.

Torres, E. F., Ibanez, P., Vinals, V., and Llaberia, J. M. 2005. Store Buffer Design in First-Level Multibanked Data Caches *Proceedings of* the *32nd International Symposium on Computer Architecture*, 469–480.

Tullsen, D. M., and Brown, J. A.2001. Handling Long-Latency Loads in a Simultaneous Multithreading Processor, *Proceedings of the 34th International Symposium on Microarchitecture*, 318–327.

Tullsen, D. M., Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., and Stamm, R.L., 1996. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor, *Proceedings of* the *23rd International Symposium on Computer Architecture*, 191–202.

Tullsen, D. M., Eggers, S.J., and Levy, H.M. 1995. Simultaneous Multithreading: Maximizing On-Chip Parallelism, *Proceedings of the 22nd International Symposium on Computer Architecture*, 392–403.

Vanderwiel, S., and Lilja, D. 2000. Data Prefetch Mechanisms, *ACM Computing Surveys*, 174–199.

Wang, Z., Burger, D., McKinley, K. S., Reinhardt, S. K., and Weems, C. C. 2003. Guided Region Prefetching: a Cooperative Hardware/Software Approach, *Proceedings of the 30th International Symposium on Computer Architecture*, 388–398.

Wilton, S.J.E., and Jouppi, N.P. 1996. CACTI: An Enhanced Cache Access and Cycle Time Model, *IEEE Journal of Solid-State Circuits* 31, 5, 677–688.

Yang, C.-L., and Lebeck, A. R. 2000. Push vs. Pull: Data Movement for Linked Data Structures, *Proceedings of the 14th International Conference on Supercomputing*, 176–186.

Yang, C.-L., and Lebeck, A. R. 2004. Tolerating Memory Latency through Push Prefetching for Pointer-intensive Applications, *ACM Transactions on Architecture and Code Optimization* 1, 4, 445–475.

Yang, Q., and Adina, S. 1994. A One's Complement Cache Memory, *Proceedings of the 1994 International Conference on Parallel Processing*, 250–257.

Yoaz, A., Erez, M., Ronen, R., and Jourdan, S. 1999. Speculation Techniques for Improving Load-Related Instruction Scheduling, *Proceedings 26th Annual International Symposium on Computer Architecture*, 42-53.

Yourst, M. T. 2007. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator, *Proceedings of the 2007 International Symposium on Performance Analysis of Systems & Software*, 23-34.

BIOGRAPHICAL SKETCH

Zhen Yang was born in 1977 in Tianjin, China. She earned her B.S. and M.S. in computer science from Nankai University in 1999 and 2002 respectively. She earned her Ph.D. in computer engineering from the University of Florida in December 2007.