The Dissertation Committee for Lakshminarasimhan Sethumadhavan certifies that this is the approved version of the following dissertation:

# Scalable Hardware Memory Disambiguation

Committee:

Douglas C. Burger, Supervisor

James C. Browne

Joel S. Emer

Stephen W. Keckler

Kathryn S. McKinley

# Scalable Hardware Memory Disambiguation

by

**Lakshminarasimhan Sethumadhavan, B.E., M.S.**

**Dissertation**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Doctor of Philosophy**

**The University of Texas at Austin**

December 2007

Dedicated to my wonderful parents:

Jayashri Raja Rao and Sanjeevi Sethumadhavan

# Acknowledgments

Eight years back, on my graduate school application, I wrote: "I look upon graduate school as formal training in research and teaching. At your university, I would like to hone the skills required to carry out independent research...academia." Looking back, I could not possibly be more satisfied with the training I have received, thanks to my advisors, committee members, collaborators and students at UT.

I thank my advisor Dr. Doug Burger and my co-advisor, Dr. Stephen W. Keckler, for giving me research guidance whenever I needed it, for teaching me architecture and VLSI design, and teaching me how to teach these subjects, for providing valuable professional mentoring at critical junctures, for writing tips and financially supporting me throughout my graduate study. I also thank them for creating an excellent environment for conducting research. I have significantly benefited from their energy, their vision, the architecture seminars they helped organize, the ample simulation bandwidth they provided and from interactions with the talented people they have recruited over the years.

I thank Dr. Kathryn McKinley for teaching a great compiler class, for valuable professional tips, for emphasizing the importance of being nice and demonstrating by example. I thank Dr. James C. Browne for sharing his infinite wisdom on a

wide array of subjects, and for his encouraging words and advice. I thank Dr. Joel Emer for his key comments at crucial times. I thank all of my committee members for taking time from their busy schedule to participate in this dissertation.

One of the best learning experiences in my life so far has been working on the TRIPS hardware prototyping project. During the initial planning phase, I learned a lot about architecture from discussions with TRIPS research fellow, Chuck Moore, thanks Chuck, and during the implementation phase I had the opportunity to work closely with Robert McDonald. I'm indebted to Robert for extensively educating me on RTL design, implementation and verification. Of course, a project of this magnitude would not be possible without teamwork. I thank the team members for the camaraderie and hard work over many years.

My work has built upon the work done by several researchers in the CART group. Thanks to Raj Desikan and Hrishi Murukkathampoondi for the sim-alpha simulator and extensions, thanks to Karu Sankaralingam and Ramdas Nagarajan for the initial GPA simulator, thanks to all of the TRIPS team for the TRIPS prototype simulator, thanks to Changkyu Kim, Haiming Liu and Nitya Ranganathan for their contributions to the TFlex simulator, thanks to Franzi Roesner for helping out with the unordered LSQ RTL implementation, and thanks to Sibi Govindan for help with the unordered LSQ power measurements.

I have also had the joy of interacting, living and playing with several gifted individuals at UT. Jaehyuk was absolutely great company! Karu, Ramdas, AK, Hrishi and Vikas were always interested in a good debate on any topic under the sun. Heather, PK and CK were always there to talk to. Additional thanks to PK for sharing his lunch with me many times. Thanks to Vinay for being a wonderful

gym partner and indulging in the death-by-workout Saturdays. Thanks to Sadia for bringing wonderful desserts, Sibi for inviting me to dinner many times, Katie for the cookies, company, and for thorough comments on my writing and to Alison and Serita for feedback on several of my talks. Thanks to Paul and Boris for organizing the socials and all of the CART and Speedway members for helping me out on various occasions.

I would like to thank Gem Naivar for shielding me from administrative matters and for helping out at several events. Thanks to Gloria Ramirez for making departmental paperwork painless. Thanks to all the departmental staff for providing and supporting a reliable computing infrastructure. And thanks to the department for providing opportunities and training to be a good teacher, researcher and scholar.

Most importantly many thanks to my parents, Jayashri Raja Rao and Sanjeevi Sethumadhavan, for the sacrifices over the years. As a small token of gratitude, I dedicate this thesis to them.

Lakshminarasimhan Sethumadhavan

*The University of Texas at Austin*

*December 2007*

# Scalable Hardware Memory Disambiguation

Publication No. _____

Lakshminarasimhan Sethumadhavan, Ph.D.

The University of Texas at Austin, 2007

Supervisor: Douglas C. Burger

This dissertation deals with one of the long-standing problems in Computer Architecture – the problem of memory disambiguation. Microprocessors typically reorder memory instructions during execution to improve concurrency. Such microprocessors use hardware memory structures for memory disambiguation, known as Load-Store Queues (LSQs), to ensure that memory instruction dependences are satisfied even when the memory instructions execute out-of-order. A typical LSQ implementation (circa 2006) holds all in-flight memory instructions in a physically centralized LSQ and performs a fully associative search on all buffered instructions to ensure that memory dependences are satisfied. These LSQ implementations do not scale because they use large, fully associative structures, which are known to be slow and power hungry. The increasing trend towards distributed microarchitectures further exacerbates these problems. As on-chip wire delays increase and high-performance processors become necessarily distributed, centralized structures such as the LSQ can limit scalability.

This dissertation describes techniques to create scalable LSQs in both centralized and distributed microarchitectures. The problems and solutions described in this thesis are motivated and validated by real system designs. The dissertation starts with a description of the partitioned primary memory system of the TRIPS processor, of which the LSQ is an important component, and then through a series of optimizations describes how the power, area, and centralization problems of the LSQ can be solved with minor performance losses (if at all) even for large number of in flight memory instructions. The four solutions described in this dissertation — partitioning, filtering, late binding and efficient overflow management — enable power-, area-efficient, distributed and scalable LSQs, which in turn enable aggressive large-window processors capable of simultaneously executing thousands of instructions.

To mitigate the power problem, we replaced the power-hungry, fully associative search with a power-efficient hash table lookup using a simple address-based Bloom filter. Bloom filters are probabilistic data structures used for testing set membership and can be used to quickly check if an instruction with the same data address is likely to be found in the LSQ without performing the associative search. Bloom filters typically eliminate more than 80% of the associative searches and they are highly effective because in most programs, it is uncommon for loads and stores to have the same data address and be in execution simultaneously.

To rectify the area problem, we observe the fact that only a small fraction of all memory instructions are dependent, that only such dependent instructions need to be buffered in the LSQ, and that these instructions need to be in the LSQ only for certain parts of the pipelined execution. We propose two mechanisms to

exploit these observations. The first mechanism, area filtering, is a hardware mechanism that couples Bloom filters and dependence predictors to dynamically identify and buffer only those instructions which are likely to be dependent. The second mechanism, late binding, reduces the occupancy and hence size of the LSQ. Both of these optimizations allows the number of LSQ slots to be reduced by up to one-half compared to a traditional organization without any performance degradation.

Finally, we describe a new decentralized LSQ design for handling LSQ structural hazards in distributed microarchitectures. Decentralization of LSQs, and to a large extent distributed microarchitectures with memory speculation, has proved to be impractical because of the high performance penalties associated with the mechanisms for dealing with hazards. To solve this problem, we applied classic flow-control techniques from interconnection networks for handling resource conflicts. The first method, memory-side buffering, buffers the overflowing instructions in a separate buffer near the LSQs. The second scheme, execution-side NACKing, sends the overflowing instruction back to the issue window from which it is later re-issued. The third scheme, network buffering, uses the buffers in the interconnection network between the execution units and memory to hold instructions when the LSQ is full, and uses virtual channel flow control to avoid deadlocks. The network buffering scheme is the most robust of all the overflow schemes and shows less than 1% performance degradation due to overflows for a subset of SPEC CPU 2000 and EEMBC benchmarks on a cycle-accurate simulator that closely models the TRIPS processor.

The techniques proposed in this dissertation are independent, architecture-neutral and their cumulative benefits result in LSQs that can be partitioned at a

fine granularity and have low design complexity. Each of these partitions selectively buffers only memory instructions with true dependences and can be closely coupled with the execution units thus minimizing power, area, and latency. Such LSQ designs with near-ideal characteristics are well suited for microarchitectures with thousands of instructions in-flight and may enable even more aggressive microarchitectures in the future.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Computers have contributed profoundly to improvements in science and society over the last 60 years. During this period, computer performance has increased an astonishing trillion times from mere 35 operations per second on the ENIAC [31], to 280 Tera operations on the fastest supercomputer at the time of this writing, the IBM Blue Gene/L [1, 76]. These contributions have been possible due to the ability to profitably manufacture smaller, faster, reliable transistors, and, perhaps most importantly, the ability to effectively translate the raw transistor speed into high performance for the end-user. Translating the power of fast transistors to more powerful computers has been largely possible due to effective computer organizations and sophisticated compilers.

Over these last 60 years, computer engineers and compiler writers have employed a number of techniques to achieve high performance. These techniques broadly fall in two categories. The first set of techniques emphasizes improving the speed at which each operation can be completed. The second set of techniques

emphasizes performing many operations in parallel. A low-cost computer, the one that completes many instructions in parallel at great speed, the Holy Grail for computer designers, however, has been difficult to design.

Broadly speaking, two unsuccessful approaches have been taken to design the elusive fast, highly parallel computer. The first approach relies on *explicit* parallelization. Usually in this model the programmer explicitly identifies regions of the program that can execute concurrently. These parallelized programs are then run on simple, fast processors to achieve high performance [1, 49]. While this approach can achieve the simultaneous goals of speed and parallelism, in most cases, it is simply not cost-effective. The task of identifying regions for concurrent execution is onerous and error prone, and often requires many programmer hours even for the simplest programs. Such a model for improving performance does not scale in terms of time and total cost of ownership.

The second approach uses *implicit* parallelization. Implicitly parallel computers rely on the compiler and/or the microarchitecture to extract the parallelism "under the covers" without any support from the programmer. While dozens of implicitly parallel computers have been built, they have been limited in their ability to extract parallelism [39, 74]. Often the goals of speed and parallelism are at odds in computer design because the hardware infrastructure required to increase the number of parallel operations typically slows down the operation and results in higher power consumption [58]. For this reason, a significant component of these implicitly parallel computers, the part that handles memory instructions – specifically the memory disambiguation hardware – has been especially difficult to scale and is the topic of this thesis. Scalable solutions to this problem will clear one of

| Program Order | Possible *Execution* Orders | |
|---|---|---|
| STORE 0xf00d, 0xcafe<br>LOAD X | STORE 0xf00d, 0xcafe<br>LOAD 0xcafe | STORE 0xf00d, 0xcafe : LOAD 0xCOO1 |

Figure 1.1: Effect of ambiguous address on parallel instruction execution: If the load is to different address than the store (right) the the load and store can execute in parallel. Otherwise, they must execute serially (middle).

the last remaining hurdles towards massively parallel fast computers.

To date, one of the most highly parallel computers that is also capable of high frequency operation is the TRIPS prototype built at The University of Texas at Austin. In this thesis, we propose and evaluate solutions to the long-standing problem of scalable memory disambiguation in the context of this processor. This chapter begins by explaining the problem of memory disambiguation, then illustrates how memory disambiguation has historically affected computer design, how it is handled in current implicitly parallel systems, and concludes with an overview of the scalable alternatives described in the thesis.

## 1.1 What is Memory Disambiguation?

Memory disambiguation is knowing with certainty that two memory instructions will or will not access the same *physical* memory location. Consider the memory instructions shown in Figure 1.1. The first instruction stores the value 0xf00d to the memory location starting at address 0xcafe. The instruction that follows the store instruction, a memory read or a load instruction, reads from an unknown location. In other words, the load's address is ambiguous.

Depending on the address of the load, two execution scenarios are possible.

If the load instruction is to a different address from the store instruction then load and store instructions can execute independently of each other, for instance, in parallel, and the program fragment can complete faster provided there are sufficient resources. Otherwise, the load instruction must necessarily wait for the store to write its value 0xf00d to location 0xcafe, before it can obtain the value from there. Thus an ambiguous load can hinder the ability to execute instructions in parallel.

To disambiguate the load, or more generally the memory instructions, one must know the data addresses that the memory instructions access. These addresses may be difficult to determine statically because of the ambiguities in the programming language, or because of dynamic linking, or because of limitations in the compilation framework or a combination of some or all of the above.

Figure 1.2 illustrates how ambiguity can arise even in simple programs on a typical (but hypothetical) modern software-hardware system. If the first fragment, "1", the compiler typically cannot determine the memory location that the pointer, p, points to because it depends on the runtime value of the function parameter *cond* which is usually difficult to determine without actually running the program. Consequently, the code following the if/else construct cannot be statically parallelized because of the ambiguous pointer. In the second fragment, "2", if the $get\_buf$ function is dynamically linked from a library then the function cannot be statically analyzed to determine if the return value of the function points to same location as the locations pointed to be the names a or b. The ambiguity again hinders parallelizing code following the function call.

Thus, several factors starting from the programming language leading all the way down to the OS/Compiler make it impossible to mechanically, statically

```
% Code fragment 1:

 1 void foo (bool cond) {
 2  int *p;
 3  int a,b;
 4  if (cond)
 5    p = &a;
 6  else
 7    p = &b;
 8  *p++;
 9  // some comptuation involving p, a and b
 0  }

% Code fragment 2:

 1 void bar (bool cond) {
 2   int *p;
 3   int a,b;
 4   // get_buf is dynamically linked
 5   p = get_buf(1);
 6   // some computation involving p, a and b
 7   //
 8 }
```

Figure 1.2: Code sequence illustrating appearance of ambiguity

and completely disambiguate memory addresses eventually hindering optimizations for parallel execution. However, scientists over the last several decades have made several innovations that side step the memory disambiguation problem. The next section highlights some of the interesting innovations and efforts.

## 1.2 Memory Disambiguation: A Brief and Incomplete History

The computers built in the 40s and 50s were simple von-Neumann machines that executed instructions serially one at a time without spatial multiplexing or temporal overlap among the instructions. These simple microarchitectures did not need memory disambiguation because their design guaranteed that all prior instructions would have finished before the next one is fetched and executed. As microarchitectures evolved from these simple, single instruction unpipelined machines to more complex, but higher throughput, pipelined, multi-functional, out-of-order, data-flow machines several innovations were required to handle memory dependences.

**Early Adopters:** The CDC6600 and the IBM 360/91 were two of the earliest machines to have special hardware to support memory dependences [75, 11]. These early processors did not have caches and hence stores to memory were long-latency operations. Store buffers were implemented to enable the overlap of computation with the completion of stores. To support correct execution, these machines required a mechanism for forwarding the store values from the "stunt" or "data buffers" to in flight loads.

**Early Avoiders:** In the late 1960s, in a short span of time, three different groups proposed a new model of execution known as the data-flow model [82]. Unlike the von-Neumann model, in the data-flow model instructions execute as soon as the input data for the instructions are available. The data-flow model removes the needless instruction-at-a-time restriction of the von-Neumann model and could, in theory, promote higher degree of parallel instruction completion.

To derive these benefits, however, the data-flow computers must overcome difficulties with handling memory dependences using special data-flow programming languages. A key feature of these languages is the absence of persistent storage — the left hand side variable in an assignment statement is always assigned a new name and copied over to a new memory location. This implementation allows the compiler to determine data dependencies by simply matching up the names of the variables. Consequently, ambiguities are impossible and the compiler, in theory, can schedule the execution of instructions to maximize parallel execution.

The data-flow computers, however, have suffered from several practical problems. First the data-flow language requirement limits the software usable on these computers; a large portion of software is written in von-Neumann style (sequential) languages and hence cannot be utilized for these data-flow machines. Second, supporting Input-Output operations is fundamentally impossible in a pure data-flow model because Input-Output operations need to maintain persistent state. Third, because of the lack of persistent storage in this model, for composite data structures like arrays, even changing one part of the data structure required copying over the entire data structure. To overcome the array-copying problem, MIT researchers, invented special hardware and language structures known as I-structures [6]. How-

ever, the size of the data structure was limited by the size of the I-structures, which were in turn constrained by the VLSI technology and eventually restricted the scope of the data-flow and the parallelism that can be exploited.

Thus, even though data-flow computers promised maximal parallelism, the Achilles heel for these computers was memory and memory disambiguation. One solution to the problem that has been investigated in detail has been to convert von-Neumann languages into data-flow representations that can execute efficiently on a processor. An analysis to establish the equivalence of pointers is an essential part of this conversion. However, even with state-of-the-art pointer analysis, precise pointer analysis is too slow and takes up too much memory for many but not all realistic programs [35]. Furthermore, as explained in the previous section, in the presence of dynamic loading, the problem of memory disambiguation becomes intractable in the general case.

**When memory was temporarily tamed:** Unlike in the case of the data-flow computers, language-based solutions have been useful and practical for some computer organizations.

During the early 1970s, supercomputer companies manufactured an enhanced von-Neumann style computer that could operate simultaneously on multiple data items using only one instruction [43]. These instructions were known as vector instructions and the computers that used these instructions came to be known as vector computers. To detect and encode vector operations from serial code, the engineers for these machines invented sophisticated compiler analyses. One notable invention was array dependence analysis, which allowed the compiler to determine whether two array accesses in the loop body pointed to the same memory location.

The results from this analysis are then used to parallelize loops and code the operations inside a loop as vector operations. Array dependence analysis was largely possible and simplified by the fact that FORTRAN, the preferred language for supercomputers, did not have any pointers. Unlike data-flow computers, the language requirement worked *in favor* of the vector computers because FORTRAN happened a popular computer language for scientific computing and therefore programs were readily available. In the 80s, however, as newer languages like C/C++ became popular, programs written in these languages used pointers heavily, which diminished the usefulness many previously useful compiler analysis including array dependence analysis.

**Modern Memory Dependence Hardware:** In the 1980s, spurred by the ability to include more transistors on a chip, microarchitects designed processors that could be fit on one chip. The microprocessors, as these were called, became increasingly sophisticated starting from simple in-order pipelines to very complex out-of-order processors, to processors that could perform in hardware the code optimizations previously performed by the compiler. The fundamental idea in several of these machines was similar to early von-Neumann machines - a set of serially fetched instructions were deposited into a structure known as the instruction window, and then the instructions whose operands were ready were issued for execution. This execution model allowed loads and stores to be reordered from the instruction window and required memory disambiguation structures known as the Load-Store-Queues (LSQs) to ensure correct execution. Patt et al. designed one of the earliest LSQs for the High Performance Substrate processor proposed in 1985 [60].

As these processors evolved, and their ability to execute multiple instructions

increased, so did the need for larger sized memory disambiguation hardware. The size of the memory disambiguation hardware limits the number of in-flight memory instructions and thus eventually limits the amount of (instruction-level) parallelism that can be exploited. These LSQs became one of the primary structures inhibiting the scaling of superscalar processors to larger processors, since they have typically supported only a few tens of in-flight loads and stores due to power and complexity limits.

In the early 2000s, these power and complexity limits in LSQs and other structures led to an industry-wide shift to chip multiprocessors (CMPs) [74, 46]. At the time of writing this dissertation, much of the current research is focusing on CMPs in which the individual are processors smaller, lower power, and simpler, effectively reducing single-thread performance. This trend is reducing the instruction-level parallelism that can be exploited, placing a higher burden on software and/or programmers to use CMPs effectively. Historically, however, most programmers have had difficulty parallelizing their applications. Additionally, compilers have had limited success in automating parallelism for popular languages like C/C++, and Java in part due to the difficulty with pointer analysis.

Alternatives to CMPs are tiled large-window microarchitectures that can scale to thousands of in-flight instructions, using traditional languages and modest and feasible compiler analysis. Memory disambiguation at this scale, however, has been a key limitation for these types of architectures because of power, latency and complexity limits.

**Summary:** Over the 40 years, since computer architects embarked on the quest for higher degree of parallelism, memory disambiguation has posed several challenges.

Architects have attempted to overcome the problem using language support, with compiler analysis and recently using limited hardware support. However, as the need for parallelism increases, and as older approaches lose favor because of economic and technical reasons, the problem of memory disambiguation threatens to stall historical performance improvements provided by computers over the past 60 years. This dissertation makes contributions towards solving this important problem.

## 1.3 Thesis Contributions

The purpose of the LSQ is to allow in flight loads and stores to be re-ordered if they are to different addresses and enforce dependencies when they are to the same address. An ideal LSQ will selectively buffer the memory instructions with true memory dependences irrespective of the ambiguity of the memory instructions, buffer the dependent instructions for the minimum duration required to satisfy the dependences, allow access to the buffered LSQ only for the dependent instructions and allow the LSQs to be physically near the unit that produced the memory operation. The first three requirements are important for reducing the size and power consumed by the LSQ and the fourth frees the designer of constraining the location of the LSQ.

A conventional LSQ, in direct contrast to the ideal LSQ, typically holds all in-flight memory instructions in a physically centralized LSQ and performs a fully associative search on all buffered instructions to ensure that memory dependencies are satisfied. These LSQ implementations do not scale because they use large, fully associative structures, which are known to be slow and power-hungry. The increasing trend towards distributed microarchitectures further exacerbates these

problems. As on-chip wire delays increase and high-performance processors become necessarily distributed, centralized structures such as the LSQ can limit scalability.

This thesis makes four contributions that propose techniques to create LSQs with near-ideal characteristics.

**LSQ Late Binding:** Traditional LSQs hold the memory instructions in the order that they are fetched by allocating slots to instructions in the early stages of the instruction execution pipeline. This allocation policy, while simplifying the LSQ hardware, results in oversized LSQs because a memory instruction does make use of the allocated slot until it is executed in the later stages of the pipeline. By delaying the allocation until instruction execution, "late-binding", the LSQ can have lower occupancy and thus can be smaller. This optimization allows the number of LSQ slots to be reduced by up to one-half with little or no performance degradation.

**LSQ Overflow Management:** LSQs for distributed architectures also need to be distributed. In distributed LSQs, it is wasteful to maximally size each of the LSQ partitions because only a fraction of the memory references will access a given partition. If the partitions are not maximally sized, however, we need schemes to deal with situations when too many instructions map to a distributed bank. Simply stalling could lead to deadlocks; simply restarting can lead to severe performance losses. We observe that in a distributed, large-window processor, LSQ banks can be treated as clients on a micronetwork, with overflows handled using traditional network flow-control techniques adapted to distributed processor microarchitectures. These flow-control techniques allow LSQs to be constructed using a set of small partitionable distributed banks with little or no performance losses.

**LSQ Access Filtering:** To mitigate the power problem, we replaced the power-hungry, fully associative search with a power-efficient hash table lookup using a simple address-based Bloom filter. Bloom filters are probabilistic data structures used for testing set membership and can be used to quickly check if an instruction with the same data address is likely to be found in the LSQ without performing the associative search. Bloom filters typically eliminate more than 70% of the associative searches and are highly effective because in most programs, it is uncommon for loads and stores to have the same data address and be in execution simultaneously.

**LSQ Area Filtering:** This technique uses a predictor to first learn about dependent memory instructions during execution and then attempts to buffer only those instructions in the LSQ. Correct execution is guaranteed by using a space-efficient Bloom Filter for the independent instructions. This scheme has the potential to reduce the size of the LSQ by 90%.

The key idea behind these contributions is to optimize the LSQ implementation for common program behavior. In many programs, although several loads and stores in a program may be statically ambiguous only a few are dynamically dependent on each other. This observation when combined with Bloom filter based implementation provide an area and power efficient implementation alternative to LSQs because they provide an efficient encoding of independent operations and thus have the potential to create near-ideal LSQs.

## 1.4 Thesis Roadmap

This thesis begins with an overview of the TRIPS processor, the TRIPS microarchitecture and then describes in detail the design of the TRIPS primary memory system – a memory system partitioned to a greater degree than prior uniprocessor memory systems – to set the stage for the discussion of LSQs for rest of the thesis. In Chapter 3, we discuss LSQ filtering optimizations. In Chapter 4, we discuss late-binding and overflow management techniques for distributed LSQs. Chapter 5 discusses related work and is followed by concluding remarks and future work in Chapter 6. The appendix includes LSQ extensions for TRIPS multiprocessors.

# Chapter 2

# Background on the TRIPS Primary Memory System

Despite the plethora of research papers on the individual components of the primary memory system, most prominently caches [1], I do not know of any research publications that thoroughly describe the internals of the primary memory system of a modern microprocessor. Without this information it is often difficult for the non-specialist reader to understand the implications of individual memory system optimizations. To address this problem, this chapter first describes the TRIPS primary memory system before describing the LSQ optimizations in the later chapters.

In addition to providing background, this chapter also documents the TRIPS memory system implementation, which is novel and interesting for several reasons. It is a fully partitioned, polymorphous memory system which supports sequential memory semantics while supporting aggressive re-ordering of loads and stores and

[1] by some estimates nearly 2000 papers have been published: J.L. Baer, HPCA 2000 Keynote Address

simultaneously ensuring that none of the traditional memory system functions are necessarily centralized. In addition, the TRIPS memory system is four times larger than several contemporary designs. For instance, each TRIPS processor can manage up to 64 outstanding misses to the memory system and handle four loads every cycle while contemporary designs can handle four misses and handle two or fewer loads every cycle.

The memory system must be fully partitioned to match the execution substrate, which, in turn, is also partitioned to address the technology challenges posed by increasing wire delays and power consumption. The memory system must be polymorphous, i.e., support sequential, streaming and threaded workloads efficiently without additional area requirements. It must also support sequential memory semantics to allow the use traditional languages like C/C++ to program the TRIPS processor. Allowing loads and stores to be aggressively re-ordered is critical for achieving greater instruction execution overlap.

We hypothesized that the TRIPS memory system could be constructed in a complexity-effective manner from independently accessible partitions that communicate using distributed protocols. Our prototype implementation validated this hypothesis and showed that although new protocols were required, these protocols were simple, and the time-to-design and verification complexity of the partitioned memory system were comparable to a centralized implementation.

The rest of this chapter describes the details of the memory system. The sections are organized as follows: Section 2.1 describes the motivation and architecture of the TRIPS processor. Section 2.2 provides an overview of the TRIPS microarchitecture. Section 2.3 describes the primary memory system a number of

important aspects of the memory system. We conclude this chapter with summary of the TRIPS primary memory system.

## 2.1 TRIPS: Motivation and Architecture

Performance depends two factors: frequency, how fast an operation can be completed, and concurrency, the number of operations that can be completed in each cycle. Towards the early part of this decade, increasing the frequency to improve performance was becoming increasingly less viable because of the effects of growing wire delays, power consumption, and the diminishing performance gains from deeper pipelines [51, 56, 40, 36]. Consequently, processor designers were forced to shift to exploiting concurrency to improve performance.

Traditional ISAs like RISC and CISC, and their conventional implementations, however, do not lend themselves profitably to concurrent implementations. Traditional ISAs were invented for von-Neumann architectures which are inherently sequential, and require a single sequencer which throttles parallelism at the front-end of processor, starving the back-end ("Flynn Bottleneck"). Although speculative techniques like branch prediction can mitigate some of this, often artificially introduced sequentiality by predicting and fetching across multiple branches, scaling these speculative methods to thousands of instructions is still an open research problem. In addition, conventional microarchitectures use many centralized structures, like register files, issue-windows, and load-store-queues which do not scale with increasing number of in flight instructions [2].

The TRIPS designers invented the EDGE ISA and microarchitecture that was more amenable to concurrent instruction execution in wire-delay dominated

technologies [13, 66]. Explicit Data Graph Execution (EDGE) ISAs sequence through blocks of instructions which are fetched and committed atomically i.e., a TRIPS block can update architectural state only when all of its memory and register outputs have been generated. Unlike von-Neumann architectures, block atomic architectures sequence through groups of instructions thus mitigating the performance impact of artificial instruction-level dependences.

The TRIPS compiler partitions the program into blocks of up to 128 instruction which are fetched, executed, and committed as atomic units [70, 50, 20]. Blocks may contain up to 32 load or store instructions. The hardware maps these blocks onto a distributed microarchitecture which executes the instructions within each block in dataflow order. Thus data-flow execution is facilitated by directly encoding the producer-consumer dependences as part of the instruction. The hardware also speculatively executes up to seven blocks so that eight blocks may be executing simultaneously. Blocks that suffer control-flow or load-store dependence mis-speculations are flushed and re-executed.

More details on the ISA, execution model and microarchitecture are described in the TRIPS ISA specification, and dissertations of Nagarajan and Sankaralingam [52, 57, 65]. The rest of this section focuses mainly on the features of the TRIPS ISA that are most relevant to the memory system.

**Instruction types:** The TRIPS architecture, like several commercial architectures, supports byte (8-bit), half-word (16-bit), word (32-bit) and double-word (64-bit) load and store instructions. These instructions can be used to load or store integer, floating point or any arbitrary sized data type. Supporting one uniform datum size could have simplified portions of the design (the LSQ and store merging

| Data Size | Load Fraction | Store Fraction |
|---|---|---|
| 64-bit | 0.58 | 0.70 |
| 32-bit | 0.23 | 0.20 |
| 16-bit | 0.03 | 0.03 |
| 8-bit | 0.15 | 0.06 |

Table 2.1: Distribution of loads and stores of different sizes on the TRIPS for 20 SPEC benchmarks.

logic), but at the expense of extra instructions that would be needed compose unsupported data types. For instance, if only doubleword memory instructions were implemented, extra instructions would be need for extracting relevant bits for byte accesses. On the other hand, if only byte sized memory instructions were allowed, multiple memory accesses would increase the memory bandwidth. We decided to support multiple data sizes because each data type comprises a significant fraction of the memory instructions (see Table 2.1).

In addition to these different data types, the TRIPS ISA also supports different types of memory attributes (mergeable/unmergeable, cacheable/uncacheable), synchronization instructions (locks), and virtual memory. Load and store instructions to mergeable regions can be combined and issued as larger load and store instructions to the memory system. Load and store instructions to uncacheable regions bypass the cache before they are issued to the memory system. The mergeability and cacheability of loads and stores is implicitly determined by the addresses they access, i.e., the attributes are applied en-masse to contiguous chunks of the address space and enforced using the TLB.

The unmergeable and uncacheable attributes are used to provide synchronization and streaming support. The unmergeable region is used to isolate lock

instructions from being merged with regular load and store instructions in the critical sections, by marking the lock address space as unmergeable. Address segments marked uncacheable are used to implement streaming operations and lock based synchronization. Lock instructions should also be marked uncacheable because the TRIPS implementation (not the ISA) does not support coherence between the L1 data caches across the multiple processors.

**Load/Store Ordering:** To determine the correct memory order and thus track load store dependences in the partitioned primary memory system, the TRIPS processor uses specially encoded program order tags called Load Store IDs (or LSIDs). An LSID is a 5-bit field in a memory instruction. In most superscalar architectures, the program order (or "age") is determined dynamically in the fetch stage and hence the LSID is not included in the instruction. In a *completely* partitioned microarchitecture like TRIPS however, the fetch mechanism is also partitioned and there are multiple distributed fetch points co-operatively fetching instruction streams. In such cases, without centralization, it is impossible to construct the total memory age order from the partial memory age orderings observed at different fetch points; hence LSIDs must be encoded as part of the instruction.

**Consistency model:** TRIPS implements a weak consistency model (similar to Power 4 [78]) that enforces load/store dependences in a thread but relaxes all execution orders and requires the programmer to insert memory barriers to realize more strict consistency. Direct hardware support of more traditional and programmer friendly consistency models on TRIPS (like in other aggressive out-of-order processors) can negatively impact performance if it is naively implemented. Con-

sider, for instance, total store ordering (TSO). Among other requirements, TSO requires that stores to different memory addresses commit in program order. To support TSO on TRIPS, a store must be constrained to update memory only after the previous store has committed. This requirement would restrict store commits to one per cycle whereas a weaker model can enable simultaneous store commits from all partitions. The slower deallocation of stores eventually leads to slower deallocation of other processor resources resulting in performance losses. Appendix A discusses implementation of traditional consistency models on the TRIPS processor.

**Block Atomic Execution:** To support block atomic execution, each TRIPS block encodes the number of memory outputs (stores) for a block, and the LSIDs of all outputs in the store-mask bit vector, in the block header. This information is broadcast to all the memory system partitions when a new block is fetched. Each block can have up to 32 memory instructions. Thus with eight inflight blocks, the microarchitecture must support 256 inflight memory instructions.

## 2.2 TRIPS Microarchitecture Overview

TRIPS is a distributed microarchitecture that seeks scalability by partitioning its microarchitectural components such as execution units and cache banks. This section gives a brief overview of the TRIPS microarchitecture as it pertains to distributed LSQs and the microarchitectural networks addressed in this dissertation. Additional details on the TRIPS core can be found in [66].

Figure 2.1 shows a diagram of a TRIPS processor composed of multiple tiles connected via a set of microarchitectural networks (micronets). The given

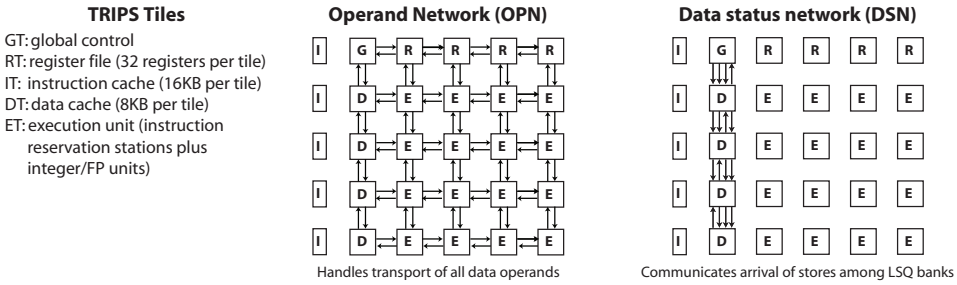



Figure 2.1: TRIPS tile organization and micronetworks relevant to the LSQ.

processor comprises five instruction cache tiles (ITs), four register file tiles (RTs), sixteen execution tiles (ETs), four data cache tiles (DTs) and one global control tile (GT). These tiles are connected via the operand network (OPN) that dynamically routes operand-size items from producers to consumers in a 5×5 mesh. When a block is fetched, its instructions are delivered to preallocated reservation stations distributed throughout the ETs. Reservation stations remain allocated to the block until the block is committed and a new block is fetched. Instructions fire when their operands arrive and subsequently deliver their results to their consumers via the OPN. Load and store instructions are delivered to the DTs via the OPN and load results are returned to consumers via the OPN. OPN links consist of a single physical channel but includes small FIFO buffers throughout the routers to increase network capacity [33]. All of the routers employ simple on-off flow control with hold signals to exert back pressure. This back pressure extends into network clients, such as the execution units, which can then be forced to stall. The network and the ETs are arranged such that dependent instructions in the same ET can be executed in back to back cycles and dependent instruction in the neighboring ET can be executed one cycle after the execution of the parent instruction.

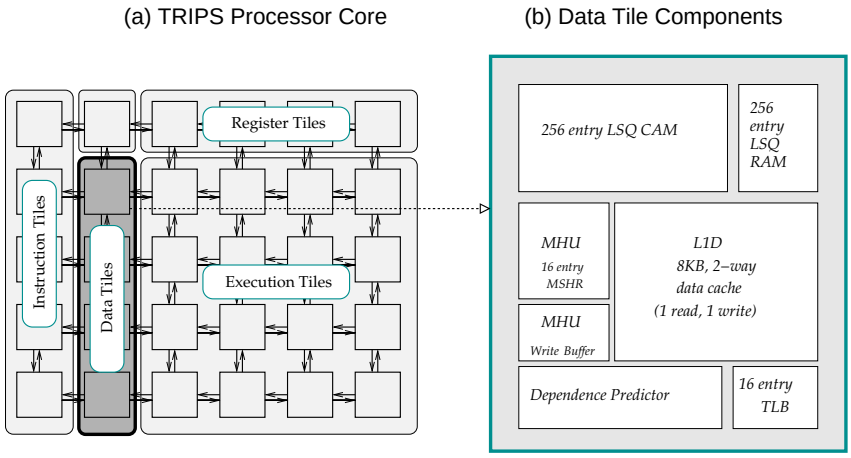



Figure 2.2: Single core of the TRIPS SMT-CMP prototype and components of a s ingle data tile

## 2.3 DT Microarchitecture Overview

The primary memory system (level-1) of the TRIPS processor prototype is made up of four partitions, each called a Data Tile (DT). Each DT (Figure 2.2) is interleaved at a cache-line granularity and includes an 8KB, 2-way associative cache bank (L1D), a local copy of the load store queue (LSQ), a local copy of the data translation look-aside buffers (TLB), a store-load dependence predictor (DPR), and miss-handling unit (MHU). Each DT is connected to three different networks: the operand network delivers loads and stores from the execution units to the DTs, the L2 network (On-chip Network or OCN [32]) is used to access the L2 cache banks on load and store misses and the status network (DSN) connects all the DTs and is used to track stores arriving at the different DTs. The store arrival information is used to determine when all block store outputs have been produced and also enable load-store dependence prediction.

Using the structures and networks described above, each DT partition:

- provides data to loads and stores
- performs address translation and protection with its DTLB bank
- handles cache misses with its MHU
- tracks load and store dependences with its LSQ
- performs load/store dependence prediction for aggressive load store issue with DPR
- detects when all store outputs for a TRIPS block have been produced
- writes stores to caches/memory when they become non-speculative
- performs store merging on L1 store cache misses

The load and store instructions can be mapped onto any of the sixteen execution units on the TRIPS processor (Figure 2.2a.) The memory instructions issue from the execution units when all their inputs are available, and are then delivered to the DT through the operand network. This section provides an overview of the basic steps involved in load and store processing in the DT and the pipelines that implement the load/store processing steps. We also describe *store tracking*, the only additional microarchitecture mechanism required for partitioning the primary memory system. We conclude this section with the method for dependence prediction on the TRIPS processor.

### 2.3.1 Load Processing

The pipeline diagram in Figure 2.3 illustrates the different stages of load processing. Every incoming load accesses (a) the TLB to perform address translation and check

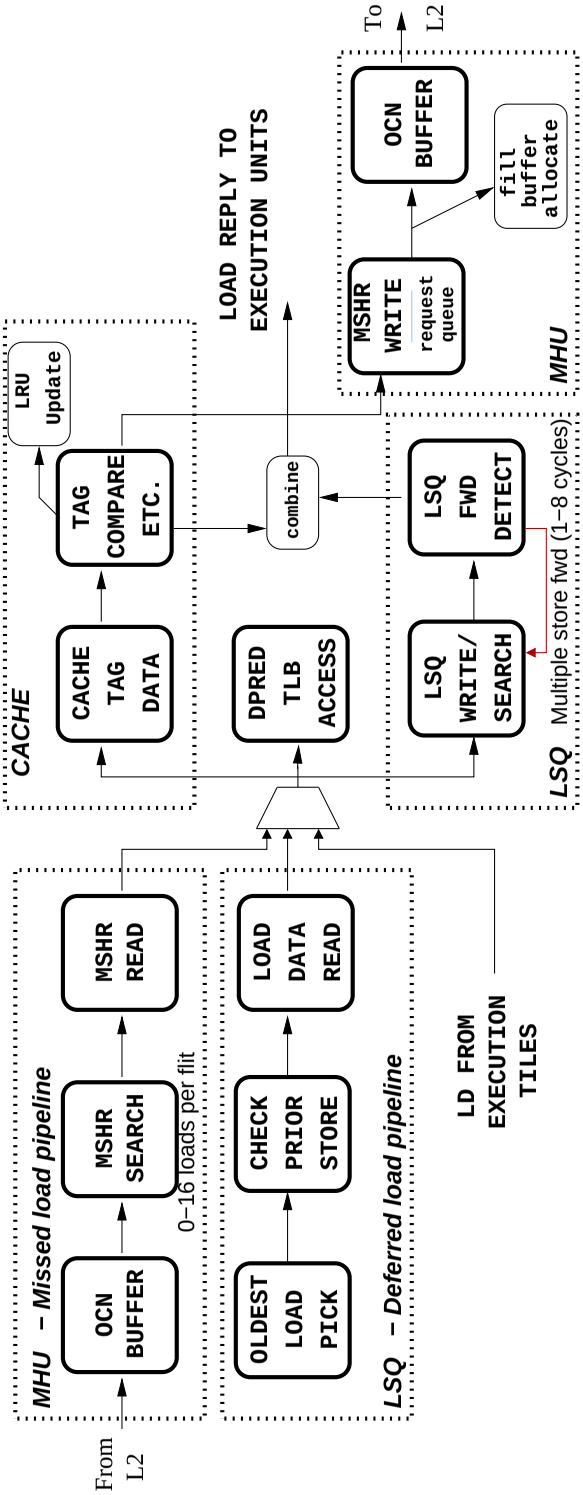





Figure 2.3: The DT Load Pipelines

| TLB | DPR | Cache | LSQ | Response |
|---|---|---|---|---|
| Miss | X | X | X | Report TLB Exception |
| Hit | Hit | X | X | Defer load until all prior stores are received |
| Hit | Miss | Hit | Miss | Forward data from cache |
| Hit | Miss | Miss | X | Forward data from L2 cache, issue cache fill request |
| Hit | Miss | Hit | Hit | Forward data from LSQ and cache |

Table 2.2: Load Execution Scenarios. X represents "don't care" state.

the protection attributes, (b) the dependence predictor (DPR) to check for possible store dependences, (c) the LSQ to identify older matching uncommitted stores, and (d) the cache tags to check for cache hits. Based on the responses (hit/miss) from the four units, the control logic decides on the course of action for that load. Table 2.2 summarizes the possible load execution scenarios in the DT.

**Load Hit:** When the load hits in the cache, and only in the cache, the load reply can be generated in two cycles. This best-case latency is likely to be the common case for most loads. When a load hits both in the cache and the LSQ, the load return value is formed by composing the values obtained from the LSQ and cache. First, the load picks up any matching store bytes from the LSQ and then reads the remaining bytes from the cache. This operation can take multiple cycles and is called as store forwarding. Section 2.5 describes store forwarding in more detail.

**DPR Hit:** A load may arrive at the DT before an earlier store on which it depends. Processing such a load right away will result in a dependence violation and a flush leading to performance losses. To avoid this performance loss, the TRIPS processor employs a simple dependence predictor that predicts whether the load processing

should be deferred. If the DPR predicts a likely dependence the load waits in the LSQ until all prior stores have arrived. After the arrival of all older stores, the load is enabled from the LSQ, and allowed to access the cache and the LSQs to obtain the most recent updated store value.

Figure 2.3 illustrates the stages involved involved in processing deferred loads. When a load is deferred, the type, size, address and target of the load are stored in the LSQ. Every cycle, the oldest deferred load checks if all prior stores for that load have arrived. If the load is ready to be undeferred, then the load waits to read data from the LSQ. If there is another load involved in store-forwarding the deferred load waits until the store forwarding is complete. When the LSQ is free, the load is read from the LSQ and the load is re-inserted into the pipeline as if the deferred load was a new load. This simple optimization, simplifies the state machines required for accessing the caches, LSQ, and different pipeline interlocks.

**Load Miss:** If the load misses in the cache, it is buffered in the Miss Status Holding Registers (MSHRs) [47] and a read request is generated and sent to the L2. When the data is returned from the L2, the loads in the MSHRs are enabled and load processing resumes. Like deferred loads, missed loads also access the LSQ and cache to pick up the most recent store values.

Figure 2.3 illustrates the stages involved in processing missed loads. In the cycle after determining that load has missed in the cache, if there are no pending memory requests, the load address is used to construct a miss request packet. As will be explained later, several bookkeeping structures are also updated in this stage. In the next stage, in the absence of store misses and cache spills, the load miss request is sent out on the network. Upon a miss return, the data is accessed in a pipelined

fashion as soon as the load data starts arriving from the memory. For each arriving 16-bytes of data (flit), loads dependent on that flit are woken up from the MSHRs and the incoming data is also written to the cache. Just in the case of deferred loads, the missed loads are also re-injected into the pipeline as if they are just arriving from the execution units. This optimization greatly simplifies the handling of several corner cases. For instance, without this optimization, loads that are woken up have to interlock with the LSQ to check for stores to the missing address that arrived while the load miss was being processed. In addition to the interlocks in the LSQ, each pipeline stage has to be checked for matching stores, increasing the number of bypasses significantly. Additional details of the miss-handling and interaction with store miss handling are discussed in Section 2.6.

### 2.3.2 Store Processing

Store processing occurs in two phases. During the first phase, each incoming store is buffered in the LSQ and the other DTs are notified about the store's arrival. During this phase each store checks for dependence violations; if any younger loads to the same address as the store are in the load-store queue, then a violation is reported to the control unit, which initiates recovery. The dependence predictor is also trained to prevent such violations in the future.

When a block becomes non-speculative, the second phase of store processing begins as illustrated in Figure 2.4. In this phase the oldest store is removed from the LSQ, checked in the TLB, and the store value is written out to the cache/memory system. If the store hits in the cache, the corresponding cache line is marked as dirty. If the store misses in the cache, the store miss request is sent to the L2. We

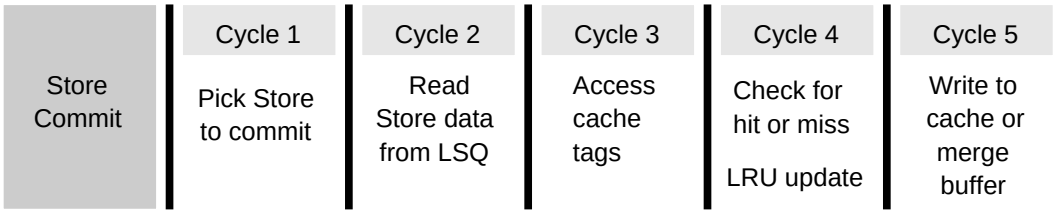



Figure 2.4: The ST Commit Pipeline

chose a write-back, write-noallocate policy to minimize the number of commit stalls.

### 2.3.3 Store Tracking

In the TRIPS execution model, a block can commit only after all of its store outputs have been generated. When a store arrives at any DT, the store arrival information is broadcasted to the other DTs through the Data Status Network (DSN). Each DT then increments a local counter that counts the number of stores that have arrived at the memory system. When all of the stores in a block have been received, the DT that received the last store sends a message to the control tile indicating that all memory outputs have been generated.

The DTs share information about the arrival of stores on the DSN. The DSN network is a staged fully connected network with dedicated physical links between the partitions. Each store packet on this network is ten bits wide – eight bits to represent the age of the store and two bits for communication exception conditions.

## 2.4 Memory-Side Dependence Processing

In the TRIPS implementation, the instruction window is partitioned across sixteen execution units (Figure 2.2) and the dependent loads and stores can be mapped to any of the execution units. A naive extension of conventional dependence processing mechanisms [18] would hold back the load *in the execution unit* until the execution of the dependent store. This strategy can increase the load latency as explained below.

The latency of dependent loads can be broken down into four parts: (1) the latency for the load to detect that the dependent store has executed, (2) the latency for the load to be delivered from the execution unit to the DT, (3) the latency to access the DT, and (4) the latency to deliver the value from the DT to the target of the load. With execution side dependence processing, no overlap is possible between any of the latencies, because the loads are issued only after the dependent stores resolve and rest of the steps must be performed in order. However, memory side dependence processing allows the overlap of steps (1) and (2).

## 2.5 LSQ Microarchitecture

The LSQ is critical for supporting aggressive memory ordering and is often considered to be one of the most complex structures in an out-of-order processor. This section describes the design of the LSQ in the context of the partitioned TRIPS microarchitecture.

The LSQ must support four major functions: (1) detect ordering violations between loads and stores to the same address, (2) forward values from uncommit-

ted stores to loads to same address, (3) buffer and wake up deferred loads, and (4) buffer and commit store values to memory. All the above functions, except the third, are also required of the LSQs in conventional architectures. The third functionality is related to dependence prediction and is most efficiently implemented in the DT/LSQs for TRIPS.

### 2.5.1 LSQ state

The TRIPS microarchitecture allows up to eight blocks to be in-flight simultaneously and each block can have a maximum of 32 memory instructions; therefore, a maximum of 256 memory instructions in-flight. To accommodate the case when all the memory instructions in a block map to the same DT bank, the LSQ is sized to hold 256 memory instructions. The logical and physical organizations of the LSQ are illustrated in Figure 2.5.

Each record in the LSQ holds: (1) the valid bit for the LSQ record, (2) the type bit to distinguish load and stores, (3) the wait bit to mark a deferred load, (4) the high order 37 bits of the virtual memory address to identify matching instructions in the LSQ, (5) an 8-bit vector (byte enable) representing the bytes accessed by the instruction; this is used to implement store forwarding efficiently, (6) the target of the load (i.e. consumer for the load data), and (7) the 64-bit store data value.

The valid, type and deferred bit (i.e. all the status bits) are constructed using flip-flop arrays. The higher order 37 bits of the address are stored in the address CAM (A-CAM) with one read, one write and one search port. The 8-bit vector representing the bytes accessed or modified by the memory instruction are

also stored in the byte-enable CAM (B-CAM) which has one write, one search but no read ports. The A-CAM supports only full exact matches while the B-CAM can identify partial matches. For load records, the load target (9 bits), the higher order address bits (37 bits) and the bit vector (8 bits) are stored in the RAM. For store records, the store data value is stored in the RAM. The RAM has two read ports and one write port. The write port is used to store incoming instructions while one of the read ports is used for committing store values, and the other is used for either store forwarding or waking up deferred loads.

### 2.5.2 LSQ operations

**LSQ Store Forwarding:** Upon receiving a new memory instruction, the LSQ decodes the age of the instruction to produce an index into the LSQ. The indexed location stores the the address, the type and the target/value of the instruction. Simultaneously, the CAMs are associatively searched with the address and byte enables of the incoming instruction to identify all matching instructions.

The store-load forwarding operation is the most complicated operation in the LSQ because the load may match an arbitrary number of stores and can forward from up to eight distinct stores because of different sized LDs/STs. To handle this case, the LSQ scans all the matching stores starting from the most recent store, processing one matching store every cycle, either until the value of every load byte has been obtained or until there are no matching stores to the unforwarded bytes.

Figure 2.6 illustrates an example of multiple forwarding where multiple stores of different sizes match to an 8-byte load (Address 0x8000, Age 24). In the first cycle, the CAMs are associatively searched and the stores 23, 22, 20, 18 and 17 are

identified as matching stores. These matches are scanned starting from the most recent store. In the first scan, byte 6 of the load is retrieved from store 23 and the load byte is marked as forwarded. In the next scan, a new associative search identifies the matching stores corresponding to the remaining unforwarded bytes. The search now returns 22, 20, 18 and 17. Store 23 does not match because it produces byte 6 which was already forwarded to in cycle 1; at the end of the second scan, byte 0 from the store 22 is forwarded to the load. After the next three scans, bytes 4 and 5 are forwarded from 20, byte 7 is forwarded from 18 and byte 1 is forwarded from 17. At this point, there are no more matching stores to the remaining unforwarded bytes (bytes 2 and 3) and therefore LSQ forwarding is terminated. The remaining bytes are obtained from the cache. In this example, forwarding takes five cycles and during this period, the LSQ is completely stalled and cannot accept new loads or stores. Note that after the first associative search which searches both the address and the byte enable CAM and the rest of the scans search only the byte enable CAM which is eight-bits wide.

**Dependence Violations:** To detect dependence violations, the LSQ detects if there are any matching, non-deferred, loads younger than store in program order and if so, reports a violation to the control unit. To recover from the violation all younger blocks starting from the load block that caused the violation are flushed and re-executed. Forward progress is guaranteed by serially executing the block that caused the violation.

**Deferred load processing:** A deferred load is marked by setting the wait bit in the load's LSQ record. Every cycle, the wakeup logic scans the deferred bits of the

all the loads and selects the oldest deferred load for processing. In the next cycle, the LSQ logic checks if all stores older than the selected load have arrived at the DTs. If they have, then all the information about the deferred load – its address, the byte-enables and the target of the load – is read from the LSQ RAM. The deferred load then accesses the LSQ and cache as if it were a new load entering the DT for the first time. When the load goes through the LSQ for the second time it picks up values from matching stores.

**Committed Store Processing:** The LSQ buffers all stores until they can be architecturally visible. When the global control sends a message to the DTs to commit the stores, the stores are picked one-by-one in program order at each of the DTs and committed to the L1 caches/miss buffers. Note that although, the stores commit in program order at each partition, they can go out-of-order between the partitions. When all the stores have been committed, the LSQ reports this information to the global control logic so that the resources allocated for the block can be freed across the processor.

### 2.5.3 Design Rationale

**Multiple forwarding in the LSQ:** Many superscalar processors avoid the complex processing required for multiple forwarding by either simply flushing [39] or replaying [78] the load when multiple matching stores are detected in the LSQ. This strategy is not feasible in a block-atomic architecture like TRIPS because it can lead to deadlocks. If there are partial matches within a block, the load will not execute until the matching stores are drained, but the matching stores cannot be drained because the load and its dependents may have to execute to produce the

block outputs.

**Unified LSQ:** Recently several processors have supported memory ordering using separate load and store buffers to increase the bandwidth and decrease the power per access. TRIPS, however, uses a unified LSQ because separate buffers will require more than 32 bits for encoding the memory instructions. This is because implementing separate LD and ST buffers requires that each memory instruction carry two different types of age tags; one tag encoding the global age and a second tag encoding the relative load/store ages. Alternatively, one can partition the TRIPS LSQ in LQ/SQ with the instruction encoding fixed at 32 bits. This strategy is disadvantageous because it restricts the number of memory instructions in the block and places hard restrictions on the number of loads and stores separately in block due to the reduced number of bits available for both the tags.

**Maximally sized LSQs:** Although on average only one fourth of the total memory instructions are expected to reach any DT partition, the LSQ in each DT is maximally sized. There are two microarchitectural reasons motivating maximally sized LSQs:

1. Deadlock avoidance: If the LSQs are undersized then with speculative execution, younger memory instructions may arrive earlier than the older instructions and may take up all slots in the LSQ preventing the older instructions from completing and eventually stalling forward progress.

2. Design Simplicity: When we started the project, it appeared to us that maximally sized conventional age-indexed LSQs would pose the least design risk

because they were well-understood and straightforward to implement.

In research conducted after the prototype implementation, described in Chapter 3, we have developed complexity-effective methods to safely reduce the LSQ size without causing deadlocks.

## 2.6 Miss Handling Unit Microarchitecture

The Miss Handling Unit (MHU) plays a key role in sustaining high levels of memory parallelism by managing multiple, outstanding L1D load and store misses. The MHU sends L1D miss requests to the L2 cache via the On Chip Network (OCN) – the chip data transfer fabric – and receives read data (for load misses) and write acknowledgements (for store misses) from the L2 cache. While much of the TRIPS MHU functionality is typical of out-of-order processors, the use of on-chip networks imposes different correctness and performance requirements.

### 2.6.1 MHU State

Figure 2.7 illustrates the components of the MHU. The MHU in each DT contains (1) sixteen MSHRs that hold information on each of the missed loads, (2) four 64-byte fill buffers that hold cache lines returning from L2 (on the OCN) before it is written to the L1D cache, (3) a four entry FIFO load request queue (LRQ) that decouples fill buffer allocation from the load miss processing, (4) a 64-byte store merge buffer that can coalesce multiple stores to the same cache line before sending it to the L2, (5) a 64-byte store transmit buffer that is used as scratch storage for holding the coalesced writes while the OCN packets are being created and sent, and

(6) a 64-byte spill buffer that holds one cache-line worth of data and is used to temporarily buffer cache spills before sending them out on the OCN.

### 2.6.2 MHU Operations

The MHU is capable of filtering redundant read requests (for load misses) and coalescing smaller write updates (store misses) into larger chunks before sending them out into the OCN. Both of these optimizations are critical to improving the packet efficiency and utilization of the OCN.

**Load Miss Processing:** On a cache load miss, the MHU allocates an MSHR to hold the information pertaining to the load. If there are no pending requests to the same cache line, the load is placed in the LRQ . To avoid deadlock conditions, the pipelines in the DT ensure that requests accepted in the DT can always be allocated in the MSHRs and the LRQ. When a free fill buffer entry becomes available, and the OCN port is available, a fill buffer is allocated for the transaction, the load is removed from the request queue and a read request is sent on the OCN.

When a read reply returns on the OCN, the reply data is placed in the fill buffer allocated for that transaction. As the data arrives from the OCN, the load(s) corresponding to the missed data are identified in the MSHR, packaged as if it were a load entering the DT for the first time and sent to access the caches and the LSQ. Re-injecting the load as new loads significantly simplifies bypassing between stages and correctness reasoning in the MHU.

**Load Return:** When a read reply returns on the OCN, the reply data is placed in the fill buffer allocated for that transaction. For each requested cache line, the

OCN reply consists of a header packet with the address, followed by four flits of data, 128 bits each. Since the address of the flit arriving the next cycle is known a cycle earlier (from the header), it is used to awaken pending loads in the MSHRs. Thus the MHU can generate wakeup signals for loads that require data from the flit arriving in the next cycle. All the pertinent load information is read from the MSHRs, packaged as if it were a load entering the DT for the first time and used to access the caches and the LSQ. If more than one load matches a flit, then one new load can be inserted into the main load pipeline each cycle after wakeup. As the missed loads are processed, the data flit corresponding to the load is written to the cache. A fill can result in a spill from the cache and each DT contains a spill buffer that temporarily holds the cache line until it can written to the memory system.

**Store Coalescing:** A OCN transaction requires one header flit and between one and four data flits depending on the size of datum. To minimize the overhead of header packets, the MHU attempts to create larger packets by coalescing multiple stores misses to same cache line. This functionality is provided by the merge and transmit buffers in the MHU. If the L1 store miss is to the same cache line as the cache line currently in the store merge buffer, then the missed store updates the merge buffer. If the store miss is to a different cache line, then the line in the merge buffer is moved to the transmit buffer and the new store is allocated in the merge buffer. Once a line is moved into the transmit buffer, the MHU logic scans and packages the cache line into fewest possible flits.

**MHU Coherence Policies:** As the MHU handles both L1 load and store misses concurrently, it needs to ensure coherency between load and store misses to the

same address. The load misses and store updates to the same cache line can arrive in three different orders at the MHU: 1) the store update arrives before the load request to the same address, 2) the store update arrives after the load request, or 3) both the load request and store update arrive at the same time.

In case 1, instead of forwarding to the load from the merge buffers, the merge buffers are flushed to the memory system. The load is then issued to the memory system as usual. This strategy avoids building complex forwarding logic between the merge buffer and the incoming load request for an uncommon execution scenario. In case 2, the strategy adopted for case 1 will not work because the load request may have already read the L2 caches and outside of the L1 none of the structures have store-to-load forwarding capabilities. In this case, the store updates the corresponding bytes of the matching fill buffer entry. When the load data arrives, it ensures that the store update bytes are not overwritten. In case 3, the store update is held back a cycle so that it becomes a write-after-read case as described in case 2.

### 2.6.3 Design Rationale

**Why fill buffers?** For deadlock-free operation, the MHU should not have any resource dependences on the OCN. Namely, the MHU can never refuse to accept incoming OCN replies while waiting for the OCN to accept new outgoing requests. Fill buffers guarantee deadlock-free operation by providing support for decoupling the OCN fills from rest of the MHU as they are always allocated prior to generation. Another alternative is to directly fill from the OCN into the caches. This strategy requires either a dedicated cache port for fills or mechanisms to premptively acquire cache ports. The former is undesirable because of area constraints and latter because

of the high complexity.

**Why Load Request Queues?** LRQ's are used to improve the utilization of the fill buffers and to avoid conservatively stalling on mergeable misses. For deadlock-free operation, two slots have to be reserved in all MHU load handling structures, so that loads already in the pipeline can be slotted. Pre-reserving two slots in the fill buffers can be constraining because there are only four fill buffers. Providing more fill buffers is expensive in terms of area and restricted by timing constraints. To work around this problem, the fill buffer allocation is decoupled from the load execution by using the request queues and pre-reserving slots in the request queue. Pre-reserving slots is area-efficient and timing-efficient because the LRQ is smaller than fill buffers (45 bits vs 512 bits).

**Sizing of structures:** The number of MHU entries was chosen so as to saturate the link between the DTs and the OCN. For uninterrupted traffic on the link, the MHU should have sufficient MSHRs to hold all incoming loads between the L1 miss detection and L1 reply for a load. Assuming one load is issued every cycle in this interval, and given that the average round trip for miss handling is 14 cycles, we would need fourteen MSHRs. An additional two MSHRs are required for deadlock-free operation, bringing the total number of MSHRs to sixteeen. If all of these loads are to different cache lines, then a 16-entry fill buffer is required. However, to meet cycle times and constraints of our ASIC methodology we restricted the number of fill buffers to four. This is unlikely to be performance critical because load misses are commonly clustered and contiguous; hence it is uncommon to have 16 back-to-back loads to different cache lines at one DT partition.

## 2.7 Physical Design Results

Each DT has an 8KB, 2-way set associative cache with 64 byte lines. The cache has one read and one write port. Each LSQ bank contains 256 entries and consists of CAM and RAM arrays. The CAM is physically constructed out of eight 32-entry CAMs (one per block) and has a read, write and search port. The DTLB is a 16 entry, 48-bit wide CAM with two search ports (one load and one store), one read port and one write port. The predictor is built using a single ported 1024-bit array.

**Area:** Figure 2.8 shows the DT floorplan after synthesis, timing and layout optimizations. The design was implemented using IBMs 130nm ASIC process and the DT measures 3.37mm $\times$ 1.188mm. The CAM is entirely synthesized from flip-flops and occupies a large fraction of the DT area. A Custom CAM is likely to be smaller than the synthesized CAM, but our design methodology did not support integration of custom CAMs into the design flow.

**Timing and Critical Paths:** The design synthesized to 3.2ns under worst-case process parameters and operating conditions. The top critical path is the detection of store forwarding in the LSQ and the generation of signals for stalling the DT pipelines during store forwarding. At a high level, this process involves generating an eight-bit mask that encodes the blocks older than the load, ANDing with another eight-bit mask that encodes blocks with matching stores, and then performing a cumulative OR on the resulting 8-bit mask. This process takes up roughly 45% of the cycle time. Then, the forwarding signal must be distributed to rest of DT pipelines to stall conflicting operations. The high fanout on this signal contributes significantly (27% of cycle time) to the total delay.

The next most critical path is the logic for extracting and packaging store misses to the L2. There are two components to the delay, the first involves the actual extraction process and the second that is a stall signal that is asserted when a multi-cycle OCN transaction has to be generated. For higher frequency implementations, adding an extra stage to the pipeline can eliminate the critical path, without affecting performance because the store writes are unlikely to be critical operations.

The third most critical path involves checking for coherence in the MHU and performing coherence updates. The source of the problem is not the delay associated with the coherence checks, but the late availability of the load address that drives the coherence checks. Speculatively performing the coherence checks a cycle earlier can result in false positives, which can complicate the design. Adding another stage (for the uncommon case of a coherence violation) in the pipeline will reduce performance by increasing the latency of missed loads.

## 2.8 Summary

The TRIPS microarchitecture includes a primary memory system that is fully partitioned and capable of supporting high levels of memory level parallelism. The memory system is made up of four Data Tiles (DTs), partitioned by interleaving based on addresses of the memory instructions. To support high levels of memory level parallelism, the DT utilizes memory side dependence predictors, deep LSQs, and an aggressive miss handling unit capable of supporting up to 16 outstanding load misses per DT (64 per core). The design, implementation and verification required 21 person months. Our design experience suggests that the design complexity of

the partitioned memory system is comparable with the complexity of a centralized memory system.

A completely partitioned memory system like TRIPS provides a complexity-effective way of increasing the capacity and bandwidth of the memory system by simply increasing the number of partitions. For instance, eight loads/stores per cycle can be supported on TRIPS with eight DTs. However, for such partitioning to be beneficial and feasible (1) the memory instructions should be placed close to the cache banks to which their addresses map, (2) the area and power overheads from replicated structures like LSQs should be minimized, and (3) the mechanisms used for communicating across the partitions should scale.

The TRIPS compiler team is currently investigating techniques for placing the memory instructions closer to DTs by array alignment analysis and sophisticated profile driven optimizations. In the following chapters, we describe solutions to area and power overheads of the LSQs Efficient store-tracking mechanism for sixteen or more partitions is a challenging problem and discussed more in future work. Solutions to these problems are the last few remaining steps towards scalable and completely distributed memory systems.

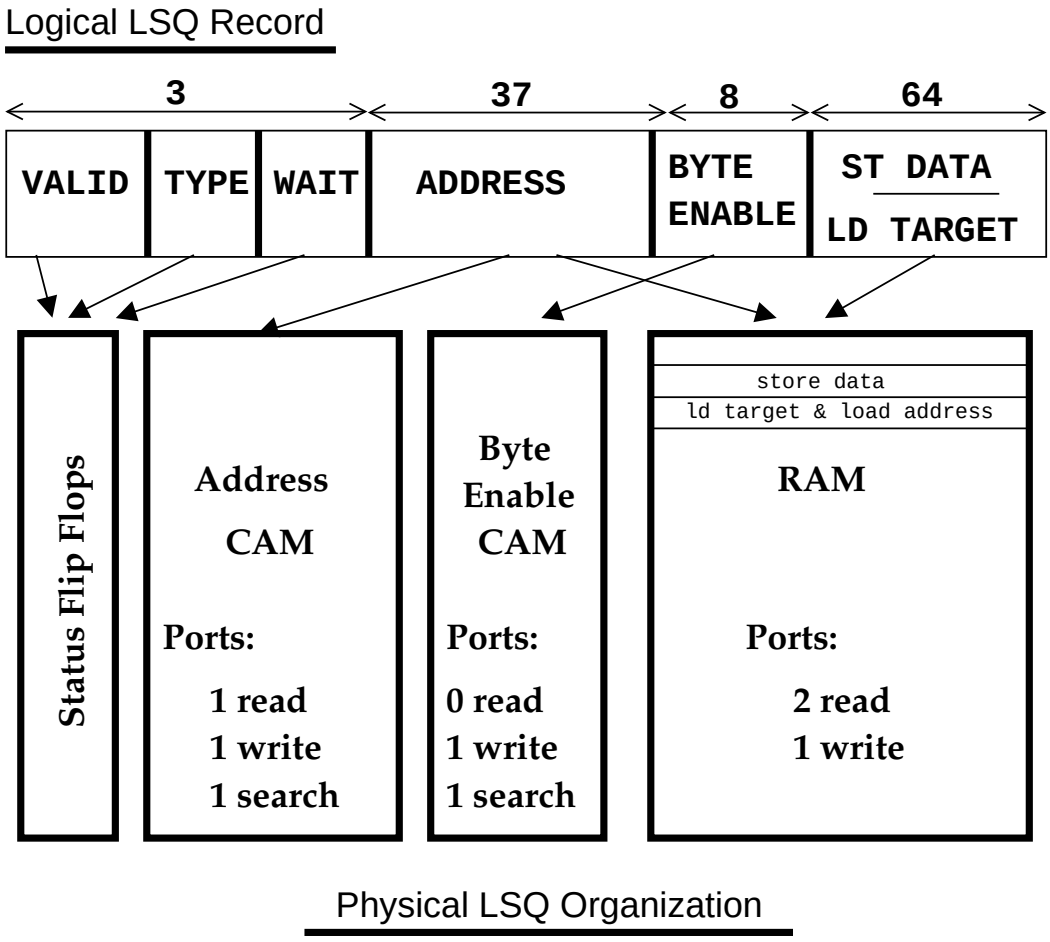



Figure 2.5: Logical and Physical Organization of the LSQ

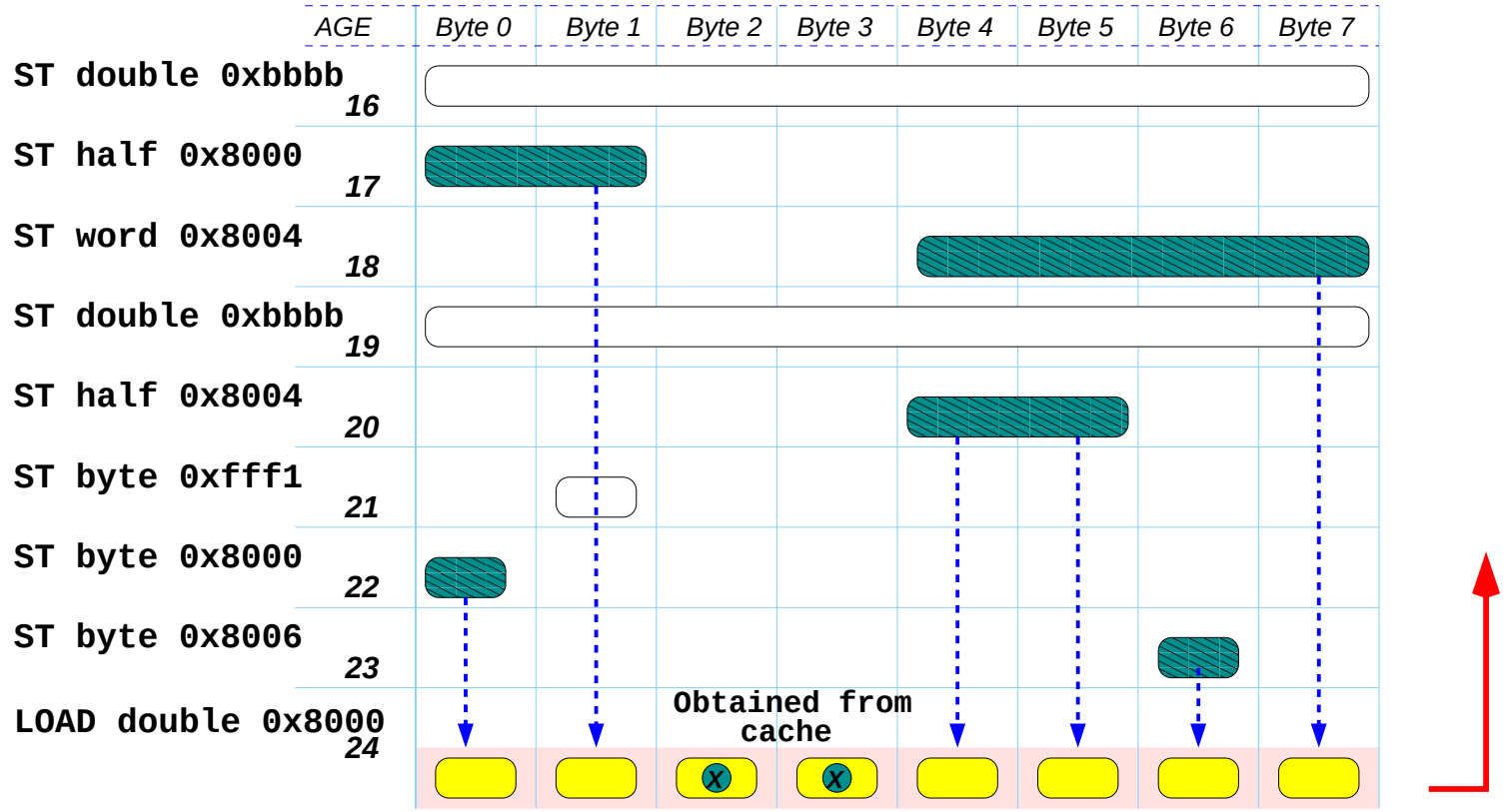



Figure 2.6: Multiple stores forwarding to loads in the LSQ

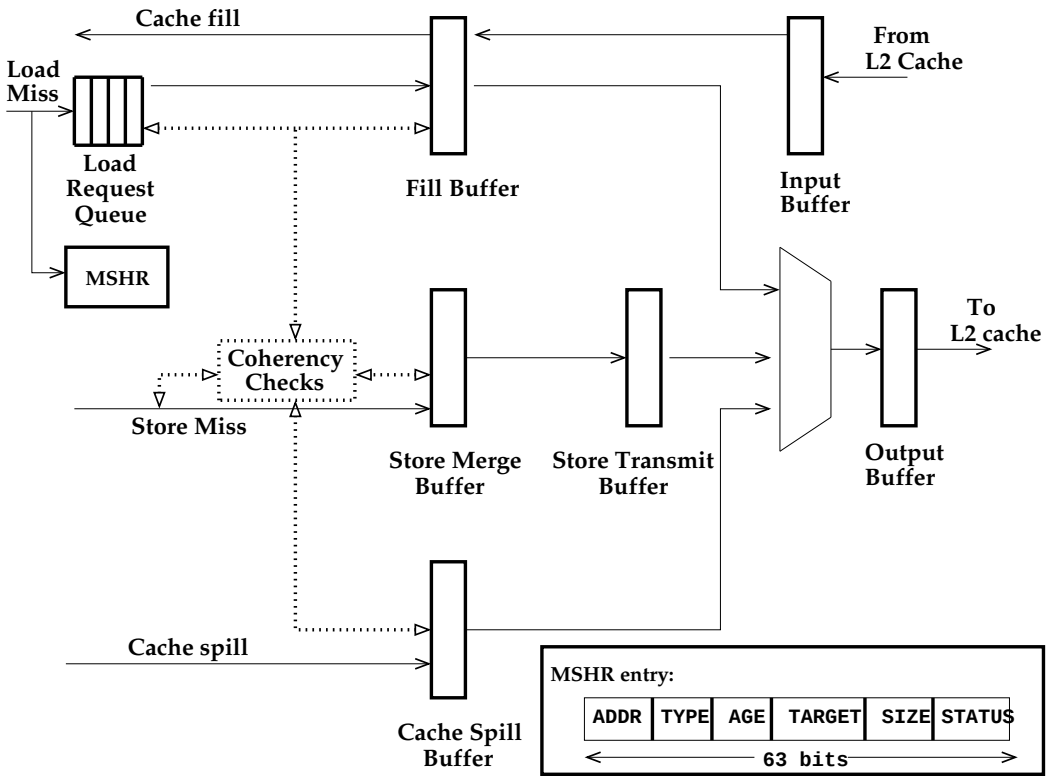



Figure 2.7: Block diagram of the MHU. Inset shows the logical structure of the MSHRs

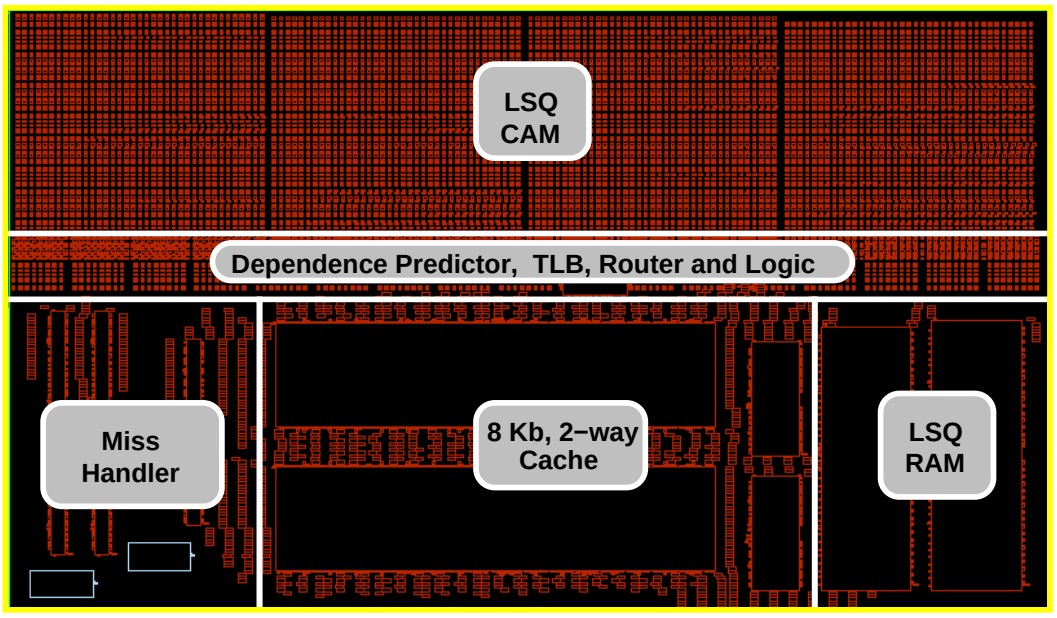



Figure 2.8: Major structures in the DT floorplan

# Chapter 3

# Late-Bound, Distributed LSQs

This chapter describes an LSQ design that provides significant area improvements over conventional LSQ designs. Two new techniques are used in these LSQs: *unordered late binding*, in which loads and stores allocate LSQ entries after they issue, and *lightweight overflow control*, which enables good performance with unordered LSQs that are divided into multiple small, address-interleaved partitions.

**Late binding:** Traditional LSQs allocate entries at fetch or dispatch time, and deallocate them at commit. Thus, entries in a conventional LSQ are physically *age-ordered*, a feature that LSQ designs exploit to provide their necessary functionality efficiently. When an LSQ reaches capacity, the microarchitecture typically throttles fetch until a load or store commits and is removed from the LSQ. Figure 3.1 shows a simple six-stage pipeline diagram with nine memory instructions (loads and stores labeled A through I) in different stages. As shown in Figure 3.1a, a conventional eight-entry LSQ is full after H is dispatched, stalling the fetch of later instructions.

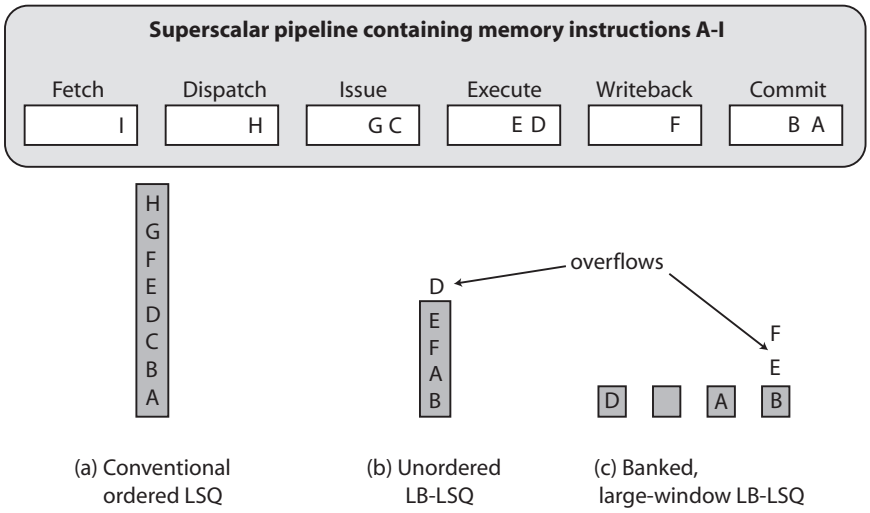



Figure 3.1: High-Level Depiction of Ordered vs. Unordered LSQs.

Unordered, late-binding LSQs (ULB-LSQs) can reduce both the average occupancy time and average number of entries in use. ULB-LSQs achieve this reduction by allocating entries only when a load or store issues, instead of when it is fetched, permitting a smaller, more efficient structure. Figure 3.1b shows the smaller ULB-LSQ that must be sufficiently large only to capture the in-flight loads and stores after they have issued. Figure 3.2 shows the potential savings of late-binding, issue-time allocation. Using this approach, on the Alpha 21264 microarchitecture, only 32 or fewer memory instructions must be buffered in the LSQ for 99% of the execution cycles, even though the original 21264 design had a combined LQ/SQ capacity of 64 entries. On the TRIPS architecture, for a synthetic benchmark that equally distributes memory references between the four partitions, the cumulative number of physical LSQ entries required is 224 whereas the TRIPS prototype implementation consists of 1024 physical entries.

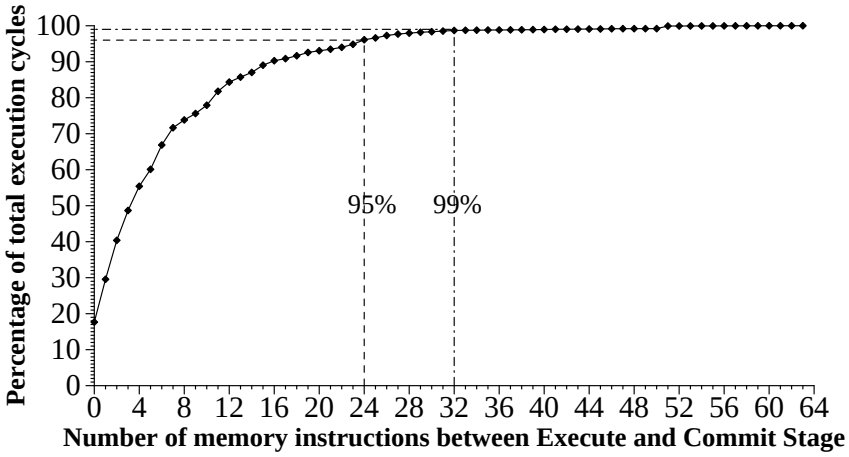



Figure 3.2: Potential for undersizing: In an Alpha 21264, for 99% of the execution cycles across 18 SPEC2000 benchmarks, only 32 or fewer memory instructions are in flight between execute and commit.

To achieve this reduction, however, the entries in an ULB-LSQ must be *unordered*; loads and stores are allocated LSQ entries in their dynamic issue order as opposed to the traditional in-order fetch sequence. Maintaining age order with issue-time allocation would require a complex and power-inefficient compacting-FIFO-like circuit [5]. We describe an ULB-LSQ design that requires only small additional complexity while performing comparably to a traditional ordered LSQ.

To compensate for the lack of ordering, the UB-LSQs take a more direct approach to determining the age by explicitly storing the age information of instructions in a seperate age CAM. The age CAM is a special type of CAM that can output greater/lesser/equal results instead of just equality matches. Using this age CAM, violation and commits can be supported in the ULB-LSQ with the same latencies as in the traditional LSQs but forwarding from multiple stores incurs ad-

ditional latencies. However, this additional latency does not increase performance because for more than 99% of the loads match with one or fewer stores.

**Low overhead overflow handling:** A second advantage above and beyond the reduced size is that ULB-LSQs lend themselves naturally to address partitioning, enabling smaller banks that are indexed by address. However, smaller banks will experience more overflows. In small-window superscalar processors, flushing the pipeline on an ULB-LSQ overflow is an acceptable policy, since the ULB-LSQ can be sized to save area and power over a conventional design while overflowing infrequently. However, ULB-LSQs can also provide efficient memory disambiguation for large-window processors, in which thousands of instructions may be in flight [66, 21, 71], by exploiting the ability to address interleave LSQ banks. The late binding and unordered organization are both necessary to support an undersized, address-interleaved LSQ, since the mapping of loads and stores to LSQ banks cannot be known at fetch/dispatch time.

In large-window designs, however, the probability of overflows grows, since the global ULB-LSQ is divided into smaller interleaved partitions that are more susceptible to overflows when the distribution of loads and stores to banks is unbalanced. Figure 3.1c shows how load imbalance for an address-interleaved ULB-LSQ increases the likelihood of overflow. In the example, each of four address-interleaved partitions holds one entry. Whereas instruction $E$ could have been held in the centralized ULB-LSQ of Figure 3.1b, $E$ conflicts in the banked example with instruction $B$, which maps to the same partition.

As the banks are shrunk and the number of overflows increases, the conventional techniques of throttling fetch or flushing the pipeline become too expensive.

By incorporating techniques to handle overflows with low overhead, an ULB-LSQ for a large-window design can consist of small partitions, resulting in low energy per load or store, while still supporting high performance across a large-window processor. In a distributed, large-window processor, LSQ banks can be treated as clients on a micronetwork, with overflows handled using traditional network flow-control techniques adapted for a distributed processor microarchitecture. We evaluate three such techniques in this chapter: *instruction replay*, *skid buffers*, and *micronet virtual channels*. These three techniques correspond to buffering the overflowing memory instructions at the execution units, or in extensions to the memory units, or in the network connecting the execution units to the memory units. The buffering space is much less expensive than the LSQ space since the buffered locations need not be searched for memory conflicts, which mitigates the area and energy overheads of large LSQs. To intuitively understand these schemes, one can imagine these schemes to be similar the "load loops" proposed by Borch et al. [12] but applied to the context of distributed architectures.

These buffering approaches effectively stall processing of certain memory instructions, which could potentially lead to deadlock. To avoid deadlocks, each of the proposed techniques, partitions the in-flight memory operations into age-ordered bins. For example, a window supporting up to 256 memory operations in-flight might assign each consecutive 32 loads or stores to one bin, permitting up to eight bins total. The oldest bin is the *high-priority* bin. The LSQ banks handle loads or stores from the high-priority bin differently from low-priority bins to guarantee forward progress. High-priority operations are either provided reserved space in each LSQ bank, or, on a high-priority overflow, cause a rare pipeline flush

followed by re-fetching of the high-priority operations and a temporary throttling of subsequent operations.

The techniques presented in this chapter work well for both small and large-window processors. For an Alpha 21264 microarchitecture with an 80-instruction window, a 32-entry ULB-LSQ using a flush-on-overflow policy provides the same performance as a 64-entry split LQ/SQ. For the 1,024-instruction window TRIPS processor, four banks of 48 entries each–using virtual channel flow control to handle overflows–provides the same performance as an idealized 1,024-entry LSQ.

The rest of this chapter is organized as follows: Section 3.1 re-visits the design of traditional LSQ implementations and identifies the deficiencies of these organizations. Section 3.2 describes the microarchitecture of the unordered, late-binding LSQ. Section 3.3 describes techniques for handling overflows in unordered LSQ. Section 3.4 measures the performance of the unordered LSQ for both superscalar and TRIPS processors.

## 3.1 Re-visiting Traditional LSQ Organizations

Most LSQs designed to date have been age indexed, because age indexing supports the physical sorting of instructions in the LSQ based on their age. The physical ordering makes some of these operations simpler to support but is not fundamentally required for any of the LSQ operations.

In an age-indexed LSQ, the address and value of an in-flight memory instruction is stored in an LSQ slot obtained by decoding the age of the memory instruction. This organization results in a LSQ that is *physically* ordered; the relative ages of two instructions can be determined by examining the physical locations

they occupy in the LSQ. For example, it is simple to determine that an instruction at slot 5 is older than an instruction at slot 8 because slot 5 is physically "before" slot 8. Additionally, this mechanism allows determining the relative order between all instructions in the LSQ that satisfy some criterion (i.e. a matching address). For example, if slots 25, 28 and 29 are occupied in the LSQ, linearly scanning the the LSQ from position 29 will reveal the most recent older instruction first (28) and then then next oldest (25) and so on. In some cases, circuit implementations exploit the physical ordering to accelerate LSQ operations. To understand the design changes that an unordered LSQ requires, it is instructive to examine how LSQ ordering supports the three functions that the LSQ provides: committing speculative stores, violation detection and forwarding from in flight stores to loads.

**Commit:** The LSQ buffers all stores to avoid potential write-after-write hazards between stores to the same address that execute out-of-order. Additionally, the stores cannot be written out until they are non-speculative. Once a store is determined to be non-speculative, the store address and value are written to the cache using the age supplied from the ROB in the case of superscalars and using the store arrival information and their ages which are available at each DT in the case of TRIPS. With ordering, age-based indexed lookup is sufficient. Without ordering, a search is necessary to find the oldest store to commit.

**Violation Detection:** The LSQ must report a violation when it detects that a load following a store in program order, and to same address, executed before the store. To support this operation, the LSQ buffers all in-flight memory data addresses, and when a store arrives at the LSQ, checks for any issued loads younger

| Operation | Search | Input | Output | Num matches | Sorting Required |
|---|---|---|---|---|---|
| Forwarding | $\geq$ | LD age | Older STs | $\geq 1$ | Yes |
| Violation | $\leq$ | ST age | Younger LDs | $\geq 1$ | No |
| Commit | == | ROB age | ST to commit | 1 | No |

Table 3.1: LSQ operations and ordering requirements.

than and to the same address as the store. If there are any matching loads, a violation is reported. For this operation, the LSQ need only determine the set of younger load instructions. It does not require sorting among the multiple matching loads based on their age. In ordered LSQ circuit implementation, the age of the incoming instruction is decoded into a bit mask and all bits "before" the decoded bit vector are set. In the case of store forwarding with multiple matches, the most recent store and successive older stores can be accomplished by linearly scanning the bit mask.

**Store Forwarding:** The LSQ must support forwarding from the uncommitted buffered stores in the LSQ to in-flight loads that issue after the stores. When a load arrives at the LSQ, it checks for older stores to the same address. If there are matching stores, the LSQ ensures that the load obtains its value from the most recent matching store preceding the load. To support this functionality when a load matches multiple stores, the LSQ sorts the matching stores based on their ages and processes the matching stores until all the load bytes have been obtained or until there are no matching stores.

The age-indexing policy requires an LSQ that is sized large enough to hold all in flight memory instructions ($2^{age}$ slots), which results in a physically ordered

LSQ. The ability to sort through multiple matching instructions is especially useful for forwarding values from multiple matching stores to a load, but a coarser age comparison is sufficient for implementing the other LSQ operations (Table 3.1). Additionally, the LSQ allocation policy is conservative. Even though the LSQ slots are occupied only after the instructions execute they are allocated early, during instruction dispatch. Traditional age-indexed LSQs are thus both over designed in terms of functionality and over provisioned in terms of size. In addition, these traditional LSQs are not suitable for partitioning because each partition should have the ability to hold all memory instructions, which increases the size of the LSQ unsustainably as in the case of the TRIPS LSQ.

## 3.2 An Unordered, Late-Binding LSQ Design

Late-Binding LSQs address the inefficiencies resulting from the "worst-case" design policies used in traditional LSQs. By allocating the memory instruction in the LSQ at issue, the sizes of ULB-LSQs are comparatively reduced. Allocating memory instructions at issue requires a set of mechanisms different from allocation in age-indexed LSQs. When a load or store instruction arrives at the ULB-LSQ the hardware simply allocates an entry from a pool of free LSQ slots instead of indexing by age. This allocation policy results in an LSQ that is physically *unordered* in which the age of the instruction has no relation to the slot occupied by the instruction.

To compensate for the lack of ordering, the ULB-LSQs take a more direct approach to determining the age by explicitly storing the age information in a separate age CAM. The age CAM is a special type of CAM that can output greater/lesser/equal results instead of just the equality matches. The LSQ functions

that used the implicit age information in the age-indexed LSQ for implementing the LSQ operations now use the explicit associative age CAM to determine younger and older instructions. Figures 3.3 and 3.4 illustrates and contrasts the structures used in the ULB-LSQ and traditional LSQ implementations, where $M$ is the memory instruction window size, and $U$ is the ULB-LSQ size.

To support commiting of stores in a superscalar processor, the age CAM is associatively searched with the age supplied by the ROB. The address and data from the exact matching entry are read out from the CAM and RAM respectively, and sent to the caches. This extra associative search is avoidable if the baseline architecture holds the ULB-LSQ slot id allocated to the store in the ROB. On a TRIPS like microarchitecture, the age CAM is searched the age of the store which is obtained by searching the store arrival vector.

To support violation detection, when a store arrives it searches the address CAM to identify matching loads, and searches the age CAM using the greater-than operator to identify younger loads. The LSQ then performs a logical OR of the results of the two searches. If any of the resulting bits is one then a violation is flagged. Detecting violations is simpler in this LSQ compared to age-indexed LSQs, since no generation of age masks is necessary.

Supporting forwarding is more involved because the ULB-LSQ does not have the total order readily available. In the case of only one match, the loss of order does not pose a problem; however when there are multiple matches, the matches must logically be processed from most recent to the oldest. In the ULB-LSQ, on multiple store matches, the age of each match is read out from the ULB-LSQ, one per cycle, and decoded into a per-byte bit vector. Bytes to forward to the load

replace bytes from other stores if the later-found store is more recent in program order. This step reconstructs the physical ordering between the matches from the ULB-LSQ, but may take multiple cycles to do so. Once the ordering is available, store forwarding proceeds exactly as in an age-indexed LSQ. Thus, compared to the age-indexed LSQ, which may already require multiple cycles to forward from multiple stores, the ULB-LSQ requires additional cycles for creating the decoded bit-vector. However, as we will show in the result section, these additional cycles rarely affect performance because multiple store forwarding is uncommon in many benchmarks.

In addition, ULB-LSQ must provide special mechanisms to handle overflows and detect deadlocks. Overflows can result if the LSQ is sized smaller than the maximum number of in flight instructions and if a in flight memory instruction arrives at a full LSQ. Deadlocks can result when the oldest memory instruction causes an overflow. When the oldest instruction overflows, it may not be possible to slot the instruction into the LSQ because none of the younger instructions can commit before the oldest instruction can be committed. But the oldest instruction cannot be committed without being slotted into the LSQ. There are several solutions from simple pipeline flushes, to invalidating older instructions, to more involved forms of buffering. These solutions are discussed in the next section.

## 3.3 Handling LSQ Overflows

Prior solutions for dealing with overflows typically involved either fetch throttling or pipeline flushing. Fetch throttling is a preventive stalling strategy in which the map stage stalls when the number of unresolved loads or stores in flight is sufficient

to fill the LSQ completely. For example, if the LSQ can hold $N$ instructions, and there are $U$ unresolved instructions in the pipeline then LSQ must stall when there are $N - U$ instructions. While this is a reasonable strategy for single LSQ banks, it is inefficient for partitioned LSQs. In partitioned LSQs, fetch throttling must begin as soon as the number of unresolved in-flight memory instructions equals the minimum available space in any of the partitions. For example, if each load queue partition can hold $N$ loads, and the fullest partition contains $N - P$ loads, then the fetch must stall as soon as $P$ additional unresolved loads are put into flight. Since this approach effectively results in the usable global LSQ size being equal that of a single local partition, fetch throttling is a poor solution for distributed LSQs. The second traditional solution to handle partition overflows is to flush all the speculative instructions and restart execution after the non-speculative instructions commit. This approach is is also unattractive because, as we show later in this section, while flushing is acceptable for small window designs, flushing is too frequent for LSQ partitions that are considerably smaller than the in-flight window.

Ideally, a distributed LSQ should be divided into equal-sized banks, where the aggregate number of LSQ entries equals the average number of loads and stores in flight, but which shows only minor performance losses over a maximally sized LSQ. When a bank overflows, however, if the microarchitecture does not flush the pipeline or throttle fetch, it must find someplace to buffer the load. We examine three principal places to buffer these instructions: in the execution units, in extensions to the memory units, or in the network connecting the execution units to the memory units. The buffering space is much less expensive than the LSQ space since the buffered locations need not be searched for memory conflicts, which mitigates the

area and energy overheads of large LSQs. The penalty associated with these schemes correspond to different "load loops" and changes as the time for load execution changes [12].

These buffering approaches effectively stall processing of certain memory instructions, which could potentially lead to deadlock. However, memory instructions can be formed into groups based on age, with all of the instructions in a group having similar ages. In a microarchitecture that is block-oriented like TRIPS, the memory instruction groups correspond to the instruction blocks. One block is non-speculative, while multiple blocks can be speculative. By choosing to prioritize the non-speculative instructions over the speculative instructions, our solutions can reduce the circumstances for deadlocks and flushing. One possible design would reserve LSQ entries for the non-speculative block, but our experiments indicated that this approach did not provide any substantive performance benefits.

### 3.3.1 Issue Queue Buffering: Memory Instruction Retry

One common alternative to flushing the pipeline in conventional processors is to replay individual offending instructions, either by retracting the instruction back into the issue window, or by logging the instruction in a retry buffer. In TRIPS retrying means sending an offending instruction back to the ALU where it was issued and storing it back into its designated reservation station. Since the reservation station still holds the instruction and its operands, only a short negative-acknowledgement (NACK) message needs to be sent back to the execution unit. The issue logic may retry this instruction later according to a number of possible policies.

Figure 3.5a shows the basics of this technique applied to LSQ overflows.

When a speculative instruction arrives at a full LSQ, the memory unit sends the NACK back to that instructions execution unit. This policy ensures that speculative instructions will not prevent a non-speculative instruction from reaching the LSQ. If a non-speculative instruction arrives at a full LSQ, then the pipeline must be flushed.

A range of policies are possible for determining when to reissue a NACKed memory instruction. If the instruction reissues too soon (i.e. immediately upon NACK), it can degrade performance by clogging the network, possibly requiring multiple NACKs for the same instruction. Increased network traffic from NACKs can also delay older non-speculative instructions from reaching the LSQ partition, as well as general execution and instructions headed to other LSQ partitions. Alternatively, the reservation stations can hold NACKed instructions until a fixed amount of time has elapsed. Waiting requires a counter per NACKed instruction, and may be either too long (incurring unnecessary latency) or too short (increasing network contention). Instead, our approach triggers re-issue when the non-speculative block commits, which has the desirable property that LSQ entries in the overflowed partition are likely to have been freed. This mechanism has two minor overheads, however: an additional state bit for every reservation station, to indicate that the instruction is ready but waiting for a block to commit before reissuing; and a control path to wake up NACKed instructions when the commit signal for the non-speculative block arrives.

### 3.3.2 Memory Buffering: Skid Buffers

A second overflow-handling technique is to store memory instructions waiting to access the LSQ in a skid buffer located in the memory unit. As shown in Figure 3.5b, the skid buffer is simple priority queue into which memory instructions can be inserted and extracted. To avoid deadlock, our skid buffers only hold speculative memory instructions. If an arriving speculative memory instruction finds the LSQ full, it is inserted into the skid buffer. If the skid buffer is also full, the block is flushed. Arriving non-speculative instructions are not placed in the skid buffer. If they find the LSQ full, they trigger a flush.

When the non-speculative block commits and the next oldest block becomes non-speculative, all of its instructions that are located in the skid buffer must be extracted first and placed into the LSQ. If the LSQ fills up during this process, the pipeline must be flushed. Like retry, the key to this approach is to prioritize the non-speculative instructions and ensure that the speculative instructions do not impede progress. Skid buffers can reduce the ALU and network contention associated with NACK and instruction replay, but may result in more flushes if the skid buffer is too small.

### 3.3.3 Network Buffering: Virtual Channel-Based Flow Control

A third approach to handle overflows is to use the buffers in the network that transmits memory instructions from the execution to the memory units as temporary storage for memory instructions when the LSQ is full. In this scheme, the operand network is augmented to have two virtual channels (VCs): one for non-speculative traffic and one for speculative traffic. When a speculative instruction is issued at

an ALU, its operands and memory requests are transmitted on the lower priority channel. When a speculative memory instruction reaches a full LSQ and cannot enter, it remains in the network and asserts back pressure along the speculative virtual channel. Non-speculative instructions use the higher priority virtual channel for both operands and memory requests. A non-speculative memory instruction that finds the LSQ full triggers a flush to avoid deadlock. Figure 3.5c shows a diagram of this approach.

This virtual channel approach has a number of benefits. First, no new structures are required, so logic overhead is only minimally increased. Additional router buffers are required to implement the second virtual channel, but our experiments show that two-deep flit buffers for each virtual channel is sufficient. Second, no additional ALU or network contention is induced by NACKs or instruction replays. Third, the higher priority virtual channel allows non-speculative network traffic to bypass speculative traffic. Thus non-speculative memory instructions are likely to arrive at the LSQ before speculative memory instructions, which reduces the likelihood of flushing.

Despite its conceptual elegance, this solution requires a number of changes to the baseline network and execution engine. The baseline TRIPS implementation includes a number of pertinent features. It provides a single operand network channel that uses on-off flow control to exert back-pressure. Each router contains a four-entry FIFO to implement wormhole routing and the microarchitecture can flush any in-flight instructions located in any tile or network router when the block they belong to is flushed. Finally, all of the core tiles (execution, register file, data cache) of the TRIPS processor connect to the operand network and will stall issue

if they have a message to inject and the outgoing network FIFO is full.

Adjusting this network to support VCs requires several augmentations: (1) an additional virtual channel in the operand network to separate speculative from non-speculative network traffic, including the standard buffer capacity and control logic needed by virtual channels, (2) virtualization of the pipeline registers, which must stretch into the execution and register tiles to allow non-speculative instructions to proceed even if speculative instructions are stalling up the virtual network, (3) issue logic in these tiles that selects non-speculative instructions over speculative logic when the virtual network is congested, and (4) a means to promote speculative instructions from the speculative virtual channel to the non-speculative channel when its block becomes non-speculative.

The trickiest part of this design is the promotion of speculative network packets to the non-speculative virtual channel when the previous non-speculative block commits. The TRIPS microarchitecture already has a commit signal which is distributed in a pipelined fashion to all of the execution units, memory units, and routers. When the commit signal indicates that the non-speculative block has committed, each router must nullify any remaining packets in the non-speculative virtual channel and copy any packets belonging to the new non-speculative block from the speculative VC to the non-speculative VC.

## 3.4 UB-LSQ and Overflow Handling Evaluation

We first evaluate the performance of the ULB-LSQ on a small-window superscalar, a Alpha 21264-like processor, then measure the performance of the ULB-LSQ for the partitioned TRIPS processor. For the Alpha-21264, we use pipeline flushing and

for the TRIPS processor we report performance for pipeline flushing, memory skid buffering, NACK scheme and Virtual channel flow control.

### 3.4.1 UB-LSQ Small-Window Performance Results

The performance of the ULB-LSQ depends on the number of entries in the LSQ, which affects the number of LSQ overflows. Performance will also be affected by the relative cost of each overflow and the additional penalty for multi-cycle store forwarding (in the case of multiple address matches). We modeled the ULB-LSQ in the sim-alpha [22] simulator and simulated single Simpoint [69] regions of 100M for the 18 SPEC benchmarks compatible with our experimental infrastructure. The compatible benchmarks are listed in Table 3.2.

In this set of experiments, ULB-LSQ overflows are handled by partially flushing the pipeline and re-starting execution from the oldest unarrived memory instruction at the time of the overflow. The penalty of an overflow is 15 cycles, which matches the cost of a branch misprediction. Table 3.2 summarizes the unordered LSQ behavior and statistics. The first two columns show the average number of memory instructions between the commit and execute stages for 95% and 99% of the execution cycles. The next eight columns show the number of overflows per 1000 memory instructions, and the performance normalized against the Alpha, for ULB-LSQ sizes ranging from 16 to 40 entries. The final column shows the percentage of dynamic loads that forward from more than one store. From the table, for 14 of the 18 benchmarks, for 99% of the cycle time, there are 32 or fewer uncommitted but executed memory instructions. This explains why a 32 entry ULB-LSQ does not show any performance degradation for the benchmarks we examined.

| Benchmark | Occupancy | | Flushes per 1K mem instr | | | | Normalized IPC | | | | Baseline IPC | % of LDs matching 0 or 1 STs |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | %program cycles | | LSQ entries | | | | LSQ entries | | | | | |
| | 95% | 99% | 16 | 24 | 32 | 40 | 16 | 24 | 32 | 40 | | |
| 164.gzip | 7 | 22 | 6 | 4 | 3 | 1 | 1.00 | 1.00 | 1.00 | 1.00 | 1.60 | 99.95 |
| 175.vpr | 14 | 21 | 10 | 2 | 0 | 0 | 0.93 | 1.00 | 1.00 | 1.00 | 0.87 | 99.72 |
| 177.mesa | 8 | 14 | 8 | 0 | 0 | 0 | 0.05 | 1.00 | 1.00 | 1.00 | 1.16 | 99.8 |
| 178.galgel | 24 | 24 | 463 | 88 | 0 | 0 | 0.04 | 0.56 | 1.00 | 1.00 | 2.70 | 100 |
| 179.art | 35 | 43 | 651 | 131 | 33 | 11 | 0.76 | 0.91 | 1.00 | 1.00 | 0.63 | 99.95 |
| 183.equake | 3 | 6 | 0 | 0 | 0 | 0 | 1.00 | 1.00 | 1.00 | 1.00 | 0.96 | 99.91 |
| 188.ammp | 11 | 15 | 4 | 0 | 0 | 0 | 1.01 | 1.00 | 1.00 | 1.00 | 1.31 | 99.91 |
| 189.lucas | 11 | 13 | 0 | 0 | 0 | 0 | 1.02 | 1.00 | 1.00 | 1.00 | 0.76 | 100 |
| 197.parser | 10 | 17 | 4 | 0 | 0 | 0 | 1.00 | 1.00 | 1.00 | 1.00 | 1.17 | 99.84 |
| 252.eon | 15 | 20 | 28 | 2 | 0 | 0 | 0.94 | 1.00 | 1.00 | 1.00 | 1.17 | 98.58 |
| 253.perlbmk | 9 | 13 | 2 | 0 | 0 | 0 | 1.00 | 1.00 | 1.00 | 1.00 | 0.83 | 98.91 |
| 254.gap | 6 | 12 | 0 | 0 | 0 | 0 | 1.00 | 1.00 | 1.00 | 1.00 | 1.11 | 99.92 |
| 256.bzip2 | 7 | 9 | 0 | 0 | 0 | 0 | 1.00 | 1.00 | 1.00 | 1.00 | 1.82 | 99.9 |
| 173.applu | 22 | 25 | 2 | 1 | 0 | 0 | 0.99 | 1.00 | 1.00 | 1.00 | 0.62 | 100 |
| 181.mcf | 51 | 51 | 360 | 251 | 171 | 98 | 1.01 | 1.01 | 1.01 | 1.01 | 0.20 | 99.92 |
| 176.gcc | 31 | 32 | 28 | 22 | 2 | 1 | 0.99 | 1.00 | 1.00 | 1.00 | 1.21 | 99.88 |
| 171.swim | 15 | 15 | 1 | 0 | 0 | 0 | 1.00 | 1.00 | 1.00 | 1.00 | 0.88 | 100 |
| 172.mgrid | 19 | 35 | 23 | 10 | 4 | 2 | 1.07 | 1.00 | 0.99 | 1.00 | 0.87 | 99.98 |
| Average | | | | | | | 0.88 | 0.97 | 1.00 | 1.00 | 1.10 | 99.79 |

Table 3.2: Performance of an ULB-LSQ on an 80-window ROB machine.

To isolate the performance effect of slower multiple forwarding, we increased the latency of store forwarding in the baseline simulator without changing the LQ/SQ size. In this experiment, if a load matches with N (N > 1) older stores, then store forwarding takes an additional N cycles even if load does not have to get its data from all N stores. The results showed no performance degradation for any of the benchmarks because loads rarely need to get data from multiple stores. From Table 3.2, the number of loads that have to forward from two or more stores is less than 0.2% on the average.

**Power**

This section examines the dynamic power consumption of an ULB-LSQ against the Alpha LQ/SQ organization. The ULB-LSQ design holds 32 memory operations, while the Alpha has a separate 32-entry LQ and SQ. Although the size of the ULB-LSQ is smaller, each memory operation has to access the same number of address CAM entries in both designs because the Alpha has a partitioned LQ/SQ. Even so, the ULB-LSQ will have higher dynamic power per access because of the additional accesses to the age CAM.

To measure the increase in power per access, we assumed that the LSQ power is approximately equal to the power consumed by the CAM and the RAM. Thus, the power-per-access of the Alpha LQ or SQ will be purely from the address CAM ($P_{addr}$) and the RAM ($P_{ram}$), while the power consumed by the ULB-LSQ will be the power from the address CAM, the age CAM ($P_{age}$) and the RAM. Since the ULB-LSQ and Alpha have the same number of entries, their sizes will be the same. Thus the power increase will be $(P_{age} + P_{addr} + P_{ram})/(P_{addr} + P_{ram})$.

We synthesized the design using IBM's 130nm ASIC methodology with the frequency set at 450MHz, and verified that the age CAM will fit in the same cycle time as the address CAM. Even though the delay of the age CAM was approximately 20% more than the address CAM, the delay was still not long enough to be on the critical path. Thus, assuming that the LB-CAMs can be run at the same clock frequency as the traditional LQ/SQ, the power increase is simply the ratio of the capacitances. From our synthesis, the capacitance of the address CAM (32x40b, 1 search, 1 read and 1 write port) was 144.5pF ($P_{addr}$) while the capacitance of the age CAM was 12.86pF ($P_{age}$). Thus the power overhead over a traditional LSQ is roughly 8% even after neglecting the power due to the RAM.

**Implementation Complexity**

The ULB-LSQ design differs from a traditional LSQ in the following ways: (1) the entries are managed as a free list, (2) multiple store forwarding requires additional control logic to scan through matching entries, and (3) the LSQ must detect and react to overflows. These differences are not a significant source of complexity because many existing microarchitectural structures implement similar functionality. For example, MSHRs and the physical register files are managed as free lists. The scanning logic has been implemented in traditional LSQs which do not flush on multiple matches. Overflow is flagged when there are no free LSQ entries, and is simple to implement. Furthermore, the ULB-LSQ does not require any modifications to the load/store pipeline even for handling the case of variable latency matches. The LSQ operations are pipelined in the exact same way as the age-ordered LSQ implementation in the POWER4 [74].

#### Supporting Processor Optimizations

Although we have described only the basic functions of the ULB-LSQ it can also be used to support processor optimizations like rolling flushes, and split data and addresses.

Rolling flushes, i.e., early misprediction detection and recovery, can reduce branch misprediction latency. To support rolling flushes it must be possible to identify all instructions younger than the flushed instruction. This operation is similar to violation detection and can be accomplished by using the age CAM. Unless flushes are common, the ordering port on the age CAM can be shared with the logic for implementing flushes.

Memory instructions are sometimes split into separate address and data instructions to reduce the number of load dependence violations. The split instructions are likely to arrive at the LSQ at different times. When a new instruction arrives, the ULB-LSQ has to identify the slot allocated for the instruction that has already arrived and, if it finds one, use the same slot for the arrived instruction. This operation is similar to the commit processing and can be accomplished by searching the age CAM based on the age of the arriving instruction.

These results showed that for small-window processors like the Alpha 21264, even with simplistic overflow handling mechanisms, the queue sizes can be reduced by half without affecting performance.

### 3.4.2 Large-window Performance Results

We implemented the UB-LSQ and the flow control mechanisms on a simulator that closely models the TRIPS prototype processor which has been validated to be within

| Parameter | Configuration |
|---|---|
| Overview | Out-of-order execution with up to 1024 instructions in-flight, Up to 256 memory instructions can be simultaneously in flight. Up to 4 stores can be committed every cycle. |
| Instruction Supply | Partitioned 32KB I-cache 1-cycle hit. Local/Gshare Tournament predictor (10K bits, 3 cycle latency) with speculative updates; Local: 512(L1) + 1024(L2), Global: 4096, Choice: 4096, RAS: 128, BTB: 2048. |
| Data Supply | 4-bank cache-line interleaved DL1 (8KB/bank, 2-way assoc, writeback, write-around 2-cycle hit) with one read and one write port per bank to different addresses. Up to 16 outstanding misses per bank to up to four cache lines, 2MB L2, 8 way assoc, LRU, writeback, write-allocate, average (unloaded) L2 hit latency is 15 cycles, Average (unloaded) main memory latency is 127 cycles. Best case load-to-use latency is 5 cycles. Store forwarding latency is variable, minimum penalty is 1 cycle. |
| Interconnection Network | The banks are arranged in 5x5 grid connected by mesh network. Each router uses round-robin arbitration. There are four buffers in each direction per router and 25 routers. The hop latency is 1-cycle. |
| Simulation | Execution-driven simulator validated to be within 11% of RTL design. 28 EEMBC benchmarks, 12 SPEC benchmarks simulated with single simpoints of 100M |

Table 3.3: Relevant aspects of the TRIPS microarchitecture

11% of the RTL for the TRIPS prototype processor. The microarchitectural parameters most relevant to the experiments are summarized in Table 3.3.

For each benchmark, we normalize the performance (measured in cycle counts) to a configuration with maximally sized, 256-entry LSQ partitions that never overflow. For these experiments, we used skid buffer sizes that are sized slightly larger than the expected number of instructions at each partition; for a 256 instruction window, each partition will likely get 64 instructions. If the partition was sized to be

72 entries then it could absorb most of the overflows. The number of skid buffers was sized to (72 - LSQ partition size). For the virtual channel scheme, we divided the four operand network buffers in the baseline equally between the two channels. Thus two buffers are provided for the speculative and non-speculative virtual channels for the VC scheme. We present results for 28 EEMBC benchmarks (all except cjpeg and djpeg) and 12 SPEC CPU 2000 (ammp, applu, art, bzip2, crafty, equake, gap, gzip, mesa, mgrid, swim and wupwise) benchmarks with Minnespec [45] medium sized reduced inputs. The other benchmarks are not currently supported in our infrastructure.

For four 48-entry LSQs and thus a total LSQ size of 192 entries (25% undersized), the flush scheme results in average performance loss of 6% for EEMBC benchmarks (Figure 3.6) and 11% for SPEC benchmarks (Figure 3.7). The worst-case slowdowns are much higher: 180% for idct and 206% for mgrid. These results support the perspective that traditional flow-control mechanisms are inadequate for distributed load-store queues. The VC mechanism is the most robust with 2% average performance degradation and less than 20% performance degradation in the worst case. As expected, the skid buffer scheme performs better than the NACK scheme because it avoids network network congestion from the NACKed packets, at the cost of extra area.

For six of the SPEC and EEMBC benchmarks, the memory accesses are unevenly distributed and cause LSQ overflows that reduce performance significantly. For instance, Figure 3.8 shows a frequently executed code sequence in idct in the EEMBC suite. The innermost loop contains two reads and two writes to two different arrays and the code generated by the compiler aligns both arrays to 256-byte

boundaries. Since the arrays are accessed by same indices, all four accesses map to the same bank. This problem is exacerbated by the aggressive loop unrolling of the TRIPS compiler. The accesses could in theory be distributed by aligning the arrays differently, but aligning data structures to minimize bank conflicts is a difficult compiler problem.

Another common code sequence that causes uneven distributions is frequent use of static scalar variables. In general, static scalar variables cannot be register allocated (precise exceptions, consistency requirements etc.) and remain allocated to one bank for the lifetime of a program. Repeated use of static variables causes imbalances; for instance, if a static file pointer that is repeatedly accessed on memory file writes. Even XOR-based indexing functions cannot reduce overflows in this case. These two cases demonstrate that imbalances in partitioned memory systems cannot be easily detected and optimized for by the compiler and that low overhead hardware mechanisms are essential for managing LSQ overflows.

**Power**

Two mechanisms improve the power efficiency in large-window processor LSQs: address partitioning and late-binding. First, partitioning the LSQ by addresses naturally reduces the number of entries arriving at each memory partition. Second, late-binding reduces the number of entries in each partition by reducing occupancy. However, the additional power may be expended in the network routers because of the flow control schemes. Wang et al. [80] show a 20% increase in power for a four fold area increase when implementing virtual channels. The power increase primarily comes from having four times as many buffers for implementing the virtual

channels. In our scheme we do not increase the number of buffers. We simply divide the number of buffers equally between the virtual channels. Hence we do not expect significant overheads from the network, but without a real implementation of a VC router it is difficult to measure the power overheads. Factoring out the network measurements, the 48-entry UB-LSQ implementation would be at least four times more power-efficient, both in terms of static and dynamic power. In the next chapter we discuss methods to reduce the power even further and report more detailed measurements.

**Area**

Among the three overflow handling mechanisms, the NACK mechanism is the most area efficient if the issue window is designed to hold instructions until explicit deallocation. On the TRIPS processor, the NACK scheme requires cumulative storage of 1024 bits to identify the NACK'ed instructions (one bit for every instruction in the instruction window) and changes to the issue logic to select and re-issue the NACK'ed instructions. The VC mechanism is next best in terms of area efficiency. The area overheads of the VCs are due to the additional storage required for pipeline priority registers in the execution units to avoid deadlocks and the combinational logic in routers to deal with promotion. The VC scheme does not require any additional router buffers since the speculative channels divide the number of buffers in the baseline. The skid buffer scheme requires the largest amount of storage, although most of the structure can be implemented as RAMs. A 24-entry skid buffer supplementing a 40-entry LSQ increases the size of each LSQ partition by 4%. Overall, using best scheme to support a 1024 instruction window – the VC mechanism –

as shown in Table 5.4, the area after optimizations is 80% smaller compared to the fully replicated LSQs at each LSQ partition.

#### Implementation Complexity

The VC scheme requires the most changes to the baseline design as it requires virtualization of not only the network routers but also the execution units that feed the router. For instance, when the low priority channel in the network is backed up, the issue logic must supply the network with a high priority instruction even though it may be in the middle of processing a low priority instruction. The NACK scheme comes second or third depending on the baseline architecture – if the baseline holds the instructions in the issue queues until commit, implementing NACK is as simple as setting a bit in a return packet and routing it back to the source instead of the destination. However, if instructions are immediately deallocated upon execution from the windows, NACK may be considerably more complex. The skid buffer solution is probably the simplest of all the solutions: it requires some form of priority logic for selecting the oldest instructions, mechanisms for handling invalidations in the skid buffer and arbitration for the LSQ between instructions in the skid buffer and new instructions coming into the LSQ partition. Despite the changes required for the schemes described here, the mechanisms are feasible and operations required have been implemented in other parts of the processor.

## 3.5 Summary

In this chapter we discussed two techniques for improving the area of the LSQ. We showed that these techniques are effective for both small and large window

processors like TRIPS.

By performing late binding and allocating LSQ entries only at instruction issue, designers can reduce the occupancy and resultant size of the load/store queues. This reduction requires that the queues be unordered. While the unordered property requires some extra overhead, such as saving the CAM index in the ROB or searching for the load or store age at commit, the design is not intrinsically more complex, and can achieve performance equivalent to an ordered LSQ, but with less area and power.

However, the most promising aspect of the ULB-LSQ approach is its partitionability, which was the original impetus for this line of research. Address-interleaved LSQ banks should be both late-bound and unordered; the ULB-LSQ design naturally permits the LSQ to be divided into banks, provided that a mechanism exists to handle the resultant increase in bank overflows. We observed that, for distributed microarchitectures that use routed micronetworks to communicate control, instructions, and data, that we could embed classic network flow-control solutions into the processor micronetworks to handle these overflows. We evaluate three such overflow control handling schemes in the context of the TRIPS microarchitecture. The best of these schemes (virtual micronet channels) enables a scalable, distributed, load/store queue, requiring four banks of only 48 entries each to support a 1024-instruction window – a 78% reduction in number of LSQ entries compared to the TRIPS processor.

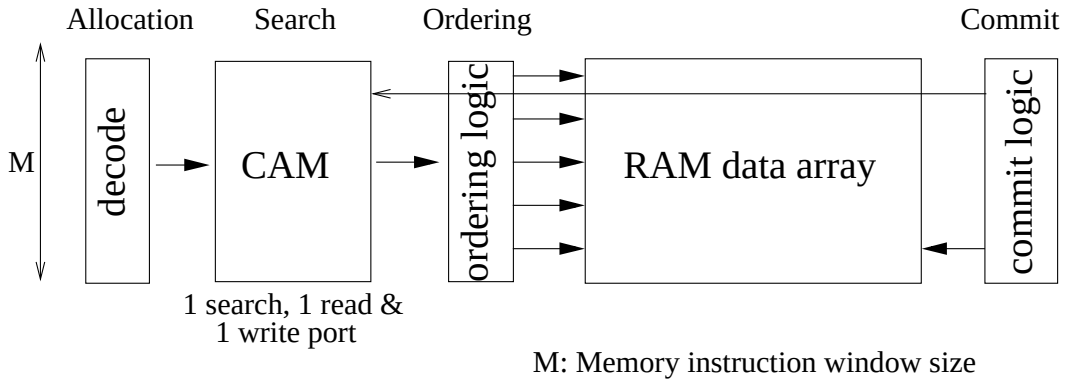



Figure 3.3: The Age-Indexed LSQ

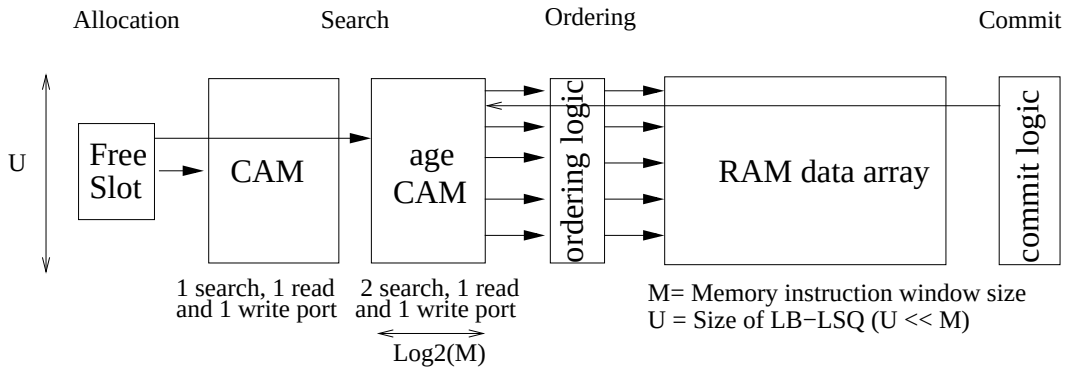



Figure 3.4: The ULB-LSQ Microarchitecture

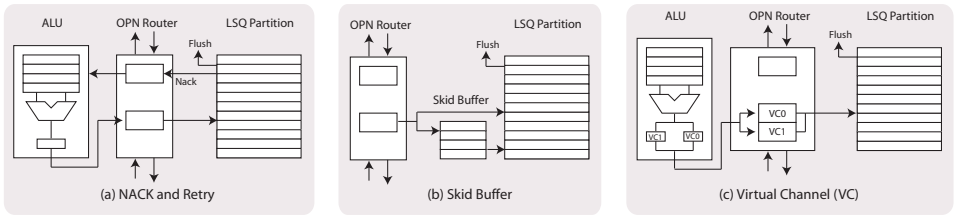



Figure 3.5: LSQ Flow Control Mechanisms.

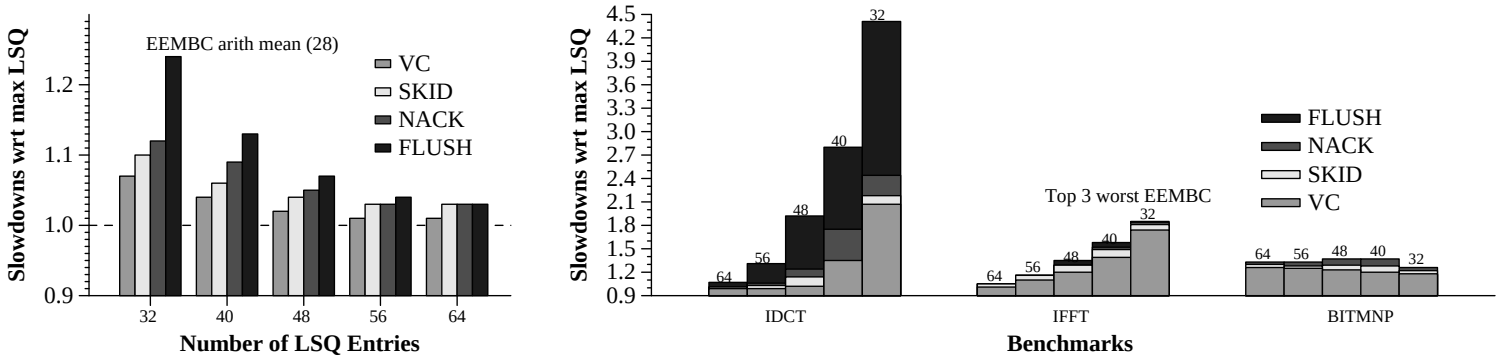



Figure 3.6: Left: Average LSQ Performance for the EEMBC benchmark suite. Right: Three worst benchmarks. bitmnp shows a different trend because there are fewer LSQ conflict violations in bitmnp when the LSQ capacity is decreased.

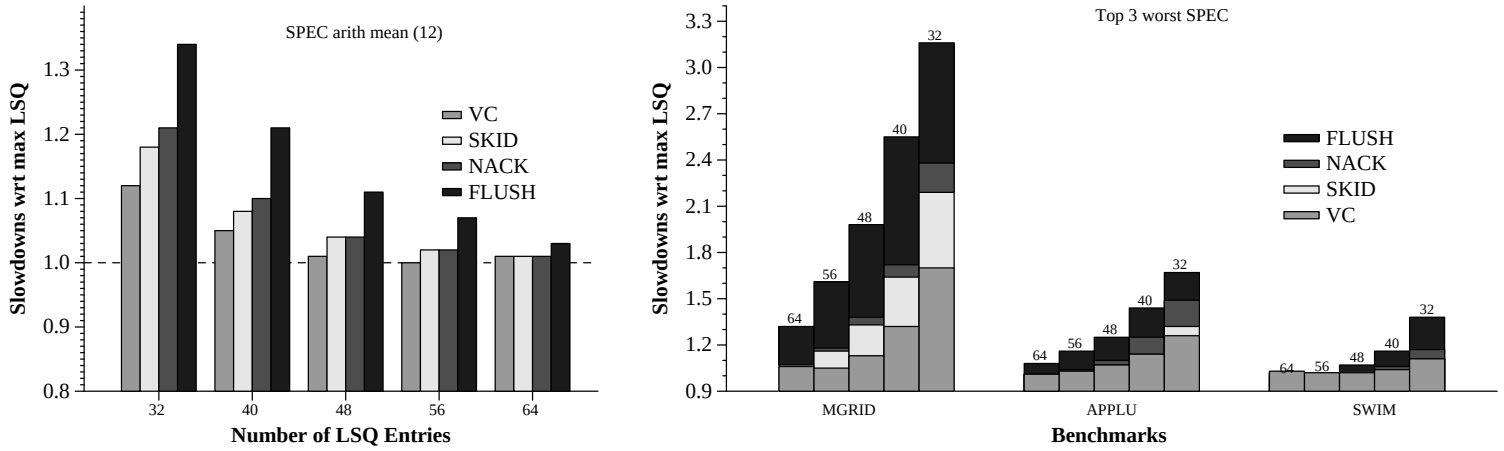



Figure 3.7: Left: Average LSQ Performance for the SPEC benchmark suite. Right: Three worst benchmarks.

```
realLow_1 = &realData_1[l_1];
imagLow_1 = &imagData_1[l_1];
realHi_1 = &realData_1[i_1];
imagHi_1 = &imagData_1[i_1];
 :
 :
realData_1[l_1] = *realHi_1 - tRealData_1;
imagData_1[l_1] = *imagHi_1 - tImagData_1;
realData_1[i_1] += tRealData_1;
imagData_1[i_1] += tImagData_1;
```

Figure 3.8: Code snippet from idct benchmark.

# Chapter 4

# LSQ Filtering Optimizations

A disadvantage with the LSQ implementations described so far (and many conventional LSQs) is that the detection of memory ordering violations and dependence handling requires frequent searches of considerable state. In a simple LSQ implementation, every in-flight memory instruction is stored in the LSQ. Thus, even with partitioning, as the number of instructions in-flight increases, so does the number of entries that must be searched in the LSQ to guarantee correct memory ordering. Both the access latency and the power requirements of LSQ searches scale roughly linearly with increases in the amount of state as the LSQ is typically implemented using a CAM structure [3].

The technique evaluated in this chapter to overcome these LSQ scalability limits is *approximate hardware hashing*. We implement low-overhead hash tables with Bloom filters [10], a structure in which a load or a store address is hashed to a single bit. If the bit is already set, there is a likely, but not a certain address match with another load or store. If the bit is unset there *cannot* be an address

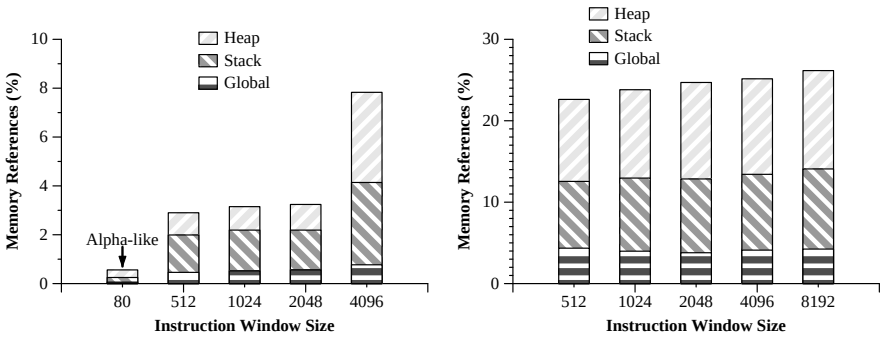
match with another load or store. We use Bloom filters to evaluate the following LSQ improvements:

- *Search filtering:* Each load and store indexes into a location in the Bloom filter (BF) upon execution. If the indexed bit is set in the BF, a possible match has occurred, and the LSQ must be searched. If the indexed bit is clear, the bit is then set in the BF, but the LSQ need not be searched. However, all memory operations must still be allocated in the LSQ.

- *Partitioned search filtering:* Multiple BFs each guard a different bank of a banked LSQ. When a load or store is executed, all the BFs are indexed in parallel. LSQ searches occur only in the banks where the indexed bit in the BF is set. This policy enables a banked CAM structure which reduces both the number of LSQ searches and the number of banks that must be searched.

- *Load state filtering:* A predictor examines each load upon execution and predicts if a store to the same address is likely to be encountered during the lifetime of the load. If so, the load is stored in the ULB-LSQ. If the prediction is otherwise, the load address is hashed in a load BF and is not kept in any LSQ. When stores execute, they check the load BF, and if a match occurs, a dependence violation may have occurred and the machine must perform recovery.

With these schemes, we show that the area required for LSQs can be reduced and, more importantly, that the power and latency for maintaining sequential memory semantics can be significantly reduced. This, in turn alleviates a significant scalability bottleneck to higher-performance architectures requiring large LSQs.




(a) LILP (b) HILP

Figure 4.1: Percentage of matching load instructions for the Alpha ISA

The rest of this chapter is organized as follows: Section 4.1 examines LSQ optimization opportunities by identifying the common case behavior. Section 4.2 describes and reports the performance of the search filtering techniques. Section 4.2.2 describes partitioned search filtering for TRIPS. Section 4.4, describes load state filtering. We conclude this chapter in Section 4.5.

## 4.1 LSQ Optimization Opportunities

Current-generation LSQs check all memory references for forwarding or ordering violations, since they are unable to diffentiate memory operations that are likely to require special handling from others that do not. Only a fraction of memory operations match others in the LSQs, however, so treating all memory operations as worst case is unnecessarily pessimistic.

To explore a range of in-flight instruction pressures and ISAs on the LSQs, we simulate three processor configurations. The first, called Perfect EDGE (PE), is intended to emulate an aggressive EDGE microarchitecture–assuming that other emerging bottlenecks are solved–to better stress the LSQs in our experiments. In

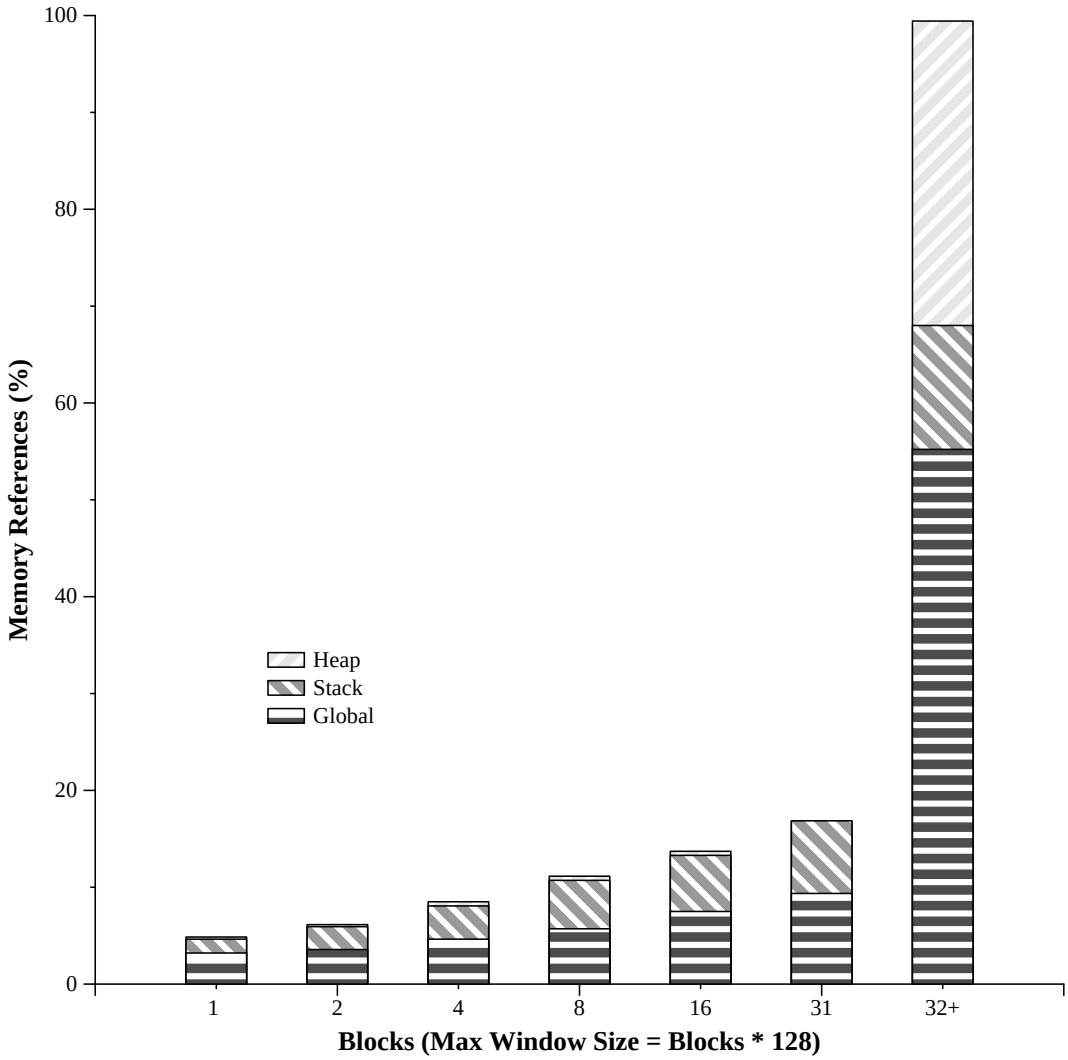



Figure 4.2: Percentage of matching load instructions for the EDGE ISA

particular, the Perfect EDGE configuration assumes perfect (oracle) load-store dependence prediction and branch prediction. The second, called Low ILP (LILP), is an Alpha 21264-like implementation with realistic microarchitecture assumptions to support window sizes larger than the Alpha 21264. The third, called High ILP (HILP) is the Alpha 21264 architecture simulated with perfect microarchitecture speculation (like PE).

The Alpha-21264 ISA results also follow similar trends and the results are presented in the Figures 4.1(a) and 4.1(b). For the LILP configuration, 512-instruction window sees 3% matching memory instructions. This rate remains essentially flat until the 4096-instruction window, at which point the matching instructions spike to nearly 8%. The HILP configuration, which has a much higher effective utilization of the issue window, has matching instructions exceeding 22% for a 512-entry window, which slowly grow to roughly 26% for an 8192-entry window.

The results of the PE experiments are presented in Figure 4.2. The results show that the matching rates are even lower for the EDGE ISA that the superscalar because of fewer fills and spills to the stack, which is in turn due to increased number of registers and the direct communication of operands within a block in the EDGE ISA.

Two results are notable from the data presented in this section. First, while the matching rates are close to two orders of magnitude greater than current architectures, three-quarters of the addresses in these enormous windows are *not* matching, indicating the potential for a four-fold reduction in the LSQ size. Second, the growth in matching instructions from 1K to 8K instruction windows is small, hinting that there may be room for further instruction window growth before the matching

| Benchmark | 8 Blocks | 16 Blocks | 32 Blocks |
|---|---|---|---|
| 168.wupwise | 9.71% | 12.63% | 14.23% |
| 171.swim | 2.85% | 3.41% | 3.45% |
| 173.applu | 2.24% | 3.26% | 4.48% |
| 175.vpr.place | 3.80% | 4.74% | 4.98% |
| 175.vpr.route | 0.20% | 0.22% | 0.76% |
| 179.art | 10.08% | 10.09% | 10.10% |
| 181.mcf | 0.36% | 0.40% | 0.48% |
| 183.equake | 2.12% | 3.44% | 3.65% |
| 197.parser | 0.63% | 0.81% | 1.10% |
| 256.bzip2 | 1.62% | 2.20% | 2.39% |
| 301.apsi | 7.32% | 8.99% | 10.23% |
| SPEC Average | 3.72% | 4.56% | 5.08% |
| a_a2time01 | 1.90% | 2.14% | 2.70% |
| a_aifftr01 | 40.49% | 41.24% | 42.09% |
| a_aifirf01 | 4.84% | 5.44% | 5.97% |
| a_aiifft01 | 43.42% | 44.23% | 45.15% |
| a_basefp01 | 3.35% | 3.89% | 4.20% |
| a_bitmnp01 | 19.37% | 19.40% | 37.16% |
| a_cacheb01 | 36.64% | 41.34% | 41.66% |
| a_canrdr01 | 6.45% | 6.60% | 6.99% |
| a_idctrn01 | 2.04% | 4.95% | 6.57% |
| a_iirflt01 | 18.14% | 19.49% | 20.67% |
| a_matrix01 | 18.68% | 20.79% | 22.98% |
| a_pntrch01 | 0.61% | 1.09% | 1.96% |
| a_puwmod01 | 8.50% | 8.62% | 8.93% |
| a_rspeed01 | 3.61% | 3.82% | 4.50% |
| a_tblook01 | 1.44% | 1.69% | 2.23% |
| a_ttsprk01 | 4.61% | 4.78% | 5.26% |
| c_cjpeg | 10.19% | 11.25% | 13.05% |
| c_djpeg | 11.07% | 11.18% | 11.81% |
| n_ospf | 2.14% | 2.16% | 2.23% |
| n_pktflow | 5.13% | 10.05% | 10.28% |
| n_routelookup | 2.57% | 2.71% | 2.77% |
| o_bezier01 | 1.34% | 1.36% | 1.40% |
| o_dither01 | 0.77% | 0.78% | 0.79% |
| o_rotate01 | 7.07% | 7.09% | 7.14% |
| o_text01 | 1.81% | 2.98% | 3.30% |
| t_autcor00 | 0.67% | 0.88% | 1.33% |
| t_conven00 | 27.65% | 39.68% | 42.73% |
| t_fbital00 | 0.05% | 0.06% | 0.10% |
| t_fft00 | 0.22% | 0.23% | 0.24% |
| t_viterb00 | 0.16% | 1.62% | 17.08% |
| EEMBC Average | 9.50% | 10.72% | 12.44% |

Table 4.1: Percentage of Loads communicating with In flight Stores with EDGE ISA, Perfect Speculation

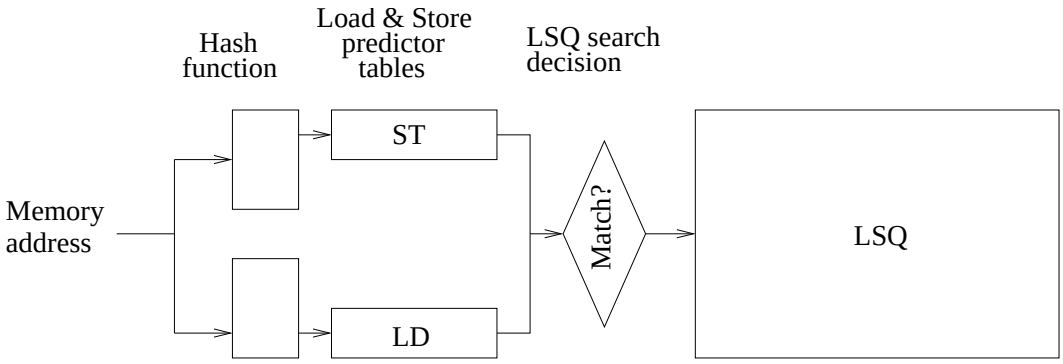



Figure 4.3: BFP Search Filtering: Only memory instructions predicted to match must search the LSQ; all others are filtered.

rate increases appreciably. Of course if the window is infinite, the matching rate will be close to 100%.

## 4.2 Search Filtering

This section describes techniques to avoid searches for memory instructions that do not match, then reduce the LSQ power consumption and latency for instructions that do match. The first technique uses a Bloom filter predictor (BFP) to eliminate unnecessary LSQ searches for operations that do not match other operations in the LSQ. We then apply BFPs to separate LSQ partitions, reducing the number of partitions that must be searched when the BFP predicts that an LSQ search is necessary. Finally, we discuss other applications of BFPs to partitioned primary memory systems.

### 4.2.1 BFP Design for Filtering LSQ Searches

The Bloom Filter Predictor (BFP) used for filtering LSQ searches maintains an approximate and heavily encoded record–as proposed by Bloom [10]–of the addresses of all in-flight memory instructions (Figure 4.3). Instead of storing complete addresses and employing associative searches like an LSQ, a BFP hashes each address to some location. In one possible implementation, each hash bucket is a single bit, which an memory instruction sets when it is loaded into the BFP and clears when it is removed. Every in-flight memory address that has been loaded into the LSQ is encoded into the BFP. If a new hashed address finds a zero, it means that the address matches no other instruction in the LSQ, so the LSQ does not need to be searched. The instruction sets the bit to 1 and writes it back. If a 1 is found by an address hashing into the BFP, it means either that the instruction matches another in the LSQ or a hash collision (a *false positive*) has occurred. In either case, the LSQ must be searched. The BFP is fast because it is simply a RAM array with a small amount of state for each hash bucket.

The BFP evaluated in this section uses two Bloom filters: one for load addresses and other for store addresses, each of which has its own hash function and $N$ locations. An issuing memory instruction computes its hash and then accesses the predictor of the opposite type (e.g. loads access the store table and vice versa). To detect multiprocessor read ordering violations, one can implement another Bloom filter with invalidation addresses is also checked by loads.

### Deallocating BFP Entries

A bit set by a particular instruction should be unset when the instruction retires, lest the BFP gradually fill up and become useless. But if multiple addresses collide, unsetting the bits when one of the instructions retires will lead to incorrect execution, since a subsequent instruction to the same address might avoid searching the LSQ even though a match was already in flight. There are several solutions to this problem.

**Counters:** One solution uses up/down counters in each hash location instead of single bits. The counters track the number of instructions hashing into a particular location. Upon instruction execution the counter at the indexed location is incremented by one and upon commit the counter is decremented by one. The counters can either be made sufficiently large so as not to overflow, or they can take some other corrective action using one of the techniques described below when they overflow. The use of counter based Bloom filters was previously proposed by Fan et al. [23].

**Flash clear:** An alternative approach to using up/down counters, is to clear all of the bits in the predictor when there are no memory instructions are in flight and hence it is safe to reset all the bits. This can be accomplished during branch mispredictions or when certain parts of bloom filter are allocated to groups of instructions. The flash clearing method has the advantage of requiring less area and complexity than the counters, but has the disadvantage of increasing the false positive rate. Encoding the instructions in each basic block into a Bloom filter and flash clearing when a basic block commits can improve the decrease the false positive rate.

### Hash Functions

To maximize the benefits of search filtering, the number of false positives must be minimized. The number of false positives depends on the quality of the hash function, the method used for unsetting the bits, and the size of the BFP tables. The BFP table must be sized larger than the number of in-flight memory operations, since the probability of a false positive is proportional to the fraction of set bits in the table.

There are two aspects that determine the efficacy of a hash function: (1) the delay through the hash function and (2) the probability of a collision in the hash table. Since the hash function is serialized with the BFP and then the LSQ search (if it is needed), we explored only two hash functions that were fast to compute, with zero or one level of logic, respectively. The first hash function, $H_0$, uses lower order bits of the address to index into the hash table, incurring zero delay for hash function computation. The second hash function, $H_1$, uses profiled heuristics to generate an index using the bits in the physical address that were most random on a per-benchmark basis. $H_1$ incurs a delay of one gate level of logic (a 2-input XOR gate). To determine $H_1$ for each benchmark, we populated a matrix by XORing each pair of bits of the address and adding the result to the appropriate position in the matrix. We then chose the bits that generated the most even number of zeros and ones, assuming that they were the most random.

### BFP Results

Table 4.2 presents a sensitivity analysis of the BFP false positives for a range of parameters, including varied predictor sizes ranging from one to four times the

| *Configuration* | | Alpha 21264 | | | LILP | | | HILP | | |
|---|---|---|---|---|---|---|---|---|---|---|
| *BFP Size* | | 32 | 64 | 128 | 128 | 256 | 512 | 128 | 256 | 512 |
| *Hash Type* | *Clearing Method* | | | | | | | | | |
| $H_0$ | Counter | 5.7 | 2.6 | 1.6 | 8.4 | 2.8 | 2.0 | 15.0 | 8.8 | 4.3 |
| $H_1$ | Counter | 3.9 | 2.3 | 1.4 | 5.7 | 3.3 | 2.0 | 10.1 | 6.1 | 4.2 |
| $H_0$ | Flash | 59.9 | 53.3 | 49.2 | 30.2 | 28.6 | 25.9 | n/a | | |
| $H_1$ | Flash | 54.0 | 49.7 | 42.2 | 27.2 | 23.9 | 20.4 | n/a | | |
| *Expected False Positives* | | 2.8 | 1.6 | 1.0 | 5.7 | 3.2 | 1.7 | 9.6 | 5.3 | 2.8 |

Table 4.2: Percentage of False Positives for Various ILP Configurations and BFP Sizes

size of each load and store queue, the two hash functions $H_0$ and $H_1$, flash and counter clearing, and the three microarchitectural configurations: the Alpha 21264, the HILP and the LILP configurations. The flash clearing results are not applicable to HILP because they rely on branch mispredictions, and HILP assumes a perfect predictor. As a lower bound, we include the expected number of false positives that would result, given the number of memory instructions in flight for each benchmark, assuming uniform hash functions[1]. The rate of false positives is averaged across the 19 benchmarks we used from the SPECCPU2000 suite. Results for the TRIPS configuration are presented in the subsequent section.

As expected, the table shows that the number of false positives decreases as the size of the BFP tables increase simply because of the reduced probability of conflicts. Flash clearing increases the number of false positives significantly over count clearing. However, the count clearing works quite effectively, especially using the $H_1$ hash function, showing less than a 2% false positive increase over the probabilistic lower bound. This result indicates that moderately sized BFPs are able to differentiate between the majority of matching addresses and those that have no match in flight. Furthermore, the lookup delay of all table sizes is less than one 8FO4 clock cycle at a 90nm technology. The power required to access the predictor tables is negligible compared to the associative lookup as the predictor tables are comparatively small, direct mapped RAM structures.

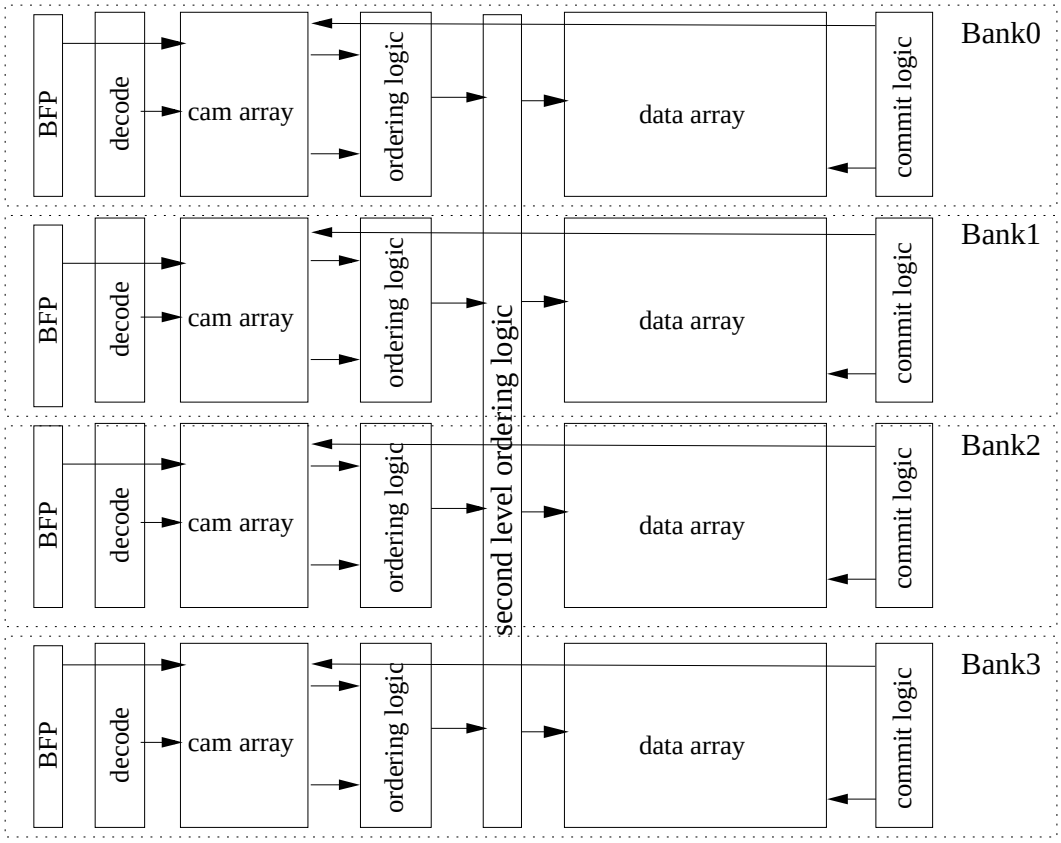
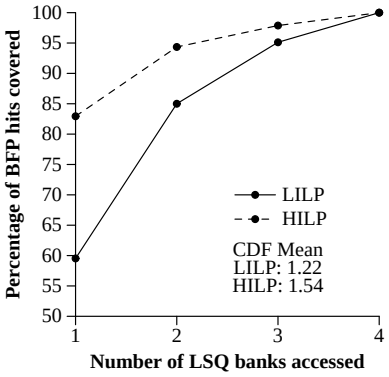



Figure 4.4: Partitioned Search Filtering: Each LSQ bank has a BFP associated with it (see far left)

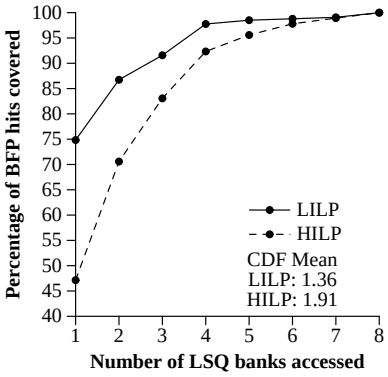
### 4.2.2 Partitioned BFP Search Filtering

The previous section described the use of BFPs to prevent most non-matching addresses from expensive LSQ searches. In this section, we describe a BFP organization that extends the prior scheme to reduce the cost of matching address searches appreciably. A distributed BFP (or DBFP), shown in Figure 4.4, is coupled with

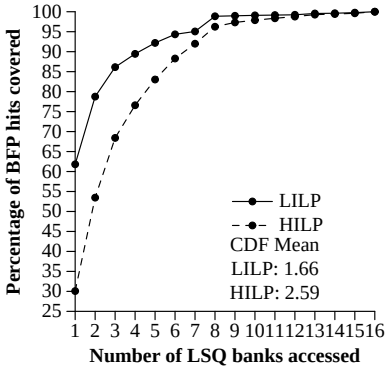
[1]Using probabilistic analysis the number of load (store) false positives assuming a uniform hash function can be estimated as:$\sum_i t_i \times (1 - (1 - \frac{1}{n})^i)$, where $t_i$ is the number of store (load) searches occurring when there are $i$ unique address in-flight loads(stores) and $n$ is the load (store) BFP size.




(a) 4 LSQ banks, 32 entries per LSQ bank, 128 entry BFP per LSQ bank, 64 entry BFP per bank




(b) 8 LSQ banks, 16 entries per bank




(c) 16 LSQ banks, 8 entries per LSQ bank, 32 entry BFP per bank

Figure 4.5: Partitioned State Filtering for Banked LSQs and BFPs

a physically partitioned but logically centralized (in terms of ordering logic) LSQ. One DBFP bank is coupled with each LSQ bank, and each DBFP bank contains only the hashed state of those memory operations in its LSQ bank. Depending on the implementation of the LSQs and the partitioning strategy, some extra logic may be required to achieve correct memory operation ordering across the partitions.

Memory instructions are stored in the LSQ just as in previous sections, but an operation is hashed into the BFP bank associated with the physical LSQ bank

into which it is entered instead of a larger centralized BFP as in the previous section. Before being hashed into the BFP bank, however, the address' hash is computed and used to lookup in all DBFP banks, which are accessed in parallel. Any bank that incurs a BFP "hit" (the counter is non-zero) indicates that its LSQ bank must be associatively searched. All banks finding address matches raise their match lines and the correct ordering of the operation is then computed by the ordering logic.

Depending on the LSQ implementation, the banking of the LSQ may have latency advantages over a more physically centralized structure. However, the power savings will be significant in a large-window machine if only a subset of the banks must be searched consistently. Figure 4.5 presents a cumulative distribution function of the number of banks that are searched on each BFP hit for both the HILP and LILP configurations, varying the number of LSQ banks from 4 to 16. The cumulative DBFP size was held at 512 entries for the different banking schemes. The results show that a DBFP can reduce the number of entries searched on a BFP hit appreciably; For the LILP configuration, 60% to 80% of the accesses result in the searching of only one bank. For the HILP configuration, 80% of the searches use four or fewer banks.

In high-ILP wider issue machines, as the number of simultaneously executing memory instructions increases, both the LSQs and the BFPs will need to be highly multiported. This section discusses organizations that still use a logically centralized, age-indexed LSQ, but exploits BFPs to facilitate a disambiguation hardware organization that matches the bandwidth of the primary memory system.

The port requirements on the BFPs can be trivially reduced by banking them, using part of the memory address as an index, to select one of the BFP banks, that will hold only memory instructions mapped to its bank. Banking the

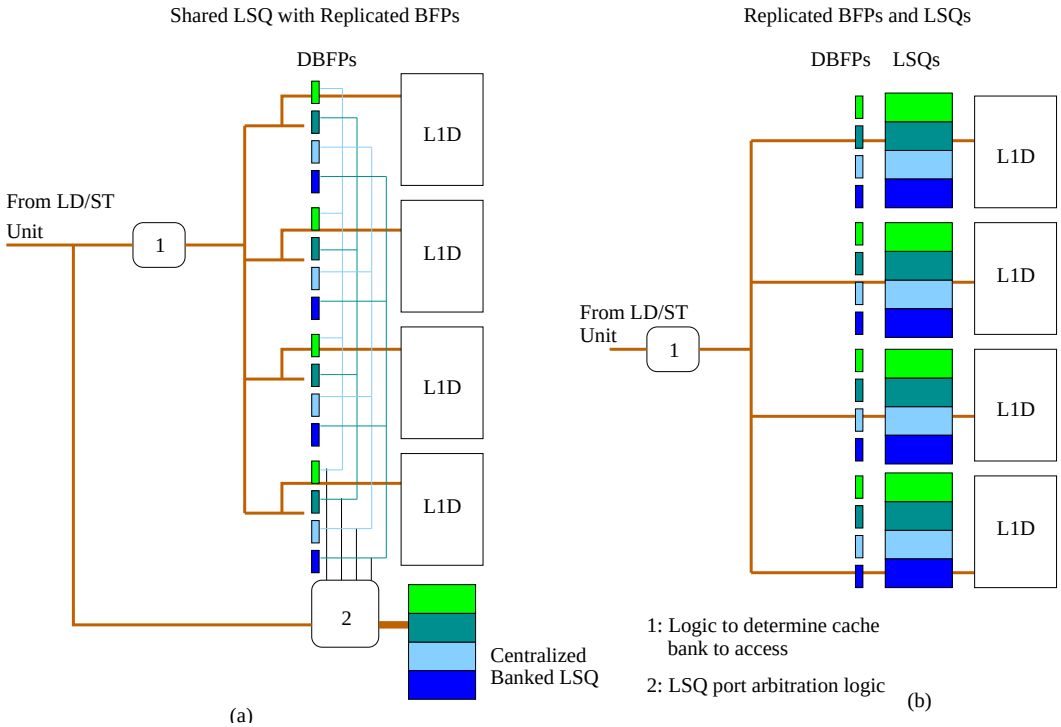



Figure 4.6: Replication of BFPs and LSQs to match L1D bandwidth

BFPs lends itself naturally to a partitioned primary memory system where the L1 data caches (L1D) are also address interleaved, as shown in Figure 4.6a. In this organization, a portion of the DBFP guarding each physical LSQ bank is associated with each L1D bank. Upon an access to the L1D cache, its DBFP banks generate a bitmask which indicates the LSQ banks that need to be searched. If the memory operation hits in none of the DBFP banks (the common case), then the LSQ search can be avoided.

Even with distributed replicated BFPs, each memory instruction must still be sent and allocated in the LSQ. As long as simultaneously executing memory instructions must are targeted into different banks of the LSQ, no contention occurs.

However, if the LSQ is to support parallel multi-banked accesses, extra circuitry must deal with buffering and collisions, increasing complexity. One simple solution to the problem is to replicate the banked LSQs as well. As with the DBFPs each replicated LSQ can be coupled with the statically address interleaved DL1 banks (Figure 4.6b), thus permitting all operations to complete locally at each partition. This scheme will also facilitate high bandwidth, low latency commit of stores to the L1D (assuming weak ordering is provided). Thus, replicated LSQs provide a complexity-effective solution but increase the area requirements significantly. In the next section we turn to schemes to reduce LSQ area, with the long-term goal being to reduce area sufficiently that completely replicated or distributed solutions become feasible.

## 4.3 TRIPS Bloom Filter Optimizations

Three mechanisms are necessary to achieve high power efficiency in large-window processor LSQs: address partitioning, late binding and associative search filtering. First, partitioning the LSQ by addresses naturally divides the number of entries arriving at each memory partition. Second, late binding reduces the number of entries in each partition by reducing occupancy. Finally, Bloom filtering reduces the number of memory instructions performing associative searches.

The Bloom filters for the TRIPS study use 8 32-bit registers, one for each in-flight block. The filters associated with each block are cleared when the block commits or is flushed – a form of flash clearing. As shown in Table 4.3, nearly 70-80% of the memory instructions (both loads and stores) can be prevented from performing associative searches in the TRIPS processor by using Bloom filtering.

| Benchmarks | *Average LSQ Activity Factor* | | | | | |
|---|---|---|---|---|---|---|
| | *VC* | | *SKID* | | *NACK* | |
| | 40 | 48 | 40 | 48 | 40 | 48 |
| SPEC | .21 | .21 | .27 | .30 | .30 | .31 |
| EEMBC | .26 | .27 | .38 | .39 | .38 | .39 |

Table 4.3: Fraction of loads performing associative searches

Using Bloom filters, however, incurs additional some additional power for reading and updating the filters for every memory instruction. Using the 130nm ASIC synthesis methodology described in the Alpha evaluation section, the capacitance of the 48-entry TRIPS LSQ was computed to be 322pF. The capacitance of the Bloom filter was 64pF. With the activity reduction of 80% the effective capacitance of the combination is 120pF which roughly is the capacitance of a 12-entry, 40-bit unfiltered CAM.

## 4.4 State Filtering

Increasing instruction window sizes lead to a corresponding increase in the number of in-flight memory instructions, making it progressively less power- and area-efficient to enforce sequential memory semantics. A future processor with an 8K instruction window would need to hold, on average, between two and three thousand in-flight memory operations. Any two in-flight memory instructions to the same address (matching instructions) must be buffered for detection of ordering violations and forwarding of store values. However, as previously shown in Figure 4.2, a small fraction of the addresses in flight are typically matching. An ideal LSQ organization would buffer only in-flight matching instructions, permitting reductions in the LSQ

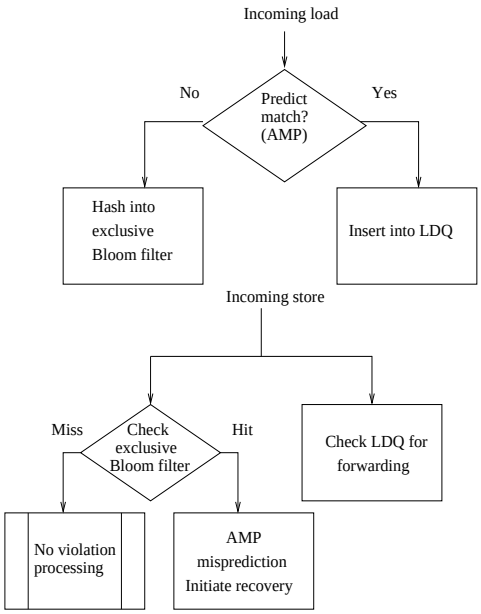



Figure 4.7: Load State Filtering

area, latency, and power. Buffering fewer operations in the LSQ facilitates efficient implementation of partitioned, address-indexed schemes that are a better match for future communication-dominated technologies.

In-flight store addresses and values must be buffered regardless of their interleaving with loads, since their values must only be written back to the memory system upon commit. The schemes presented in this section therefore attempt to reduce only the number of loads contained in the LSQs, by attempting to buffer only matching loads. .

Figure 4.7 shows one possible scheme to reduce the loads that must be saved in the LSQ. An *address match predictor* (AMP) predicts whether a load is likely to match a store. If a load is predicted *not* to match, it is hashed into a structure called

an *exclusive Bloom filter* (EBF), which contains an approximate hardware hash of all in-flight loads *not* contained in the LSQ. If the load is predicted to match, it is placed into the LSQ, which may be guarded by an *inclusive* Bloom filter for efficient accessing, as described in the previous section. Issued stores check the LSQ as usual, but they also search the EBF. A store hashing to a set bit in the EBF indicates either a false positive hit, or a possible memory ordering violation due to incorrect hashing into the EBF by the AMP. Since the two cannot be differentiated, the processor must take corrective action, possibly culminating in a pipeline flush.

Like the Bloom filter organization for search filtering, we take advantage of block-atomic ISA to improve state filtering. In our implementation, the EBF is partitioned by blocks to reduce the effect of false positives. A load predicted to match by the AMP is slotted in the EBF of the block the load belongs to. When a block commits all the entries in the EBF are cleared. When a store searches the EBF it searches the EBFs of blocks that are younger than the store or at least as old as the store. This organization can potentially reduce the false positives from aliasing from older blocks.

We use another optmization to avoid false positives from stores that are older than the loads but in the same block. We augument each entry in the EBF to indicate the age of the last load instruction that updated a particular EBF entry. When a store checks the EBF if the age of the store is greater than the load then no violation is flagged. As an additional storage optimization, instead of storing the full five bits for age, we store only approximate age information to save space. From our experiments most benchmarks 3-bit age tags perform as well as full 5-bit age tags. With 3 bits, a value of 000 indicates no load and 001 indicates a load between

0 and 7. Rest of the loads are grouped in fours and tagged with successively higher age-tags.

The AMP is a simple 1-bit table that is indexed by address. On a violation, the AMP table is trained with the address of the store (the address of the load is unavailable because it has been hashed). If the AMP uses the same hash function as EBF, then by training the AMP with the store address will ensure that any loads that are likely to cause false positives or violations will be directed to the LSQ the next time they arrive. If the AMP is never reset, over time, all the AMP bits will be 1, and the AMP will lose it's ability to filter loads. To avoid this, the AMP is periodically reset. The reset interval is programmable.

Tables 4.4, 4.5  4.6 summarize the results of state filtering. The tables show the slowdowns with respect to maximally sized LSQs for a 32 and 24 entry LSQs with state filtering optimization, for five hand optimized benchmarks, six SPEC INT benchmarks and eight SPEC FP benchmarks. For each of these benchmarks, the table also shows the performance of an undersized, unordered LSQ with NACK replay scheme to handle overflows. For the state filtering scheme, the loads were filtered using 64 bits EBF per block and an AMP of 64 bits per partition. For all benchmarks except vector add (vadd), 3 bits were used to represent the age tag.

The results indicate that for the hand coded benchmarks and for almost all of the SPEC benchmarks, a 32 entry LSQ with state filtering performs as well as a 48 entry LSQ with the NACK scheme. Furthermore, for the hand optimized and SPEC INT benchmarks a 24 entry LSQ performs as well as a maximally sized LSQ. These results can be attributed to lower LSQ pressure because of fewer in flight memory operations for SPEC INT benchmarks and the benefits of state filtering.

| Benchmark | LSQ NACK | | | LSQ | % lds | Replay | Flushes | | Age | Hash |
|---|---|---|---|---|---|---|---|---|---|---|
| | 48 | 40 | 32 | SF 24 | filtered | per ld | LSQ | EBF | bits | type |
| conv | 1 | 1.08 | 2.27 | 1 | 97% | 0 | 0 | 2 | 3 | xor |
| ct | 1 | 0.98 | 0.96 | 0.98 | 92% | 0.1 | 0 | 3 | 3 | high |
| genalg | 1 | 1 | 1 | 1 | 41% | 0.01 | 0 | 1 | 3 | xor |
| matrix | 1 | 1 | 1.03 | 1 | 79% | 0 | 0 | 0 | 3 | xor |
| vadd | 1.05 | 3.51 | 5.01 | 1.03 | 99% | 0.04 | 27 | 0 | 4 | low |
| Average | 1.01 | 1.514 | 2.054 | 1.002 | 82% | 0.03 | | | | |

Table 4.4: Performance of a 24 entry LSQ with state filtering optimization for hand optimized benchmarks. The slowdowns are with respect to a maximally sized LSQ. The results show that 24 entry LSQ with state filtering performs as well as a 48 entry LSQ with NACK scheme.

| Benchmark | LSQ | | %lds | Replay | Flushes per KI | |
|---|---|---|---|---|---|---|
| | NACK 32 | SF 24 | filtered | per ld | LSQ | EBF |
| vpr | 1 | 1.01 | 50% | 0 | 0 | 0.3 |
| gzip | 1.07 | 1.04 | 39% | 0.02 | 0.07 | 0.15 |
| bzip2 | 1 | 1 | 16% | 0 | 0 | 0.01 |
| crafty | 1 | 1.02 | 25% | 0 | 0.02 | 0.11 |
| vortex | 1 | 1 | 40% | 0 | 0.06 | 0.02 |
| perlbmk | 1 | 1 | 47% | 0 | 0 | 0.04 |
| Average | 1.01 | 1.01 | 36% | 0.00 | 0.03 | 0.10 |

Table 4.5: Performance of a 24 entry LSQ with state filtering optimization for SPEC INT benchmarks. The slowdowns are with respect to a maximally sized LSQ.The results show that 24 entry LSQ with state filtering performs as well as a 32 entry LSQ with NACK scheme.

| Benchmark | LSQ NACK | | LSQ | %lds | Replay | Flushes per KI | |
|---|---|---|---|---|---|---|---|
| | 48 | 32 | SF 32 | filtered | per ld | LSQ | EBF |
| apsi | 1.03 | 1.29 | 1.37 | 4% | 0.24 | 0.17 | 0.08 |
| mesa | 1.01 | 1.02 | 1.07 | 36% | 0.01 | 0.03 | 0.21 |
| swim | 0.99 | 1.84 | 1 | 99% | 0 | 0 | 0 |
| applu | 1 | 1.07 | 1.03 | 32% | 0.05 | 0 | 0.15 |
| mgrid | 2.75 | 3.25 | 1.04 | 79% | 0.01 | 0.01 | 0.04 |
| equake | 1.05 | 1.23 | 1.05 | 39% | 0.04 | 0.06 | 0.05 |
| wupwise | 1 | 1 | 1 | 34% | 0 | 0 | 0.01 |
| sixtrack | 1 | 1.01 | 1.05 | 44% | 0.01 | 0 | 0.14 |
| Average | 1.23 | 1.46 | 1.08 | 46% | 0.04 | 0.03 | 0.08 |

Table 4.6: Performance of a 32 entry LSQ with state filtering optimization for SPEC FP benchmarks. The slowdowns are with respect to a maximally sized LSQ. The results show that 24 entry LSQ with state filtering performs better than a 48 entry LSQ with NACK scheme.

For the remaining benchmarks, better hash functions can increase the benefits of filtering.

A key constraint in state filtering is to ensure that the state added to implement filtering (EBF and AMP) is not larger than the state saved in the LSQ. The state savings EBF and AMP for 32-entry LSQ with 5-bits of age is roughly 160 bytes acorss four partitions. The state savings for a 24-entry LSQ with 3-bits for the age tag is roughly 1.2KB across four LSQ partitions. While the program characteristics indicate that only few loads or stores have dependences, the hardware mechanisms are severely stunted without quality hash functions. Either better hash functions or compiler support will be needed to reduce the size of the LSQ further.

## 4.5 Summary

In this chapter, we proposed a range of schemes that use approximate hardware hashing with Bloom filters to improve LSQ scalability. These schemes fall into two broad categories (see Figure 4.8): *search filtering*, reducing the number of expensive associative LSQ searches, and *state filtering*, in which some memory instructions are allocated into the LSQs and others are encoded in the Bloom filters.

The search filtering results show that by placing a 4-KB Bloom filter in front of an age-indexed, centralized queue, 73% of all memory references can be prevented from searching the LSQ, including the 95% of all references that do not actually have a match in the LSQ. By banking the age-indexed structure and shielding each bank with its own Bloom filter, a small subset of banks are searched on each memory access; for a 1024-entry LSQ, only 12 entries needed to be searched on average. We also proposed placing Bloom filters near partitioned cache banks, preventing a slow,

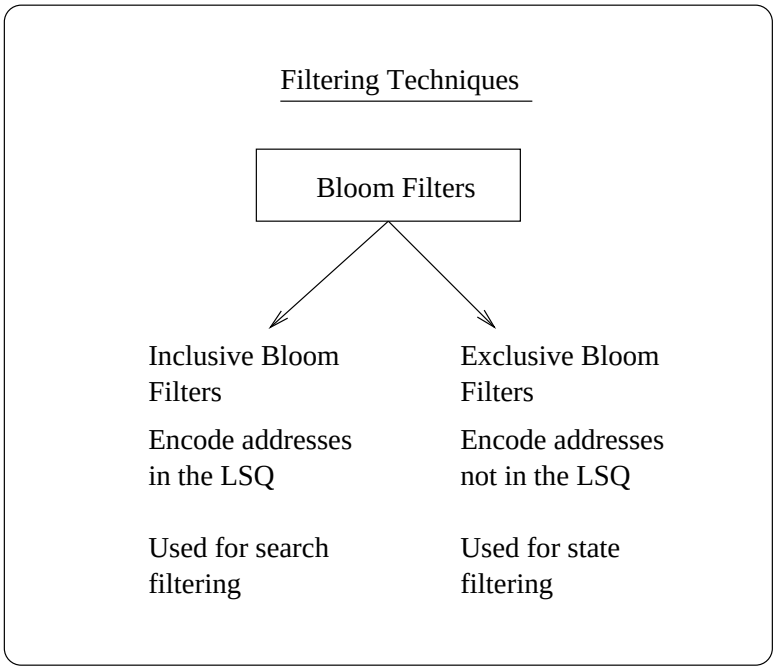



Figure 4.8: Summary of filtering techniques

centralized LSQ lookup in the common case of no conflict.

For state filtering, we coupled an address match predictor with a Bloom filter to place only predicted dependent operations into the LSQs, encoding everything else in a Bloom filter and intiating recovery when a memory operation finds its hashed bit set in the Bloom filter. With this scheme the number of entries in the LSQ can be reduced further but our results indicate that the LSQ area savings are offset by the increase in predictor area. With better hash functions state filtering can perhaps be made more profitable.

# Chapter 5

# Related Work

In this chapter, we will describe the evolution of LSQs and then qualitatively and quantitatively differentiate the optimizations described in this dissertation with recently proposed LSQ optimization. We conclude with a review of related work on other aspects of the primary memory system.

## 5.1 Historical Background on Memory Disambiguation

Initially, simple sequential machines executed one instruction at a time and did not require hardware for enforcing the correct ordering of loads and stores. With the advent of speculative, out-of-order issue architectures, the buffering and ordering of in-flight memory operations became necessary and commonplace. However, the functions embodied in modern LSQ structures are the result of a series of innovations much older as described in this chapter.

**Store Buffers:** In early processors without caches, stores were long-latency operations. Store buffers were implemented to enable the overlap of computation

with the completion of the stores. Early examples were the stunt box in the CDC 6600 [75] and the store data buffers in the IBM 360/91 [11]. More modern architectures separated the functionality of the store buffers into pre-completion and post-commit buffers. The pre-completion buffers, now commonly called store queues, hold speculatively issued stores that have not yet committed. Post-commit buffers are a memory system optimization that increases write bandwidth through write aggregation. Both types of buffers, however, must ensure that *store forwarding* occurs; when later load to the matching address are issued, they receive the value of the store and not a stale value from the memory system. Both types of store buffers must also ensure that two stores to matching addresses are written to memory in program order.

**Load Buffers:** Load buffers were initially proposed to temporarily hold loads while older stores were completing, enabling later non-memory operations to proceed [60]. Later, more aggressive out-of-order processors–such as IBM's Power4 [74] and Alpha 21264 [19]–permitted loads to access the data cache speculatively, even with older stores waiting to issue. The load queues then became a structure used for detecting *dependence violations*, and would initiate a pipeline flush if one of the older stores turned out to *match* (have the same address as) the speculative load. Processors such as the Alpha 21264 and Power4 also used the load queue to enforce the memory consistency model, preventing two matching loads from issuing out of order in case a remote store was issued between them.

As window sizes increased, the probability that matching memory operations would be in-flight increased, as did the chance that they would issue in the incorrect order, resulting in frequent pipeline flushes. Memory dependence predictors were

developed to address this problem, allowing loads that were unlikely to match older stores to issue speculatively, but deferring loads that had often matched in-flight stores in the past.

Dependence predictors have been an active area of research since the late part of the last decade. Early work on dependence predictors by Moshovos and Sohi, and Henson et al. identified the potential of memory speculation for out-of-order processors and proposed solutions that worked well for small window processors. Moshovos and Sohi proposed a predictor that identified recurring RAW memory violations using a couple of a CAM tables one each for the Store PC and the Load PC [55]. Hesson et al. proposed a scheme where loads that caused violation were marked using reserved bits in the instruction encoding and stored the store barrier cache. When any marked load was inflight the memory execution was completely serialized in their design irrespective of matching addresses. [37].

Two papers, one in 1998 and other in 1999, have had great influence on dependence prediction research. Chrysos and Emer described the store sets predictor which identified sets of matching loads and stores and made dependent loads wait on particular dependent stores [18]. Yoaz et al. used a much simpler but very effective predictor based on distance between dependent loads and stores to improve performance [84]. Several researchers have built upon both of the designs. Notably, Sha, Martin and Roth enhanced the store sets predictor with path based information and proposed training on both violations and forwardings [68]. Similarly, Subramanium and Loh extended the distance predictor with partial tags and confidence estimates to improve its accuracy even further [73].

The dependence predictor used for state filtering is a very coarse filtering

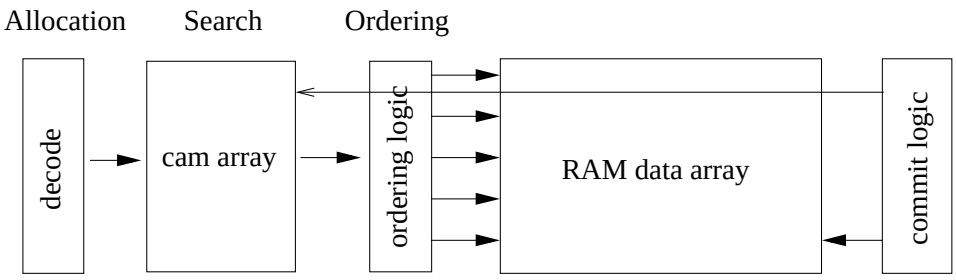



Figure 5.1: A simplified LSQ datapath

dependence predictor that makes history based predictions on the granularity of blocks. This is the first block based dependence predictor. We envision this predictor being used as a pre-filter to filter out references to the concurrent work on counting dependence predictors.

## 5.2 Age-Indexed LSQs

The majority of LSQ designs so far have been *age indexed*, in which memory operations are *physically ordered* by age i.e. memory instructions are slotted into a specific row in the CAM and the RAM based on the LSIDs. The age indexed queues are managed as circular buffers and use fully associative search on the entire queue contents to enforce dependences. Although fully associative structures are expensive in terms of latency and power, age indexing permits simpler circuitry for allocating entries, determining conflicts, committing stores in program order, and quick partial flushes triggered by mis-speculations because the instructions are physically ordered by age.

Dynamically scheduled processors, such as those described by Intel [17],

IBM [24], AMD [42] and Sun [59], use age-indexed LSQs. An LSQ slot is reserved for each memory instruction at decode time, which it fills upon issue. To reduce the occurrence of pipeline stalls due to full LSQs, the queue sizes are designed to hold a significant fraction of all in-flight instructions (two-thirds to four-fifths). For example, to support the 80-entry re-order buffer in the Alpha 21264, the load and store buffers can hold 32 entries each. Similarly, on the Intel Pentium 4, the maximum number of in-flight instructions is 128, the load buffer size is 48, and the store buffer size is 32.

Additional information of specific aspects of the age-indexed LSQs are described the following patents: Justification and design of an unified LSQ over separate LSQ are presented in US patent numbers 5832297 from Sun, Store-to-load forwarding is described in detail by several designers from IBM and Intel in patent numbers 6141747, 6021485, 5931957 and 6301654.

Ponomarev et al. [61] proposed an age-indexed but segmented LSQ, in which a fully associative LSQ is broken into banks through which requests are pipelined, accessing one bank per cycle. This strategy ultimately saves little power or latency, since all entries must be searched in the common case of no match, and an operation must wait until a number of cycles equal to the number of banks has elapsed to determine that there were no conflicts. This scheme lends itself to efficient pipelining of LSQ searches for faster clock rates, but not necessarily higher performance.

## 5.3 Address-indexed LSQs

To reduce the state that must be searched for matches, partitioning of LSQs is desirable. Address-indexed LSQs logically break the centralized, fully associative LSQ

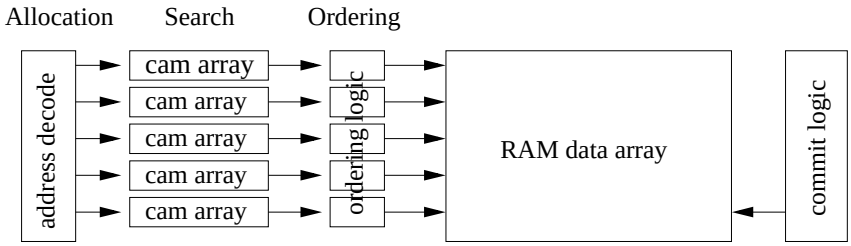



Figure 5.2: Address-indexed LSQ datapath

into a set-associative structure. As shown in Figure 5.2, a portion of an memory instruction's address chooses the LSQ set, and then only the entries in that set are searched for a match. While this does reduce the number of entries that are searched, address-indexed organizations suffer from two major drawbacks. First, address-partitioned LSQs can have frequent overflows (since set conflicts are possible), resulting in more flushes than an age-indexed LSQ. Second, ordering and partial flushing become more difficult in an address-indexed LSQ because instructions to be flushed may exist in different sets. For the same reason, in-order commit of stores to memory is also more expensive.

To mitigate the conflict problem, sets can be made larger, in which case the latency, power, and partitioning advantages diminish. Alternatively, the sets can be made more numerous, in which case the LSQ may requiring more area than a pure centralized design and the average utilization of the entries will be low. Thus, even though address-indexed LSQs can support sets residing in separate banks with localized ordering logic (enabling de-centralized LSQs), they incur both performance and complexity penalties.

A number of proposed or implemented designs have used address-indexed

LSQs to facilitate partitioning. The Itanium-1 microarchitecture uses a violation detection table called an ALAT [41], which is a 32-entry, 2-way set associative structure. Because conflicts can overflow a set in an address-partitioned LSQ, more entries can reduce the probability of conflicts; the ALAT can hold 32 entries, even though a maximum of 20 memory instructions can be in-flight. The Itanium-2 microarchitecture [53] implements a 32 entry fully associative structure reducing the probability of conflicts even more.

Both the IA-64 [29] compiler and the Memory Conflict Buffer paper [26] emphasize static disambiguation analysis to store only instructions whose addresses either have a true dependence or cannot be statically disambiguated, thus reducing the size of the hardware ALAT or MCB structures. An IA-64 study on dependence analysis [83], however, concedes that relying completely on static analysis is ineffective for programs that cannot tolerate the compile-time analysis cost (e.g. JITs) or non-native binaries for which source access is not available, and that static analysis is much less effective for many pointer-intensive codes. Static analysis can play a role but cannot address LSQ scaling issues comprehensively.

Finally, the MultiScalar processor proposed an address-indexed disambiguation table called the Address Resolution Buffer (ARB) [25]. When an ARB entry overflows (an ARB set has too many memory addresses), the MultiScalar stages are squashed and the processor rolls back. The MultiScalar compiler writers focused strongly on minimizing the probability of conflict in the ARB, trying to reduce the number of subsequent squashes.

## 5.4 Recently Proposed LSQ Organizations

An LSQ fulfills three necessary functions: (1) it forwards store values to later loads when their addresses match, (2) it detects load misspeculations, if a load issued with older unresolved stores in flight that were later found to match the load's address, and (3) it acts as a store buffer that is used to commit stores to memory upon retirement.

Recent work (see Figure 5.3) has proposed separating some or all of the three LSQ functions into separate structures, to accelerate some of the individual structures. This approach results in increased complexity, and occasional area increases due to redundant information held in multiple structures, but the potential power savings, and in some cases, the access latency improvements, are considerable. In this paper, we propose a LSQ organization that provides all the benefits of functional decomposition without the increased complexity of the other mechanisms.

**Functionally Decomposed LSQs:** The LSQ typically supports 1) forwarding from stores to later loads 2) commitment of speculative stores to memory and 3) detection of load misspeculations. Recent work has proposed breaking all three LSQ functions into separate structures, to accelerate some of the individual structures. This approach results in increased complexity, and occasional area increases due to redundant information held in multiple structures, but the potential power savings are considerable. Both Baugh and Zilles [8] and Roth [64] take this approach. For the speculative forwarding, both papers propose a small, fast, centralized, unordered, fully associative forwarding buffer, and both papers propose a non-associative FIFO for buffering and committing stores. The papers differ on the misspeculation detection function: Baugh and Zilles use a centralized, addressed-

indexed structure, whereas Roth uses enhancements of a load re-execution proposed by Cain and Lipasti [14]. Cain and Lipasti's paper proposed eliminating the misspeculation detection structure (i.e. the load queue) by having loads save their value in the reorder buffer and compare the values at commit time to detect misspeculations.

Torres et al. [77] propose a distributed, unordered, address-interleaved store-load forwarding buffer but a centralized, age-ordered store queue for speculation checking and commit. While the distributed store forwarding buffers increase forwarding bandwidth, the centralization of the other two structures forces all loads and stores to be routed to a central place for verification and commit, hampering scalability.

Stone et. al. [72] suggest using a set associative cache for forwarding, a non associative FIFO for commit and an address-indexed timestamp table for checking speculation. The timestamp table never produces false negatives but may produce false positives due to address aliases and conservative handling of partial flushes. As with most address-indexed LSQs, such designs have to be oversized for good performance. For instance, the authors use a 8K entry, 8bit wide timestamp table and a 80bit wide, 512 set, 2way forwarding cache even though there can be only 1K instructions in-flight at any time.

Sha et al. [67] extend Roth's scheme by using a modified dependence predictor to match loads with the precise store buffer slots from which they are likely to receive forwarded data. This solution completely eliminates the associative store forwarding buffer but instead requires large multi-ported dependence/delay predictors (approximately 16KB combined), thus effectively improving power at the expense

of area. These functionally decomposed schemes save power and in some cases energy, but still rely on centralized structures that will be challenging to partition effectively.

**Hierarchical LSQs:** Other work has looked at reducing LSQ delay and power by building hierarchy into the memory disambiguation hardware. While effective, these techniques add complexity and do not directly lend themselves to facilitating distributed memory disambiguation.

Akkary et al. [4] propose two-level store buffers which are both centralized, fully associative and age ordered. Stores are first entered in the L1 store buffer, and when it overflows they are moved to the L2 store buffer. Both buffers support forwarding and speculation checking but stores commit from the second level buffer. This scheme reduces power, but still requires a worst-case sized L2 and uses area-inefficient CAMs.

Gandhi et al. [27] propose an area-efficient disambiguation hierarchy by dividing operations into two categories on a long-latency L2 cache miss. Stores dependent on the L2 miss bypass the store queue and are allowed to speculatively update the L1-D cache, whereas independent memory operands go to a non-associative FIFO. The dependent cache updates are discarded when the memory operations dependent on the L2 cache miss are re-executed. It is unclear that this technique will be effective for distributed, high-ILP cores that attempt to tolerate many L2 misses concurrently.

Finally, Cristal et al. [21] are investigating a complete set of mechanisms for kilo-instruction processors, an interesting approach that will likely exploit high ILP and tolerate long L2 memory latencies. They advocate a combination of hierarchical

techniques to make their centralized LSQ power and area scalable to future large windows.

**Distributed age-indexed LSQs:** Research proposals for clustered architectures [85, 7] employ multiple partitions of an age-indexed LSQ, but instead of reserving a slot in each of the LSQ partitions, they use memory bank predictors [9] to predict a target bank and reserve a slot there. If the bank prediction is low-confidence, slots are reserved in all banks. While this approach is better than conservatively reserving a slot in each partition, it still wastes space because of conservative dispatch allocation.

**Miscellaneous:** Jaleel et al. point out that blindly scaling the larger window LSQs can be detrimental to performance due to the increase in the number of replay traps [44]. In their study on a scaled Alpha-21264 core, such traps can occur when load instructions violate the consistency model, when load needs to partially obtain the data from the LSQ and the cache, when a load miss cannot be serviced because of structural hazards and when a load instruction executes prematurely. TRIPS avoids these traps and does not suffer from the performance losses described in [44]. In particular, TRIPS avoids load-load traps with weak memory ordering, wrong-size traps by supporting partial forwarding in LSQ, and load-miss traps by using larger MSHRs. Like the Alpha, TRIPS also uses a dependence predictor to reduce the number of load-store replay traps which occur when a load instruction is executed prematurely.

Garg, Rashid and Huang propose another mechanism for eliminating associative LQ/SQs [28]. In the first phase of two-phase processing, loads and stores speculatively obtain their value from a L0 cache. In the second phase, the memory

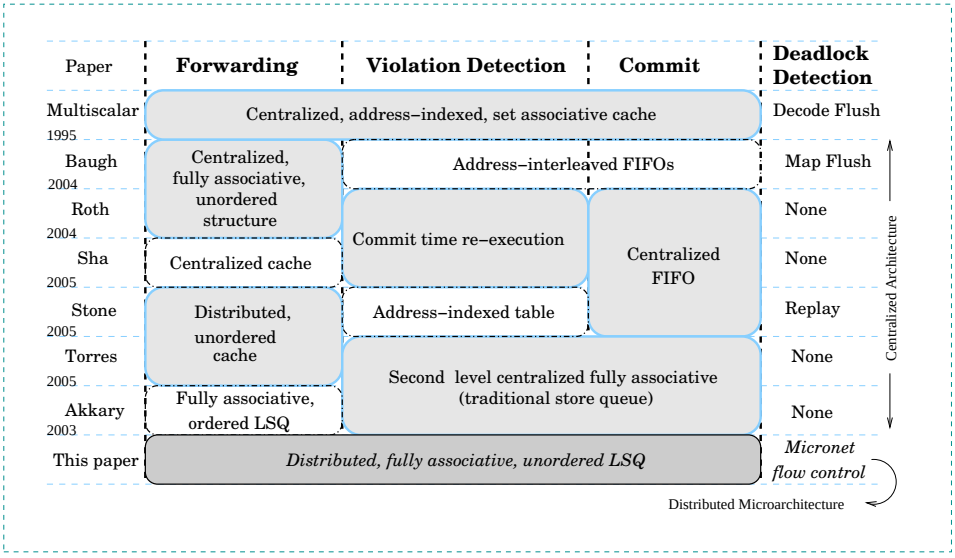



Figure 5.3: LSQ Related Work.

instructions are re-executed in program order, without any speculation, and access the regular L1 cache. Any difference in the load values between the two phases results in corrective action. This mechanism, while eliminating the CAM, requires a 16KB L0 cache and an age-ordered queue for holding the values read during the first phase.

Table 5.4 summarizes the ROB size, the area of the LSQ before and after the optimizations, the size of the supporting structures required by several of the recently proposed optimizations (but which may be already be present in the design for performance reasons), and finally the ratio of total area required for memory disambiguation, before and after the optimizations. We computed the area of the memory structures, in bytes of storage, assuming the CAM cell area to be three times larger than the RAM cell. We also assumed 40-bit addresses and 64-bit data,

| Scheme | ROB Size | Unoptimized | | | | Optimized | | | | | | | | | Unoptimized :optimized Ratio |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Depth | | Storage (KB) | | Depth | | CAM Width | | RAM Width | | Storage (KB) | | Supporting Structures | |
| | | LQ | SQ | LQ | SQ | LQ | SQ | LQ | SQ | LQ | SQ | LQ | SQ | (KB) | |
| SVW - NLQ | 512 | 128 | 64 | 1.00 | 1 | 128 | 64 | 0 | 12 | 92 | 64 | 1.44 | 0.78 | 1 | 1.61 |
| SQIP | 512 | 128 | 64 | 1.00 | 1 | 128 | 64 | 0 | 0 | 92 | 64 | 1.44 | 0.5 | 23.5 | 12.72 |
| Garg et al. | 512 | 64 | 48 | 0.50 | 0.75 | 64 | 48 | 0 | 0 | 92 | 0 | 0.72 | 0 | 16 | 13.38 |
| NoSQ | 128 | 40 | 24 | 0.31 | 0.375 | 40 | 0 | 0 | 0 | 92 | 0 | 0.45 | 0 | 11 | 16.65 |
| FnF | 512 | 128 | 64 | 1.00 | 1 | 128 | 0 | 0 | 0 | 92 | 0 | 1.44 | 0 | 23.75 | 12.59 |
| Stone et al. | 1024 | N/A | | | | N/A | | | | | | | | 18 | N/A |
| LateBinding-Alpha | 80 | 32 | 32 | 0.25 | 0.5 | 32 | | 12 | | 92 | | 0.5 | | 0.0625 | 0.75 |
| LateBinding-TRIPS | 1024 | 1024 | | 16 | | 192 | | 12 | | 92 | | 3 | | 0.25 | 0.20 |

Table 5.1: Area for LSQ and supporting structures for recent related work

and that all the unoptimized designs had 12-bit partial addresses in CAMs and rest of the address bits in the RAMs. The depth of the queues, however, is different for each of these structures. The table shows that the proposed schemes add area overhead between factors of 1.5 to 16.5. When discounting the dependence predictor, the area overheads are between factors of 1.5 and 13.

In contrast to all of the above schemes, the design proposed in this paper uses late binding to reduce the area and latency without any additional state outside of the LSQ. Dynamic power reduction, however, requires additional state in the form of address-based Bloom filters described in Chapter 4. These structures take up only few hundreds of bytes and can even be further reduced by optimizations suggested by Castro et al. [15].

Other researchers have also applied issue-time binding, explicitly or implicitly, to improve the efficiency of microarchitectural structures. Monreal et al. use late allocation to reduce the number of physical registers [54]. The effectiveness of some LSQ optimizations like address-based bloom filters or the small associative forwarding buffers [8, 64, 77] can also be explained in part by late allocation.

# Chapter 6

# Conclusions

## 6.1 Contributions and Impact

### 6.1.1 Memory Disambiguation Research

For more than a decade now, computer architects have proposed designs that attempt to improve performance by dynamically extracting parallelism from a large pool of in-flight instructions. The Achilles heel for most of these proposals has been the design of the memory disambiguation hardware, also known as a load-store-queue (LSQ), which ensure that sequential memory semantics are satisfied. Conventional approaches to LSQ designs do not scale with increasing number of in-flight instructions because they use associative memories, which are power-hungry, large and slow. This dissertation described new LSQ organization techniques that will scale to large instruction windows by minimizing use of associative memories. We have published two papers on this topic: the first paper described the problems with scaling LSQs and proposed optimizations for scaling centralized LSQs, and was

presented at the 36th International Symposium of Microarchitecture in 2003 (MICRO 2003). The second paper is on design and scalability of distributed LSQs was presented at the 40th International Symposium on Computer Architecture (ISCA 2007).

**Centralized LSQs:** Our MICRO 2003 paper was one of the first academic papers to articulate the difficulties associated with scaling LSQs. In addition to framing the problem, the paper made three key contributions:

1. I observed that even when a large number of memory instructions were simultaneously in-flight, only a small fraction of these memory instructions were to the same address (matching addresses). However, LSQ designs treated all memory operations equally and hence were under-optimized for the common case, which led to power and area problems.

2. To mitigate this power problem, I proposed using simple address-based Bloom filters. Bloom filters replace the power-hungry, fully associative search with a power-efficient hash table lookup in the common case; associative search occurs only when a matching instruction is likely to be found in the LSQ. The Bloom filter is constructed by hashing the data address of every executing load and store into a single bit. Then, when a load or store executes, it checks the Bloom filter by hashing its own data address. If the bit at the hashed location is not set, then there cannot any other instruction to the same data address in the LSQ and thereby making the associative search unnecessary. As we predicted in our paper, Bloom filters have enjoyed wide applicability with architects in recent days. They have been applied to several structures outside of LSQ with very high impact.

3. To mitigate the area problem, we proposed a technique in which the LSQ is sized to hold only instructions that are likely to have matching addresses, and the instructions that are unlikely to match are summarized in a Bloom filter. If an instruction is incorrectly slotted into the Bloom filter, then a pipeline flush is initiated. Although the initial results were affected by the low accuracy of dependence predictors and false positives in the Bloom Filter other researchers have built upon these techniques/observations and showed significant LSQ area savings but added numerous dependence predictors which diminished the benefits of LSQ area reduction. In follow on work I invented new dependence predictors that are very area efficient and improve this accuracy significantly and thus make state filtering profitable.

**Distributed LSQs:** As on-chip wire delays increase, and high-performance processors necessarily become more partitioned, centralized structures like the LSQ can severely limit scalability. In work published in ISCA 2007, I proposed schemes to partition and distribute the LSQs. The two key requirements of efficient, partitionable LSQs are 1)they should support late binding and 2) they should support low-overhead mechanisms for dealing with LSQ overflows.

1. **Late binding:** Conventional LSQs reserve a slot in the LSQ as soon as a memory instruction is decoded. Employing the same approach for distributed LSQs will result in a slot being wastefully reserved in all the partitions even though the instruction will go only to one partition. One solution to this problem is to delay allocation until the partition is known. However, delaying allocation, i.e., late binding requires new LSQ organizations and algorithm that have traditionally been considered difficult to implement. I noted that the

implementation is complex only if all execution possibilities are treated equally. Implementation can thus be simplified by optimizing only the common case since 99% of accesses consist of a load forwarding from one or fewer stores. Our measurements showed no performance degradation from increasing the delay of the uncommon case. An added benefit of late binding is the reduction in the size of the LSQ. Late allocation reduces the occupancy of the LSQ, resulting in a smaller LSQ.

2. **Overflow handling:** In distributed LSQs, it is wasteful to maximally size each of the partitions because only a fraction of the memory references will access a given partition. If the partitions are not maximally sized, however, we need schemes to deal with situations when a memory instruction arrives at a full LSQ. Simply stalling could lead to deadlocks because the LSQ is allocated out-of-order; simply flushing the pipeline can lead to severe performance losses. To solve this problem, we proposed three low-overhead mechanisms for handling overflows. The first method buffers the overflowed instructions in a separate buffer near the LSQs. The second NACKs the overflowing instruction and sends it back to the issue window from which it is re-tried later. The third scheme, the most novel of the three, uses the buffers in the interconnection network between the execution units and memory to buffer instructions, and it uses virtual channels to avoid deadlocks. The network-buffering scheme is the most robust of all the schemes for large instruction windows.

For nearly a decade now, the lack of robust, low-overhead overflow schemes has been a severe impediment to distributing the LSQ and hence to completely distributed microarchitectures; I believe that the techniques presented in this this

dissertation provide a solution to this problem and will likely scale to very large window processors.

### 6.1.2 TRIPS System Design Experience

For the TRIPS prototype chip, I was the lead designer of the primary memory system and I was involved in the design and verification of the SDRAM controller. The processor core is built from five tiles and the primary memory system was composed of tiles called Data tiles or DTs. As a tile owner, I was responsible for the initial specification, RTL entry for large portions, timing closure and floorplanning. Robert McDonald, the chief engineer on the project, was involved in all aspects of the design, wrote some RTL and was the verification lead on the primary system. The DT is considered one of the most complex of all tiles because of the number of components and interaction between the components. The SDRAM controller was considered the mostly likely to break because of IBMs documentation of the component was sketchy. Disciplined design and sophisticated verification has ensured that both these components are bug free.

The TRIPS primary memory system posed a significant challenge because it had to match the bandwidth of the execution core, and it had to support high levels of memory parallelism and re-ordering (256 in-flight memory instructions). Growing on-chip wire delays and the scale of this problem necessitated a fully partitioned memory system. To match the bandwidth requirements, the primary memory system is partitioned into four independently accessible partitions, the DTs, which are interleaved based on the addresses of the memory instructions. To support high memory parallelism, the DT uses memory-side dependence predictors, deep LSQs,

and an aggressive miss-handling unit capable of supporting up to 16 outstanding load misses per DT (64 per core). These capacities are much higher than the state-of-the-art in high performance processors To the best of my knowledge, the TRIPS processor is the first processor to feature a fully partitioned memory system that is capable of maintaining sequential memory semantics in which none of the memory functions are necessarily centralized. Also, I believe that paper in the International Conference on Computer Design, ICCD 2006, is the first paper to provide an in-depth description of the pipelines in a complex memory system and how they work together.

**SDRAM controller:** I worked with Changkyu Kim to integrate and verify the SDRAM memory controller. Since the SDRAM controller works at a different clock frequency than the main chip, we used custom synchronizers for moving across the clock domains. I was closely involved in the design and was the verification lead for this component. I designed and wrote a sophisticated random test generator, which was successful at catching various integration bugs and exposing rare corner cases in the design.

## 6.2 Looking Back

At the time of the publication of the initial TRIPS study, one of the major concerns was the feasibility of implementing the primary memory system. If anecdotal evidence can be believed, even less ambitious primary memory systems of commercial processors had proved to be very challenging. At the time I started working on the project in Spring 2003, however, I was unaware of the complexities and had my VLSI design experience was limited to graduate class work. The research and pro-

totyping that followed over the next three years has proved very valuable because it not only provided design experience and validated many of our initial research ideas but also opened up new avenues of research. In this section, I will re-visit upon some important design decisions.

Our design philosophy was to go with the least risky design option which was often arrived at after discussing design time, area, performance impact and robustness and crispness of the solution. I aim to illustrate this process using a discussion of two of the biggest intellectual and verification challenges, and using a performance optimization as case studies. I will discuss the complexity of pipelining to obtain maximum throughput within the tiles, the distributed protocol to communicate between the tiles and the dependence predictor.

**Pipelining:** Contrary to conventional wisdom, the design effort involved in setting up the pipeline control logic and optimizing it for peak throughput was much greater than the design of any of the separate sub-structures (like the LSQ) and probably even surpassed all of them together. This problem was complicated because of the area constraints, bypasses, the number of possible execution scenarios, and the number of distinct pipelines.

Area was at a premium in the TRIPS prototype design (and will continue to be so even in the future "billion transistor era" because of power and limited chip real estate). To save area we often shared costly resources in the data tile (like the LSQ register file) and sometimes even cheaper resources like pipeline latches between exclusive pipelines. While these optimizations saved area, they increased the intellectual and verification complexity of the design. The reason behind sharing pipeline latches (although their area overhead was low) was to minimize the number

of separate structures that had to be controlled on pipeline stall conditions and flushes. Such a design style is likely to be useful when goal is to minimize the fanout and the signal propagation delay to achieve competitive clock frequencies.

Another source of complication with the pipelining was the bypassing of intermediate values in the pipeline (say from a store to a load to the same address). Since we had seven distinct pipelines in the DT accomplishing different operations, supporting several types of instructions ((un)cacheable, (un)mergeable, locks etc.) and handling different data types, each bypass stage required very careful planning. Correctly handling exceptions and deallocating pipeline state was another source of bugs in the early implementation stages. We did not find simpler ways to implement these functions; however, the current implementation is unlikely to diminish the scalability because we do not foresee the individual pipeline stages and the bypasses growing significantly.

**Distributed Protocol:** We went through multiple iterations of the distributed protocol used for communicating between the DTs before settling on the all-to-all DT protocol in the prototype. The main complications in designing this protocol were from interactions of local/ remote store arrival messages across different tiles with block commits and block flushes which flowed across tiles from the GT, combined with the processing required for the reissue of dependent loads (which were based on the store arrivals from local/remote tiles), and ensuring all of this happened with predictable latencies. Despite these complexities, the key factor that greatly simplified the design of this protocol was the meticulous specification and enforcement of interfaces between the different tiles from early on in the project. Details of the networks and interfaces are described in Nagarajan's thesis [57].

An undesirable feature of the current implementation is the all-to-all communication between the different DTs. This method for completion detection is unlikely to scale as the number of partitions increases. An alternative (that we considered seriously for the prototype) is to compute the store completion in a distributed manner. Basically, each tile, every cycle, will exchange with its nearest neighbor the last store prior to which all stores had arrived. This solution is similar to the distributed leader election problem, where the leader is the store information. For the interested reader, Erik Reeber has formally verified the nearest-neighbor protocol and details of this protocol can be found in his dissertation [63]. Though completely distributed by construction, the disadvantage of this is solution is its poor performance robustness because the completion latency depends heavily on the arrival order of the stores. Thus, we traded performance robustness for an unscalable design because we feared that the unpredictable latencies may have complicate code optimization and compiler development for the hardware. However, such scalable distributed protocols and their performance robustness are an important avenue for future research in scalable distributed architectures.

**Dependence Predictor:** While the prototype implementation satisfied the bandwidth and latency requirements of the aggressive execution core, one aspect of the design that can be optimized further is the dependence predictor. We implemented a simple 1-bit dependence predictor that waited for all prior stores to arrive before a deferred load could be woken up. Even this simple predictor improves performance over aggressive execution by nearly 17% on SPEC and EEMBC benchmarks, but more performance improvements are likely with advanced predictors. The reason to go with the simplest possible predictor was in part because of proto-

type design schedule constraints. The TRIPS tool chain was developed in parallel with the hardware prototype and thus optimized TRIPS binaries were not available until very late in the prototyping phase (2005). Without real TRIPS binaries and exact quantitative measurements based on those binaries it was difficult to justify more sophisticated predictors and additional schedule delays they would incur. In fact, our design philosophy was not to inadvertently decrease performance because of false positives from the dependence predictor. To minimize such performance slowdowns we made the two aspects of the dependence predictor configurable. First, the dependence predictor can be completely turned off if the programmer suspects slowdowns due to false aliases or if (s)he wants to minimize the effects of dependence predictor for controlled measurements of new optimizations. Second, if the dependence predictor is on, the number of cycles between clearing the dependence predictor is configurable. Contrary to most designs, the dependence predictor is cleared based on number of blocks rather than number of cycles to make the performance more predictable across runs.

**ISA improvements:** Minor ISA extensions that could be included in the next revision of the EDGE ISA include instructions that flush specific addresses (lines) from the L1 caches to improve speed DMA transfers and vector load and store instructions to improve to reduce the load store queue size even further.

## 6.3 Looking Forward

**Towards Ideal LSQs:** In this thesis we proposed four techniques to improve LSQs. Table 6.1 summarizes these techniques. Search filtering mechanism achieves eight-ninths of its possible potential. Better hash functions may help achieve the remaining

| Ideal LSQ Characteristic | Optimization | Potential Achieved | Current Limitations | Possible Solutions |
|---|---|---|---|---|
| Only dependent instructions are bufferred | State Filtering | 3/9ths | STs are not filtered, False positives | Speculative store commit, Better hash functions |
| Only dependent instructions access the LSQ | Search Filtering | 8/9ths | Matching instructions are checked, False positives | Better hash functions |
| Instructions remain in the LSQ only for as long as they are needed to satisfy dependences | Late binding | Unknown | Instructions are held until block commit | Early release of loads: Policies include release of loads after all prior stores have arrived and after forwarding |
| Distribution | Flow Control | Yes | Complexity | Simpler protocols |

Table 6.1: Towards Ideal LSQs

potential. Late binding reduces the size of the LSQ to half of the age-ordered LSQ; additionally, early release of LSQ entries can decrease the LSQ even further. State filtering achieves only one third of its possible potential but can be improved with better hash functions. Finally, the LSQ distribution can be implemented in a complexity effective manner with simpler protocols. In addition to these optimizations, some interesting related questions are discussed below.

**An Alternative to LSQs:** Load-Store Queues are used to satisfy dependencies between speculative memory instructions in a single thread of execution. Cache coherence ensures that memory dependences are satisfied across different threads. An interesting question is if both mechanisms can be combined together. Researchers have proposed speculative versioning caches (SVC) [79] to solve this problem in the context of a hardware cache coherent system. The disadvantages of the SVC proposal are 1) high bookkeeping overheads in maintaining versions of data in the caches and 2) complexity of the multi-step version management algorithm. I believe that an in-place versioning system possibly designed using data compression is an promising alternative that needs to be explored further. Such a memory system can also be used to support hardware transactional memory quite effectively.

**A question on Design Complexity:** During my proposal examination, Dr. Joel Emer, asked me to compare the complexity of LSQ design proposed in this thesis against a traditional LSQ. I have pondered if this question can be quantitatively answered. Complexity is usually only intuitively perceived and there are different flavors of complexity like performance complexity, circuit complexity, verification complexity and physical design complexity. Of all the above complexities,

I think verification complexity closely matches the microarchitects intuitive perception of complexity and taken together with physical design complexity mostly determines the time-to-market. Measuring these complexities and co-relating with user studies may prove to be an interesting exercise that can scientifically aid designers pick a good choice and also help them communicate their designs better.

**Universal Memory Systems:** Due to increased application diversity, the type of memory system that best suits an application today widely varies. Some applications require hardware coherent shared memory for high performance, others work better with streaming memory model, and yet others need emerging transactional memory models. A truly polymorphous memory system will support for all these three types of models without significant additional area than any of the individual designs and at no or little additional design complexity. Some design issues need to solved to overcome before such a memory system can be created: How should the caches be configured and sized? What should be the organization of the routed network for optimal performance in these three modes? For instance, should we have a wide network for DMA transfers or can we use a narrower network? How do we ensure quality of service when certain cores in the CMP are dedicated to streaming applications and certain other cores are dedicated for general purpose applications? Answers to these questions will lead to an energy-efficient memory system that can work well across multiple domains.

# Appendix A

# Multiprocessor Extensions

The TRIPS processor differs considerably from conventional processors because it utilizes block-atomic execution and implements a large instruction window. The purpose of this chapter is to examine the TRIPS consistency model and investigate the interaction between block-atomic execution model and parallel programming. In particular, we are interested in answering the following questions: Does block-atomic execution offer new opportunities to simplify parallel programming? Can block-atomic execution improve the performance of parallel programs? Can block-atomic execution simplify implementation of memory consistency in hardware? How do all these optimizations interact with LSQs?

## A.1 Sequential Consistency on TRIPS

Sequential consistency requires that all operations in a single thread execute in program order. This follows directly from Lamport's definition of sequential consistency [48]:

*"A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."*

At first glance, a sequentially consistent memory may seem conceptually at odds with block-atomic execution model because, irrespective of the program order of reads and writes, the reads from a block will always reach the memory system before the writes from that block can be entered into the memory system. If the other threads see the reordering of reads and writes within the block, then sequential consistency violations may result.

The key to achieving a sequentially consistent memory with block atomic execution is to make reordered reads and writes from one thread invisible to the other threads if the readwrite sets of the threads conflict. Once a violation is suspected, if all but one of the threads is throttled, then the other threads cannot see the reordering, and the resulting execution is same as if the memory was updated sequentially. Normal block-atomic execution can resume once the violation conditions are handled.

To implement sequential consistency based on the above principle, two mechanisms are required: one for detecting violations across threads and another for throttling and resuming execution. Violations can be detected by comparing the read-write sets of the threads either during the commit processing stage of block execution (late detection) or just before the blocks are committed (in-time detection) or as and when memory instructions reach the primary cache tiles (early detection). To recover from violations, either all blocks subsequent to the violating blocks can

be squashed or selective reexecution can be used to save some of the speculative work. The timing of the operations is illustrated in Figure A.1.

Before discussing each of these mechanisms, it may be useful to list out some execution scenarios that can result in sequential consistency violations and use these to guide the design of the detection algorithms. Here are some of the cases (assuming MESI protocol):

1. If any two loads in a block are reordered, and the younger (program order) load reads from a cache line in any state other than modified or exclusive, then other threads may have cached copies and the other threads may observe the reordering of loads if they happen to read the cache line.

2. If two stores within a block are reordered, and either of the stores is to shared/invalid lines, then there is danger of other threads observing the reordering through the out-of-order invalidate/request messages.

3. If a load and store within a block are reordered and either one of these operations accesses cache lines accessed by other threads, then the reordering can be indirectly observed.

## A.2 Early Detection Mechanism

1. As and when loads and stores reach a cache bank, other cache banks re notified of their arrival. Based on the arrival time stamp, the operations that reached the cache banks out-of-order can be identified.

2. If a reordered load reads a cache block in "shared" state then all of the other

copies of the cache block are marked "possibly inconsistent". The cache blocks is returned to shared state when the block with the reordered load commits.

3. Stores speculatively fetch cache block write permissions when they reach the cache banks. In addition, if a store executes out-of-order then its cache block address is held locally in table at each cache bank.

4. When a load reads a cache line marked as "possibly inconsistent" or when a store attempts to fetch write permissions held by a different thread, a sequential consistency violation may have occurred and recovery should be initiated.

5. On violations, the thread that observed the out-of-order effect is rolled back to the last block boundary and all side-effects on the memory system due to that thread should be undone (e.g. "possibly inconsistent" markings)

Early detection scheme requires little additional hardware support over the baseline TRIPS model; even the functionality for "possibly inconsistent" markings can be emulated by maintaining a table at each processor that holds the addresses of reordered shared cache blocks instead of modifying the cache coherence protocol. However, the system-wide broadcast of reordered loads may be problematic for performance. Also, write invalidations from misspeculated blocks may pose additional difficulties in achieving high performance. The next scheme tries to workaround these problems.

## A.3 In-time Detection Mechanism

1. As and when loads and stores reach a cache bank, other cache banks re notified of their arrival. Based on the arrival time stamp, the operations that reached

the cache banks out-of-order can be identified. This step is same as Early detection.

2. All the reordered load addresses are encoded in to a Bloom filter(BF); all the reordered store addresses are encoded into another Bloom filter.

3. After the block is determined to be non-speculative, the BFs are broadcast to all processors. Commit begins after all processors ack the broadcast.

4. If some thread's load or store BF matches the broadcast BF value, then the thread re-starts execution from the violating block.

**Notes:** A global arbiter must provide special mechanisms to ensure forward progress when two threads continuously conflict. This scheme avoids costly the per-load broadcast message and replaces it with a BF thus possibly reducing the performance losses due to network congestion. But false positive matches in the BF's may negatively impact performance. Also, the store commits are held up until all processors ack the BF broadcast. This can potentially reduce the performance due to slower de-allocation of blocks. The next algorithm tries to minimize these performance losses.

## A.4 Late Detection Mechanism

The late detection algorithm is same as in-time detection until step (3). The difference is to allow commits (updates to memory) to speculatively proceed before all the acks come back from the other processors. The recovery mechanism is more involved than the other detection algorithms; in addition to rolling back the execution of the conflicting blocks, the memory updates of one of the conflicting blocks

have to be undone. This can be accomplished by either using a log to hold the old values or a combination of logs and inplace multi version memory. This scheme is the most complex of all mechanisms but is likely to be the best performing among the three schemes.

## A.5 Related Work

In 1997, Hill recommended that hardware should present a simple and easy to understand memory interface for the programmer and that microarchitectural techniques should be utilized to mitigate any performance losses due to the simple interface definition [38]. Most of the community agrees that this simple intuitive model is Sequential Consistency. Of course, there is no empirical studies to back this belief. A user study may be needed to ascertain the benefits of sequential consistency over other consistency models like linearizability and serializability.

After Hill's position paper, Ranganathan et al. and Gnaidy et al. proposed and evaluated microarchitectural innovations to implement SC effectively [62, 30]. Both of these studies used aggressive microarchitectural configurations for that time, a 64-entry instruction window processor, and showed that performance of the programmer friendly SC model aided by speculation can match the performance of the RC models. The implementation in this chapter is different for the original implementations which used history files, and additionally the scale of the TRIPS microarchitecture requires different kind of speculative mechanisms than the ones proposed in the earlier schemes. Recently Wenisch et al. and Ceze et al., have proposed mechanisms to implement SC efficiently by using early and in-time detection mechanisms respectively for superscalar and TLS architectures [81, 16]. We arrived

at these ideas independently and simultaneously. In addition our work is the first to define SC-like consistency for Block-atomic architectures.

**TCC vs Proposed Mechanisms:** In TCC-style transactional memory [34], consistency is defined only at transaction boundaries The model we are attempting to implement is much stronger compared to TCC because we attempt to define consistency on a per-instruction basis.

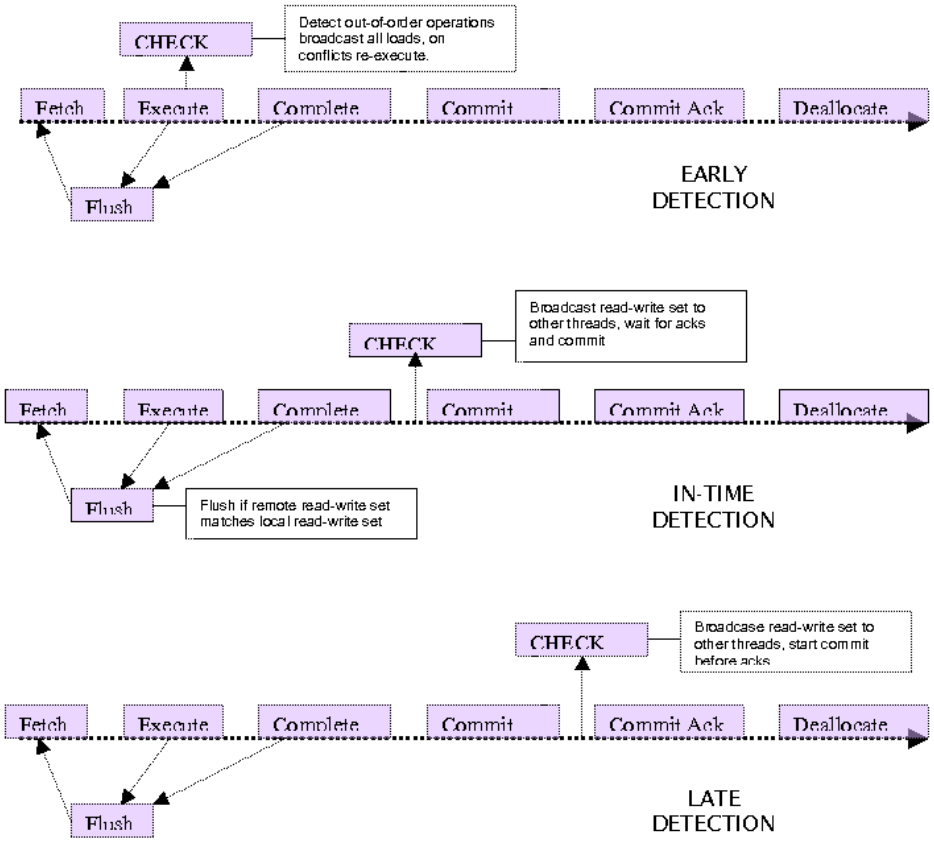



Figure A.1: Three Block SC Implementations

# Bibliography

[1] N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. Bright, J. Brunheroto, C. Caşcaval, n. J Casta W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. Cipolla, P. Crumley, K. Desai, A. Deutsch, T. Domany, M. Dombrowa, W. Donath, M. Eleftheriou, C. Erway, J. Esch, B. Fitch, J. Gagliano, A. Gara, R. Garg, R. Germain, M. Giampapa, B. Gopalsamy, J. Gunnels, M. Gupta, F. Gustavson, S. Hall, R. Haring, D. Heidel, P. Heidelberger, L. Herger, D. Hoenicke, R. Jackson, T. Jamal-Eddine, G. Kopcsay, E. Krevat, M. Kurhekar, A. Lanzetta, D. Lieber, L. Liu, M. Lu, M. Mendell, A. Misra, Y. Moatti, L. Mok, J. Moreira, B. Nathanson, M. Newton, M. Ohmacht, A. Oliner, V. Pandit, R. Pudota, R. Rand, R. Regan, B. Rubin, A. Ruehli, S. Rus, R. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B. Steinmacher-Burow, K. Strauss, C. Surovic, R. Swetz, T. Takken, R. Tremaine, M. Tsao, A. Umamaheshwaran, P. Verma, P. Vranas, T. Ward, M. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. Krolak, C. Li, T. Liebsch, J. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. Wait, J. Wittrup,

M. Bae, K. Dockser, L. Kissel, M. Seager, J. Vetter, and K. Yates. An overview of the BlueGene/L supercomputer. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–22, 2002.

[2] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[3] V. Agarwal, S. W. Keckler, and D. Burger. Scaling of microarchitectural structures in future process technologies. Technical Report TR2000-02, The University of Texas at Austin, February 2000.

[4] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 2003 International Symposium on Microarchitecture*, pages 423–435, 2003.

[5] Alper Buyuktosunoglu and David H. Albonesi and Pradip Bose and Peter W. Cook and Stanley E. Schuster. Tradeoffs in power-efficient issue queue design. In *Proceedings of the 2002 international symposium on Low power electronics and design*, pages 184–189, 2002.

[6] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: data structures for parallel computing. *ACM Transactions on Programming Language and Systems*, 11(4):598–632, 1989.

[7] R. Balasubramonian. *Dynamic Management of Microarchitecture Resources in Future Microprocessors*. PhD thesis, University of Rochester, 2003.

[8] L. Baugh and C. Zilles. Decomposing the load-store queue by function for power reduction and scalability. *IBM Journal of Research and Development*, 50(2/3):287–297, 2006.

[9] S. Bieschewski, J.-M. Parcerisa, and A. Gonzalez. Memory bank predictors. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 666–670, 2005.

[10] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[11] L. Boland, G. Granito, A. Marcotte, B. Messina, and J. Smith. The IBM system/360 model 91: Storage system. *IBM Journal of Research and Development*, 11(1):54–69, January 1967.

[12] E. Borch, E. Tune, S. Manne, and J. Emer. Loose loops sink chips. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 299–310, 2002.

[13] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the End of Silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[14] H. W. Cain and M. H. Lipasti. Memory ordering: A value-based approach. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 90–101, 2004.

[15] F. Castro, L. Pinuel, D. Chaver, M. Prieto, M. Huang, and F. Tirado. Dmdc:

Delayed memory dependence checking through age-based filtering. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 297–308, 2006.

[16] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. Bulksc: bulk enforcement of sequential consistency. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 278–289, 2007.

[17] M. F. Chowdhury and D. M. Carmean. Method, appartus, and system for maintaining processor ordering by checking load addresses of unretired load instructions against snooping store addresses. U.S. Patent Application Number 6,484,254, 2000. Patent assigned to Intel.

[18] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 142–153, 1998.

[19] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.

[20] K. E. Coons, X. Chen, S. K. Kushwaha, D. Burger, and K. S. McKinley. A spatial path scheduling algorithm for EDGE architectures. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–140, October 2006.

[21] A. Cristal, O. J. Santana, M. Valero, and J. F. Martínez. Toward kilo-instruction processors. *ACM Transactions on Architecture and Code Optimizations*, 1(4):389–417, 2004.

[22] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 266–277, 2001.

[23] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[24] K. A. Feiste, B. J. Ronchetti, and D. J. Shippy. System for store forwarding assigning load and store instructions to groups and reorder queues to keep track of program order. U.S. Patent Application Number 6,349,382, 2002. Patent assigned to IBM.

[25] M. Franklin and G. S.Sohi. ARB: a hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, 1996.

[26] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. mei Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proc. of the Sixth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, Oct 1994.

[27] A. Gandhi, H. Akkary, R. Rajwar, S. T. Srinivasan, and K. Lai. Scalable load and store processing in latency tolerant processors. In *Proceedings of the 32th International Symposium on Computer Architecture*, pages 446–457, 2005.

[28] A. Garg, M. W. Rashid, and M. Huang. Slackened memory dependence enforcement: Combining opportunistic forwarding with decoupled verification. In

*ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 142–154, 2006.

[29] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambigution methods for C programs. In *Proc. of the ACM SIGPLAN 2001 conference on Programming Language Design and Implementation*, pages 47–58, 2001.

[30] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is sc + ilp = rc? In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 162–171, 1999.

[31] H. H. Goldstine and A. Goldstine. The electronic numerical integrator and computer (ENIAC). *IEEE Annals of the History of Computing*, 18(1):10–16, 1996.

[32] P. Gratz, C. Kim, R. McDonald, S. W. Keckler, and D. Burger. Implementation and Evaluation of On-Chip Network Architectures. In *Proceedings of the 2006 IEEE International Conference on Computer Design*, pages 477–485, 2006.

[33] P. Gratz, K. Sankaralingam, H. Hanson, P. Shivakumar, R. McDonald, S. W. Keckler, and D. Burger. Implementation and Evaluation of Dynamically Routed Processor Operand Network. In *Proceedings of the 1$^{st}$ ACM/IEEE International Symposium on Networks-on-Chip*, pages 7–17, 2007.

[34] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (tcc). In *ASPLOS-XI: Proceedings of the 11th international conference on Ar-*

*chitectural support for programming languages and operating systems*, pages 1–13, 2004.

[35] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. *SIGPLAN Notices*, 42(6):290–299, 2007.

[36] A. Hartstein and T. R. Puzak. Optimum power/performance pipeline depth. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 117–128, December 2003.

[37] J. Hesson, J. LeBlanc, and S. Ciavaglia. Apparatus to dynamically control the out-of-order execution of load-store instructions. U.S. Patent Application Number 5,615,350, 1995. Patent assigned to IBM.

[38] M. D. Hill. Multiprocessors should support simple memory-consistency models. *Computer*, 31(8):28–34, 1998.

[39] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 1, February 2001.

[40] M. Hrishikesh, K. Farkas, N. P. Jouppi, D. Burger, S. W. Keckler, and P. Sivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 14–24, May 2002.

[41] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the IA-64 architecture. *IEEE Micro*, 20(5):12–23, Sept./Oct. 2000.

[42] W. A. Hughes and D. R. Meyer. Store to load forwarding using a dependency link file. U.S. Patent Application Number 6,549,990, 2003. Patent assigned to AMD.

[43] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, Inc., New York, NY, USA, 1990.

[44] A. Jaleel and B. Jacob. Using virtual load/store queues (vlsqs) to reduce the negative effects of reordered memory instructions. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 191–200, 2005.

[45] A. J. KleinOsowski and D. J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *IEEE Computer Architecture Letters*, 1(1):7, 2006.

[46] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, 2005.

[47] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proc. of the 8th annual symposium on Computer Architecture*, pages 81–87, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[48] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 46(7):779–782, 1997.

[49] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. S. Pierre, D. S. Wells, M. C. Wong-Chan, S.-W. Yang, and R. Zak. The network architecture of

the connection machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, 1996.

[50] B. A. Maher, A. Smith, D. Burger, and K. S. McKinley. Merging head and tail duplication for convergent hyperblock formation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 65–76, December 2006.

[51] D. Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, September 1997.

[52] R. McDonald, D. Burger, S. W. Keckler, K. Sankaralingam, and R. Nagarajan. TRIPS Processor Reference Manual. Technical report, Department of Computer Sciences, The University of Texas at Austin, 2005.

[53] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 23(2):44–55, Mar./Apr. 2003.

[54] T. Monreal, A. González, M. Valero, J. González, and n. Victor Vi˙ Delaying physical register allocation through virtual-physical registers. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 186–192, 1999.

[55] A. Moshovos. *Memory Dependence Prediction*. PhD thesis, University of Wisconsin-Madison, Department of Computer Sciences, December 1998.

[56] T. Mudge. Power: a first-class architectural design constraint. *IEEE Computer*, 34(4):52–58, April 2001.

[57] R. Nagarajan. *Design and Evaluation of a Technology-Scalable Architecture for Instruction-Level Parallelism*. PhD thesis, The University of Texas at Austin, Department of Computer Sciences, May 2007.

[58] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Archtecture*, pages 206–218, 1997.

[59] R. Panwar and R. C. Hetherington. Apparatus for restraining over-eager load boosting in an out-of-order machine using a memory disambiguation buffer for determining dependencies. U.S. Patent Application Number 6,006,326, 1999. Patent assigned to Sun Microsystems.

[60] Y. Patt, S. W. Melvin, W. Hwu, and M. Shebanow. Critical issues regarding HPS, a high performance microarchitecture. In *Proceedings of the 18th annual workshop on Microprogramming*, pages 109–116, 1985.

[61] D. V. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the 2001 International Symposium on Microarchitecture*, pages 90–101, 2001.

[62] P. Ranganathan, V. S. Pai, and S. V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 199–210, 1997.

[63] E. Reeber. *Combining Advanced Formal Hardware Verification Techniques.*

PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 2007.

[64] A. Roth. Store vulnerability window (svw): Re-execution filtering for enhanced load optimization. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 458–468, 2005.

[65] K. Sankaralingam. *Polymorphous Architectures: A Unified Approach for Extracting Concurrency of Different Granularities*. PhD thesis, The University of Texas at Austin, Department of Computer Sciences, October 2006.

[66] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 480–491, December 2006.

[67] T. Sha, M. M. K. Martin, and A. Roth. Scalable store-load forwarding via store queue index prediction. In *Proceedings of the 2005 International Symposium on Microarchitecture*, pages 159–170, 2005.

[68] T. Sha, M. M. K. Martin, and A. Roth. Nosq: Store-load communication without a store queue. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 285–296, 2006.

[69] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01:*

*Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, 2001.

[70] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. Dataflow predication. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 89–100, December 2006.

[71] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines: Achieving resource-efficient latency tolerance. *IEEE Micro*, 24(6):62–73, 2004.

[72] S. S. Stone, K. M. Woley, and M. I. Frank. Address-indexed memory disambiguation and store-to-load forwarding. In *Proceedings of the 2005 International Symposium on Microarchitecture*, pages 171–182, 2005.

[73] S. Subramaniam and G. H. Loh. Fire-and-forget: Load/store scheduling with no store queue at all. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 273–284, 2006.

[74] J. M. Tendler, J. S. Dodson, J. J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 26(1):5–26, January 2001.

[75] J. Thornton. *Design of a Computer System: the Control Data 6600*. Scott, Foresman, and Company, 1970.

[76] TOP 500 SuperComputers Website. http://www.top500.org.

[77] E. F. Torres, P. Ibanez, V. Vinals, and J. M. Llaberia. Store buffer design in first-level multibanked data caches. In *Proceedings of the 32th International Symposium on Computer Architecture*, pages 469–480, 2005.

[78] S. Vetter, S. Behling, P. Farrell, H. H. ff, F. O'Connell, and W. Weir. *The Processor Introduction and Tuning Guide*. IBM, 2001.

[79] T. N. Vijaykumar, S. Gopal, J. E. Smith, and G. Sohi. Speculative versioning cache. *IEEE Transactions on Parallel Distributed Systems*, 12(12):1305–1317, 2001.

[80] H.-S. Wang, X. Zhu, L.-S. Peh, and S. Malik. Orion: a power-performance simulator for interconnection networks. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305, 2002.

[81] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 266–277, 2007.

[82] P. G. Whiting and R. S. V. Pascoe. A history of data-flow languages. *IEEE Ann. Hist. Comput.*, 16(4):38–59, 1994.

[83] Y. Wu, L.-L. Chen, R. Ju, and J. Fang. Performance potentials of compiler-directed data speculation. In *Proc. of the 2003 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, pages 22–31, Mar 2003.

[84] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In *ISCA '99: Proceedings of the*

*26th annual international symposium on Computer architecture*, pages 42–53, 1999.

[85] V. V. Zyuban. *Inherently Lower-Power High Performance SuperScalar Architectures*. PhD thesis, University of Notre Dame, March 2000.

# Vita

Lakshminarasimhan Sethumadhavan (also Simha Sethumadhavan) was born in Chennai, India on June 4th 1978, to Jayashri Raja Rao and Sanjeevi Sethumadhavan. He received a Bachelor of Science degree in Computer Science and Engineering from University of Madras in May 2000. Immediately after that, he entered the graduate program in Computer Science at the University of Texas at Austin. He received a Master of Science degree in December 2005.

Permanent Address: 3357 Lake Austin Blvd. Apt B

Austin TX 78703

This dissertation was typeset with LaTeX $2_\varepsilon$ by the author.