MULTIVERSIONING IN THE STORE QUEUE IS THE ROOT OF ALL STORE-FORWARDING EVIL

BY

SAMUEL S. STONE

B.S., Virginia Polytechnic Institute and State University, 2003

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

ABSTRACT

As semiconductor technologies have continued to scale according to Moore's Law, complexity, power consumption, and energy dissipation have become first-order considerations in microprocessor design. In processors that issue instructions out-of-order, store-load forwarding is a source of significant complexity and energy dissipation. To decrease the complexity and improve the energy efficiency of store-load forwarding, this thesis proposes the forwarding cache (FC), an address-indexed, set-associative alternative to the age-indexed, fully associative store queue (SQ).

The SQ is a content-addressable memory (CAM) that holds in-flight stores in program order. Because the SQ is age-indexed, a load's address may match one or more stores located anywhere in the SQ. Thus, the SQ search is fully associative and priority-encoded. In today's wide-issue processors, the SQ is large (24 to 32 entries), multiported (to accommodate the issue of multiple loads in a single cycle), and fast (no slower than an L1 data cache hit). The energy and complexity required to perform a fast search in a highly associative, multiported CAM are substantial.

The contributions of this work are as follows. First, this thesis shows empirically that address multiversioning and the accompanying age-ordered, priority-encoded search are rarely necessary to perform store-load forwarding correctly. While others have observed the same empirical result on a particular processor configuration using a particular load speculation policy, this thesis extends the analysis to a broad variety of processors using several load speculation policies. Second, this thesis proposes the forwarding cache (FC), an address-indexed, set-associative cache that performs store-load forwarding. Third, this thesis investigates the sensitivity of the FC's performance and energy dissipation to several design parameters, including size, associativity, number of banks, and number of ports. The results show that a small, simple, set-associative FC performs comparably to the complex, fully associative SQ on both control-intensive and scientific workloads, while dissipating nearly ten times less energy than the SQ.

To my parents, Harry and Mary Ann, who have always encouraged and supported me, and to Chip Coulter, the very best computer science teacher any student could hope to find.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Matthew I. Frank, for encouraging me to study cache-based alternatives to conventional store queues. Thanks to Kevin M. Woley, who assisted with an earlier version of this work, and Gene Wu, who assisted with microarchitectural modeling in our performance simulator and helped prepare some of the tables and figures. Thanks to Dr. Frank for drafting the Related Works and Motivation chapters of an earlier version of this work, and to Kshitiz Malik for assisting with the editing and with the preparation of the tables and figures. Thanks also to all the past and present members of Dr. Frank's Implicitly Parallel Architectures group for developing the simulation infrastructure. Finally, many thanks to my lovely wife Kirstin for shouldering so much of the load at home that I can be a husband, a father, a graduate student, and well-rested all at the same time.

This thesis builds on work that was previously published by Samuel S. Stone, Kevin M. Woley, and Matthew I. Frank under the title "Address-Indexed Memory Disambiguation and Store-to-Load Forwarding" in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, November 2005, © IEEE. The work reported in this thesis was supported in part by the National Science Foundation under grant CCR-0429711 and by the Gigascale Systems Research Center, one of five research centers funded by the Semiconductor Research Corporation under the Focus Center Research Program. Computational resources were supported by an equipment donation from Advanced Micro Devices (AMD) and the National Science Foundation under grant EIA-0224453. The Information Trust Institute and Coordinated Science Laboratory of the University of Illinois also donated computing time on the the Trusted ILLIAC cluster. This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

In the past decade, complexity, power consumption, and energy dissipation have become first-order considerations in microprocessor design. As feature sizes continue to scale according to Moore's Law, microarchitectural complexity has an increasingly adverse effect on reliability and manufacturability [1]. And in a world of ubiquitous mobile devices, expensive server rooms, and massive data centers, there is much value in improving the energy and power efficiency of microprocessors [2].

In processors that issue instructions out-of-order, store-load forwarding is a source of significant complexity and energy dissipation. Store-load forwarding occurs when a store forwards its value to a dependent load via the microarchitecture rather than the memory system. Store-load forwarding allows loads and their dependent slices to issue and execute long before the store that produces the load's value commits to the cache-memory hierarchy.

Modern microprocessors use store queues to perform store-load forwarding. The store queue (SQ) is a content-addressable memory (CAM) that holds dispatched, not-yet-committed stores in program order. Because the SQ is age-ordered rather than address-indexed, a load's address may match one or more stores located anywhere in the SQ. Thus, the SQ search is fully associative and priority-encoded. In today's wide-issue processors, the SQ is large (24 to 32 entries), multiported (to accommodate the issue of multiple loads in a single cycle), and fast (no slower than an L1 data cache hit). The energy and complexity required to perform a fast search in a highly associative, multiported CAM are substantial. In modern processors, a load instruction dissipates nearly as much energy accessing the SQ as it does accessing the L1 data cache (see Chapter 6).

Much of the store queue's complexity and energy dissipation stem from *address multiversioning*. Address multiversioning is analogous to register renaming. A register renamer maps definitions of the same architectural register to different physical registers, thereby eliminating anti- and output-dependences from the instruction stream and increasing the available instruction-level parallelism [3–5]. Likewise, the SQ performs address multiversioning because it maps definitions (stores) of the same address to different entries in the SQ. By so doing, the SQ eliminates write-after-read (WAR) and write-after-write (WAW) dependences from the stream of load and store instructions and enables forwarding of any not-yet-committed versions

of an address's value to dependent loads. The results and analysis in Chapter 3 demonstrate that address multiversioning incurs significant complexity but rarely increases performance.

To decrease the complexity and improve the energy efficiency of store-load forwarding, this thesis proposes the forwarding cache (FC), an address-indexed, set-associative alternative to the age-indexed, fully associative store queue. Stores write their values to the FC speculatively, as soon as their addresses and data are ready. Loads access the FC in parallel with the L1 data cache access. Because the FC does not perform address multiversioning, FC searches do not incur the complexity of the SQ's fully associative, priority-encoded searches. Furthermore, the FC's address-indexed organization is amenable to address banking, which increases the effective bandwidth of store-load forwarding without increasing the complexity or the energy dissipation.

The contributions of this work are as follows. First, this thesis shows empirically that address multiversioning (and the accompanying age-ordered, priority-encoded search) are rarely necessary to perform store-load forwarding correctly, even when loads and stores execute out-of-order and there are multiple in-flight stores to the same address. While others have observed the same empirical result on a particular processor using a particular load speculation policy [6], this thesis extends the analysis to a broad variety of processors using several load speculation policies. Second, this thesis proposes the forwarding cache (FC), an address-indexed, set-associative cache that performs store-load forwarding. Third, this thesis investigates the sensitivity of the FC's performance and energy dissipation to several design parameters, including size, associativity, number of banks, and number of ports. The results show that a simple, set-associative FC performs comparably to the complex, fully associative SQ on both control-intensive and scientific workloads, while dissipating nearly an order of magnitude less energy than the SQ.

The remainder of this thesis is organized as follows. The next chapter provides background and discusses related works. Chapter 3 analyzes address multiversioning in the context of store-load forwarding, and Chapter 4 presents the design of the forwarding cache. Chapter 5 explains the methodology for measuring the performance, energy, power, latency, and area of various store queue and forwarding cache configurations. Chapter 6 compares the performance and complexity of the SQ and FC across a broad range of benchmarks and processor configurations. Chapter 7 discusses conclusions and future work.

CHAPTER 2
BACKGROUND AND RELATED WORK

A modern microprocessor typically uses a store queue to perform store-load forwarding (Figure 2.1). Store-load forwarding occurs when a store forwards its value to a dependent load via the microarchitecture rather than the memory system. Forwarding allows a load and its dependent slice to issue and execute long before the store that produces the load's value commits to the memory system.

The store queue (SQ) is a content-addressable memory (CAM) that holds the addresses and values of dispatched, not-yet-committed stores in program order. Stores allocate entries from the tail of the SQ as they dispatch and release entries from the head of the SQ as they commit to the memory system. When a store issues, it calculates its address and then writes its address and value into the SQ entry that was allocated at dispatch.

When a load issues, it searches the SQ and the level-one (L1) data cache in parallel. Because the SQ is age-ordered rather than address-indexed, the load's address may match the address of a store located anywhere in the SQ. Thus, the SQ search is fully associative. Furthermore, the load's address may match the addresses of multiple stores, or match the address a store that is later than the load in program order. Therefore, the SQ must priority encode the matching stores to select the *closest matching store*, i.e., the matching store that dispatched most recently before the load dispatched. If the load's search in the SQ returns a value, the load simply uses that value and discards the value retrieved from the L1 data cache.

In today's wide-issue processors, the SQ is large, multiported, and fast. First, the capacity of the SQ must increase as the size of the instruction window increases. Otherwise, the SQ tends to fill before the instruction window fills, yielding pipeline stalls and under utilization of other processor resources. In modern processors, the SQ capacity is typically 24 or 32 entries [8],[9]. Second, the SQ bandwidth must increase as the processor's issue width increases. Modern processors typically issue up to two loads per cycle, necessitating multiporting or replication of the SQ [8–10]. Third, the SQ's search latency must not exceed the hit latency of the L1 data cache. In short, the energy and complexity required to perform a fast search in a highly associative, multiported CAM are substantial. In modern processors, a load instruction dissipates nearly as much energy accessing the SQ as it does accessing the L1 data cache (see Chapter 6).

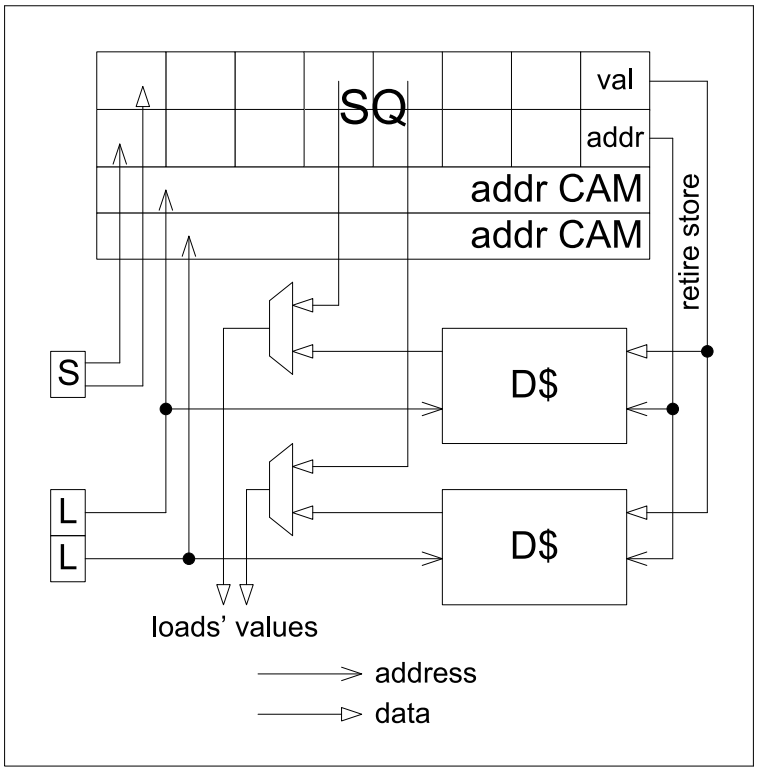


Figure 2.1 Load/store datapath with a conventional store queue. With two CAMs, the SQ permits two loads to issue and execute in parallel. This figure is derived from Figure 2 in [7].

2.1 Store-Load Forwarding

Researchers have proposed many techniques for increasing the capacity and bandwidth of the store-load forwarding mechanism. There has also been much effort directed at reducing the complexity and energy dissipation of store-load forwarding. This section discusses the proposed schemes and differentiates the work presented in this thesis from related works.

One approach is to segment and pipeline the store queue [11],[12]. The segments can then be searched serially or concurrently. For an SQ of a given capacity, a parallel search of the segments achieves lower latency than a conventional search in a monolithic SQ. The latency and energy dissipation of a serial search depend on the number of segments searched.

The hierarchical SQ described in [13] includes a small level-one (L1) SQ that holds the N most recently dispatched stores, a larger level-two (L2) SQ that holds all other in-flight stores, and a membership test buffer that filters searches of the L2 SQ. Both the L1 and L2 SQs include CAMs to support associative searches. A related proposal eliminates the CAM from the L2

SQ by replacing it with a small cache for store-load forwarding and a simple FIFO that buffers stores for in-order commitment to the memory system [14].

Torres et al. [15] proposed a distributed, hierarchical SQ in conjunction with a banked L1 data cache and a sliced memory pipeline. The hierarchy comprises small L1 SQs that hold the N stores most recently dispatched to each L1 data cache bank, and a centralized L2 SQ that handles overflows from the L1 queues. A bank predictor steers loads and stores to the appropriate cache bank and SQ.

Franklin and Sohi [16] proposed the address resolution buffer (ARB), a store-load forwarding mechanism that is both address-indexed and age-ordered. The ARB consists of address-interleaved banks, with each bank holding the addresses and values of its stores in age order. The banked structure increases the store-load forwarding bandwidth. However, because the stores in each bank are age-ordered, each store must allocate an ARB entry at dispatch, before the store's address is known. Thus, each store allocates an entry in all ARB banks, which diminished the ARB's effective capacity.

In related work, Roth proposed a forwarding store cache (FSC) in which each set of a small, four-way associative cache is managed as a store queue [17]. The FSC sets support address versioning (there can be multiple stores to the same address in the same set), but the stores in each FSC set are not held in program order. Thus, when a load's address matches the address of multiple stores in an FSC set, the FSC must perform a priority encoding on the matching, unordered stores. This unordered priority encoding is more time-consuming than the ordered priority encoding of a conventional store queue. Sethumadhavan et al. [18] have recently proposed an unordered, late-binding SQ that is conceptually similar to the FSC. The unordered, late-binding SQ scales to very large instruction windows and handles bank over-subscription efficiently.

Search filtering and *state filtering* have been proposed as techniques for increasing the SQ's effective bandwidth and capacity, respectively [12],[17],[19],[20]. In processors with large instruction windows, the percentage of loads that obtain their values from the SQ ranges from 7% to 44%, depending on the processor and the application [17]. The goal of search filtering is to identify the loads that will not obtain their values from the SQ and to prevent those loads from searching the SQ. The goal of state filtering is to identify the stores that will not forward their values before committing to memory and to prevent those stores from allocating entries in the SQ. By conserving SQ bandwidth and increasing effective SQ capacity, search filtering and state filtering not only increase overall performance but also reduce the complexity and energy dissipation of the SQ.

Various forms of *memory renaming* have been proposed [21–25]. In this context, memory renaming refers to the class of store-load forwarding mechanisms in which a load obtains its value directly from an earlier store without performing an address-based search. In general, memory renaming schemes use sophisticated memory dependence predictors in the pipeline's front end to identify store-load dependences prior to calculating effective addresses. Store-load forwarding then occurs speculatively using a low-complexity datapath.

Sha et al. [25] proposed NoSQ, a memory renaming scheme in which the SQ is completely eliminated. NoSQ predicts store-load dependences using the StoreSets predictor, which predicts store-load dependences with 99.8% accuracy. The StoreSets predictor is algorithmically similar to the conventional store-set predictor [26], but is organized as a hybrid predictor that includes two set-associative memory dependence predictors. One of the predictors has a path-sensitive indexing function while the other does not. Stores and loads that are predicted to participate in store-load forwarding simply modify the register alias table (RAT) in the renamer to collapse DEF-store-load-USE chains into DEF-USE chains, effectively using the register file to forward a store's value directly from the instruction that computed it to the instructions that consume it.

Earlier memory renaming proposals include [21] and [22], in which some stores forward their values to dependent loads using structures similar to physical register files. The collapsing of some DEF-store-load-USE chains into DEF-USE chains is also proposed in [22]. Finally, in [23] a load predicts the distance (number of stores) between itself and the store that produces its value. The load obtains its value directly from that store's SQ entry, without performing an associative search. Likewise, in [24] a store predicts the distance (number of loads) between itself and the load that consumes its value. The store then uses that predicted distance to deposit its value directly into the consumer load's entry in the load queue.

The forwarding cache proposed in this thesis is most similar to the cache-like structures used for store-load forwarding in [6], [27], and [28]. During the runahead execution described in [27], the processor uses a runahead cache to forward the values of pseudo-retired runahead stores to runahead loads. Stone et al. [28] proposed a store forwarding cache (SFC) that is conceptually similar to the FC. However, this thesis evaluates the FC in terms of performance, bandwidth, and energy dissipation; neither the bandwidth nor the energy dissipation of store-load forwarding is reported in [28]. Also, the evaluated SFCs have capacities 4 to 16 times greater than the capacities of the FC's evaluated in this thesis. Finally, the SFC incurs some additional complexity to support partial flushes, which the FC does not support.

Garg et al. [6] proposed storing speculatively written L1 cache lines in a special level-zero (L0) cache to alleviate memory bandwidth pressure caused by re-executing loads to validate memory ordering. The capacity of the L0 cache is half that of the L1 cache, and the two caches have the same number of read and write ports. In the slackened memory dependence enforcement (SMDE) system, all loads and stores access the L0 cache during execution. If a load or store misses in the L0 cache, the L0 obtains the desired line from the L1 cache. However, dirty L0 lines are never written back to other levels of the cache-memory hierarchy. In the back end of the pipeline, all stores commit to the the L1 cache and all loads validate their speculative values in the L1 cache.

Despite the substantial increase in area and energy dissipation incurred by replacing the SQ with a large L0 cache and by failing to filter load re-execution, the performance of the SMDE system is not competitive with that of the conventional SQ. To increase performance, two enhancements to the naive SMDE system are proposed. First, the 8-entry write buffer is augmented with a CAM to support store-load forwarding. Second, a fully associative search in a 16-entry fuzzy disambiguation queue (FDQ) detects potential memory ordering violations shortly after loads and stores issue, triggering selective re-execution of the offending load (and its dependent slice) or the offending store. The FDQ and the selective re-execution collectively perform the same function as a conventional memory dependence predictor, reducing the rate of load violations by a factor of 20. With these two enhancements, the SMDE system substantially outperforms a system equipped with a conventional load queue, a conventional store queue, and no memory dependence predictor.

2.2 Memory Disambiguation

Dynamically scheduled processors expose parallelism by executing instructions speculatively and out of order, without violating the data dependences among the instructions. To expose additional parallelism, processors may execute load instructions prior to resolving their dependences on earlier stores. A load that issues speculatively may violate a read-after-write (RAW) dependence on an earlier store. Therefore, a system that performs load speculation must detect dependence violations. The process by which load violations are detected is *memory disambiguation*.

Conventional processors use a load queue (LDQ) to disambiguate memory references [29]. The load queue is an age-ordered, fully associative buffer containing all in-flight loads. When a store completes, it searches the load queue for a later, completed load that matches the store's

address. If such a load is found, a load violation has occurred. Processors typically recover from a load violation by flushing the load and all subsequent instructions.

Given the complexity of its circuitry and the dynamic power consumed by each fully associative, age-ordered search, the load queue does not scale well as the size of the instruction window increases. There have been many proposals related to reducing the complexity and energy dissipation of memory disambiguation. Filtered load re-execution, the disambiguation technique used in this paper, is discussed below.

In a system equipped with filtered load re-execution, any load that may have violated a true memory dependence accesses the data cache just before retirement [7],[30]. If the value obtained during nonspeculative re-execution does not match the value obtained during speculative execution, then a load violation has occurred. While [31] observed that load re-execution is viable only if the fraction of loads that must re-execute is low, the SMDE system of [6] challenges that observation by using an L0 cache for store-load forwarding and a separate L1 cache for unfiltered load re-execution.

The proposed schemes for filtered load re-execution are distinguished by the algorithm used to filter loads that need not re-execute. Cain and Lipasti required re-execution of loads that issue before an earlier store with an unresolved address [30]. Roth's store vulnerability window, which is used for disambiguation in all simulations presented in this thesis, assigns sequence numbers to stores and sequence ranges (vulnerability windows) to loads, then passes those sequence numbers through an address-indexed filter to detect potential memory ordering violations [7].

2.3 Memory Dependence Prediction

In processors that allow loads and stores to issue as soon as their operands are available, the rate of pipeline flushes caused by load violations can be fairly high. Several schemes for reducing the rate of load violations have been proposed. These schemes, which rely on memory dependence predictors either to identify loads that are likely to violate read-after-write (RAW) dependences or to identify sets of stores and loads in which RAW dependences are likely to exist, constrain the issue order of loads and stores to avoid potential violations. These predictors vary in their complexity and in their effectiveness at decreasing load violations without substantially reducing instruction-level parallelism. The predictors used in this thesis are discussed below.

A load wait table (LWT) dynamically identifies loads that have violated dependences in the past [29],[32],[33]. Loads that hit in the LWT are not permitted to issue speculatively, but rather must wait until all earlier stores have executed. By eliminating speculation on loads that are likely to have dependences on in-flight stores, the load wait table reduces the rate of misspeculation at the expense of exposing less parallelism.

Alternatively, the store-set predictor identifies a (possibly empty) set of in-flight stores upon which a load is likely to depend, and the scheduler constrains that load and those stores to execute in-order [26]. That is, the store-set predictor predicts store-load dependences, and the scheduler enforces those predicted dependences. This scheme's goal is to ensure that a load's producer store has executed before the load issues, while delaying the load just long enough to avoid violating its true data dependence.

CHAPTER 3
MOTIVATION

The motivation for the forwarding cache is the empirical observation that the percentage of loads that require address multiversioning in the store queue is quite low across a wide variety of processor configurations and benchmarks. The SQ performs address multiversioning by mapping definitions (stores) of the same address to different entries in the SQ. This technique eliminates write-after-read and write-after-write dependences from the stream of load and store instructions and enables forwarding of any not-yet-committed version of an address's value to a dependent load. However, despite the fact that loads and stores issue out-of-order and there are often multiple stores to the same address in-flight, simulations show that when the store queue forwards data to a load, the entry in the store queue that forwards the data nearly always corresponds to the most recently *issued* store matching the load's address.[1] The SQ's failure to leverage this property of store-load forwarding is the source of considerable complexity.

Table 3.1 Percentage of loads that require multiversioning in the store queue. Results are averaged across the SPECINT2K benchmarks.

SPECINT2K		pipeline width		
		2-wide	4-wide	8-wide
MDP	NONE	0.01	0.01	0.02
	LWT	0.02	0.18	1.01
	SSET	0.01	0.05	0.39

Table 3.2 Percentage of loads that require multiversioning in the store queue. Results are averaged across the SPECFP2K benchmarks.

SPECFP2K		pipeline width		
		2-wide	4-wide	8-wide
MDP	NONE	0.01	0.04	0.15
	LWT	0.06	0.22	1.53
	SSET	0.02	0.05	0.19

Tables 3.1 and 3.2 show the percentages of loads that require address multiversioning on three different processors using three different techniques to enforce memory dependences. The three processors are 2-wide, 4-wide, and 8-wide superscalars. The MDP-NONE configuration allows all loads and stores to issue as soon as their source operands become ready. The load-wait table (MDP-LWT) and store-set predictor (MDP-SSET) configurations were described in Chapter 2.3. The experimental methodology used to obtain this data is described in

[1] Others have observed the same empirical result on an 8-wide processor with no memory dependence predictor [6]. This thesis extends the analysis to a wide variety of processor configurations and load speculation policies.

Chapter 5. In each table, the 2-wide, 4-wide, and 8-wide configurations correspond precisely to the `SQ-INF,LS2` configuration in Figure 6.2, the `SQ-INF,LS4` configuration in Figure 6.4, and the `SQ-INF,LS3` configuration in Figure 6.6, respectively.

Each entry in Tables 3.1 and 3.2 gives the percentage of dynamic loads for which the address multiversioning capability of the SQ actually performs a useful function. Specifically, multiversioning performs a useful function when (a) the SQ forwards a value to a load, (b) the SQ contains at least two previously issued stores to the load's address, and (c) the correctly forwarded value is obtained from some store other than the most recently issued store to the load's address. With no memory dependence prediction, in the worst case 0.15% of the loads required multiversioning (SPECFP2K, 8-wide). With the store set predictor, 0.39% of the loads required multiversioning in the worst case (SPECINT2K, 8-wide). On narrower pipelines the percentage of loads requiring multiversioning decreases because the effective size of the instruction window also decreases, yielding less instruction-level parallelism and a lower degree of out-of-order execution.

When the load wait table is used, the percentage of loads requiring multiversioning increases substantially, peaking at 1.53% (SPECFP2K, 8-wide). By delaying a load that is likely to depend on an earlier store until all earlier stores have executed, the LWT increases the rate of write-after-read dependence violations. That is, the LWT delays some loads so long that subsequent stores matching the load's address execute while the load is waiting. The delayed loads therefore need the SQ's address multiversioning mechanism to discard the values of those later, matching stores.

The small benefit provided by address multiversioning in the SQ incurs substantial complexity. First, to support multiversioning, stores must allocate SQ entries in program order, at dispatch time, before their addresses are known. There is no relation between a store's address and its location in the SQ. Therefore, loads must perform fully associative (content addressable) searches of the entire SQ. Second, unlike a conventional content addressable memory, in which there can only be one entry with any particular tag, the SQ has the property that a load's tag might match multiple entries. If multiple SQ entries do match the load's address, the SQ must perform a selection operation (priority encoding) to choose the matching store that dispatched most recently before the load dispatched. Only then can the SQ forward a value to the load.

Eliminating the SQ's multiversioning capability yields substantial benefits. First, the unbanked, fully associative queue can be replaced with a banked, set-associative forwarding cache. Second, the priority encoder can be eliminated, thereby conserving additional power

and latency. The experiments in Chapter 6 show that a forwarding cache with the same area and latency as a conventional, CAM-based store queue achieves the same performance as the SQ while dissipating an order of magnitude less power.

CHAPTER 4
DESIGN

The forwarding cache (FC) is an address-indexed, set-associative cache that performs store-load forwarding without store multiversioning. Figure 4.1 depicts the load/store datapath of a processor equipped with the FC. After a store calculates its address, it indexes into the FC and writes its value into the corresponding FC line. Likewise, loads access the FC in parallel with the L1 data cache lookup. If a load hits in the FC, it simply obtains its value from the corresponding FC line.

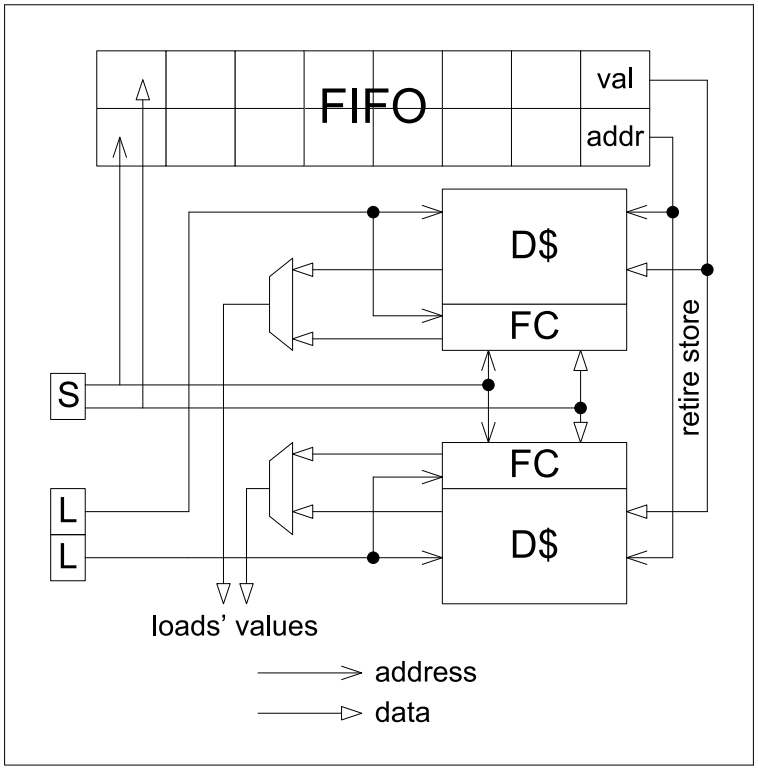


Figure 4.1 Load/store datapath with a forwarding cache. The FC and the L1 data cache each have two banks. This figure is derived from Figure 2 in [7].

The FC is compatible with filtered load re-execution, a scalable, low-complexity technique for disambiguating memory references. See Chapter 2 for a discussion of disambiguation via load re-execution. The experiments in this thesis use a modified version of Roth's store vulnerability window [7] to detect *load violations* (loads that violate true memory dependences). In

Roth's store vulnerability window (SVW), the decoder assigns a store sequence number (SSN) to each store. The SSNs represent a total ordering on all the stores in the processor. When stores forward their values to loads, they also forward their SSNs. The SSN of the store that forwarded its value to a load defines the load's *store vulnerability window* because the load is only vulnerable to memory dependence violations with respect to stores such that `store.SSN > load.SVW`. If a load obtains its value from the data cache rather than the SQ, the load's SVW is set to the SSN of the most recently committed store. Finally, all stores and loads access a direct-mapped, address-indexed filter (the SSBF) as they retire. Each store writes its SSN into the corresponding entry. Each retiring load compares its SVW to the SSN in the corresponding filter entry, and if `store.SSN > load.SVW`, the load re-executes nonspeculatively.

Replacing the SQ with the FC changes the SVW algorithm slightly. When the FC is used, a load may obtain its value from a store that is later in program order than the load. To detect such cases of erroneous store-load forwarding (which are caused by write-after-read dependence violations that do not exist when the SQ's address multiversioning logic is present), a load must also re-execute if it obtains its value from the FC and `store.SSN < load.SVW`. Likewise, a load may obtain part of its value from a later store and part of its value from the L1 cache. Loads that obtain their values partially from the data cache and partially from the FC are rare, so the load re-execution pipeline simply re-executes them. This thesis refers to the modified version of the SVW algorithm as *extended SVW* (E-SVW).

Each FC line consists of a tag, a valid bitmask, a double-word of data, and two store sequence numbers (see Figure 4.2). When a store executes, it uses its address to index into the FC. If the store's address is already in the FC, the store simply overwrites the existing data, sets the corresponding bits in the valid bitmask, and overwrites the existing SSN. Otherwise, if no free lines in the store's set are available, a least-recently-stored (LRS) replacement policy is used to evict an FC entry in the store's set.

Tag	Valid	Data	Upper SSN	Lower SSN
(4 bytes)	(1 byte)	(8 bytes)	(2 bytes)	(2 bytes)

Figure 4.2 Forwarding cache line.

Double-word stores and single-word or subword stores are treated somewhat differently. When a double-word store writes to an FC line, it overwrites all eight bytes of data, the entire valid mask, and both SSNs. By contrast, a single-word or subword store overwrites only half the data (the upper or lower word), half the valid mask (the bits corresponding to the upper

or lower word), and one SSN (upper or lower). The rationale for this scheme is as follows. First, with the compiler and ISA used in this study, both double-word and single-word stores are common, so the FC does not optimize for one class at the expense of the other. Reducing the granularity of each FC entry to a single word would force the double-word stores and loads to access two FC banks; increasing the granularity of all FC writes to a double-word would create contention between single-word stores to the upper and lower word of the same FC line. Second, to ensure correctness, the granularity of the SSN must correspond to the granularity of the data. That is, when the upper and lower words of an FC line contain data from two different stores, the FC line must hold the SSNs of both stores. Otherwise, the SVW algorithm would not necessarily detect all memory ordering violations.

The FC requires no special handling related to the commitment or cancellation of stores. Stores on mispredicted paths write their values into the FC, speculatively, just as stores on correctly predicted paths do. When a mispredicted branch resolves, the valid bits in the FC are simply cleared, thereby removing the data deposited by any wrong-path stores. The stores that are pending in the write buffer write their values back into the FC after the mispredicted branch retires. When a store commits, its data remains in the FC until it is overwritten by another store to the same address or until the FC line is evicted by a younger store.

When a load executes, it accesses the FC in parallel with the data cache. If the load's address matches a line in the FC, the load obtains its value and its SVW from that line. If the load is a double-word load and the SSNs in the FC line are not equal, the load simply sets a bit indicating that it should re-execute before it retires. Such cases are rare.

Because the FC is address-indexed, it can be banked by address to increase the store-load forwarding bandwidth while conserving energy, power, and area. As shown in Chapter 6, address banking gives the FC significant advantages over the SQ in terms of bandwidth and complexity. However, banking is not without cost. A controller is necessarily to manage pending loads and stores when banks are oversubscribed.

CHAPTER 5
METHODOLOGY

5.1 Energy, Power, Latency, and Area

This thesis compares the energy dissipation, average power dissipation, latency, and area of various forwarding cache and store queue configurations for a 90 nm process technology using CACTI 4.2 [34].[1] The SQ is modeled as a fully associative cache. Because the SQ model does not include the priority encoder, CACTI's estimates of the SQ's area, latency, and read energy are conservative. Likewise, the FC is modeled as a set-associative cache, and that model does not include the area or energy devoted to the FC controller. Write energy is not included in the analysis because writes to the SQ and FC dissipate very little energy. Furthermore, a processor equipped with a forwarding cache must support in-order commitment of stores by buffering its in-flight stores in a simple FIFO.

Because the access times of many FC and SQ configurations differ by 30% or more, direct comparison of the energy and average power dissipation among different FC and SQ configurations is inadequate. Therefore, for each FC or SC configuration, a first-order model of voltage scaling is used to normalize the access time reported by CACTI. The energy and power dissipation are then computed using the scaled value of the supply voltage and the normalized access time. For more details, refer to Appendix A. Using the voltage scaling model, the access times of each FC and SQ are normalized to 0.67 ns. The normalized access time corresponds to two clock cycles on a 3 GHz clock, which is reasonable given that the SQs of the AMD Athlon 64, AMD Opteron, and Intel Core 2 processors are all accessed in two clock cycles [9],[10] at frequencies ranging from 2 GHz to 4 GHz.

5.2 Performance

The performances of the forwarding cache and the store queue are compared using execution-driven, cycle-level simulation of several superscalar processor configurations. The simulator

[1] The interface to CACTI 4.2 contains a documented bug that prevents the tool from modeling caches with certain combinations of banks and ports. We have modified the interface to correct this bug.

executes a variant of the 64-bit MIPS ISA and faithfully models the effects of branch mispredictions and load violations, including execution of misspeculated instructions in the simulator's back end.

All simulations use the SPECINT2K and SPECFP2K benchmarks [35] and the MinneSPEC lgred inputs [36]. The benchmarks are compiled with gcc -O3. We lack compiler support for the Fortran 90 benchmarks (`galgel`, `lucas`, `facerec`, and `fma3d`) and runtime library support for `wupwise`, `sixtrack`, and `eon`. Also, the benchmark source code has been modified to compile and link with the limited support of our libraries, which are largely derived from FreeBSD. For example, some double-precision floating point values or computations have been reduced to single-precision, and some library calls have been replaced with other calls that perform the same function. These modifications should affect neither the core computations performed by the benchmarks nor the validity of the results obtained from their execution.

For most benchmarks, the simulator skips the first 300M instructions (warming the caches), then executes the next 200M instructions. However, `applu` and `mgrid` have only 376M and 244M instructions, respectively, in our ISA. For `applu` the simulator skips 200M instructions, then runs to completion. For `mgrid` the simulator executes the entire benchmark. All results are validated against a trace obtained from an architectural simulator.

To understand how the performance trade-offs between the FC and the SQ vary with the underlying architecture, three superscalar processor configurations are simulated, with issue widths of 2, 4, and 8. As Table 5.1 shows, the rate of instruction supply, the size of the instruction window, and the degree of out-of-order execution increase with the issue width. In all configurations, the processor frequency is 3 GHz and the latency to main memory is 300 cycles (100 ns). Each configuration has sufficient physical registers to avoid stalls. There are no rename alias table (RAT) checkpoints; the processor initiates recovery from a branch misprediction or load violation when the misspeculated branch or load reaches the head of the reorder buffer. Also, the 2-wide and 4-wide processors do not wake up the dependents of load instructions speculatively. By contrast, the 8-wide processor wakes up the dependents of load instructions speculatively using a perfect cache-hit predictor.

The issue bandwidth to the load/store unit is varied as shown in Table 5.2. The AMD Athlon 64 and Opteron have two load/store ports (allowing any combination of two loads and/or stores to issue each cycle), while the Intel Pentium 4 and Core 2 processors have one load port and one store port (allowing up to one load and one store to issue each cycle) [8–10]. In addition to those conventional load/store bandwidths, increased load/store bandwidth is simulated to

Table 5.1 Simulated processor configurations. The rate of instruction supply, the size of the instruction window, and the degree of out-of-order execution increase with the pipeline width.

		2 Wide	4 Wide	8 Wide
Pipeline Width	Decode/Issue/Retire	2/2/2	4/4/4	8/8/8
Instr Window	ROB	64	128	512
	Scheduler	16	32	128
	Write buffer	4	8	16
Instr Supply	ICache	16 KB, 128 lines, 2-way	32 KB, 256 lines, 4-way	perfect perfect
	Bpred	 4 KB gshare, 14 history bits	12 KB tournament 4 KB gshare, 14 history bits 4 KB local 4 KB select	48 KB tournament 16 KB gshare, 16 history bits 16 KB local 16 KB select
	Fetch through taken branches	No	No	Yes
	Minimum branch mispred penalty	13 cycles	13 cycles	17 cycles
Data Supply	DCache	16 KB, 256 lines, 2-way	32 KB, 512 lines, 4-way	32 KB, 512 lines 4-way
	L2Cache	512 KB, 4 K lines, 8-way	1 MB, 8 K lines, 8-way	1 MB, 8 K lines, 8-way
Scheduling	Issue ports (int/fp/branch)	1/1/1	2/2/1	4/4/2
	Wakeup of load dependents	conservative	conservative	aggressive, perfect prediction
Load-Store	Issue ports	See Table 5.2	See Table 5.2	See Table 5.2
	SQ/FC	Varies	Varies	Varies
	FC bank conflict controller entries	8	12	16
	Memory dependence predictor	LWT, 1 K entry	Store sets, 1 K entry SSIT	Store sets, 2 K entry SSIT

determine whether the forwarding cache's high load/store bandwidth translates to increased performance.

The capacities and bandwidths of the SQ and FC are varied over a broad range. The Pentium 4 and the Alpha 21264 have SQs with 24 and 32 entries, respectively [8],[29], so SQs

with 16, 24, and 32 entries are modeled. To determine the upper bound on SQ performance, SQs that have as many entries as the reorder buffer are also modeled. This thesis refers to SQ configurations using the format SQ-N, where N denotes the number of SQ entries (N=INF denotes a SQ with as many entries as the ROB). Likewise, to determine the upper bound on FC performance, an 8-way associative FC with 64 sets, 8 banks, and up to 2 read/write ports per bank is modeled. To find the most complexity-effective configurations, many small, simple FC's are also modeled. This thesis refers to FC configurations using the format FC-16S-4W-2B-1P, where S denotes the number of sets, W denotes the associativity, B denotes the number of banks, and P denotes the number of read/write ports per bank. Finally, when the SQ is used, stores complete in two cycles (one cycle for address calculation and one cycle to write into the SQ). When the FC is used, stores complete in three cycles. The additional cycle corresponds to the tag check in the FC.

Table 5.2 Load/store issue ports. The specified combinations of exclusive load ports, exclusive store ports, and shared load/store ports are modeled in the simulations described in Chapter 6.

Label	Ports
LS1	1 load/store port
LS2	2 load/store ports
LS3	3 load/store ports
LS4	4 load/store ports
L1S1	1 load port, 1 store port
L2S2	2 load ports, 2 store ports

Other features of load/store handling are held constant across the processor configurations. Memory disambiguation is performed via the extended store vulnerability window (E-SVW) with a 512-entry store sequence bloom filter (SSBF). The load/store bandwidth to the SSBF is the same as the issue bandwidth to the load/store unit. All statements regarding FC capacity assume that store sequence numbers (SSNs) are 16 bits, and the simulator does not model SSN overflow. The L1 data cache has two read ports and two read/write ports which are shared among executing loads, re-executing loads, prefetching stores, and committing stores. The simulator does not model contention or bandwidth restrictions on the memory bus. Finally, when modeling the store-set predictor, we use the store-set alias table (SSAT) rather than the last-fetched store table (LFST) to enforce store set dependences. As described in [37], the

SSAT simplifies the scheduling of loads and stores in store sets. The simulator models 32 store set identifiers (SSIDs), a 32-entry SSAT, and sufficient SSAT tags to avoid stalls.

CHAPTER 6
EVALUATION

6.1 Energy, Latency, and Area

This section compares the complexity of the forwarding cache (FC) and the store queue (SQ) in terms of energy dissipation, access latency, and chip area. As described in Chapter 3, a load that participates in store-load forwarding almost always obtains its value from the most recently executed store matching the load's address. Thus, the address multiversioning and priority encoding implemented by the SQ, while costly, are rarely necessary to ensure that stores forward their values to loads properly. The FC leverages this observation to reduce the complexity of store-load forwarding.

As Figure 6.1 shows, the per-read energy dissipation of a store queue of reasonable capacity and bandwidth far exceeds the per-read energy dissipation of a forwarding cache of greater capacity and bandwidth. In this figure, the SQs and FCs are clocked at 3 GHz and accessed in 2 cycles. As a frame of reference, the dashed line represents the energy dissipation of a 32 KB, dual banked L1 data cache with 1 read port and 1 read/write port per bank.

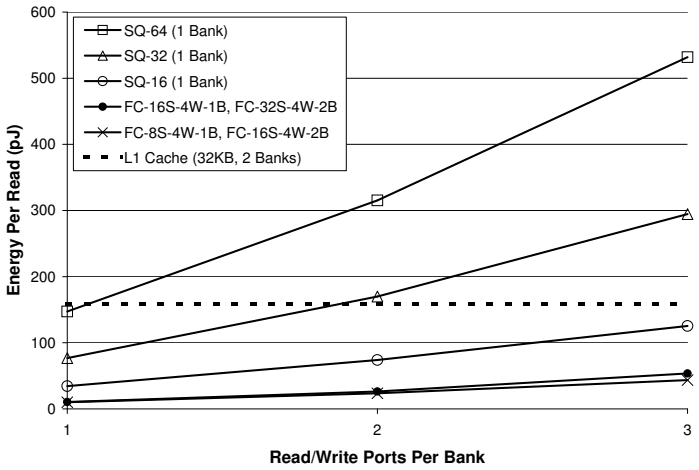


Figure 6.1 Read energy as a function of store-forwarding capacity and bandwidth. The dashed line represents the per-read energy dissipation of a 32 KB, 4-way associative, dual-banked L1 data cache with 1 read port and 1 read/write port per bank. The larger FC configurations (FC-16S-4W-2B with 2 ports per bank, for example) achieve the same performance as the 32-entry SQ.

With regards to the SQ, two trends are evident in Figure 6.1. The SQ's energy dissipation increases superlinearly not only with the number of SQ entries, but also with the number of ports. The magnitude of the SQ's energy dissipation is also significant. Consider a 32-entry SQ with 2 R/W ports, clocked at 3 GHz and accessed in 2 cycles. Such an SQ is representative of the SQs used on the Alpha 21264, the Intel Core 2, and the AMD Opteron. As Table 6.1 shows, this SQ dissipates slightly more energy per read than does a 32 KB, dual-banked L1 cache. In short, the per-read energy dissipation of a conventional SQ is slightly greater than the per-read energy dissipation of a conventional data cache, and increasing either the capacity or the bandwidth of the SQ increases that energy dissipation super-linearly.

Because the FC is set-associative and banked, it dissipates much less energy than an SQ of comparable capacity and bandwidth. Consider a 4-way associative FC with 32 sets, 2 banks, and 2 R/W ports per bank, clocked at 3 GHz and accessed in 2 cycles. This FC has four times the capacity and twice the bandwidth of the SQ discussed in the previous paragraph, yet the FC dissipates 6.4X less energy per read. Furthermore, the trends relating the FC's energy dissipation to its capacity and bandwidth are somewhat different than the trends observed in the SQ. For example, increasing the capacity of a given FC by increasing the number of sets has little impact on the FC's energy dissipation. On the other hand, the energy dissipation increases linearly as the number of ports per bank increases, but increases dramatically when the number of banks exceeds two, because the energy dissipated during routing dominates the energy dissipated accessing a bank. In short, as long as the number of banks is restricted to two, the FC offers more capacity and bandwidth than the SQ while dissipating substantially less energy.

Relative to the SQ, the FC exhibits not only lower energy dissipation, but also lower access time. Referring to Table 6.1, which lists the access times and chip areas for a variety of SQ and FC configurations with comparable capacities and bandwidths, the access times of the 16-entry SQs and the 32-entry SQs are roughly 35% higher and 50% higher, respectively, than the access times of comparable FCs. Because the store-forwarding logic is on the processor's critical path for loads and stores, the timing slack introduced by the FC is quite valuable. Depending on the other factors affecting the critical path, the slack might enable a faster clock frequency or might increase the manufacturing yield at a given frequency.

The chip areas of the FCs and SQs in Table 6.1 are roughly comparable, and represent a small fraction of the area devoted to the L1 data cache. While a rigorous analysis of leakage energy is beyond the scope of this paper, area and leakage are often strongly correlated. Thus, we assume that the leakage energy of the FC and the SQ are roughly comparable.

Table 6.1 Complexity of store-forwarding circuits. The access time, area, and read energy are obtained by modeling the SQs and FCs in CACTI at 90 nm. The read energy at 0.67 ns (2 cycles at 3 GHz) is obtained using the scaling model described in Appendix A. The size of the L1 cache is 32 KB, not including tags. The size of each FC is slightly greater than 1 KB, including tags.

Device	Sets	Ways	Banks	Ports Per Bank	Access Time (ns)	Area (mm^2)	Read Energy (pJ)	Read Energy @ 0.67ns (pJ)
L1 Cache	128	4	2	1R,1R/W	0.89	3.44	97.1	159.1
FC-16S-4W-2B-1P	16	4	2	1R/W	0.68	0.07	9.7	10.1
FC-16S-4W-2B-2P	16	4	2	2R/W	0.70	0.30	21.8	23.6
SQ-16	1	16	1	1R/W	0.94	0.04	18.7	34.5
SQ-16	1	16	1	1R,1W	0.96	0.08	18.9	36.3
SQ-16	1	16	1	2R/W	0.96	0.13	37.9	73.9
SQ-32	1	32	1	1R/W	1.05	0.05	31.7	76.8
SQ-32	1	32	1	1R,1W	1.08	0.11	32.2	83.4
SQ-32	1	32	1	2R/W	1.08	0.17	64.6	170.0
SQ-32	1	32	1	2R,2W	1.03	0.47	83.9	193.2

6.2 Performance

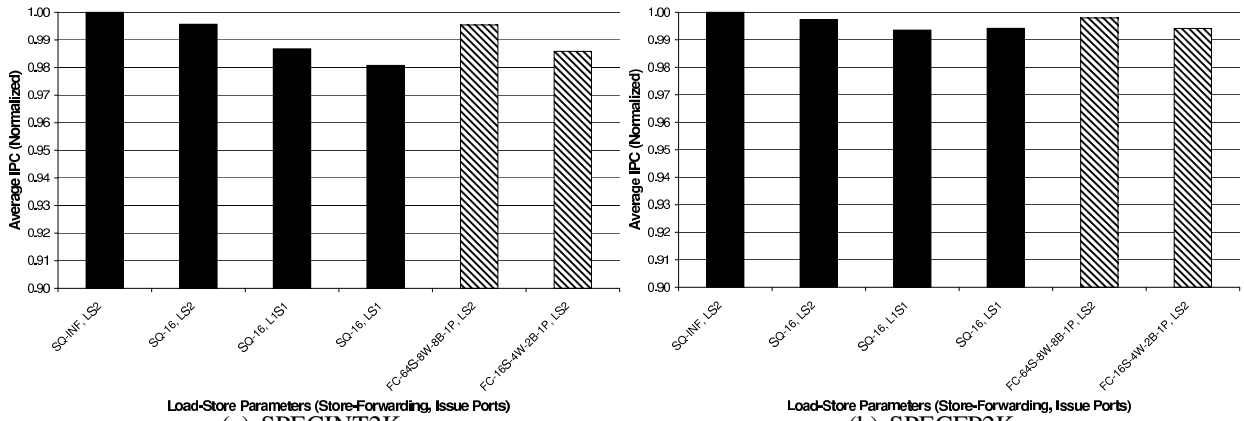
This section presents the performance results for various SQ and FC configurations in superscalar processors with 2-wide, 4-wide, and 8-wide pipelines. For each superscalar processor, this section examines the average performance of SQs and FCs with a broad range of capacities and bandwidths, including configurations with very high and very low complexity. Based on that exploration of the store-forwarding design space, the most complexity-effective store-forwarding configurations are selected. Performance is then compared on each of the individual benchmarks.

6.2.1 2-wide processor

Figure 6.2 shows the average performance of the 2-wide processor across the SPECINT2K and SPECFP2K benchmarks. The performance of each store-load forwarding configuration is normalized to the performance of an SQ that has the same capacity as the reorder buffer and bandwidth equal to the processor's issue width. Clearly, a 16-entry SQ with either one read port and one write port or a single read/write port achieves nearly all the performance of the

baseline SQ with substantially less complexity. We select the 16-entry SQ with one read port and one write port as the complexity-effective SQ because it yields slightly better performance than the SQ with a single read/write port, yet the two SQs have essentially the same complexity (see Table 6.1).

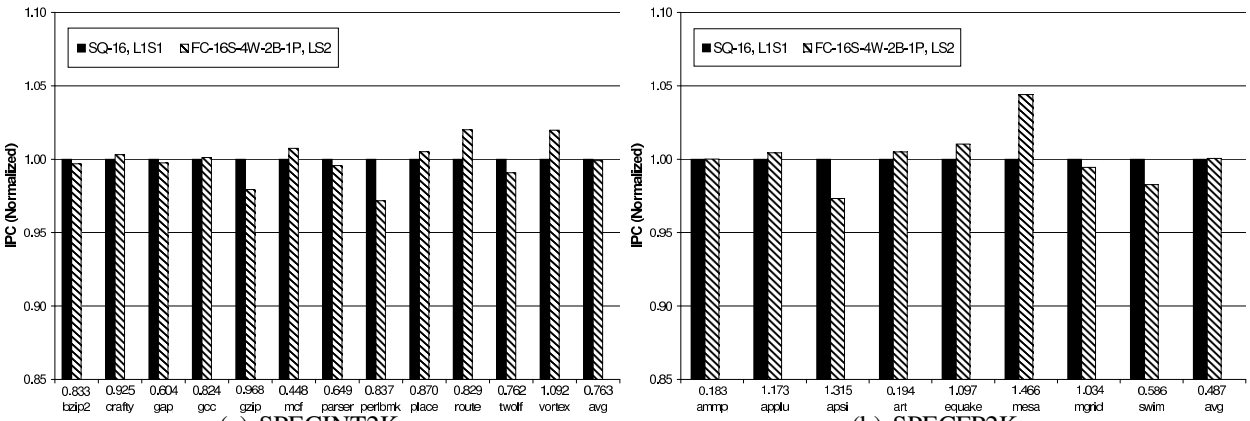
2-wide figures



(a) SPECINT2K (b) SPECFP2K

Figure 6.2 Average performance of store-forwarding circuits on a 2-wide processor. IPCs are averaged across (a) SPECINT2K and (b) SPECFP2K benchmarks. The results are normalized to the performance of an ideal SQ, which has the same capacity as the ROB and bandwidth equal to the issue width. The FC-64S-8W-8B-1P configuration represents an approximate upper bound on the performance that can be achieved with the FC.

The FC-64S-8W-8B-1P configuration in Figure 6.2 demonstrates that the upper bound on the forwarding cache's performance is within 0.5% of the upper bound on the store queue's performance. This configuration represents an 8-way FC with 64 sets partitioned into 8 banks and with 1 read/write port per bank. Increasing the associativity, capacity, or bandwidth beyond this configuration does not appreciably increase performance. The upper bound on the FC's performance is lower than the upper bound on the SQ's performance precisely because the FC does not support multiversioning. Although store-forwarding patterns that require multiversioning are rare (see Chapter 3), they do exist. When an FC-equipped processor encounters such store-load forwarding patterns, the processor either suffers an increased rate of load violations or trains the memory dependence predictor to impose a more conservative ordering on the offending stores and loads. In either case, this small reduction in performance is the fundamental cost of replacing the SQ with the FC.



(a) SPECINT2K (b) SPECFP2K

Figure 6.3 Complexity-effective SQ and FC configurations on a 2-wide processor. Results for (a) SPECINT2K and (b) SPECFP2K benchmarks are normalized to the performance of a 16-entry SQ with 1 read port and 1 write port. The 1 KB FCs are dual-banked and 4-way associative. Table 6.1 lists the chip area and energy dissipation for the SQ and FC configurations.

Figure 6.3 shows the performance of the 2-wide processor for the most complexity-effective SQ and FC. Several benchmarks suffer performance degradations of 2%-3% when the FC is used. In the case of `apsi` the degradation is fundamental, as 0.36% of all loads require multiversioned store-forwarding. The only other benchmark in which more than 0.08% of all loads require multiversioned store forwarding is `parser`, and in 13 of the 20 benchmarks, fewer than 1 in 100 000 loads require multiversioned store forwarding. In the case of `gzip` the degradation is not fundamental. As the capacity, associativity, and bandwidth of the FC decrease from FC-64S-8W-8B to FC-8S-2W-2B, the load violation rate increases by an order of magnitude (from 1 violation per 2000 retiring loads to 1 violation per 238 retiring loads). We conclude that capacity and conflict misses in the FC are responsible for the increased rate of load violations. In the case of `swim` the cause of the performance drop is unclear. Finally, several benchmarks experience speedups when the FC is used, notably `route`, `vortex`, and `mesa`. In `vortex` and `mesa`, the increased bandwidth to the store-forwarding logic decreases the pressure on the 16-entry scheduler, yielding fewer stalls. In `route`, the cause of the speedup is unclear.

In summary, for the 2-wide processor, a complexity-effective FC achieves the same performance as a complexity-effective SQ, while using the same chip area and considerably less energy. Specifically, a 1 KB, 4-way forwarding cache with 2 banks and 1 read/write port per bank achieves average performance within 0.1% of a 16-entry SQ that has 1 read port and 1

write port. The FC uses 0.07 mm^2 of chip area, compared to 0.08 mm^2 for the SQ. More significantly, the FC dissipates 3.4X less energy per read than the SQ does. Across the SPECFP2K benchmarks, the average power dissipated by loads accessing the SQ is 11.3 mW, compared to 3.2 mW for the FC. For the SPECINT2K benchmarks, the corresponding figures are 23.9 mW in the SQ and 6.7 mW in the FC.

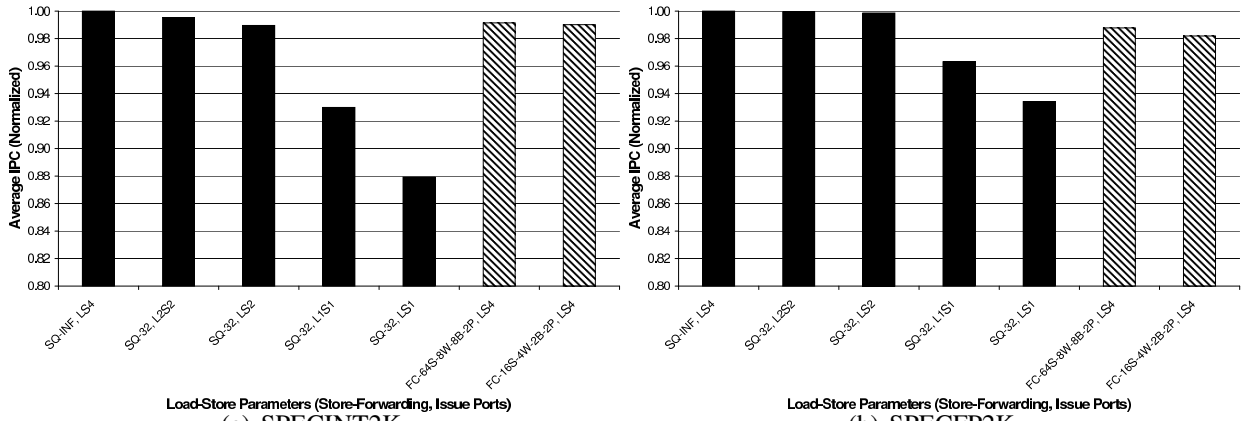
6.2.2 4-wide processor

Figure 6.4 shows the average performance of the 4-wide processor across the SPECINT2K and SPECFP2K benchmarks. Again, the performance of each store-load forwarding configuration is normalized to the performance of an SQ that has the same capacity as the reorder buffer and bandwidth equal to the processor's issue width. In this case, a 32-entry SQ with two read/write ports achieves nearly all the performance of the baseline SQ with substantially less complexity. While the SQ with one read port and one write port is considerably less complex than the SQ with two read/write ports, that reduction in complexity is accompanied by a substantial decrease in performance. Therefore, we select the 32-entry SQ with two read/write ports as the complexity-effective SQ for this processor.

The FC-64S-8W-8B-2P configuration in Figure 6.4 demonstrates that the upper bound on the FC's average performance is within roughly 1% of the upper bound on the SQ's average performance. Again, the decrease in performance from the ideal SQ to the ideal FC represents the fundamental cost of replacing the SQ with the FC.

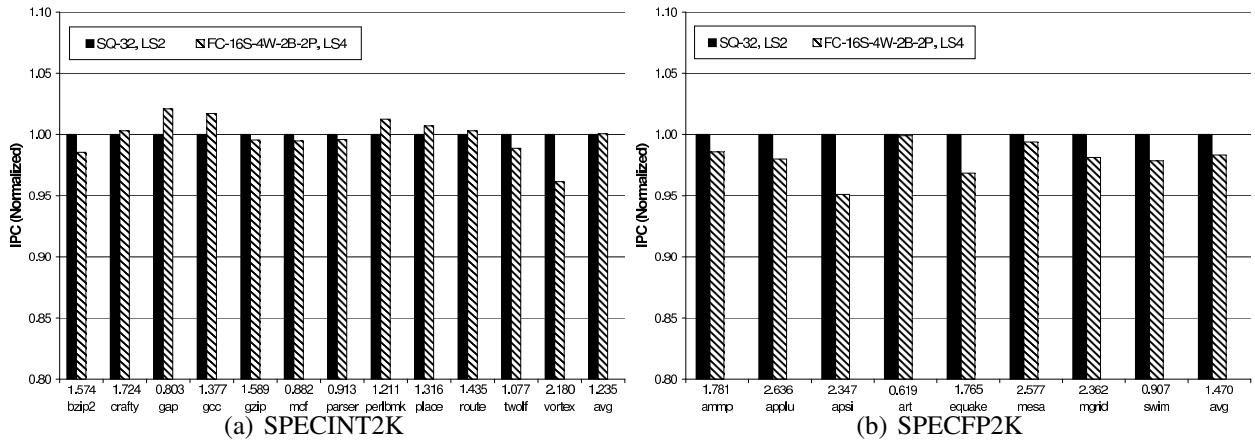
Figure 6.5 shows the performance of the 4-wide processor for the most complexity-effective SQ and FC. Across the individual benchmarks, the complexity-effective FC performs comparably to the complexity-effective SQ. These results are quite similar to those observed on the 2-wide processor. Note that the complexity-effective FC configuration in Figure 6.5 has the same associativity, same number of sets, and same number of banks as the complexity-effective FC configuration in Figure 6.3. However, in the 2-wide processor, an FC with one port per bank provided sufficient load/store bandwidth to maximize performance. In the 4-wide processor, an FC with two ports per bank is needed. Experiments (not shown) indicate that on the 4-wide processor, several benchmarks lose substantial performance when a low-bandwidth FC is used.

In summary, for the 4-wide processor, a complexity-effective FC achieves nearly the same performance as a complexity-effective SQ, while using roughly twice as much chip area but considerably less energy. Specifically, on the SPECFP2K benchmarks, a 1 KB, 4-way FC with 2 banks and 2 read/write ports per bank achieves average performance within 1.7% of a 32-entry SQ that has 2 read/write ports. On the the SPECINT2K benchmarks, the FC actually



(a) SPECINT2K (b) SPECFP2K

Figure 6.4 Average performance of store-forwarding circuits on a 4-wide processor. IPCs are averaged across (a) SPECINT2K and (b) SPECFP2K benchmarks. The results are normalized to the performance of an ideal SQ, which has the same capacity as the ROB and bandwidth equal to the issue width. The FC-64S-8W-8B-2P configuration represents an approximate upper bound on the performance that can be achieved with the FC.



(a) SPECINT2K (b) SPECFP2K

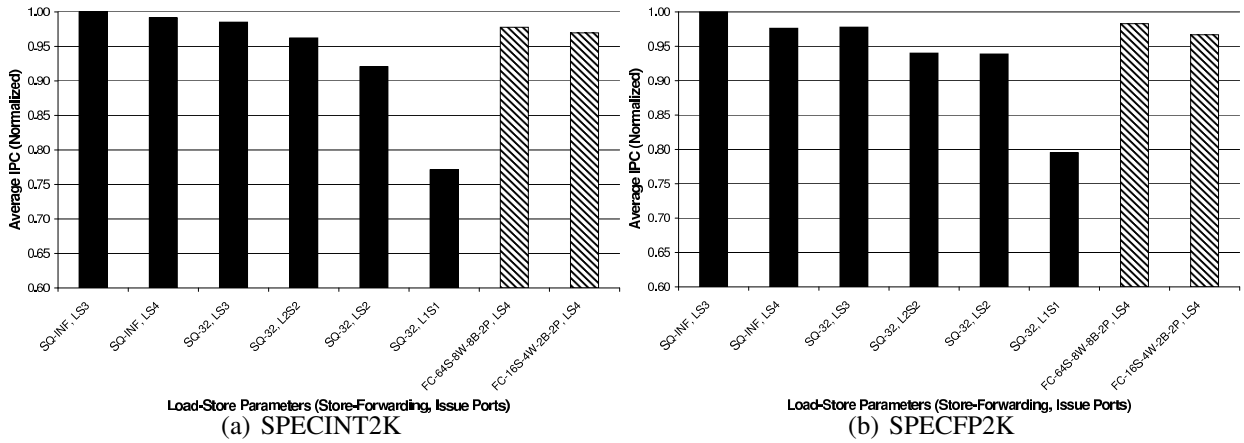
Figure 6.5 Complexity-effective SQ and FC configurations on a 4-wide processor. Results for (a) SPECINT2K and (b) SPECFP2K benchmarks are normalized to the performance of a 32-entry SQ with 2 read/write ports. The 1 KB FCs are dual-banked and 4-way associative. Table 6.1 lists the chip area and energy dissipation for the SQ and FC configurations.

outperforms the SQ by 0.1%. The FC uses 0.30 mm^2 of chip area, compared to 0.17 mm^2 for the SQ. However, the FC dissipates 7.2X less energy per read than the SQ does. Across the SPECFP2K benchmarks, the average power dissipated by loads accessing the SQ is 161 mW,

compared to 22 mW for the FC. For the SPECINT2K benchmarks, the corresponding figures are 197 mW in the SQ and 28 mW in the FC.

6.2.3 8-wide processor

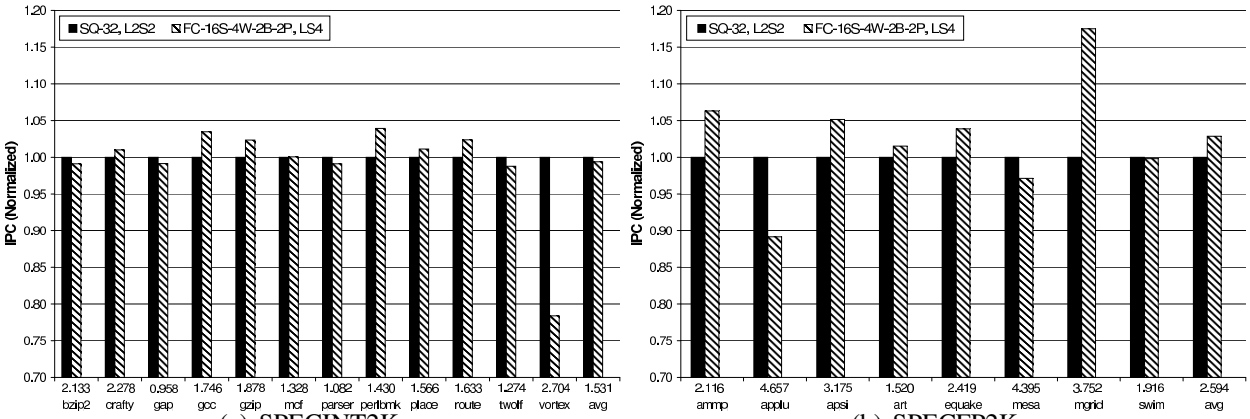
Figure 6.6 shows the average performance of the 8-wide processor across the SPECINT2K and SPECFP2K benchmarks. The performance of each store-load forwarding configuration is normalized to the performance of an SQ that has the same capacity as the reorder buffer and three read/write ports. In this case, a 32-entry SQ with two read ports and two write ports achieves roughly 95% of the performance of the baseline SQ with substantially less complexity than any of the higher-performing SQs. Therefore, we select the 32-entry SQ with two read ports and two write ports as the complexity-effective SQ for this processor.



(a) SPECINT2K (b) SPECFP2K

Figure 6.6 Average performance of store-forwarding circuits on an 8-wide processor. IPCs are averaged across (a) SPECINT2K and (b) SPECFP2K benchmarks. The results are normalized to the performance of an SQ which has the same capacity as the ROB and three read/write ports (increasing the number of read/write ports actually decreases performance). The FC-64S-8W-8B-2P configuration represents an approximate upper bound on the performance that can be achieved with the FC.

The FC-64S-8W-8B-2P configuration in Figure 6.6 demonstrates that the upper bound on the FC's average performance is within 1.7% of the upper bound on the SQ's average performance for the SPECFP2K benchmarks; the corresponding figure for the SPECINT2K benchmarks is 2.2%. Again, the decrease in performance from the ideal SQ to the ideal FC represents the fundamental cost of replacing the SQ with the FC. Clearly that fundamental cost is increasing as the issue width and instruction window of the processor increase, because the percentage of loads that require address multiversioning increases with the issue width and the size



(a) SPECINT2K (b) SPECFP2K

Figure 6.7 Complexity-effective SQ and FC configurations on an 8-wide processor. Results for (a) SPECINT2K and (b) SPECFP2K benchmarks are normalized to the performance of a 32-entry SQ with 2 read ports and 2 write ports. The 1 KB FCs are dual-banked and 4-way associative. Table 6.1 lists the chip area and energy dissipation for the SQ and the FC configurations.

of the instruction window. However, when complexity-effective restrictions on the capacities and bandwidths of the FC and SQ are considered, the FC's additional capacity and bandwidth compensate for this fundamental performance loss.

Two significant results are evident in Figures 6.6 and 6.7. First, the performance of a small, low-energy forwarding cache (FC-16S-4W-2B-2P) exceeds the performance of the complexity-effective SQ on average and on 12 of the 20 benchmarks. The SQ suffers not only from capacity stalls, but also from reduced load-store bandwidth relative to the FC. Second, the FC performs quite poorly on `vortex`. Even with the store set predictor, 3.0% of the loads in `vortex` require multiversioning in the store-forwarding circuit. As expected, this store-forwarding pattern translates to a high load violation rate. In the FC-16S-4W-2B-2P configuration, the load violation rate observed in `vortex` (1.23 load violations per 100 retired loads) actually exceeds the misprediction rate for conditional branches. Hence, the performance loss is substantial.

In summary, for the 8-wide processor, the complexity-effective FC outperforms the 32-entry SQ on average and on 12 of the 20 benchmarks, while using less area and considerably less energy. Specifically, a 1 KB, 4-way FC with 2 banks and 2 read/write ports per bank outperforms a 32-entry SQ with 2 read ports and 2 write ports by 2.8% on the SPECFP2K benchmarks. On the SPECINT2K benchmarks, the FC achieves performance within 0.6% of the SQ. The FC uses 0.30 mm^2 of chip area, compared to 0.47 mm^2 for the SQ. Furthermore,

the FC dissipates 8.2X less energy per read than the SQ does. Across the SPECFP2K benchmarks, the average power dissipated by loads accessing the SQ is 348 mW, compared to 44 mW for the FC. For the SPECINT2K benchmarks, the corresponding figures are 322 mW in the SQ and 42 mW in the FC.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

Address multiversioning in the store queue is a source of considerable complexity in the store-load forwarding circuitry of modern microprocessors. This thesis has shown empirically that address multiversioning is rarely needed to perform store-load forwarding correctly, even in wide superscalars with large instruction windows. Eliminating support for address multiversioning substantially decreases the latency and energy dissipation of the store-load forwarding logic. In particular, replacing the monolithic, fully associative store queue with a banked, set-associative forwarding cache reduces the energy dissipation and the average power dissipation of the store-forwarding logic by a factor of eight, while maintaining the same performance.

APPENDIX A
VOLTAGE SCALING MODEL

Because the access times of many FC and SQ configurations differ by 30% or more, direct comparison of the energy and average power dissipation among different FC and SQ configurations is inadequate. Therefore, for each FC or SC configuration, a first-order model of voltage scaling is used to normalize the access time reported by CACTI. The energy and power dissipation are then computed using the scaled value of the supply voltage and the normalized access time.

The relationship between supply voltage (V_{DD}) and frequency is given by Eq. (A.1), where f is the device frequency, V_{DD} is the supply voltage, V_{TH} is the threshold voltage, and α is a technology-dependent parameter that decreases with feature size [38]. In our computations we set α to 1.2 [39],[40] and assume that V_{TH} is 0.5 V (CACTI does not report V_{TH}).

$$f \;\; \alpha \;\; V_{DD}/(V_{DD} - V_{TH})^{\alpha} \tag{A.1}$$

Next, the standard model of technology scaling, given by Eqs. (A.2) and (A.3), and the scaled value of V_{DD} [41] are used to compute the energy dissipation of the forwarding cache or store queue. In short, power and energy are directly proportional to the square of V_{DD}. Thus, given the base energy dissipation and supply voltage reported by CACTI, the scaled energy dissipation (after scaling V_{DD} to normalize the access time) is given by Eq. (A.4).

$$P = C * V_{DD}^2 * f \tag{A.2}$$

$$E = C * V_{DD}^2 \tag{A.3}$$

$$E_{SCALED} = E_{BASE} * (V_{DD\_SCALED}/V_{DD\_BASE})^2 \tag{A.4}$$

Finally, this thesis reports the average power dissipation for various configurations of the FC and SQ. As Eq. (A.5) shows, the average power dissipation is computed as the product of the scaled energy (joules per access), the activity factor (accesses per cycle), and the clock

frequency (cycles per second). The activity factors are measured using a performance simulator as described in Chapter 5.2.

$$P_{SCALED} = E_{SCALED} * activity\_factor * f \tag{A.5}$$

Clearly, the comparison of energy and power dissipation among various FC and SQ configurations relies on high-level scaling models that may introduce substantial imprecision in the computations. However, we are interested not in the raw values obtained via the scaling models, but rather in the order of magnitude of the difference between the FQ's power and energy as compared to the SQ's power and energy. To that end, the first-order models are adequate.

REFERENCES

[1] P. Bose, D. H. Albonesi, and D. Marculescu, "Guest editors' introduction: Power and complexity aware design," *IEEE Micro*, vol. 23, no. 5, pp. 8–11, 2003.

[2] M. J. Irwin and J. P. Shen, "Revitalizing computer architecture research," http://www.cra.org/Activities/grand.challenges/architecture/home.html, Dec. 2005.

[3] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, Jan. 1967.

[4] R. Keller, "Look-ahead processors," *ACM Computing Surveys*, vol. 7, no. 4, pp. 177–195, 1975.

[5] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS-IV)*, 1991, pp. 176–189.

[6] A. Garg, M. W. Rashid, and M. Huang, "Slackened memory dependence enforcement: Combining opportunistic forwarding with decoupled verification," in *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA-33)*, 2006, pp. 142–154.

[7] A. Roth, "Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization," in *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA-32)*, 2005, pp. 458–468.

[8] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the Pentium™4 processor," in *Intel Technology Journal*, 2001, no. Q1.

[9] "Intel® 64 and IA-32 architectures optimization reference manual," http://www.intel.com/products/processor/manuals/index.htm, May 2007.

[10] "Software optimization guide for AMD64 processors," http://www.amd.com/us-en/Processors/TechnicalResources/, Sept. 2005.

[11] H. Akkary and K. Chow, "Processor having multiple program counters and trace buffers outside an execution pipeline," 2001. U.S. Patent Number 6,182,210.

[12] I. Park, C. L. Ooi, and T. N. Vijaykumar, "Reducing design complexity of the load/store queue," in *Proceedings of the 36th Annual Symposium on Microarchitecture (MICRO-36)*, 2003, pp. 411–422.

[13] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint processing and recovery: Towards scalable large instruction window processors," in *Proceedings of the 36th Annual Symposium on Microarchitecture (MICRO-36)*, 2003, pp. 423–434.

[14] 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 32nd International Symposium on Computer Architecture (ISCA-32)*, 2005, pp. 446–457.

[15] E. F. Torres, P. Ibanez, V. Vinals, and J. M. Llaberia, "Store buffer design in first-level multibanked data caches," in *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA-32)*, 2005, pp. 469–480.

[16] M. Franklin and G. S. Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," *IEEE Transactions on Computers*, vol. 45, pp. 552–571, May 1996.

[17] A. Roth, "A high-bandwidth load-store unit for single- and multi- threaded processors," Department of Computer and Information Science, University of Pennsylvania, Technical Report MS-CIS-04-09, June 2004.

[18] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler, "Late-binding: Enabling unordered load-store queues," in *Proceedings of the 34th International Symposium on Computer Architecture (ISCA-34)*, 2007, pp. 347–357.

[19] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler, "Scalable hardware memory disambiguation for high ILP processors," in *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO-36)*, 2003, pp. 399–410.

[20] L. Baugh and C. Zilles, "Decomposing the load-store queue by function for power reduction and scalability," *IBM Journal of Research and Development*, vol. 50, no. 2/3, pp. 287–297, 2006.

[21] G. S. Tyson and T. M. Austin, "Improving the accuracy and performance of memory communication through renaming," in *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-30)*, 1997, pp. 218–227.

[22] A. Moshovos and G. S. Sohi, "Streamlining inter-operation memory communication via data dependence prediction," in *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-30)*, 1997, pp. 235–245.

[23] T. Sha, M. M. Martin, and A. Roth, "Scalable store-load forwarding via store queue index prediction," in *Proceedings of the 38th Annual International Symposium on Microarchitecture (MICRO-38)*, 2005, pp. 159–170.

[24] S. Subramaniam and G. H. Loh, "Fire-and-forget: Load/store scheduling with no store queue at all," in *Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO-39)*, 2006, pp. 273–284.

[25] T. Sha, M. M. Martin, and A. Roth, "NoSQ: Store-load communication without a store queue," in *Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO-39)*, 2006, pp. 285–296.

[26] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-25)*, June 1998, pp. 142–153.

[27] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *Proceedings of the 9th Annual International Symposium on High-Performance Computer Architecture (HPCA-9)*, 2003, pp. 129–140.

[28] S. S. Stone, K. M. Woley, and M. I. Frank, "Address-indexed memory disambiguation and store-to-load forwarding," in *Proceedings of the 38th Annual International Symposium on Microarchitecture (MICRO-38)*, 2005, pp. 171–182.

[29] R. E. Kessler, "The Alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, 1999.

[30] H. W. Cain and M. H. Lipasti, "Memory ordering: A value-based approach," in *Proceedings of The 31st International Symposium on Computer Architecture (ISCA-31)*, 2004, pp. 90–101.

[31] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *Proceedings of the 1991 International Conference on Parallel Processing*, 1991, pp. 355–364.

[32] G. Reinman and B. Calder, "Predictive techniques for aggressive load speculation," in *Proceedings of the 31st Annual International Symposium on Microarchitecture (MICRO-31)*, 1998, pp. 127–137.

[33] S. Subramaniam and G. H. Loh, "Store vectors for memory dependence prediction and scheduling," in *Proceedings of the 12th Annual International Symposium on High-Performance Computer Architecture (HPCA-12)*, 2006, pp. 64–75.

[34] D. Tarjan, S. Thoziyoor, and N. P. Jouppi, "Cacti 4.0," HP Laboratories Palo Alto, Technical Report HPL-2006-86, June 2006.

[35] The Standard Performance Evaluation Corporation, *Welcome to SPEC*. http://www.specbench.org/.

[36] A. KleinOsowski and D. J. Lilja, "MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research," *Computer Architecture Letters*, vol. 1, pp. 7–10, June 2002.

[37] S. S. Stone, K. M. Woley, K. Malik, M. Agarwal, V. Dhar, and M. I. Frank, "Synchronizing store sets (SSS): Balancing the benefits and risks of inter-thread load speculation," University of Illinois at Urbana-Champaign Center for Reliable and High-Performance Computing, Technical Report CRHC-06-14, Nov. 2006.

[38] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 473–484, Apr. 1992.

[39] A. Iyer and D. Marculescu, "Power efficiency of voltage scaling in multiple clock, multiple voltage cores," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD-2002)*, Nov. 2002, pp. 379–386.

[40] N. Ranganathan and A. K. Murugavel, "A microeconomic model for simultaneous gate sizing and voltage scaling for power optimization," in *Proceedings of the 21st International Conference on Computer Design (ICCD-21)*, Oct. 2003, pp. 276–281.

[41] S. Borkar, "Design challenges of technology scaling," *IEEE Micro*, vol. 19, no. 4, pp. 23–29, 1999.