Efficient Methods for Out-of-Order Load/Store Execution for High-Performance Soft Processors

Henry Wong, Vaughn Betz and Jonathan Rose
Department of Electrical and Computer Engineering, University of Toronto

***Abstract*—As FPGAs continue to increase in size, it becomes increasingly feasible and desirable to build higher performance soft processors. Preserving the familiar single-threaded programming model can be done with an out of order processor. The ability to execute memory loads and stores out of order has a large impact on performance, but this is difficult to do because the dependencies between stores and loads are not known until addresses are computed. Out of order memory disambiguation is traditionally done with CAMs in the load queue and store queue, but large CAMs are inefficient on FPGAs. Store Queue Index Prediction (SQIP) and NoSQ propose to replace CAMs with store-load forwarding prediction and load re-execution.**

We implement four memory disambiguation schemes (in-order, CAM, SQIP, NoSQ) on a Stratix IV FPGA and evaluate the area and delay trade-offs. We find that CAM area and delay degrade quickly with load/store queue size, while SQIP and NoSQ have little degradation with queue size but have area overhead for prediction and predictor training hardware. SQIP and NoSQ use less area than CAMs beyond 32 and 16 load/store queue entries, respectively, and have higher maximum frequency beyond 4 entries.

I. Introduction

Many FPGA systems are comprised of both software and hardware components, and the choice between the two is often driven by performance requirements. In FPGA systems, processing is sometimes done using soft processors, as software requires lower development effort. However, the ability to assign a greater portion of the processing to software is limited by the performance of soft processors, which has been little changed for more than a decade since soft processors have been in use commercially. With FPGA capacity still continuing to grow, it is possible to spend much more resources on a soft processor on higher-performance soft processors.

There have been many proposed soft processor architectures with improved performance, such as using multithreading [1], VLIW processors [2], vector processors [3], and even architectures developed from a first-principles examination of an FPGA's capabilities [4]. However, in order to preserve the familiar single-threaded programming model, superscalar and out of order processors are required, even though they are generally less efficient than specialized architectures.

In out-of-order processors, data dependencies between instructions are determined in hardware. Determining data dependencies through registers is done using register renaming, but determining data dependencies for memory operations ("memory disambiguation") is difficult because memory addresses are not known until they are computed during instruction execution.

Traditionally, memory disambiguation is performed using CAMs (content addressable memory), by searching for stores whose location overlaps with a load, or vice versa. Some recent work has proposed to replace the address-based CAMs with value-based load re-execution, which speculates on whether a load is dependent on a store, then verifies this by re-executing the load when it commits [5]. Since CAMs are particularly expensive on FPGAs [6], re-execution based schemes are attractive for FPGA soft processors. Due to the hardware cost, early out-of-order processors executed arithmetic instructions out-of-order, while still executing memory operations in-order. As processors become more aggressive with larger instruction windows, it becomes more important to be able to execute memory operations out-of-order. We measured 40% improvement in IPC between in-order and out-of-order memory execution in a cycle-level simulation of a two-issue out-of-order processor. Other work has shown even more impressive gains for more aggressive designs [7].

This work examines the suitability for FPGA implementation of four different memory speculation and disambiguation schemes: In-order, Out-of-order using CAMs, Store Queue Index Prediction (SQIP) [8], and NoSQ [9]. Both SQIP and NoSQ are re-execution based and do not use CAMs. We focus on the FPGA resource usage and maximum frequency of these four schemes when implemented on an FPGA, as earlier work has shown that the two re-execution based schemes have similar IPC (instructions per cycle) to using CAMs [8], [9].

II. Background

A. Memory Execution

In an out-of-order processor, memory execution is concerned with performing loads and store operations after the virtual address has already been computed (address generation). Figure 1 shows a typical processor pipeline with the memory execution hardware highlighted. For load instructions, the virtual memory address is translated to a physical address, the L1 cache is accessed, and some memory dependence speculation or checking is performed, depending on the memory disambiguation scheme. Stores work similarly, but data is stored into a store queue rather than directly into the L1 cache.

The memory execution system also has its own scheduler (labeled "Replay"), as a memory operation can fail for a number of reasons (L1 cache miss, TLB miss, load delayed because it depends on a store, etc.) A separate scheduler is used because memory scheduling waits for events such as

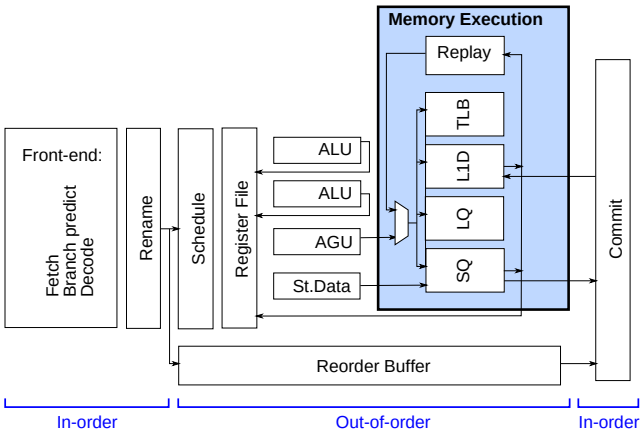

Figure 1. CPU Block Diagram. This work focuses on memory execution.

cache and TLB line fills rather than the availability of register operands.

B. Out-of-Order Memory Execution

An aggressive memory execution scheme performs several functions not found in a basic in-order execution scheme: Store-to-load forwarding, out-of-order load execution, memory dependence speculation, and memory disambiguation.

Executing load instruction out-of-order is desirable as it reduces load latency. Out-of-order execution requires knowing whether a load instruction depends on an earlier store. The most basic form (no speculation) delays load instructions until the load is known to be independent from all earlier store instructions. More aggressive reordering is enabled by **memory dependence speculation**, which speculates whether a load is dependent on some earlier store and only delays those that do. Earlier work has shown that memory dependencies are highly predictable [7]–[10].

Using speculation creates a new requirement of needing to verify the prediction using a **memory disambiguation** scheme. There are two main classes of memory disambiguation schemes: Address-based, and value-based. Address-based schemes compare the addresses of loads and stores to determine whether memory locations overlap. Value-based schemes re-execute (some) loads at instruction commit time and compare whether the non-speculative (correct) load value matches the speculative value loaded earlier. CAM-based disambiguation is address-based, while SQIP and NoSQ are value-based.

All three out-of-order schemes we evaluated perform store-to-load forwarding, out-of-order load execution, memory dependence speculation, and memory disambiguation. The in-order scheme performs none of these.

C. In-order Memory Execution

The basic in-order memory execution scheme executes loads in-order, delaying loads until all earlier stores have committed into cache. Stores are held in a store queue RAM until they are committed. This scheme does not require any CAMs to check dependencies between stores and loads, thus it has a low hardware cost.

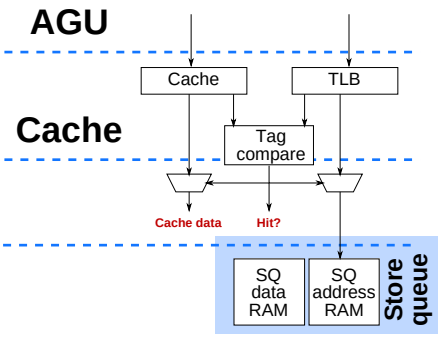

Figure 2. In-order Memory Execution

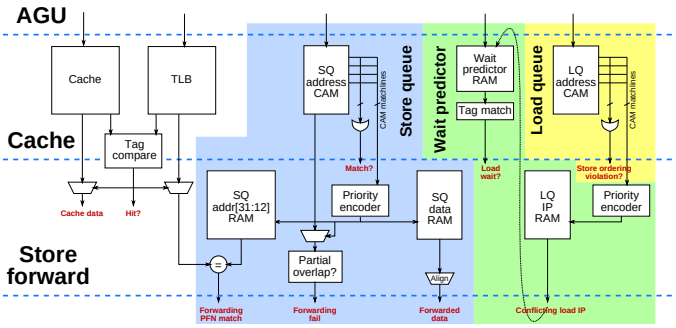

Figure 3. CAM Out-of-Order Memory Execution

Figure 2 shows our implementation of a hardware pipeline needed to implement an in-order memory execution scheme. A virtual addresses is used to look up a TLB for address translation and a virtually-indexed, physically-tagged data cache. Cache data is available the cycle after cache access, for a total load latency of three cycles (AGU, cache/TLB, writeback/bypass).

D. CAM-based Out-of-Order Memory Execution

Figure 3 shows our implementation of CAM-based memory execution. The store queue now becomes a CAM to allow load instructions to search for stores that overlap in memory location. There is also a simple "wait" predictor for memory dependence speculation, and a load queue CAM for memory disambiguation.

When a load executes, the store queue is searched for all *earlier* stores that overlap with the load. There are four possible outcomes of this search: No match, match, ambiguous, or conflict. Ambiguous loads are usually predicted to not conflict, unless the wait predictor says that load has been recently mispredicted. Mispredictions are detected when stores search the load queue CAM for any *later* loads that overlap in location and have already executed (i.e., improperly reordered).

In our pipeline design, the cache, TLB, store queue, wait predictor, and load queue accesses occur in parallel. The total load latency is 3 cycles for cache hits (AGU, cache/TLB, bypass/writeback), and 4 cycles for loads that require store-to-load forwarding.

E. Store Queue Index Prediction

Store Queue Index Prediction (SQIP) replaces the CAMs used for memory disambiguation with a predictor that *predicts*

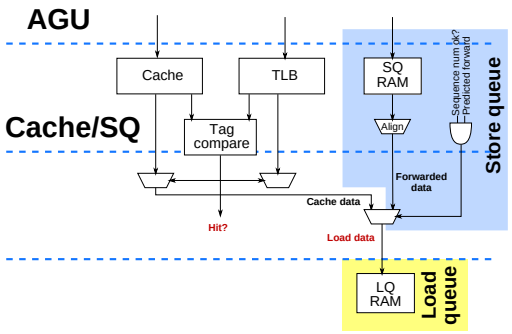

Figure 4. SQIP Memory Execution

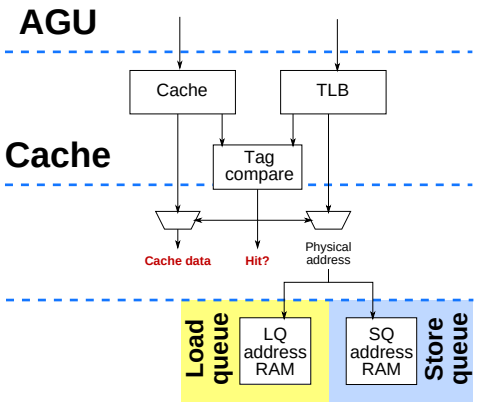

Figure 5. NoSQ Memory Execution

whether a load depends on a store, and also from which entry in the store queue the load should forward, allowing the store queue to be implemented in a RAM. A load that is predicted to not need forwarding reads its data from the cache. Correctness is ensured through in-order re-execution. Loads are first filtered using a store vulnerability window (SVW) filter [11] that allows most loads to skip re-execution.

Figure 4 shows our implementation of the load execution portion of SQIP, shown here without the predictors. We implemented the predictor and predictor training hardware as well, but those are outside the critical memory execution hardware and not shown in this figure.

The SQIP hardware we implemented has a 3-cycle load latency for both cache hits and loads that are forwarded from a store.

F. NoSQ

NoSQ goes further than SQIP by removing the store queue out of the load critical path. This is done by using the memory dependence predictor to *rename* store to load dependencies through the register file (speculative memory bypassing [12]). Instead of taking a data value from a register, storing it to memory, then loading it into another destination register, speculative memory bypassing uses the register renamer to point the final destination register's renamer table entry at the source register of the store, thus bypassing the memory system entirely.

Figure 5 shows the load hardware for NoSQ. Because all of the loads that require forwarding have been bypassed, the load pipeline is nearly identical to the in-order scheme: Only the cache and TLB are accessed. We chose to include a load queue and store queue address RAM to store addresses for use by re-execution, which we believe is lower cost than recomputing addresses for re-execution as originally proposed by NoSQ. Like SQIP, the total load latency is 3 cycles for loads that hit in the cache, but are even faster for loads that are bypassed.

III. Methodology

We performed both cycle-level simulation of the various memory execution schemes and implemented the memory execution hardware and associated predictors on an FPGA.

A. Cycle-Level Simulation

We simulated the memory execution schemes on a cycle-level simulator derived from Bochs [13], a functional full-system x86 emulator. We replaced the CPU simulation with an execute-in-execute cycle-level model of an out-of-order x86 CPU. Since our focus in this work is on the area and delay trade-offs of the various predictors, we omit further details of our simulation setup. We note that our simulations agree with the many IPC results that have already been published in previous work [7]–[9].

B. Hardware

To make area and operating frequency measurements, we implemented the memory execution portion of the processor on a Stratix IV FPGA, using Quartus II 13.0 SP1. Because the cache and TLB are closely coupled with the load and store queues, we also implemented a mock-up of the cache and TLB that includes the caches, read ports, and tag comparison logic, but without the more complicated but less timing critical cache miss handling and page table walking logic. In all cases, we use a 2-way 8 KB cache (in M9K block RAM) with 64-byte cache lines and a 2-way 64-entry TLB (in MLAB LUT RAM). For SQIP and NoSQ, we default to predictor sizes roughly $1/16^{th}$ the size as originally proposed[1], as we found that this gave significant area and frequency improvements with a 1-3% loss in IPC.

We report FPGA resource utilization using the "Logic utilization" metric reported by Quartus, which takes into account how often logic functions of various sizes can be packed into a dual-output fracturable LUT (Stratix IV ALM). We also include the area of used memory blocks, based on Stratix III relative tile areas reported in [6]. Frequency and area results are the average over 20 random seeds.

IV. Results

A. Performance over Varying Queue Sizes

Figures 6 and 7 show the maximum frequency and area of the in-order, CAM-based out-of-order, SQIP, and NoSQ memory execution schemes. We evaluated load queue and store queue sizes from 2 through 128 entries. In the plot of area (Figure 7), the area of the cache and TLB alone is also marked with a dashed horizontal line. Table I gives a breakdown of

[1]SQIP: 256-entry FSP, SPCT, and SSBF, 32-entry SAT; NoSQ: 256-entry predictor, untagged SSBF, 64-entry SRQ

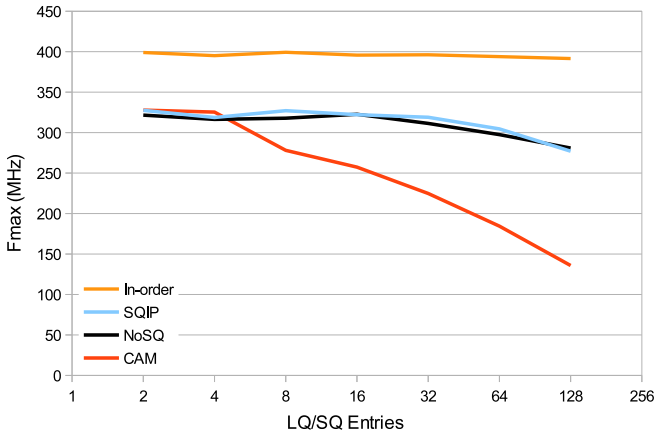

Figure 6. Maximum Frequency for varying LQ and SQ size

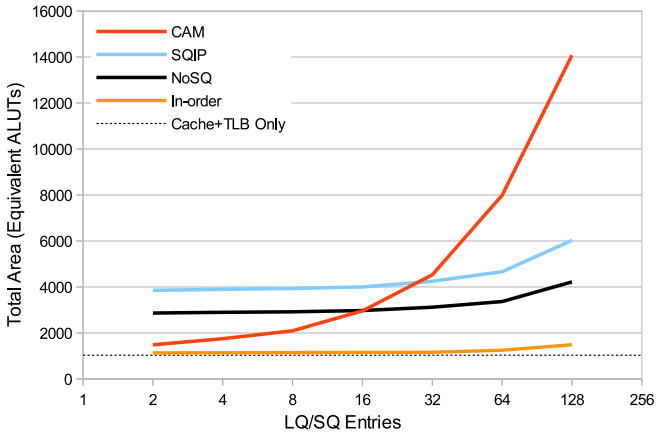

Figure 7. Total area for varying LQ and SQ size

FPGA resource usage for the four schemes at 64 load queue and store queue entries.

a) In-order: This scheme has a low hardware cost. Its operating frequency of around 400 MHz is limited by the delay through the TLB for the range of queue sizes we evaluated. Area usage does not noticeably grow until additional MLABs are needed for store queues beyond 64 entries.

b) CAM: Not surprisingly, the CAM-based load/store queues are sensitive to the number of queue (and CAM) entries. The delay increases slightly quicker than logarithmic in the number of entries, while area use grows linearly with the number of CAM entries.

c) SQIP and NoSQ: SQIP and NoSQ behave similarly with varying queue size. The critical path is through various sequence number and tag comparisons, which grow slowly with increasing queue size. The predictor tables and associated logic to do prediction, re-execution, and predictor training adds a considerable amount of area independent of queue length, thus NoSQ and SQIP have greater area usage than the CAM-based scheme below 16 and 32 entries, respectively.

V. Conclusions

As single-threaded soft processors increase in performance and complexity, there will be a need for out-of-order execution of memory operations and memory dependence speculation.

Scheme	Logic Utilization (ALUTs)	M9K	Total Area (Equiv. ALUTs)	f_{max} (MHz)	μPC
Cache and TLB only	570	8	1029	395	–
In-order	789	8	1248	394	0.86
CAM	7529	8	7988	184	1.21
SQIP	3746	16	4664	305	1.13
NoSQ	2678	12	3367	298	1.19

Table I
AREA AND μPC FOR OUR DEFAULT CONFIGURATION: 256 PREDICTOR ENTRIES, 64 LQ AND SQ ENTRIES.

However, the traditional method of using load queue and store queue CAMs is particularly inefficient on FPGAs. In this work, we evaluated four memory execution schemes on a Stratix IV FPGA: In-order, CAM-based out-of-order, and two schemes that do not use CAMs: SQIP and NoSQ. Previous work has shown, and we have confirmed, that in-order execution of memory operations significantly reduces IPC.

We find that the area of the CAM-based scheme grows quickly with the number of load queue and store queue entries, while SQIP and NoSQ have a greater fixed area overhead independent of queue size, with NoSQ being more efficient. In our implementations, SQIP and NoSQ are more area efficient than using CAMs beyond 32 and 16 entries, respectively. We observe similar trends in maximum frequency, where the CAM-based frequency decreases more than SQIP or NoSQ with increasing queue size, with a break-even point at around 4 queue entries. For high-performance soft processors, in-order memory execution is unattractive because even if the rest of the processor could be designed to run at 400 MHz, the higher clock frequency does not make up for the loss in IPC (21% vs 40% for 64-entry LQ/SQ).

Finally, this work demonstrated practical methods to build aggressive out-of-order memory execution hardware on FPGAs, a key piece of a high-performance FPGA soft processor.

References

[1] M. Labrecque and J. Steffan, "Improving pipelined soft processors with multithreading," in *Proc. FPL*, 2007, pp. 210–215.
[2] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster, "An FPGA-based VLIW processor with custom hardware execution," in *Proc. FPGA*, 2005, pp. 107–117.
[3] A. Severance and G. Lemieux, "VENICE: A compact vector processor for FPGA applications," in *Proc. FPT*, 2012, pp. 261–268.
[4] C. E. LaForest and J. G. Steffan, "Octavo: an FPGA-centric processor family," in *Proc. FPGA*, 2012, pp. 219–228.
[5] H. W. Cain, "Memory ordering: A value-based approach," in *Proc. ISCA*, 2004, pp. 90–101.
[6] H. Wong, V. Betz, and J. Rose, "Comparing FPGA vs. Custom CMOS and the Impact on Processor Microarchitecture," in *Proc. FPGA*, 2011, pp. 5–14.
[7] A. Moshovos and G. Sohi, "Memory dependence speculation tradeoffs in centralized, continuous-window superscalar processors," in *Proc. HPCA*, 2000, pp. 301–312.
[8] T. Sha, M. M. K. Martin, and A. Roth, "Scalable store-load forwarding via store queue index prediction," in *Proc. Micro*, 2005, pp. 159–170.
[9] ——, "NoSQ: Store-load communication without a store queue," in *Proc. Micro*, 2006, pp. 285–296.
[10] A. Moshovos, "Dynamic speculation and synchronization of data dependencies," pp. 181–193, 1997.
[11] A. Roth, "Store vulnerability window (SVW): A filter and potential replacement for load re-execution," *JILP*, vol. 8, 2006.
[12] A. Moshovos and G. S. Sohi, "Speculative memory cloaking and bypassing," *Int. J. Parallel Program.*, vol. 27, no. 6, Dec. 1999.
[13] K. P. Lawton, "Bochs: A portable pc emulator for unix/x," *Linux J.*, vol. 1996, no. 29es, Sep. 1996.