A Position-Insensitive Finished Store Buffer

Erika Gunadi and Mikko H. Lipasti
Electrical and Computer Engineering Department
University of Wisconsin - Madison
{egunadi, mikko}@ece.wisc.edu

Abstract

This paper presents the Finished Store Buffer (or FSB), an alternative and position-insensitive approach for building a scalable store buffer for an out-of-order processor. Exploiting the fact that only a small portion of in-flight stores are done executing (i.e. finished) and waiting for retirement, we are able to build a much smaller and more scalable store buffer. Our study shows that we only need at most half of the number of entries in a conventional store queue if we buffer only the stores that have finished execution. Entries in the store buffer are allocated at issue and disallocated on retirement. A clever encoder circuit is used to provide positional searches without an explicitly positional queue structure. While reducing the access latency and power consumption significantly, our technique has virtually no detrimental effect on per-cycle performance (IPC).

1. Introduction and Motivation

In a modern out-of-order microprocessor, load-store queues are structures designed to keep track all the in-flight memory instructions. The load-store queues are usually composed into a pair of age-ordered queues, one for loads and one for stores. Those queues served three purposes: maintaining the order of in-flight memory instructions, forwarding stores data to later loads, and detecting memory ordering violations. To facilitate the last two purposes, load-store queues are equipped with associative searching capability to determine matching loads or matching stores. On execution, loads search the store queue to get data from older stores while stores search the load queue to find younger matching loads that have executed prematurely.

As microprocessor designs get wider and deeper and contain larger out-of-order windows, more memory instructions are in flight in the out-of-order window. Thus, larger queues are needed to keep all the in-flight loads and stores. Wider issue also requires the structure to have more forwarding logic to facilitate a larger number of loads or stores to be executed. Unfortunately, increasing the size of the queues and the number of ports could significantly impacts the access time and power consumption of the structure. The access time of the store queue is particularly critical because it is a component of the load-to-use latency. It is important to keep the store queue access time no larger than the data cache access time as store-to-load forwarding will require replay in speculatively scheduled machines otherwise.

There have been many papers on scalable store and load queues. Many prior techniques rely on program characteristic behavior of memory instructions. The most commonly exploited one is the predictability of load-store forwarding behavior. A forwarding store is likely to keep forwarding while a forwarded load is likely to keep getting data from the forwarding logic. This characteristic is often used to reduce the size or the number of ports of the CAM structure through a combination of filtering [1][2][5], caching [3][4][5], or segmentation [1].The latest work even exploits this characteristic to remove the entire store queue [8][9]. This prediction method is commonly paired with load re-execution [11] as a validation method.

As most of the prior work relies on predicting program characteristic behavior, such schemes could possibly fail in pathologically misbehaving cases, resulting in significant impact in performance. Most of the prior work also relies on load re-execution for validation. Though adding a pipeline stage for re-executing loads prior to retirement stage has minimal performance impact, it adds complexity to the already complex datapath. Additional structures needed for the predicting and bookkeeping purpose also add complexity, increase area, and consume additional power.

In this paper, we propose an alternative way to build a store queue. This technique will be orthogonal or complementary to some of the prior work [1][2][3][4] as described in Section 6. Although we are unable to eliminate the forwarding logic or even the whole store queue as some of the prior work proposed [6][7][8][9], we shows in Section 5 that our technique generally results in less performance impact. Since we do not need any structures for bookkeeping purposes, our design is also simpler and easier to implement.

Our work is the closest in nature with the work of Sethumadhavan [10], which also used un-ordered load-store queues, with significant differences in overflow handling policy and multiple-store forwarding capability. Late allocation itself has also been applied to other microprocessor structures such as the physical register file as proposed in [18].

We exploit the fact that only a portion of in-flight stores are executed and waiting to retire and that only these stores need to be searched for forwarding purposes. Thus, our store buffer (no longer a queue) can be much smaller than the conventional one, leading to less power consumption and better access time latency. One issue with this solution is that the design no longer provides the position-based ordering available in conventional store queue, making it harder to do priority

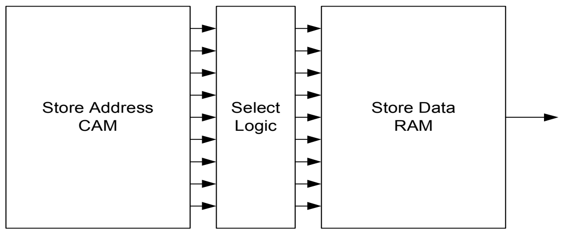


Figure 1. Store Queue Block Diagram.

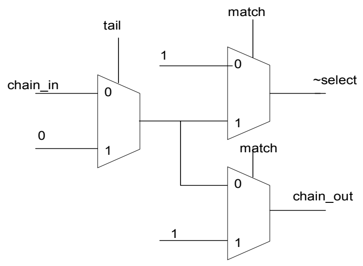


Figure 2. One Passgate Block of Circular Priority Encoder.

encoding, allocation, deallocation, flush, and commit. We have this problem in common with prior efforts that change the store queue structure from age-indexed into address-indexed. To solve this issue, we use a circuit, presented in Section 3 that enables efficient position-based searches even when the entries in the buffer are not stored in program order. We also show that our technique is scalable as the design gets wider and contains more in-flight instructions. Our technique could also be applied to the load queue. However, in this paper we only focus on the application to store queue design as store queue access latency is much more important than load queue access latency.

The rest of the paper is organized as follows. Section 2 explains conventional store queue design with its scalability challenges. Our proposed store buffer design is presented in Section 3. Methodology and modeling details are described in Section 4. Section 5 presents and discusses the results. Prior work is described in Section 6 and Section 7 concludes the paper.

2. .Understanding the Store Queue

In this section, store queue design is described in detail to understand why the store queue is not scalable in term of access latency and power consumption.

A store queue serves two main purposes: to maintain the order of in-flight stores and to forward store data to later loads. It is commonly designed as a circular buffer with entries allocated on dispatch and deallocated on retirement. As shown in Figure 1, its forwarding logic consists of a CAM structure and some select logic to pick the youngest older matching store to forward from.

Each in-flight store is identified using a tag, commonly referred to as store color. It is a sequence number assigned on dispatch of each store. The store color is also used to maintain the relative ordering between store and

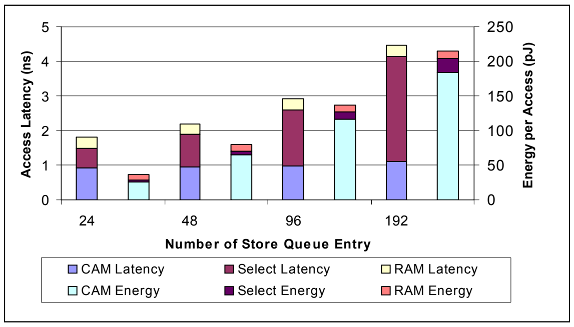


Figure 3. Store Queue Access Latency and Energy per Access Comparison.

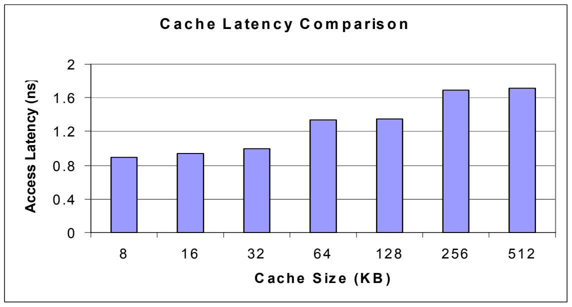


Figure 4. Cache Access Latency Comparison.
Caches: 64B block, 4-way associativity, and 1 RW port.

load instructions. The current store color is assigned to loads as they enter the instruction window.

Each load instruction needs to search the store queue for any matching older stores as it gets executed. If it is found, the load uses the forwarded data rather than the data fetched from the cache.The time it needs to find a forwarding store, if any, is time critical as it is part of the load-to-use latency. A shorter latency means that a load can wake up its dependents sooner, which results in better performance, especially if there are many cases where mispredicted branch instructions are depending on load results.A longer latency also means that more penalty is imposed due to speculative load misscheduling, which also results in poorer performance.

One might argue that the cache latency should be the critical path rather than the load forwarding path as the cache is a larger structure separate from the out-of-order core area. It is important to note that though larger, cache has much lower associativity compared to the store queue, thus is more scalable. One example to show that store-to-load forwarding path is likely to be the critical path is the IBM Power 4 [21]. In the IBM Power 4, store-to-load forwarding takes longer than cache access, resulting in load misscheduling when a matching store is found in the store queue. This problem gets worse as the store queue gets bigger to accommodate more in-flight stores in a wider microprocessor, since CAMs are widely known to scale poorly. Similarly, power consumption will also increase proportionally to the number of entries in the store queue becauseeach CAM entry needs to perform a match on every access.

Shown in Figure 1, the latency component of a store

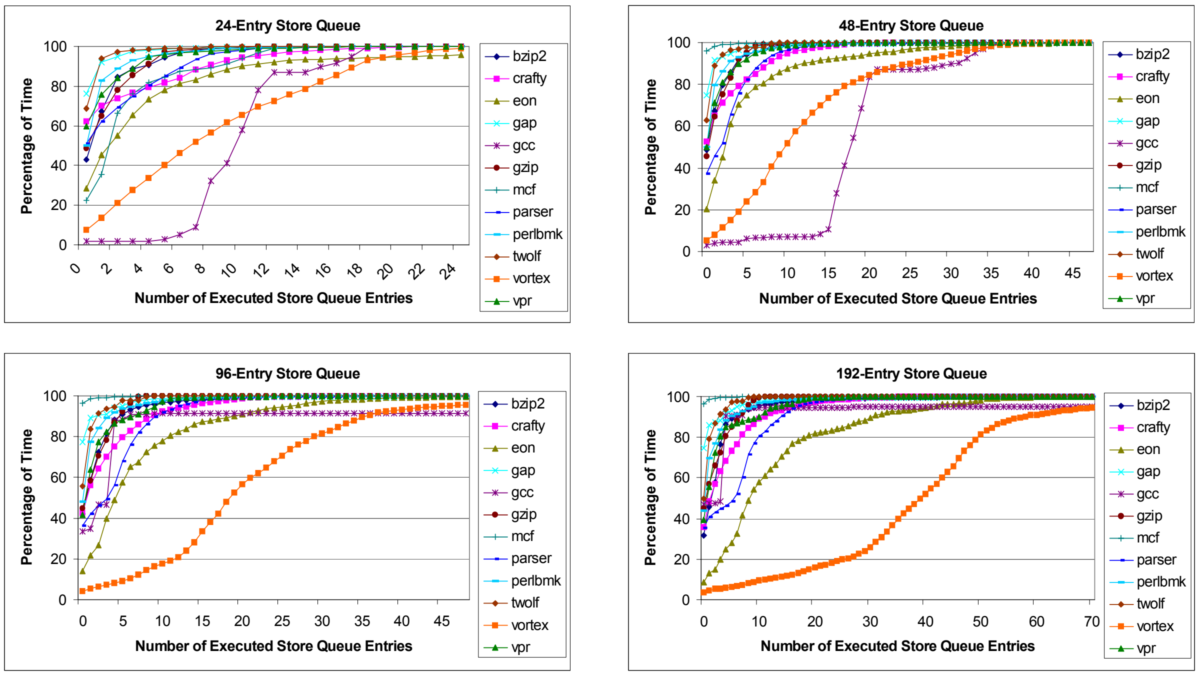


Figure 5. Store Queue Occupancy Study on Different Configurations.

queue access consists of address CAM, select logic, and RAM data read. There is also an age masking to invalidate younger stores that is done in parallel with the address CAMing but is not critical. As the store queue is usually implemented as a circular buffer with a head and a tail, a circular priority encoder is needed as the select logic. We implement the select logic using circuitry patented by Intel [16]. This circular priority encoder is used as it is simple yet fast; each level only has a depth of two muxes. This circuitry is not the only way to implement a circular priority encoder, a possible alternative design is a hierarchical priority encoder as described in [13] with some modification. However one can easily see that [16] is simpler and faster.

Naively, this circular encoder consists of a chain of passgate blocks connected one after another in a circular manner. A tail signal informing the passgate block that it is supposed to be the start of the circular chain and a matching signal are provided to each block. The block will then computes the select signal and the chain_out signal to the next block. As the select signal is computed serially from one block to another, one can envision that the latency will not be scalable as the number of entries in the store queue increases. To make the latency better, a hierarchical technique is applied so that the chain_out signal is computed in a group of four. Though helpful, it still does not help the fact that the select logic latency will increase in a linear or logarithmic manner. Figure 2 shows the logic of one passgate block.

Figure 3 shows the scaling of access latency and power consumption of store queues with different size. CAM and RAM data is taken from CACTI [20] while the select logic is synthesized using a standard-cell design flow. Both CACTI and synthesis are using 110 nm technology. Methodology details are explained in Section 4.

As shown in Figure 3, both access latency and energy per access increases linearly as the number of entries goes up. The increase in access latency is mostly dominated by the select logic. Contrary to popular belief, the CAM latency does not increase proportionally to the the number of entry. The reason is the sub-banking/subarray technique that has been applied to the memory array structure. By dividing the CAM array into several physical structure, the length of bitline needs to be travelled in the structure remains approximately constant.

Although the sub-banking technique prevents the access latency from soaring with the increase in the number of entries, it does not prevent the increase in energy consumption by the CAM array as each entry still has to perform data matching. The energy per access increases in proportion to the number of entries.Thus, a feasible store queue will be achieved only if we can keep the structure to a reasonable number of entries.

For comparison purpose, we also show cache access latency for different cache sizes in Figure 4. All caches have 64B block and 4-way set associative. Cache access latency scales much better compared to the store queue access latency as the subbanking/subarray approach can be applied to the whole cache. This means that without a good solution, store-to-load forwarding path will likely be a cycle-time bottleneck in the future.

3. Finished Store Buffer (FSB)

Our Finished Store Buffer is based on the simple hypothesis that only a fraction of stores are done executing and ready to retire at any given time and that a load instruction only needs to search these finished stores for any forwarded data.To verify our hypothesis, we do some occupancy studies as described in Section 3.1. We then explain our design in Section 3.2.

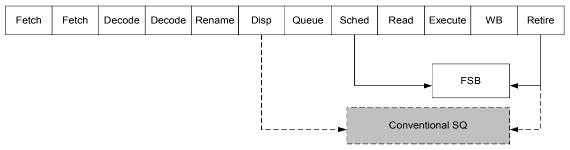


Figure 6. Pipeline Diagram with SQ and FSB allocation. Allocation and release for conventional store queue is shown in dotted line and shadowed box.

3.1. Initial Study

Figure 5 shows the results of our occupancy study on various store queue sizes. The base configuration is a 24-entry store queue in a 4-wide, 12-stage pipeline with 128-entry ROB as shown in Table 1. A 24-entry store queue is chosen for a 128-entry instruction window as store instructions represent roughly 20% of the in-flight instructions; this choice was validated with a sensitivity study resulting in less than 2% IPC slowdown compared to an unlimited store queue. Assuming that microprocessors will keeps getting wider and the number of ROB entries will keep increasing in the future, we also simulate a 48-entry store queue in a 8-wide machine with 256 ROB, a 96-entry store queue in a 16-wide machine with 512 ROB, and a 192-entry store queue in a 32-wide machine with 1024 ROB. Across different configurations, we assume the same pipeline depth.

Figure 5 shows that the percentage of time when more than half of in-flight stores are waiting to retire is close to 0%. In fact, at all times there are at most 12 stores that are done executing and ready to retire in a 24-entry store queue. Although the number of store instructions in flight increases in proportion to the size of the instruction window, the number of finished store instructions does not seem to scale proportionally. In a 48-entry store queue, 20 seems to be the upper limit of the number of finished stores in any given time. While in a 96-entry and a 192-entry store queue, 32 and 52 seems to be enough capacity for the finished store instructions.

In brief, Figure 5 confirms our hypothesis that a smaller store buffer is potentially possible to build. A smaller store buffer should result in lower access latency and less dynamic energy per access. Also, a smaller structure will result in less leakage power.

3.2. Details of the Design

A store queue serves two main purpose, which are to keep track of the age ordering of the in-flight stores and to forward data to load instructions as necessary. As mentioned previously, the forwarding logic only needs to keep track of the finished stores waiting to retire.The number of finished stores, as shown in Figure 5, is less than half of the number of store queue entry. Thus, the store queue can be built much smaller than it is right now. As for the other purpose of the store queue (keeping track of stores in order), there is another structure which can serve that purpose: the ROB.

In contrast to a conventional store queue, a store does not allocate an entry in the FSB at dispatch.The store simply allocates an entry in the ROB. An FSB will be

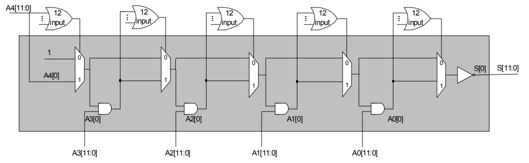


Figure 7. Basic Algorithm for Youngest Select Logic. The above assumes 12-entry FSB with 32-entry maximum of in-flight stores (5-bit store color). The block in shadow is repeated 12 times.

cation will be done in a similar manner as in the scheduler. A store is simply inserted in the next available entry. A store color is still assigned to the store to identify each store in the window. The store color ID can be kept in payload RAM temporarily before it is inserted into the store buffer.

If a store is ready to issue and there is an available entry in the FSB, an entry is allocated for the issued store. The allocated entry will remain allocated until the store is retired from the window. Figure 6 illustrates the difference between the conventional store queue and proposed FSB allocation policy. If the FSB is full, store issue is simply disallowed by the scheduler. Our store queue is now viewed by the scheduler as one of the resources that it needs to keep track of for scheduling purposes. To avoid a deadlock situation that can happen when there is no empty slot in the FSB while the store to be issued is the oldest store in the window, an empty slot is reserved in the store queue and can only be used if the issued-to-be store is the oldest store in-flight. Identifying the oldest store in-flight can be done using the store color in the same way as SVW [5].

The current store color is also assigned to every branch instruction. On branch misprediction, the store color of the mispredicted branch will be used to determine which entries in the FSB need to be invalidated. To keep the in-order store retire easy, the FSB index allocated for the store instruction is kept in the ROB entry for that store. As the store is ready to retire, the FSB index is used to get the data and the address from the FSB.

Issued loads access the FSB to see if there is an aliased store in the same way as in a conventional store queue. Similarly, a premature load will not find any matching store in the FSB and will be detected when store executes and searches the load queue.

One major difference between our FSB and the conventional store queue is the fact that the allocation algorithm is not based on the age of store instructions. Instead it is allocated based on available slot and first-ready stores. Thus, an entry's position does not reflect its relative age anymore. Nonetheless, our FSB still has to pick the youngest among older matching store without relying on the stores position in the FSB.

Fortunately, picking oldest/youngest instruction is not a completely new issue. It has always been a problem to solve in instruction schedulers. Solutions to this problem that don't rely on position have been presented in [15] by Intel and in [14] by Buyuktosunoglu. [15] uses a specific age encoding technique, grouping, and global OR-ing across different groups to determine the oldest entries. Though originally proposed as a pseudo-FIFO

Table 1. Machine Configurations.

Out-of-order Execution	4-wide fetch/issue/commit, 128 ROB, 96 PRF, 32LQ, 24SQ, 32-entry scheduler, 12-stage pipeline, fetch stop at first taken branch in a cycle
Branch Predictions	Combined bimodal (16k entry) / gshare (16k entry) with a selector (16k), 16-entry RAS, 4-way 1k-entry BTB
Functional Units	2 integer ALU (1-cycle), 1integer mult/div (3/20-cycle), 1general memory ports (1+2 cycle)
Memory System	L1 I-Cache: 64KB, DM, 64B (2-cycle) L1 D-Cache: 64KB, 4-way, 64B (3-cycle) L2 Unified: 2MB, 8-way, 128B (8-cycle) Off-chip memory: 150-cycle latency
Store-to-Load	Same as L1 D-Cache latency (2-cycle)

algorithm, [15] can be easily modified to be an exact FIFO algorithm by removing the grouping of entries. Different from [15], [14] does not require a special encoding technique and uses a slightly more complex logic to determine the oldest entry. We choose a solution similar to [14] for our FSB as it uses fewer bits, and is thus more power-friendly.

Figure 7 illustrates the basic algorithm for the youngest select logic [14] used in our FSB. Here, 00000 is considered the oldest while 11111 is considered the youngest.The algorithm walks from the most significant bits to the least significant bits. For each significant bit, it checks whether the current significant bit of all entries is able to pass and add information from the previous significant bit. If so, AND the result from previous-bit with the current one to invalidates older stores in current bit position. If not, just pass along the previous result. To handle a wrap-around store color, one more bit is added to the store color and a simple reverse logic is added to the most significant bits of the select logic circuitry.

One problem with the algorithm above is that it performs the checks in a serial manner for each significant bit, ORing as many bits as the number of FSB entries for as many of the number of bits that are used in the store color. Thus, the delay could be quite bad. however, the algorithm can be easily restructured hierarchically, so that the checking part happens in parallel for each group of four, leaving only the MUXing and the ANDing to be done serially.

4. Methodology

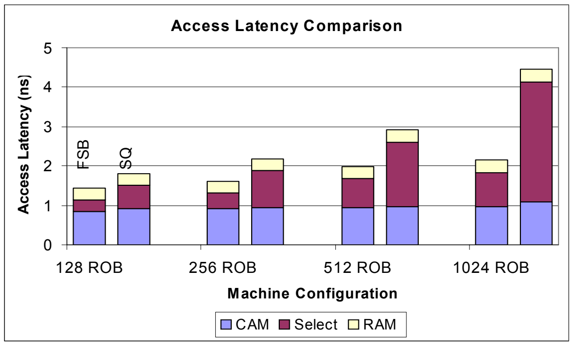
For our microarchitectural study, we use a modified *Simplescalar / Alpha* 3.0 tool set [17], a suite of functional and timing simulation tools for the Alpha AXP ISA. Specifically, we extended sim-outorder to perform full speculative scheduling. In this pipeline, instructions are scheduled in the scheduling stage, assuming instructions have constant execution latency and any latency changes (e.g. cache misses or store aliasing) cause all dependent instructions to be re-scheduled. Our simulator also models aggressive load-store reordering with store set predictor [22]. The machine configuration used for the baseline is shown in Table 1.

Table 2. IPC for Benchmark Programs Simulated.

Benchmark	128-ROB	256-ROB	512-ROB	1024-ROB
bzip2	1.690	2.467	3.124	3.318
crafty	1.595	2.712	3.660	4.131
eon	1.369	2.122	2.580	2.755
gap	1.339	1.809	2.119	2.195
gcc	1.444	2.559	4.087	5.534
gzip	1.392	1.867	2.260	2.354
mcf	0.133	0.140	0.143	0.146
parser	1.358	1.820	2.162	2.307
perlbmk	1.322	2.001	2.558	2.668
twolf	1.255	1.637	1.798	1.930
vortex	1.684	2.888	3.988	4.462
vpr	1.238	1.624	1.830	1.960
ammp	1.069	1.485	1.891	2.227
applu	1.227	2.095	3.462	4.923
apsi	1.434	2.565	4.256	6.287
art	0.854	1.472	2.289	3.053
equake	0.569	0.896	1.432	2.342
facerec	1.489	2.126	2.762	3.731
fma3d	1.325	2.173	3.214	4.554
galgel	1.218	1.425	1.591	1.694
lucas	0.918	1.177	1.605	2.584
mesa	1.888	3.243	5.095	5.959
mgrid	1.688	3.320	5.477	8.405
sixtrack	2.007	3.550	4.218	4.768
swim	2.126	3.277	4.776	7.116
wupwise	1.915	3.278	5.188	6.832

More aggressive configurations to estimate FSB in future machines are also studied. They include 8-wide with 256 ROB, 16-wide with 512 ROB, and 32-wide with 1024 ROB. All resources such as SQ, LQ, scheduler, PRF, functional units, branch predictor, and caches are simply scaled proportionally. This means that the number of entries in the conventional store queue are 24, 48, 96, and 192 for each corresponding machine configuration. For the FSB, the number of entries are 12, 20, 32, and 52 respectively. These numbers are taken from occupancy study described in Section 3.1. For simplicity, we do not change latencies or pipeline depth as the machine gets more aggresive. We compare our results to the SQIP approach (described further in the prior work section), and use the same SQ configuration as in [6] for 512 window, which are 4K-entry 2-way FSQ, 4K-entry 2-way DDP, 256 entry SAT, 2K-entry SSBF, and 2K-entry SPCT. It is scaled accordingly for larger and smaller windows.

The SPEC2000 integer and floating point benchmark suites are used for the results presented in Section 5. All benchmarks were compiled with the DEC C and C++ compilers under the OSF/1 V4.0 operating system [23] using -O4 optimization. Reference input sets and SMARTS statistical sampling methodology were used



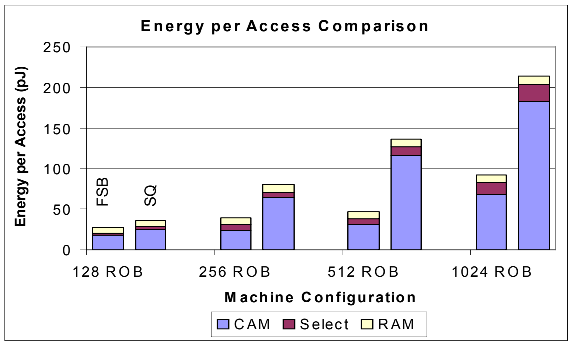


Figure 8. Access Latency and Energy Per Access Comparison between Conventional SQ and FSB.

for all benchmarks.The baseline IPC for each benchmark program simulated on different machine configurations are shown in Table 2.

For accurate modeling of timing and power, the select logic for both the conventional and the proposed store queue designs are implemented in Verilog and synthesized using Synopsis Design Compiler and LSI Logic's gflxp 0.11 micron CMOS standard cell library. Due to tool inavailability, designs have not been placed and routed. The synthesis tool does include an estimate of the wire delay using fanout-based wire load models as the tool does not yet know the exact wire lengths and the capacitive loads. According to [19], synthesis reasonably estimates the wire load of the multitude of short wires and is quite accurate. We also explored wire delay calculation using distributed RC model as described in [24]. However as the wire latency for each structure are all less than 0.1 ns, we think that it is not important. Latency for CAM and RAM structure are computed using CACTI [20].

5. Experimental Results

As seen from Figure 8, the latency and the energy for our technique is more scalable than the conventional store queue. Due to fewer entries, the latency of the select logic is much less than the circular priority encoder of the conventional store queue. Fewer entries also helps to reduce the energy of the CAM structure as less entries are performing comparisons.

Figure 9 compares the normalized IPC number of SQIP, FSB, and FSBmin. Each number is normalized to the IPC of each baseline configuration shown in Table 2. FSB uses store queue number of entries described in Section 3, which are 12, 20, 32, and 52 for 128, 256, 512, and 1024 ROB respectively. In FSBmin we use our scheme to the most aggressive limit. The premise here is that we only need enough entries so that the retirement stage is not stalled by unfinished stores. Assuming that stores represent 20% of the total instructions, the upper limit would be 20%*width-of-machine*issue-to-retire-stages. It results at 5, 10, 20, and 40 for each instruction window. To get a fair IPC comparison, we use the same 2-cycle store queue latency for all configurations. One can envision that our technique could easily take advantage of the store queue latency difference from Figure 8 to get a better IPC or higher frequency.

The SQIP technique is chosen since its performance should be quite a representative of the latest prior work such as SVW [5], FnF [8], and NoSQ [9], as all of them are using similar store-set style predictor and a filtered load replay recovery. As shown, SQIP works quite well with less than 5% slowdown for most benchmarks despite the additional three pipeline stages for load commit. However, there are some benchmarks where the predictor does not work very well such as vortex that has 15%-20% slowdown across different configurations.

As seen, our FSB technique performance impact is less than 1% for most of the benchmark. Nonetheless we experience around 5% slowdown on sixtrack with 1024-ROB. This is possibly due to retirement stall of unfinished stores. By increasing the number of reserved entries into two rather than one, the slowdown can be reduced into less than 1%. Figure 9 also shows that even a minimal number of store entries (FSBmin) has minimal impact on IPC.

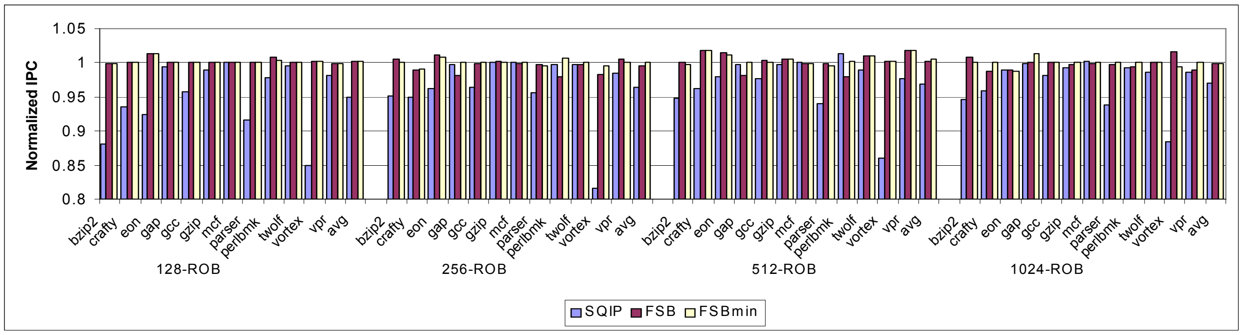
There are cases where FSB slightly outperforms the baseline IPC. The reason is because in FSB, the store queue size does not limit instructions dispatched to the out-of-order window. Thus, it is possible that a higher ILP is achieved during periods when the pressure on the store queue is quite high. There are also cases where FSBmin outperforms FSB. It is probably caused by the different ordering of instruction executions resulting in different wrong path instructions being executed.

6. Prior Work

Franklin [12] proposes ARB, an address-indexed memory disambiguation hardware used in Multiscalar. Loads and stores would index to this structure where age tags were stored to assist in forwarding values and detecting ordering violation.

Sethumadhavan [1] proposed to use a bloom filter to reduce the number of associative search in the store queue. Stores update the bloom filter upon entering and leaving the window. Loads only access the store queue on match. As the bloom filter only results in false positive, no correcting technique is needed.This idea is extended further to build a banked store queue. The bloom filter is partitioned into multiple smaller ones that are accessed simultaneously by load instructions. Loads only needs to associatively search the corresponding bank where the bloom filter partition hits.

A similar idea with a different filtering structure is proposed by Park [2]. A store set predictor is used to pre-



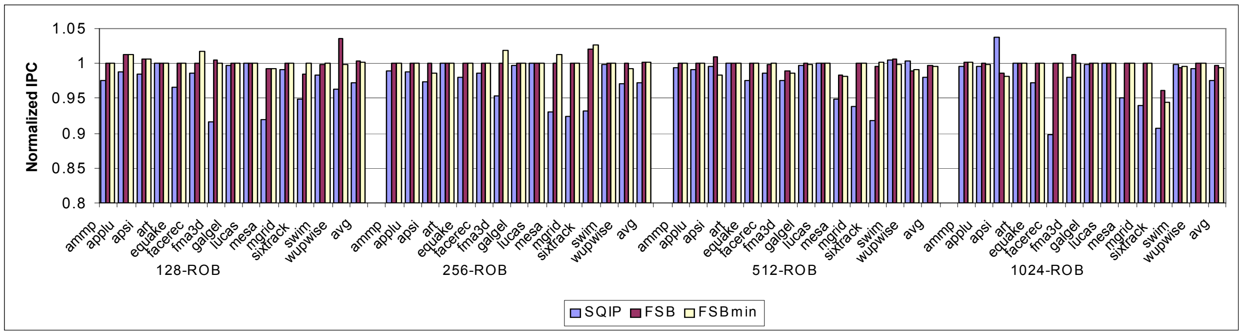


Figure 9. Normalized IPC Comparison Conventional SQ, FSB, and FSBmin. FSB scheme uses 12, 20, 32, and 52 store queue entries for each machine configuration, while FSBmin uses 5, 10, 20, and 40 store queue entries.

dict if a load is a forwarding one. Only predicted forwarding loads need to access the store queue. Thus, the number of search is reduced and less ports are needed. As it is speculative, load queue search is moved to the retirement stage rather than execution stage to catch the mispredicted loads. Both [1] and [2] are orthogonal to our technique and can be combined to reduce store queue ports and the number of access. In fact our FSB provides a nice solution to the banked store queue proposed in [1] to reduce the number of store queue ports.

Work by [3] decomposed store queue by functionality, which are store forwarding and correctness enforcement. To determine which store is in the forwarding group, this work proposes an extension to the ISA to add a prediction bit in the instruction or dynamically storing it with the instruction in the I-cache. The prediction bit is then set on the first encountered misprediction. Once the predictor is in placed, the load-store forwarding is much like a traditional store queue, only using a smaller buffer named forwarding store buffer. Memory validation queue is used to ensure correctness due to load-store ordering violations, consistency violations, and mispredictions. This structure is banked by address to reduce power. Again, our FSB technique can be used hand-in-hand with this technique to further reduce the size of the forwarding store queue, hence improving its latency and power. Our technique also provides a good solution to all the complication raised in this work regarding the banked-by-address memory validation queue.

A two-level store queue is proposed by [4]. Using a forwarding predictor, only predicted forwarding stores are put in the first level. The first level is allocated on execute stage and is removed using LRU policy, while the second level keeps all stores from dispatch to commit. The main idea is to keep the access time for the first level to be the same as data cache hit and let second level do the validation work at its own speed. Similarly, we can combine this with our work to make the first level store queue even smaller and faster, and the second level store queue to be much smaller than the conventional one by only putting the executed-ony stores.

SVW [5] proposed a store queue optimization by separating forwarding and non-forwarding stores. It is done by breaking store queue functionality into RSQ, FSQ, and FSC. SSBF, SPCT, and FSQ are structures needed for the prediction and correction mechanism. Only forwarding stores need to be placed in the FSQ and only forwarding loads need to search it. All correcting mechanism is done by re-executing loads prior to commit stage. This work also proposed the SVW technique to filter load re-execution.

A more aggressive approach named SQIP is proposed by [6] by removing the associative-search capability from store queue. Instead, a store-set dependence predictor is extended to predict the index of a forwarding store queue entry. A forwarding mis-prediction is detected using a pre-commit re-execution, which results in a pipeline flush. Along similar lines, AIMB [7] proposed to replace the store forwarding and violations checking of load-store queue using three structures: a store forwarding cache, a memory disambiguation table, and a store FIFO. The store forwarding cache is address indexed and used to forward data. The memory disambiguation table is also an address indexed structure to keep track the latest executed loads and stores. The store FIFO is a buffer to keep the ordering of stores. A modification of store-set predictor is used to predict whether forwarding or violations would occur.

Both FnF [8] and NoSQ [9] utilized store-set predictors but in a different way. FnF used the predictor to predict the index of the forwarded load in the load queue if any. Stores access the prediction and put the data in the load queue as necessary. NoSQ used the store-set predictor to predict if the load is a bypassed load. In that case, the load will get its data from the predicted register.

SVW, SQIP, FnF, and NoSQ are not combineable with our work. These works are similar in a way that they exploited the same program characteristic, used a similar store-set prediction mechanism, and re-execution validation method. As seen from the result in their paper, those techniques did not work consistently for all benchmarks. Since they are based on predicting certain characteristics, there are some benchmarks that those techniques fail to predict. Those works are also quite complex as they need many structures for bookkeeping and prediction. They also make the datapath more complex by the need to have an access path from the ROB to the cache for load re-execution.

The latest store queue optimization work is ULB-LSQ [10]. Similar to ours, ULB-LSQ proposes an unordered store queue that is allocated at issue time, hence enabling much smaller store queues. The main differences between [10] and our technique are the overflow handling policy, banking approach, and multiple-store forwarding policy. [10] uses flush to handle overflow in a small window and three techniques, instruction replay, skid buffers, and micronet virtual channels, to handle overflow in large window microprocessor. Since their technique does not have select logic, they also do not handle forwarding from multiple stores.

7. Conclusion

We describe the Finished Store Buffer (FSB) as an alternative way to build the store queue. A conventional store queue is built to contain all stores in flight in the window, and scales poorly as machines are getting more aggressive with wider pipeline and larger instruction windows. Instead of putting all stores in the store queue, thus requiring a large queue, we propose to only put stores that have finished executing and are ready to retire, which are only a fraction of the in-flight stores. It is built on observation that loads only need to search executed stores and that only a small portion of in-flight stores have been executed.

Our study shows that we can build a much smaller store buffer, with lower access latency and less power consumption with less than 1% slowdown on IPC for most benchmarks. We also explore the possibility of a minimum-entry store queue with only enough number of entry to keep 20% of instructions between issue and retirement. Our result shows that the aggresive scheme does not have a significant impact on performance.

This technique can also be easily applied to a load queue as only a portion of in-flight loads are finished executing at any certain time. In contrast to stores, most loads have dependent instructions waiting for their results. Thus the number of load queue entries cannot be reduced too aggressively or a better load scheduling policy is needed. Load queue study is left as future work.

8. Acknowledgements

This research was supported in part by the National Science Foundation under grants CCR-0133437, CCF-0429854, and CCF-0702272, as well as grants and equipment donations from IBM and Intel.

9. References

[1] S.Sethumadhavan et al., Scalable Hardware Memory Disambiguation for High ILP Processors, in *MICRO-36*, 2003.

[2] I.Park et al., Reducing Design Complexity of the Load/Store Queue, in *MICRO-36*, 2003.

[3] L.Baugh and C.Ziller, Decomposing the Load-Store Queue by Function for Power Reduction and Scalability, in *IBM-JRD*, October 2004.

[4] E.F. Torres et al., Store Buffer Design in First-Level Multibanked Data Caches, in *ISCA-32*, 2005.

[5] A.Roth, Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization, in *ISCA-32*, 2005.

[6] T.Sha et al., Scalable Store-Load Forwarding via Store Queue Index Prediction, in *MICRO-38*, 2005.

[7] S.S.Stone et al., Address-Indexed Memory Disambiguation and Store-to-Load Forwarding, in *MICRO-38*, 2005.

[8] S.Subramanian and G.H.Loh, Fire-and-Forget: Load/Store Scheduling with No Store Queue at All, in *MICRO-39*, December 2006.

[9] T.Sha et al., NoSQ: Store-Load Communication without a Store Queue, in *MICRO-39*, December 2006.

[10]S.Sethumadhavan et al., Late-Binding: Enabling Unordered Load-Store Queues. in *ISCA-34*, 2007.

[11]H.W.Cain and M.H.Lipasti, Memory Ordering: A Value-Based Approach, in *ISCA-31*, 2004.

[12]M.Franklin and G.S.Sohi, ARB: a Hardware Mechanism for Dynamic Reordering of Memory References, *IEEE Transactions on Computer*, vol 45, no. 5, 1996, pp. 552-571.

[13]S.Palacharla et.al., Complexity-Effective Superscalar Processors, in *ISCA-24*, 1997.

[14]A. Buyuktosunoglu et al., An Oldest-First-Selection Logic Implementation for Non-Compacting Issue Queues, in *ASIC/SOC-15*, 2002.

[15]R.W.Robery et al., Reservation station with a pseudo-FIFO circuit for scheduling dispatch of instructions. Intel Corporation, U.S. Patent 5684971, November 4, 1997.

[16]F.Chen, Method and Apparatus for Implementing Circular Priority Encoder. Intel Corporation, U.S. Patent 6696988 B2, February 24, 2004.

[17]D.Burger and T.Austin. The Simplescalar toolset, version 2.0. *Technical Report TR-97-1342*, University of Wisconsin-Madison, 1997.

[18]T.Monreal et al., Delaying Physical Register Allocation Through Virtual-Physical Registers, in *MICRO*, 1999.

[19]R.Ho et al., The Future of Wires, *Proceedings of the IEEE*, vol 89, no. 4, April 2001, pp. 490-504.

[20]P.Shivakumar and N.P.Jouppi, CACTI 3.0: An Integrated Cache Timing, Power, and Area Model.

[21]J.M. Tendler, et.al., Power4 System Microarchitecture, *IBM Journal of Research and Development*, vol. 46, no. 1, January 2002.

[22]G.Z.Chrysos and J.S.Emer, Memory Dependence Predictor using Store Sets, in *ISCA-25*, 1998.

[23]R.E.Wunderlich et al, SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling, in *ISCA-30*, 2003

[24]H.E. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 3rd edition, May 2004.