A Scalable Low Power Store Queue for Large Instruction Window Processors

Abstract

In modern out-of-order superscalar processors, efficient load store queues which enable loads to be speculatively issued even before all older stores have retired is important for achieving higher IPC. However, earlier proposals use conventional Content Addressable Memories (CAMs) and are hence power hungry and limit the frequency scalability of the processor. In this paper, we propose the Scalable Low power Store Queue (SLSQ). SLSQ predicts the store queue indices of the stores that are most likely to forward to a load. This ability to accurately predict the indices allows SLSQ to drastically reduce the number of comparisons and consequently reduce the energy consumed by the store queue forwarding circuit by 78.4%. SLSQ also cuts down the store queue forwarding circuit latency by 49.9%.

1 Introduction

Out-of-order superscalar processors require the ability to issue loads while older stores are in-flight. Forcing loads to wait for all older stores, including those on which they may not be dependent on, to retire and write to the cache would reduce IPC and take away almost all the benefit of out-of-order execution [4]. On the other hand, maintaining functional correctness while allowing loads to execute in the presence of stores in-flight requires the ability to forward data from the most recent older in-flight store to the same address. Such forwarding, in conventional microprocessors, involves a CAM match of the 64 bit physical address field of each of the store queue entries. Further, the forwarding circuit needs to able to select the most recent older store when compared to the load if multiple matches occur.

The store queue data forwarding logic is thus a significantly high-latency circuit and could limit the frequency scalability of the entire design [13, 7]. Additionally, the power consumption is also considerable as it involves a fast, wide, multiple entry CAM. Note that the forwarding circuit is also trying to detect dependences between loads and stores, similar to a load disambiguation predictor. The primary difference between the two
is that while disambiguation occurs between loads to be issued and pending stores, forwarding happens from completed stores to executing loads.

Our proposal, the Scalable Low power Store Queue (SLSQ), predicts the store queue indices of the stores that are most likely to forward to a load. The ability to accurately predict these indices allows SLSQ to drastically reduce the number of comparisons and consequently, improve energy efficiency. Further, we observe that the loads which get a forwarded value themselves are not in the critical path of the program execution and hence forwarding latency is not critical to performance. While conventional CAM based store queue forwarding logic addresses these non critical instructions which limits the frequency scalability of the design, SLSQ completely obviates the need for a CAM. SLSQ effectively alleviates the store queue data forwarding logic latency problem and allows the design to be clocked at a higher frequency.

In this paper, we identify that delaying only the loads which forward from stores does not effect IPC. This leads us to the conclusion that detecting or predicting occurrences of store to load forwarding accurately is more important that being able to complete the data forwarding in one cycle. We then observe that most disambiguation predictors predict the dependences between loads and pending stores but totally ignore completed stores. Instead, they rely on full CAMs in the store queue to resolve these by forwarding data from the completed store to the load once the load is issued. Finally, we propose, implement and evaluate a store queue design which makes use of the aforementioned observations and eliminates the CAM in the store queue. This reduces store to load forwarding circuit latency and power consumption without any noticeable loss in IPC.

This paper is organized as follows. We describe the Store Vector Disambiguator [5], an efficient load scheduling mechanism on which our proposal is based, and motivate our work in Section 2. Details of our SLSQ microarchitecture are presented in Section 3. In Section 4, we evaluate the performance of our scheme. We discuss and compare SLSQ with other related works in Section 5 and conclude in Section 6.

2 Background and Motivation

In this section, we first describe the Store Vector Disambiguator (SVD) as our SLSQ design is based on and improves SVD in terms of energy efficiency or frequency scalability. We then examine the criticality of loads which forward from stores.
2.1 Store Vector based Disambiguation

Consider the example in Figure 1 wherein the static load and store instructions are indicated in a control flow graph along with two potential dynamic sequences. The static load instruction corresponding to Load E has had ordering violations with the static instruction corresponding to Store C and Store A. The Store Vector based Disambiguator (SVD) remembers the relative store queue indexes of stores which are likely to alias with the load in the form of a bit vector as shown in Figure 1.

Figure 2 depicts the various structures involved in the Store Vector based Disambiguation (SVD) technique. The Store Vector Table stores the store vectors and is indexed using the load PC. When a load is dispatched, the initial vector is obtained from the Store Vector Table and is barrel shifted to accommodate the fact that the least significant bit in the vector corresponds to the most recent store in the store queue. The vector is then bitwise ANDed with the valid and completed bits of the store queue to ensure that the load does not end up waiting on a non-existent or completed store.

This vector is then written to the Load Scheduling Matrix. This matrix has as many rows as the load queue depth and as many columns as the store queue depth. When a store completes, it clears all the bits in the column corresponding to its store queue entry. When all the bits in a row corresponding to a load are zero, the load is speculatively issued even though there may be earlier pending stores. Thus, the store vector predicts the dependence of loads with respect to earlier stores in-flight. The load address is then compared with the address of all previous completed stores in the store queue. If a match, i.e., store to load data forwarding occurs, then

---

1 We avoid repeatedly mentioning “static load corresponding to” and loosely use Load E or Store C in this discussion to refer to the static instruction corresponding to the dynamic instance throughout this discussion.
the store data is read out of the store queue and is used by the load instruction. Thus, even if the store vector corresponding to the load did not have this bit set, the true dependence between the older store and the load are resolved without any memory ordering violations. However, if there is an older pending (address not available) store to the same address\(^2\), for which the bit in the store vector indicated no dependence, a memory ordering violation will occur. When the store in question completes, it checks the load queue for any younger loads to the same address that did not forward correctly.

This approach has been shown [5] to be much simpler to implement than previous Store Sets [4, 13] based approaches. The SVD predictor brings with it almost all the IPC benefit that an oracle disambiguation predictor has over a blind disambiguation predictor.

### 2.2 Motivating the SLSQ Design

Earlier studies observe that a conventional CAM based store queue forwarding logic limits the frequency of the entire design similar to large monolithic issue queues. One of the reasons for the scalability limitation is the complexity of the associative search for matching the load addresses with every entry in the store queue. In this section, we examine whether the instructions that the forwarding structure is intended to speed up, i.e., the loads which get forwarded values from stores, are on the critical path of the execution. To this end, we delay all

\(^2\)If forwarding is to occur, the pending store must be younger than the forwarding store
loads that forward from stores by a few cycles. In Figure 3 we plot the IPC penalty that is caused by delaying all such loads by two cycles, i.e. a load latency of 5 cycles. This results in a loss of only 0.2% over SPEC FP and no discernible loss over SPEC Int. Details of the machine configuration and the methodology used can be found in Section 4.

From the above results, we conclude that the store queue forwarding logic serves only a small portion of the instruction stream, which are not on the critical path. Thus, a highly accurate forwarding predictor coupled with a complexity-effective and potentially multicycle forwarding circuit should result in minimal IPC degradation when compared to a much more difficult to implement fast forwarding circuit.

It is interesting to note that most disambiguation predictors including SVD, predict dependences between loads and stores only between loads to be issued and stores which do not have their address ready yet. This is because such designs assume a full forwarding CAM to detect and forward from older in flight completed stores to the same address. We observe that disambiguation predictors can be modified to predicts dependences between loads and all stores, not just pending stores.

We also observe that the SVD lends itself to simplifying the forwarding circuit as the store vector is nothing but a bitmap of which entries in the store queue a load is likely to be dependent on. This could alleviate the need for fully associative searches in the store queue.

3 SLSQ Microarchitecture

The SLSQ proposed in this paper has a Load Scheduling Matrix (LSM) which is similar to the one used in the Store Vector based Disambiguation (SVD) scheme, and a load PC indexed Disambiguation and Forwarding Predictor (DFP). The Disambiguation and Forwarding Predictor is structurally similar to the Store Vector Table.
Figure 4: The SLSQ microarchitecture

used in SVD but is different in the way it is used and trained as explained later in this section. SLSQ also adds the capability to store the predicted store vector for every load in the load queue as shown in Figure 4.

During the dispatch stage, when a load instruction is allocated an entry in the load queue, the Disambiguation and Forwarding Predictor (DFP) outputs a bit vector where the least significant bit corresponds to the youngest store older than the current load instruction. The bit vector is barrel shifted so that bit 0 corresponds to the first store queue entry. We call this rotated vector the Disambiguation Prediction (DP). Unlike the SVD, this Disambiguation Prediction is modified so that only one bit is set and saved as the Forwarding prediction (FP) in the load queue entry. Then for each bit set in the DP, the store queue is checked to see if the store queue entry is valid or if store instruction has completed. If the entry is invalid or the instruction has completed, the bit is reset. This processed store vector is then written to the Load Scheduling Matrix (LSM) in the appropriate row.

When a store completes, it clears the corresponding column in the LSM. Once all the bits corresponding to the load queue entry are cleared, it is ready to be issued speculatively. Thus, the disambiguator prediction for a load in SLSQ is similar to SVD.

As observed in Section 2.2, loads that get a forwarded value are not in the critical path of execution. Thus, simplifying the forwarding structure may not significantly affect the performance (IPC). Therefore, we avoid the associative search, i.e., the store to load forwarding CAM in Figure 2 and use a single ported RAM to

---

4 A store instruction is completed if the store has its address and data. Note that the store writes to the cache only in the retire stage.
accomplish store to load forwarding instead. When a load is issued, the store queue is probed if and only if one bit was set in the Forwarding Prediction in the load queue. If one bit was set, then, only the store queue entry corresponding to the set bit is read and compared with the load address. Thus, the store to load forwarding circuit latency is reduced significantly, potentially improving the frequency scalability of the entire processor. We observe that the bit selection scheme used to reduce the DP to the FP does not significantly affect performance. The performance delta between selecting the first bit (youngest store) or the last bit (oldest store) was not noteworthy. We used the first set bit and cleared the rest to obtain the FP in all our studies.

Note that if there is a match when the load probes the store queue in any of the SLSQ designs, the store tag of the store is stored in the load queue. This is done to detect the scenario where there are multiple stores to the same address in-flight at any given point of time and the load wrongly gets forwarded data from an older store to the same address. Such a scenario could arise due to a misprediction in the disambiguation stage. In SLSQ, such a scenario could also occur if the DFP mispredicts the store queue indices to compare with for store to load forwarding.

In order to detect occurrences of memory ordering violations, when a store retires, it queries the load queue to check if there is a younger load to the same address that has not forwarded from the store. There are two situations to consider here. When the load was issued, no matching store may have been found and hence the forwarding store tag was set to invalid. Alternatively, a matching store was found, but was older than the retiring store. So, the memory ordering violation is detected by comparing the store address and the instruction tag of the retiring store with the load address and the forwarding store tag saved in the load queue. Thus, a memory ordering violation is detected when the store address is identical to the load address and the forwarding store tag is either invalid or older than the retiring store tag. When such a violation is detected, the relative age of the store with respect to the load is determined. The corresponding bit is set in the DFP so that future instances of that load will wait for this store to complete. Thus, the predictor is trained and future memory ordering violations are minimized. Since all the bits are likely to be set eventually, the table is flash cleared once every few thousand cycles\(^5\) as in the Alpha 21264 [3]. Note that this involves a CAM in the load queue. However, compared to the forwarding CAM in the store queue which is probed for all speculative loads, this CAM is only probed only for retiring stores. Since the bandwidth required from the load CAM is smaller, it can be designed to consume less power.

\(^5\)We clear the table once every sixteen thousand cycles
3.1 SLSQ Energy Consumption

In the base machine, every load that is issued to the data cache also probes the store queue. This involves a fully associative search, and thus necessitates a CAM. The CAM is used to compare the load address with the addresses of all the older stores in-flight. Since, we assume a split load store queue structure, the logic to determine the youngest store older than the load is required in the face of multiple hits. We assume an aggressively gated baseline which compares the load address only with store queue entries which contain valid stores.

The SLSQ extensions replace the CAM with a RAM that is read only when one bit in the FP is set. As we observed earlier, both memory ordering violations and instances of forwarding are few and far between. Therefore, the RAM is not activated very often. Thus, both the number of store address reads and the number of address compares are reduced. This, leads to significant reduction in energy consumption.

3.2 SLSQ Latency

SLSQ only considers at most one or two set bits while reading the LSM. This enables the store forwarding logic to completely eliminate the CAM structure. As illustrated in Figure 4, the store queue can be implemented as a single ported RAM like structure, wherein two entries are read and then compared to the address of the newly issued load. Note that the latency of this RAM structure is also (marginally) lower than a conventional RAM because it does not require a decoder. This follows from the design decision to store and use the bit vector of the entries in the load queue to access the store queue physical addresses in the RAM structure.

4 Performance Evaluation and Methodology

4.1 Simulation Methodology

We modified Sim-Alpha, a cycle accurate Out-of-Order superscalar simulator validated against the Alpha 21264 [8]. The simulator was extended to implement blind disambiguation, store vectors and SLSQ. We model a 6 wide machine with parameters similar to but marginally more aggressive than the 21264. We assume a branch misprediction latency of 12 cycles as indicated in Table 1. The data cache has 2 read-write ports. The simulator models limited outstanding cache misses, bus contention and detailed memory transactions.

We simulate the entire SPEC CPU2000 benchmark suite except fma3d which required a system call that was not implemented in Sim-Alpha. We use the early simulation points [11] to identify and simulate phases
Table 1: Microarchitectural Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out-of-Order</td>
<td>4/6/4 wide fetch/issue/commit</td>
</tr>
<tr>
<td>execution</td>
<td>100 ROB, 30 Int IQ, 25 FP IQ,</td>
</tr>
<tr>
<td></td>
<td>32 LQ, 32 SQ</td>
</tr>
<tr>
<td>Functional</td>
<td>4 int ALUs, 4 int MUL,</td>
</tr>
<tr>
<td>units</td>
<td>2 fp ALU, 1 fp MUL/DIV</td>
</tr>
<tr>
<td>Memory sub-system</td>
<td>L1 Data 8-way 32KB (3 cycle),</td>
</tr>
<tr>
<td>(latency)</td>
<td>L1 Inst 2-way, 64KB (1 cycle),</td>
</tr>
<tr>
<td></td>
<td>L2 Unified 16-way 1MB (10 cycles), main memory</td>
</tr>
<tr>
<td></td>
<td>(min 150 cycles)</td>
</tr>
<tr>
<td>Branch prediction</td>
<td>Alpha 21264 hybrid predictor [8]</td>
</tr>
<tr>
<td>Misprediction latency</td>
<td>12 cycles</td>
</tr>
</tbody>
</table>

Figure 5: IPC of SLSQ

that are representative of the entire program and simulate 100 million instructions from the early simpoint. All binaries for the benchmarks were taken from the SimpleScalar website. Table 2 shows the benchmarks used. The reference input sets were used for all the programs. We use geometric means when considering averages of measured metrics including IPC. Arithmetic means are used whenever normalized values or percentages are considered.

The circuit latency and the energy consumption of the store queue CAM circuit were estimated using fully associative and direct mapped configurations of CACTI 3.2 [12]. We modified CACTI for the required tag (address) width and to output a detailed stage-wise energy consumption for fully associative address comparison.
4.2 Impact of SLSQ on IPC

We now examine the performance of the various design alternatives. Figure 5 shows the IPC for a 1K entry (32KBytes) SVD and SLSQ with 1 and 2 RAM ports. The single ported SLSQ experiences a minor average IPC loss of 0.02% and 0.06% for the integer and floating point benchmarks respectively. While, adding a second port to the RAM eliminates these some of these losses, it incurs the additional overhead of having to split the store vector into 2 vectors, each with only one bit set so that each vector can be used in one port. Since the IPC degradation encountered by the single ported SLSQ is very low, we use only this RAM configuration for all further studies.

4.3 Why SLSQ Works

In this section, we examine the reasons behind the very low IPC degradation faced by the SLSQ design. Note that there are 2 sources of additional memory ordering violations in SLSQ when compared to SVD. The first is because we consider only the entries whose corresponding bits in the DP are set. Additionally, we only consider one of these entries as SLSQ is implemented using a single ported RAM. To better understand how these approximations affect performance, we introduce the SLSQ-CAM variant. SLSQ-CAM is different from SLSQ in that it uses a conventional CAM for store forwarding. But, unlike the base machine, SLSQ-CAM entries are compared with only if the corresponding bit in the DP is set. Also, the CAM itself is activated

---

Note that SLSQ has an extra cost of storing the forwarding predictions in the load queue. The 1K entry SLSQ requires a 128 bytes more than the SVD which requires 4KB for the predictor table and 128 bytes for the LSM.
only if at least one bit is set in the DP. Thus, SLSQ-CAM significantly reduces broadcasts and compares and consequently reduces power consumption. In terms of accurate store to load forwarding, it is obvious that SLSQ-CAM is right in between the perfect base machine and the SLSQ described in the previous section. However, SLSQ-CAM offers no forwarding circuit latency improvement over the base case.

First, we look at the number of loads which forward from the store queue and the prediction accuracy of the SLSQ and SLSQ-CAM designs. We classify a prediction as inaccurate only if it results in a memory ordering violation. Thus, a pessimistic prediction in which all the bits are set is always accurate\textsuperscript{7}. Table 2 presents the number of loads which forward from stores, and memory ordering violations per thousand instructions for the base SVD configuration with a 1K entry SVT.

We observe that the number of forwarding instances per thousand instructions is greater than 10 (i.e., greater than 1%) in only 3 of the 25 benchmarks. Additionally, on an average, there are less than 2 instances of forwarding per thousand instructions (0.2%). Thus, the forwarding instances are few and far between. Figure 6 indicates the forwarding store queue entry prediction accuracy of the SLSQ and the SLSQ-CAM designs. We find that for the SLSQ-CAM, the accuracy is higher than 99.5% on an average across all the benchmarks. However, for SLSQ, the prediction accuracy is a little lower at 98.811% for the integer benchmarks and 96.34% for the floating point benchmarks. In fact, for Art, SLSQ is only correct 54% of the time. Even this does not translate to a significant loss in IPC as the number of loads which forward from stores is only 0.6 loads per thousand instructions for Art.

\textsuperscript{7}Such a pessimistic prediction results in a few extra compares in SLSQ-CAM and impacts only its energy efficiency.
Table 2: Frequency of Forwarding, Memory Ordering Violations and Bits set per prediction in 1K-entry SVD

<table>
<thead>
<tr>
<th>Bmk</th>
<th>Forwarding instances (per 1000 instrs)</th>
<th>Mem. ordering violations (per 1000 instrs)</th>
<th>Bits set per prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bzip</td>
<td>10.6372</td>
<td>0.2737</td>
<td>1.2291</td>
</tr>
<tr>
<td>Crafty</td>
<td>0.5019</td>
<td>0.2957</td>
<td>0.0172</td>
</tr>
<tr>
<td>Eon</td>
<td>18.6583</td>
<td>3.4745</td>
<td>0.2833</td>
</tr>
<tr>
<td>Gap</td>
<td>3.4602</td>
<td>0.3779</td>
<td>0.0526</td>
</tr>
<tr>
<td>Gcc</td>
<td>0.1637</td>
<td>0.1804</td>
<td>0.1741</td>
</tr>
<tr>
<td>Gzip</td>
<td>3.3759</td>
<td>0.0828</td>
<td>0.0401</td>
</tr>
<tr>
<td>Mcf</td>
<td>0.0012</td>
<td>0.0056</td>
<td>0.0001</td>
</tr>
<tr>
<td>Parser</td>
<td>8.9223</td>
<td>0.1991</td>
<td>0.0566</td>
</tr>
<tr>
<td>Perl</td>
<td>6.5034</td>
<td>0.5583</td>
<td>0.0975</td>
</tr>
<tr>
<td>Twolf</td>
<td>1.9922</td>
<td>2.2600</td>
<td>0.1023</td>
</tr>
<tr>
<td>Vortex</td>
<td>1.8827</td>
<td>0.7247</td>
<td>0.0372</td>
</tr>
<tr>
<td>Vpr</td>
<td>35.1472</td>
<td>0.4057</td>
<td>0.0733</td>
</tr>
<tr>
<td>Int AVG</td>
<td>1.8656</td>
<td>0.3051</td>
<td>0.0514</td>
</tr>
<tr>
<td>Ammp</td>
<td>6.5845</td>
<td>0.7309</td>
<td>0.0956</td>
</tr>
<tr>
<td>Applu</td>
<td>7.4257</td>
<td>2.4175</td>
<td>0.2178</td>
</tr>
<tr>
<td>Apsi</td>
<td>1.0503</td>
<td>0.5722</td>
<td>0.0353</td>
</tr>
<tr>
<td>Art</td>
<td>0.6363</td>
<td>0.9832</td>
<td>0.0793</td>
</tr>
<tr>
<td>Equake</td>
<td>2.6338</td>
<td>0.1106</td>
<td>0.0261</td>
</tr>
<tr>
<td>Facerec</td>
<td>0.0736</td>
<td>0.2466</td>
<td>0.0839</td>
</tr>
<tr>
<td>Mesa</td>
<td>6.5984</td>
<td>1.0068</td>
<td>0.0987</td>
</tr>
<tr>
<td>Mgrid</td>
<td>3.1300</td>
<td>0.2410</td>
<td>0.0072</td>
</tr>
<tr>
<td>Sixtrack</td>
<td>0.0488</td>
<td>0.1433</td>
<td>0.0196</td>
</tr>
<tr>
<td>Swim</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0001</td>
</tr>
<tr>
<td>Wupwise</td>
<td>5.7204</td>
<td>0.2185</td>
<td>0.0426</td>
</tr>
<tr>
<td>FP AVG</td>
<td>0.0925</td>
<td>0.0471</td>
<td>0.0117</td>
</tr>
</tbody>
</table>

We now turn our attention to the average number of bits set in every SLSQ-CAM forwarding prediction that has at least one bit set. Given that SLSQ, which only considers one bit, performs as well as it does, we expect to find that very few bits are set on an average. This is indeed the case as all benchmarks, except Bzip and Art, have less than 1.5 bits set. The average number of bits set over all the benchmarks is 1.2 as indicated in Figure 8. The large number of bits set in SLSQ-CAM for Art clearly indicates why the prediction accuracy of SLSQ is very low for that benchmark. Note that these are the average number of bits set per prediction with at least one bit set. If all the predictions (including all predictions with all bits in the vector being reset) are included, the average drops to 0.05 and 0.01 bits per prediction as indicated in Table 2.
We now look at the total memory ordering violations encountered in the SVD, SLSQ, and SLSQ-CAM designs. As stated previously, the base machine employs SVD with a 1K entry SVT and an ideal CAM based forwarding logic. We find that on an average, SLSQ CAM encounters 0.26% and 0.11% more violations for the integer and floating point benchmarks respectively. However, SLSQ faces an increase of 13.94% and 4.6% respectively. The high averages are due to large relative increases in bzip, parser, art and wupwise. Again, while the relative increases are very high for these benchmarks, the absolute number of memory ordering violations that cause a load and all subsequent instructions to be squashed and reexecuted is small. Further, the impact of these violations on IPC is negligible. We investigate the performance of SLSQ at different hardware budgets and several other minor variations in greater detail in the full report [15].

4.4 Circuit Timing Results

The small IPC loss incurred by SLSQ would be acceptable only if there is a significant reduction in the issue latency. In this section, we present the latency results of the SLSQ designs. As described in Section 3, SLSQ-CAM and the base SVD machine are similar in terms of forwarding latency. As mentioned earlier, the timing results were obtained using the fully associative and direct mapped configurations in Cacti 3.2. We measured the circuit latencies for both the 90 nm and the 65 nm process.

In the base SVD machine the forwarding latency consists of the CAM address broadcast to the store queue CAM and comparison ($T_{CAM}$), and the latency of Reading the store queue data RAM ($T_{RAM}$). Thus the forwarding latency for a conventional CAM based store to load data forwarding logic is given by the following formula:
For the 65nm process node we find that $T_{CAM}$, $T_{RAM}$ to be 0.6373ns, 0.1463ns respectively, corresponding to a total latency of 0.7836ns.

In SLSQ, the store vector prediction indicates exactly for which store queue entry address needs to be compared. The address which are stored in a RAM are read out and compared with the address of the new load. The latency to read the store address for address comparison is $T_{Rd}$. Note that reading the store queue RAM does not require any decoding, as the decoded store vector is available. The time to compare the store address and the load address is called $T_{Comp}$. Next, if a match occurs, the same vector is used to read the contents of the store queue data RAM as before. Thus, the forwarding latency of SLSQ consists of the Address read latency ($T_{Rd}$), Address compare latency ($T_{Comp}$) and the store queue data RAM latency ($T_{RAM}$).

For SVD on the 65nm process node, we find that $T_{Rd}$, $T_{Comp}$ and $T_{RAM}$ to be 0.0908ns, 0.1603ns and 0.1418ns respectively. Thus, the ideal CAM based forwarding logic has a latency of 0.7836ns while that of SLSQ is only 0.3929ns. This corresponds to a latency improvement of 49.9%. We also observe that the latency improvement is almost identical for the 90nm proces as well, indicating that SLSQ latency scales well with process technology.
4.5 Energy Consumption

We now evaluate the reduction in energy consumption achieved by the SLSQ designs. To this end, first, we look at the relative number of broadcasts/reads and comparisons required for SLSQ when compared to the base machine which uses SVD. Note that the SLSQ-CAM design broadcasts the address like the base machine while the SLSQ reads it out like a conventional RAM structure. For this study, we will consider the read from the RAM in SLSQ to be conceptually similar to the tag broadcast in the base machine and SLSQ-CAM. In the SLSQ design, the number of broadcasts / reads would be far fewer than the base machine. This because a broadcast occurs in SLSQ only if at least one of the bits in the initial store vector that is stored in the load queue.

---

Note that no decoder is required for this RAM as the store vector directly stores the decoded information.
is set unlike the base machine which broadcasts the load address on every load. Figure 9 shows the fraction of broadcasts/reads avoided by SLSQ. On an average, SLSQ-CAM avoids broadcasts 86% and 94.7% of the time for the integer and floating point benchmarks. Similarly, SLSQ cuts down on the number of reads significantly. Reads are avoided 86% and 94.7% of the time for the SPEC Int and FP suites respectively.

Next we consider the number of comparisons reduced by SLSQ. Once the broadcast/read has occurred, the store physical address is then compared to the newly issued load. The number of compares for each broadcast in the case of the base machine is equal to the number of valid store queue entries at that time. For SLSQ-CAM, it is equal to the number of bits that are set in the initial store vector while for SLSQ, it is exactly equal to the number of RAM ports, which in our case is 1. Figure 10 shows the reduction in compares when compared to the base machine with a conventional CAM based forwarding logic. SLSQ CAM also reduces the number of comparisons by 93.9% and 98.3% respectively. SLSQ avoids comparisons 96.7% and 99% of the time for the integer and floating point benchmarks. Bzip2 is the only exception to this clear trend. This is primarily because many of the bits in the DFP are set due to memory ordering violations. Thus, SLSQ almost always broadcasts the load address (SLSQ CAM) or reads the store address (SLSQ).

We now measure the energy consumed for broadcasts/reads and compares in the store queue followed by the store data RAM read on a forwarding match, as discussed earlier. The energy results described below were obtained using corresponding CACTI estimates for identically sized and ported fully associative and direct mapped caches. Table 4 shows the energy consumption for each critical phase of the forwarding logic for the 90 nm and the 65 nm process nodes. Let $E_{brd}$ and $E_{RAM}$ correspond to the energy consumed in broadcasting the load address and reading the data from the store RAM when a match occurs. The energy consumed $E_{rd}$ in reading the store address in the case of SLSQ is almost 2-3 times as much as $E_{brd}$. However, SLSQ requires only one compare per read as opposed to the base machine that requires as many compares as there are valid entries in the store queue. Note also that, while this results in SLSQ not being as energy efficient as SLSQ-CAM, the RAM organization allows to cut down the forwarding latency by 49.9% as discussed in Section 4.4.

The energy consumption of the forwarding logic for each benchmark for the base machine, and for SLSQ CAM can be estimated as:

Energy Consumption for SVD, SLSQ-CAM = $N_{brd}E_{brd} + N_{cmp}E_{cmp} + N_{Fin}E_{RAM}$

$N_{cmp}$ represents the total number of comparisons required for the benchmark. $E_{cmp}$ refers to the per entry
Table 4: Energy Consumed by Store Queue Activities for the 90 nm and 65 nm process

<table>
<thead>
<tr>
<th>Process node (nm)</th>
<th>Config</th>
<th>Broadcast Energy (E_{brd}(\text{nJ}))</th>
<th>Read Energy (E_{rd}(\text{nJ}))</th>
<th>Per Entry Compare Energy (E_{cmp}(\text{nJ}))</th>
<th>RAM access Energy (E_{RAM}(\text{nJ}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>Base</td>
<td>0.0563</td>
<td>-</td>
<td>0.0010</td>
<td>0.1579</td>
</tr>
<tr>
<td>90</td>
<td>SLSQ-CAM</td>
<td>0.0563</td>
<td>-</td>
<td>0.0010</td>
<td>0.1579</td>
</tr>
<tr>
<td>90</td>
<td>SLSQ</td>
<td>-</td>
<td>0.1136</td>
<td>0.0003</td>
<td>0.0812</td>
</tr>
<tr>
<td>65</td>
<td>Base</td>
<td>0.0262</td>
<td>-</td>
<td>0.0005</td>
<td>0.1106</td>
</tr>
<tr>
<td>65</td>
<td>SLSQ-CAM</td>
<td>0.0262</td>
<td>-</td>
<td>0.0005</td>
<td>0.1106</td>
</tr>
<tr>
<td>65</td>
<td>SLSQ</td>
<td>-</td>
<td>0.0799</td>
<td>0.0001</td>
<td>0.0562</td>
</tr>
</tbody>
</table>

address compare energy. \(N_{brd}\) refers to the total number of times the address of a load was broadcasted to the entries in a CAM based forwarding logic and \(E_{brd}\) refers to the energy consumed for each such operation. \(N_{inst}\) and \(E_{RAM}\) refers to the number of times store to load forwarding occurs in the program while and the energy to read the store data respectively. Note that we consider an aggressive base machine which compares the addresses of an issuing load with addresses of only valid store queue entries. For the 65nm node, we find \(E_{brd}, E_{cmp}, \) and \(E_{RAM}\) to be 0.0262nJ, 0.0005nJ and 0.1106nJ respectively. Note that these energy values are per occurrence, i.e although \(E_{cmp}\) is much smaller than \(E_{RAM}\), it occurs more frequently and thus constitutes a significant portion of the total store queue energy.

The energy consumption of SLSQ is given by:

\[
\text{Energy Consumption for SLSQ} = N_{rd}E_{rd} + N_{cmp}E_{cmp} + N_{inst}E_{RAM}
\]

\(N_{rd}\) refers to the total number of times a store queue entry was read in SLSQ because one of the bits in the predicted store vector was set (this would have resulted in a broadcast in a CAM based forwarding logic) and \(E_{rd}\) refers to the energy consumed by each such read. For the 65nm node \(E_{rd}, E_{cmp}, E_{RAM}\) are found to be 0.0799nJ, 0.0001nJ, 0.0562nJ.

We modified the Sim-Alpha simulator to output the energy consumption based on the above formulae and values listed above. Again, we report average forwarding logic energy consumed over the entire SPEC CPU2000 suite. Figure 11 shows the energy savings. On an average, SLSQ CAM saves 78.4% and 91.6% of the store queue forwarding logic energy over the SPEC INT and FP benchmarks respectively. SLSQ however saves only 61.4% and 84.6% of the energy for the two suites. This is because while the Address RAM avoids expensive associative searches and has a faster latency, it consumes more energy per read. Thus, for bzip which has a very high number of such reads, SLSQ actually consumes 52.3% more energy than the base case. Bzip,
however, is the only such benchmark wherein SLSQ is more energy consuming than the base machine.

Figure 12 shows the reduction in energy delay ($ED$) product for SLSQ and SLSQ-CAM when compared to the base machine with SVD and a conventional naive CAM based forwarding logic. Since SLSQ-CAM has no impact on the latency, its energy delay savings are identical to its energy savings described above. However, SLSQ reduces the latency of the forwarding circuit by 49.9% as discussed in Section 4.4. If the latency of this circuit was on the critical path for the whole design, then SLSQ reduces the energy delay product by 80.5% and 92.3% for the integer and floating point benchmarks respectively. Even for Bzip, SLSQ outperforms the base SVD machine in Energy Delay product by 23.7%. Note that all these results are for store queue forwarding circuit energy only.

5 Related Work

Several studies have proposed modifications to the conventional age ordered store queues [13, 1] that improve its scalability. Sethumadhavan et al. [1] use a Bloom filter that conservatively encodes the addresses of in-flight stores. Only loads whose addresses hit in this filter access the store queue. This scheme is vulnerable to false positives from loads that match younger stores in-flight.

Park et al. [13] use a store load disambiguation predictor that uses a modified and more accurate version of the store sets [4] principle. A Store set corresponds to the set of stores, tracked using the least significant bits of the PC, that could potentially write to the same address as a given load and potentially be in-flight when the load is issued. Store forwarding is made more scalable by dividing the store queue into segments and pipelining the
store queue access. As this scheme is based on the store sets principle, it introduces more structures capable of associative searches into the microarchitecture. This would directly and adversely impact the energy efficiency of the design.

Other proposals including the one by Baugh et al. [7] and another one by Roth [2] propose to scale store queues by splitting them into two structures, one for store to load forwarding and one for memory ordering. Only loads which resulted in ordering violations earlier probe the store queue on execution. The forwarding structures continue to be fully associative CAMs, albeit a little smaller than before. Since these designs rely on CAMs, they are orthogonal to SLIQ and could be combined to yield better energy efficiency and latency.

Address-indexed store queues such as [1] offer much higher load execution bandwidth. However, such designs suffer from address conflicts and do not lend easily to designs which incorporate data and control misspeculation due to the inherent lack for support for multiple in-flight stores to the same address.

Our SLSQ design is based on the Store Vector based Disambiguator design [5]. However, there are some important differences which lead to significant improvement in the forwarding latency and energy consumption with a negligible loss in IPC. While the scheduling mechanism for issuing loads speculatively is similar in SVD and SLSQ, that's where the similarities end. SVD only predicts dependences between loads and pending stores, i.e., stores which do not have their address ready yet. It assumes the presence of a CAM in the store queue to detect and resolve dependences between loads and completed stores. SLSQ obviates the need for a CAM by predicting dependences between loads and pending stores (DP) as well as completed stores (FP). This allows
SLSQ to reduce the store queue circuit latency as well as power consumption with no discernible loss in IPC.

Sha et al. [9], use a modification of the store sets principle to predict which store will forward to the given load. This work is similar to ours in that the predictor predicts the store queue index of the store. This eliminates the need for a CAM in the store queue and improves scalability, but incurs a loss in IPC when compared to a fast fully associative store queue. Also, this scheme necessitates a selective replay mechanism to handle scenarios when the store queue latency is higher than the data cache latency. This proposal and others [14, 16] use a Store Vulnerability Window based approach that requires re-execution of loads before commit instead of an associative load queue. This results in greater contention for data cache bandwidth. Since the predictor can only track one store index per dynamic load, another predictor which delays problematic loads is used. Load scheduling now involves both the forwarding index predictor and the delay index predictor. SLSQ on the other hand, predicts the store queue index accurately with minimal changes to existing structures and a reduction in latency as well as energy consumption. SLSQ does not require a Store Vulnerability Window implementation in the base machine.

6 Conclusions

The design requirement for larger store queues, a direct consequence of the requirement for larger instruction windows, could result in the store queue forwarding logic becoming the timing critical path of the entire design. Typically store queue data forwarding logic is implemented using CAMs. Every newly issued load compares its address with every older completed store in the store queue. This is also wasteful in terms of energy consumption as most loads do not forward from stores. Further, we observe that identifying the loads that are likely to get forwarded results (from an in-flight store) is more important than forwarding the results themselves quickly. Thus, simply predicting occurrences of store-to-load forwarding in an energy-efficient manner is a practical alternative to existing power-hungry CAM based Load Store Queue design.

Based on the above observation, we propose and evaluate the Scalable Low power Store Queue (SLSQ) design. SLSQ builds on the Store Vector based Disambiguator [5]. In SLSQ, only when a load results in a memory ordering violation, the store vector for that load is trained. For future instances of that dynamic load, SLSQ compares the load address with only the relevant store queue entries, instead of blindly comparing with all valid store queue entries. SLSQ accurately predicts the store queue index of the store which will forward to the given load at the cost of a minimal increase in hardware budget due to the overhead of storing the vectors in the load queue. In SLSQ, the CAM in the forwarding logic is replaced with a single ported RAM. The RAM
structure facilitates a 49.9% reduction in the store-to-load data forwarding latency. SLSQ incurs a minimal IPC degradation of under 0.1% on average for the entire SPEC CPU2000 benchmark suite. This allows aggressive designs to have better clock frequency scalability with minimal loss of IPC. Since, most loads instructions never alias with any store, they do not need to access the store queue at all. This enables SLSQ to significantly reduce the energy consumption and the energy-delay product by 72.3% and 86.4% on an average for the entire SPEC CPU2000 INT and FP benchmark suites respectively.

References


