

Fire-and-Forget: Load/Store Scheduling with No Store Queue at All

Samantika Subramaniam Gabriel H. Loh
 Georgia Institute of Technology
 College of Computing
 {samantik,loh}@cc.gatech.edu

Abstract

Modern processors use CAM-based load and store queues (LQ/SQ) to support out-of-order memory scheduling and store-to-load forwarding. However, the LQ and SQ scale poorly for the sizes required for large-window, high-ILP processors. Past research has proposed ways to make the SQ more scalable by reorganizing the CAMs or using non-associative structures. In particular, the Store Queue Index Prediction (SQIP) approach allows for load instructions to predict the exact SQ index of a sourcing store and access the SQ in a much simpler and more scalable RAM-based fashion. The reason why SQIP works is that loads that receive data directly from stores will usually receive the data from the same store each time.

In our work, we take a slightly different view on the underlying observation used by SQIP: a store that forwards data to a load usually forwards to the same load each time. This subtle change in perspective leads to our “Fire-and-Forget” (FnF) scheme for load/store scheduling and forwarding that results in the complete elimination of the store queue. The idea is that stores issue out of the reservation stations like regular instructions, and any store that forwards data to a load will use a predicted LQ index to directly write the value to the LQ entry without any associative logic. Any mispredictions/misforwardings are detected by a low-overhead pre-commit re-execution mechanism.

Our original goal for FnF was to design a more scalable memory scheduling microarchitecture than the previously proposed approaches without degrading performance. The relative infrequency of store-to-load forwarding, accurate LQ index prediction, and speculative cloaking actually combine to enable FnF to slightly out-perform the competition. Specifically, our simulation results show that our SQ-less Fire-and-Forget provides a 3.3% speedup over a processor using a conventional fully-associative SQ.

1. Introduction

The recent past has seen processor performance driven by an incredibly fast ramping of processor clock frequencies. However, recent processor designs have backed away from this “hyper-pipelined” mentality due to unmanageable com-

plexity and power consumption. In pursuit of more performance while maintaining manageable clock speeds, current processor microarchitectures have moved back toward ILP techniques. For example, the upcoming Intel® Core 2 processor is reported to provide wider superscalar execution with larger buffers as compared to previous Intel® Pentium-M and Core 1 microarchitectures. However, conventional implementations of out-of-order instruction windows are limited with respect to how large they can be scaled before negatively impacting the overall processor clock speed and power consumption. In this paper, we specifically address the load and store queues which allow the processor to perform out-of-order scheduling.

Memory operations occur very frequently in most applications (1 out of every 3 or 4 instructions), and therefore a high-ILP processor must have large structures to buffer all of these loads and stores. The load and store queues buffer these instructions, and they also perform the additional tasks of memory disambiguation and store-to-load value forwarding. Standard implementations of these queues require a large amount of fully-associative search logic (i.e. content addressable memories or CAMs) that end up limiting the queue sizes for a given clock frequency. Furthermore, the CAMs consume a large amount of power which affects the thermal design power (TDP) of the overall processor.

There is an ironic aspect in the difficulty of scaling the load and store queues in that, for a large fraction of memory instructions, the costly CAM logic will not actually result in hits/matches. Even for very large instruction windows, many stores end up writing to the cache before any loads from the same address are ready to receive the data, implying that the data-forwarding mechanism of the store queue is underutilized. In a complementary fashion, many load instructions search the store queue for an earlier write to the same address, but do not find a match. These loads end up retrieving their data values from the data cache, and do not effectively make use of the store queue’s CAMs. The processor includes all of this associative logic to make sure that loads and stores execute correctly, but it slows down the common case of no forwarding.

Conventional load and store queues only provide a few basic functions for the correct operation of the processor. In particular, they (1) enforce the proper ordering between

loads and stores, and (2) forward data from stores to loads of the same address. The store queue also (3) keeps stores in program order to facilitate in-order writeback to the cache hierarchy. While the load and store queues do accomplish these tasks, we can potentially use *any* mechanism that fulfills these functions. Previous proposals based on load re-execution have described different ways of removing associative logic from the load and/or store queues. In this work, we present a new approach to memory scheduling that supports a non-associative load queue, and it *completely eliminates the store queue*.

Our main technique is to predict the relative location of a store's data-dependent load (if such a load exists at all) and speculatively forward the data. After forwarding to a predicted load queue entry, the store never again worries about data forwarding and simply waits to commit and write its results back to memory: we call this "Fire-and-Forget." In addition to simplifying the memory scheduling logic, this approach enables a store to forward a value before its corresponding store address has been computed and/or before the receiving load entry's effective address has been computed thus enabling memory cloaking [16]. As a result, our Fire-and-Forget memory scheduling microarchitecture provides better performance than previously proposed techniques for simplifying the load and store queues, and at the same time Fire-and-Forget requires substantially less hardware.

The next section provides a brief review of the load and store queues and describes the most recent proposals for simplifying the hardware. Section 3 describes our Fire-and-Forget forwarding mechanism, and Section 4 provides our performance evaluation. Section 5 takes a closer look at a few benchmarks to explain why Fire-and-Forget works. Section 6 presents a summary of our findings and discusses future work in this area.

2. Background

2.1. Conventional Load and Store Queues

A conventional processor partitions the memory scheduling logic into two separate queues: the load queue (LQ) and the store queue (SQ). Load instructions wait in the load queue until the processor determines that the load's dependencies have been resolved and that the load is ready to issue. When a load issues, it retrieves a value from the data cache, and it also performs an associative search of the store queue (to be described shortly) to check if there are any earlier stores to the same address that have not yet written their results back to the cache. Store instructions wait in the store queue for their effective address and data operands. When both address and data become ready, the store broadcasts both of these to the load queue. Every valid load queue entry monitors the store broadcast bus, and captures the forwarded data value if its address matches the store's and it

comes later in program order than the store.¹ In a conventional LQ/SQ implementation, both stores searching for data-dependent children and loads searching for parents require content addressable memories as illustrated in Figure 1(a). It is important to note that the LQ/SQ CAMs are much more complex than the CAMs used in the reservation stations (RS) for scheduling non-memory instructions. In particular, the RS CAMs match on relatively small tags ($\lceil \log_2 \text{Physical RF Size} \rceil$, typically 6-8 bits) whereas the LQ/SQ typically match on memory addresses (32-64 bits), although tricks such as staggered/partial tag matching and virtual-indexing/physical-tagging can be used to reduce the overhead. Furthermore, the LQ/SQ's also broadcast and compare some form of age identifier because there may be more than one load (store) from (to) the same address.

2.2. Load Re-Execution

An alternative to the fully-associative load and store queues is *load re-execution* prior to commit. The idea is to allow loads to speculatively issue out of program order, and then re-execute the load later in the pipeline to verify the results. When a load commits, it is the oldest instruction in the processor, implying that any earlier stores have already written their results to the cache hierarchy.² Therefore, any value retrieved from the cache hierarchy at commit will be correct. If the re-executed load value matches the preliminary speculative value, then the load's dependents also received the correct value and the processor's state is correct. Otherwise, it is possible that the load's dependents have consumed incorrect values. In this situation, we flush the pipeline and refetch the instructions following the load. An interesting aspect of load re-execution is that it is a *value-based* approach [4]. Loads that execute in an order that would normally be considered as an ordering violation or incorrect with respect to the consistency model may still be treated as "correct" if re-execution reveals that the speculative *value* of the load is the same as the real value. This may occur due to Silent Stores [13] or just dumb luck (e.g., many values in a processor are zero [27], and so reading a zero from the wrong store, the wrong address, or anywhere else may still result in the correct value!).

Procrastinating the load checking until commit removes the need for a lot of the complex LQ and SQ hardware. However, naïve load re-execution forces every load to effectively issue twice. This places a large burden on the lim-

¹Some LQ designs assume that it does not hold the forwarded value. Instead the load reads it from the store queue and propagates it to its dependents. For the purpose of this work we assume that the LQ has additional storage for holding the data value. We also account for this extra storage in our hardware budget calculations.

²For a superscalar processor that commits multiple instructions per cycle, this commit-time re-execution also needs to check for any earlier stores retiring in the same cycle. However, the complexity of this logic is quite reasonable as it only needs to scale with the commit width rather than the LQ or SQ sizes.

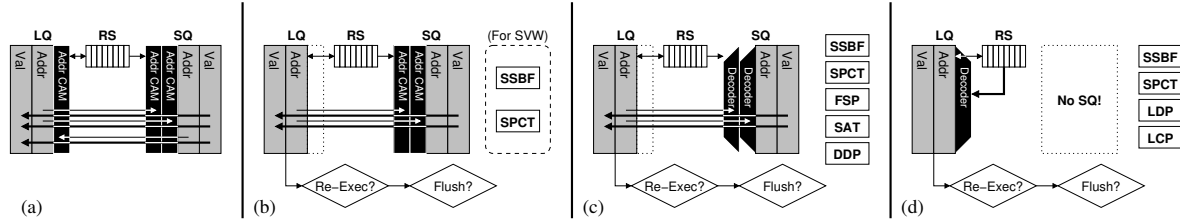


Figure 1. Memory scheduler designs: (a) conventional fully-associative load and store queues, (b) non-associative load queue with optional SVW filtering, (c) Store Queue Index Prediction, and (d) Fire-and-Forget store forwarding.

ited data cache ports and tends to have a substantial negative effect on performance due to pipeline flushes and cache port contention. Much work based on load re-execution has focused on reducing the complexity of the LQ and/or SQ while attempting to reduce the overhead of re-execution.

2.2.1. Non-Associative Load Queue

To address the power concern of the load queue's associative search, Cain and Lipasti proposed the Non-associative load queue (NLQ) [4] which uses load re-execution. When a load issues, it performs an associative search on the store queue to find any earlier stores to the same address, with a hit resulting in data being forwarded from the matching store to the load. However, a store does not “execute” in that even when its address and data are ready, the store makes no attempt to communicate this information to other instructions. The store queue entry simply holds on to its operands until it reaches commit, and then writes its value to the data cache. As a result, the NLQ approach removes all of the associative CAM logic from the LQ, as shown in Figure 1(b). The NLQ work also proposed a simple heuristic to reduce the re-execution overhead. In particular, a load that issues when there were no unresolved store addresses will have received the correct value, and therefore need not be re-executed at commit. Cain and Lipasti also proposed two additional heuristics that guarantee correct operation with respect to multi-processor consistency.

2.2.2. Store Vulnerability Window

To improve on the non-associative load queue design, Roth proposed the idea of a Store Vulnerability Window (SVW) [19]. The SVW technique is a re-execution filter based on the observation that when a load issues, the processor only contains a small number of stores that could even potentially conflict with the load. For each load, this technique tracks this small set of stores (i.e. its SVW), and a load only re-executes at commit if any store from its SVW had executed after the load. Any other store not in the load's SVW will not cause re-execution because, by construction, the store cannot have a conflict with the load. For the purposes of our work, the reader can simply consider SVW as a very effective means of reducing load re-executions when using a NLQ (Roth showed an 85% reduction).

To implement the SVW scheme, Roth proposed to assign each store instruction a unique, monotonically increasing identifier called the *store sequence number* (SSN). In practice, the SSN will have a finite bit-width and some mechanism for handling SSN overflow/wrap-around. For each dynamic load, its vulnerability window can be represented by the SSN of the youngest older store that already committed when the load was dispatched (i.e. the most recent store to which this load is not vulnerable). A store with a lower SSN (an older store) already committed its results to the cache prior to this load instruction entering the out-of-order core of the processor, and therefore the load cannot be vulnerable to this store. SVW also uses an address-based *store sequence Bloom filter* (SSBF) that conservatively filters unnecessary load re-executions. The SVW scheme also used a Store PC Table (SPCT) that performs memory dependence prediction to reduce the frequency of load-store ordering violations. It is important to note that the auxiliary structures (SSBF and SPCT) are all off of the critical load and store execution paths and therefore do not impact the scalability of the LQ and SQ.

2.2.3. Store Queue Index Prediction

SVW provides a means of implementing the non-associative load queue (NLQ) while significantly reducing re-execution overhead. However, SVW_{NLQ} (SVW on top of NLQ) still requires associative search logic in the store queue. In particular, when a load executes, it still searches the SQ to find earlier stores that have already computed their addresses from which data may be read. Sha et al. proposed a novel approach called *Store Queue Index Prediction* (SQIP) that works on top of SVW_{NLQ} [21]. Prior work on memory dependence prediction has already shown that there are stable patterns between loads and the store instructions they receive their values from [5, 15, 25]. That is, a static load that is dependent on an earlier store is usually dependent on the same static store for each dynamic instance of that load, and furthermore the relative position of the store will also be the same. As shown in Figure 1(c), SQIP replaces the SQ's costly associative logic with much

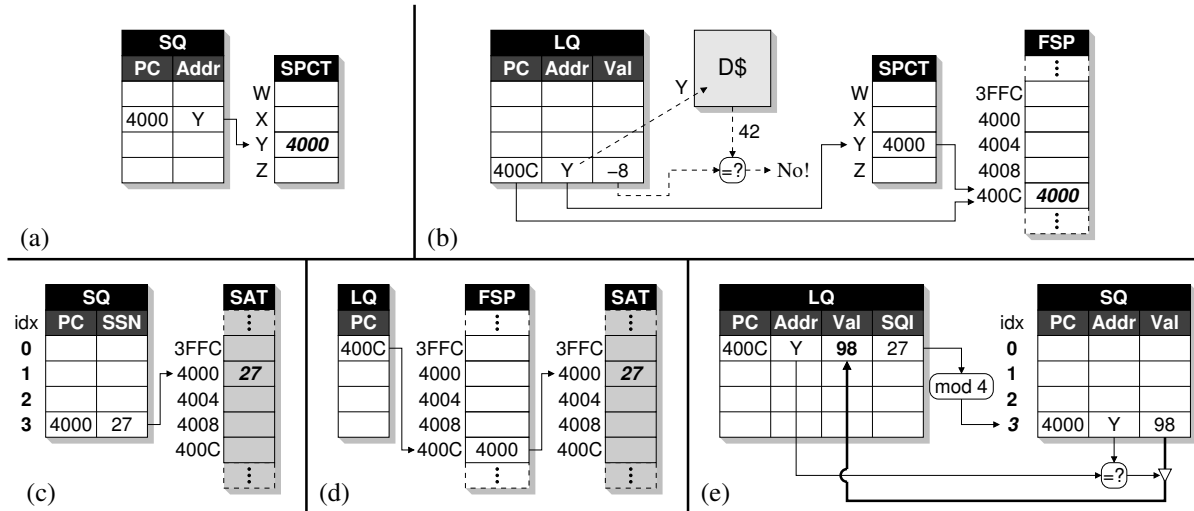


Figure 2. Store Queue Index Prediction example illustrating (a) a store updating the SPCT, (b) a misforwarded load using the SPCT to update the FSP, (c) a later instance of the same store updating the SAT, (d) a later instance of the load using the FSP and SAT to compute a SQ index, and (e) the load using the predicted index to directly access the predicted SQ entry.

simpler direct-mapped indexing.

SQIP starts with SVW_{NLQ} with all loads directly reading their values from the data cache (no SQ search). SQIP uses two tables to track loads and stores: the *Forwarding Store Predictor* (FSP) and the *Store Alias Table* (SAT). The FSP is a set-associative table indexed by the PC of a load instruction, where the entry stores a small set of partial PCs for store instructions that recently forwarded values to the load. The FSP also contains valid bits, partial tags, and a training/confidence counter. The SAT is a direct-mapped untagged table indexed by a store’s PC. A SAT entry contains the SSN of the most recent store to hash to the entry. The Store PC Table (SPCT) tracks pairs of loads and stores that have had ordering violations. Similar to the underlying SVW scheme, SQIP’s extra hardware structures (FSP, SAT, etc.) are all accessed off of the critical path of load-store execution.

An Example: For each part of the example in Figure 2, we only include the fields of the SQ and LQ entries that are used in that step. Figure 2(a) shows that when a store commits, it indexes the SPCT with the address it wrote to (Y) and records its own PC (4000). Next, a later load that should have received its data from the store re-executes at commit and reveals that it did indeed receive the wrong value (b). The load now reads the SPCT to find out which store it should have received a value from, and then records the identity of this store in its FSP entry. A later instance of the same store (c) writes its SSN into the SAT during the register rename stage. The SAT is not a large structure, but it requires additional ports and checkpointing to repair it after

pipeline flushes. When we next encounter the load (d), the load checks the FSP and finds that it had a previous conflict with the store at PC=4000. The load then accesses the SAT to find the SSN of the most recent instance of the forwarding store (SSN=27). From this SSN, the load can directly compute the store queue index (SQI) by taking the SSN modulo the SQ size (e). When the load issues, it can use the predicted SQ index to directly access a single SQ entry (i.e. no associative search). In this case, the load’s address matches the address in the predicted store queue entry, and so the load uses this value instead of the value in the data cache. Note that underneath the SQIP scheme, SVW is still running to guarantee the overall correctness of execution. Sha et al. also proposed a Delay Distance Predictor (DDP) that reduces the frequency of pipeline flushes due to difficult-to-predict loads. Overall, they demonstrate that SQIP performs within 0.6% of a realistic fully-associative SQ.

2.3. Other Related Work

Our Fire-and-Forget approach can be viewed as a logical progression in the line of load/store processing optimizations as illustrated in Figure 1(d). However, there have been other proposed techniques for optimizing the load and store scheduling structures. Address-Indexed Memory Disambiguation (AIMD) uses a structure that acts like a small, speculative L0 cache [24]. Stores speculatively write to this cache and along with sequence numbers that are similar in spirit to Roth’s SSNs, and ordering violations can be detected which forces loads to re-execute. Note that with AIMD, the load execution occurs in the main out-of-order core which requires already issued instructions to somehow

be put back into the scheduler. Also, the use of sequence numbers alone does not allow for the relaxation to the value-centric approach of commit-time load re-execution. Sethumadhavan et al. proposed a partitioned LSQ design [20]. Each segment of the LSQ still uses CAMs for address matching, but having fewer entries per segment relaxes timing constraints. They also use Bloom filters to filter unnecessary CAM activity when it can be proven that a match will not occur. Jaleel et al. propose to use a convention LSQ, but define a *virtual* load-store queue (VLSQ) comprised of a subset of the physical LSQ entries [10]. Limiting out-of-order memory scheduling to only the instructions within the VLSQ allows them to reduce the LSQ power consumption by gating the CAMs outside of the VLSQ, and overall the technique also reduces the frequency of ordering violations. Previous work has also capitalized on the observation that few stores actually forward their values. Roth [18] and Baugh and Zilles [3] independently proposed store queue organizations that partition the SQ into one piece that buffers stores for in-order retirement, and a smaller piece that forwards values to loads (the forwarding store queue or FSQ). The expensive CAM logic is limited to only the small FSQ, which keeps the hardware costs under control. Some form of load re-execution is still required to guarantee correct load execution.

A variety of other address-indexed load and/or store queues have also been proposed, for example MultiScalar's ARB [7], and the banked store buffer [26]. The difficulties with address-indexed LSQs are that address/bank/port conflicts can occur, allocation of entries is more complicated as it typically occurs at execution, and finding the right instructions to squash on a pipeline flush is not straightforward.

Sha et al. have independently proposed some concurrent work in which store-load forwarding can be performed without a store queue [22]. Their NoSQ technique makes use of speculative memory bypassing [16] to remove store-to-load communications, and as a result remove the need for a store queue. Both FnF and NoSQ techniques make use of a form of distance prediction as well as SVW-based re-execution filtering [19]. FnF and NoSQ provide two design alternatives for the elimination of the store queue and simplification of load-store scheduling in general.

3. Fire-and-Forget Store-to-Load Forwarding

We now describe our Fire-and-Forget (FnF) store forwarding design. Sha et al.'s Store Queue Index Prediction (SQIP) exploited the fact that a load which receives a forwarded value usually receives its value from the same store. In this work, we take a subtly different view of this observation: a store that forwards a value usually forwards it to the same load. While this may seem like two sides of the same coin, the slight change in perspective leads us to a different memory scheduling scheme that ultimately removes not only the

CAMs or decoders of the store queue, but eliminates the store queue completely!

3.1. Load Queue Index Prediction

Store queue index prediction takes a *load-pull* approach to store-to-load data communication. When the load has computed its address, it checks its predicted source SQ entry and then possibly pulls a value from this store. We propose a *store-push* approach that can be considered as the dual of SQIP. For each store, we make a prediction for the LQ index that the store should forward its value to.

Similar to SQIP, Fire-and-Forget starts with SVW_{NLQ} and augments it with three auxiliary tables. The first is a modified SPCT that tracks a store's PC as well as its position relative to the LQ entries. The second is the *load distance predictor* (LDP) that tracks the relative distance from a store to the load it should forward to. The last table is the *load consumption predictor* (LCP) that determines whether a load should make use of a forwarded value. In the following text, we first explain how FnF works with a stripped down SQ, and then we explain how to remove the entire SQ.

We define a *load sequence number* which is analogous to SVW's SSN. The processor assigns a unique and monotonically increasing identifier to each dynamic load. When the processor dispatches a store instruction, the store records the LSN corresponding to the most recently dispatched load (MRDL), as shown in Figure 3(a).³ Although we do not place stores in the load queue, this MRDL number indicates where in the load sequence the store would have hypothetically been inserted. When a store commits, it writes its MRDL and PC number into the SPCT, shown in Figure 3(b). Later, when a load re-executes and discovers that it received an incorrect value, it will use its effective address to check the SPCT to see if there was a recent store to this same address, shown in Figure 3(c:①). Assuming the load finds a valid SPCT entry, the processor subtracts the recorded MRDL from the load's current LSN to compute the distance in load queue entries from the hypothetical store insertion point to the consuming load (②). The processor then stores this distance in the LDP based on the *store's* PC (③). The load also updates the PC-indexed LCP so that in the future it will know that it should receive a value from a store forwarding (④). Each entry of the LCP is a single bit, and we only update the LCP on a load re-execution-induced pipeline flush. The bit is set to zero if the load should have accessed the cache, and one if the load should have used a forwarded value. When the processor next encounters this same static store, it searches the LDP and finds a non-zero

³In our terminology, "dispatch" refers to the point in the machine where resource allocation occurs and instructions are inserted their respective ROB and RS entries. In other contexts, this is sometimes called "alloc" in which case MRDL could be called MRAL instead.

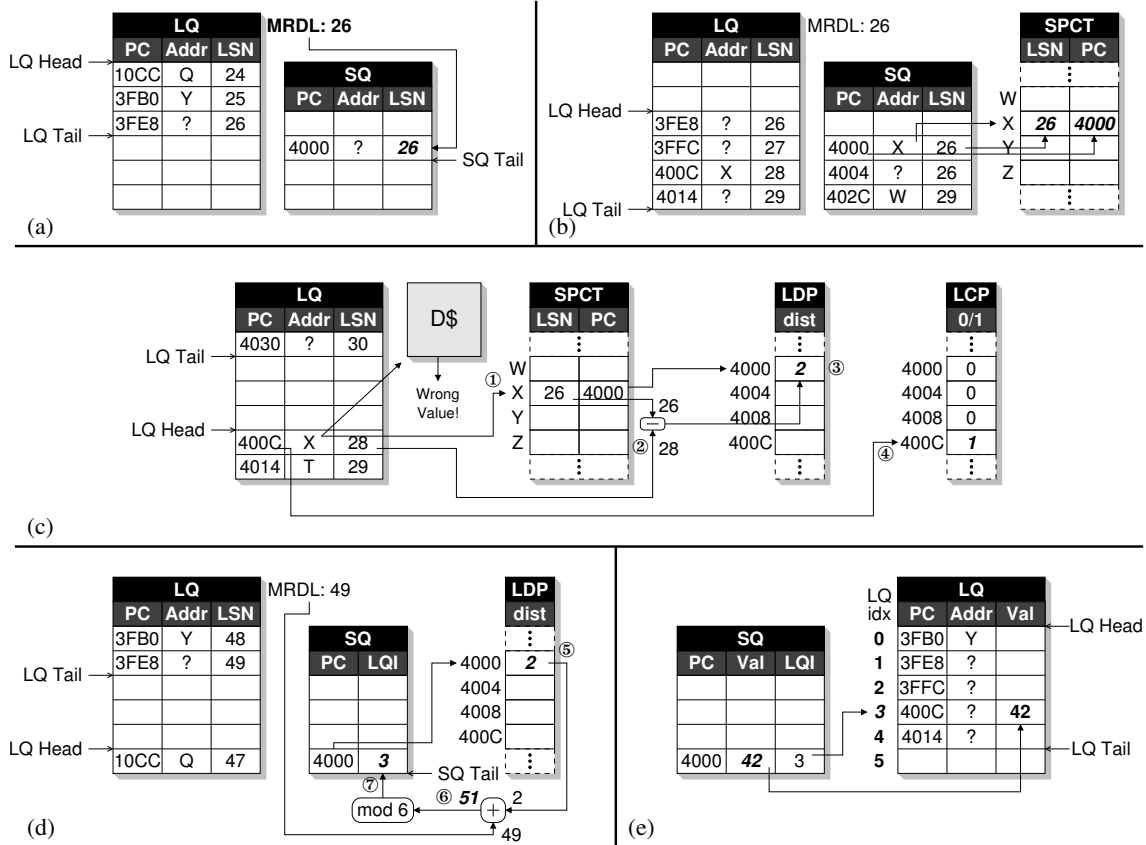


Figure 3. Fire-and-Forget example illustrating (a) a store tracking the LSN at dispatch, (b) the store updating the SPCT at commit, (c) a misforwarded load tracking its distance for future use by the store, (d) a later instance of the store computing the index for the predicted LQ entry, and (e) forwarding a value to the LQ entry.

load distance, shown in Figure 3(d:⑤). The store computes a predicted LSN to forward to by taking the current MRDL and adding the distance from the LDP (⑥). Taking this predicted LSN modulo the load queue size provides the load queue index (LQI) of the target load (⑦).⁴

As described above, FnF forwarding could be used with a non-associative SQ. A store would wait in its respective SQ entry until it is ready to issue, and then it simply forwards its data to the predicted load queue entry, shown in Figure 3(e). Note that the data movement from stores to loads is the sole responsibility of the store instructions. A load never searches, polls or otherwise accesses the store queue to try to find data (i.e. no CAMs, no decoders, no way at all for the load to access the SQ).

When a load dispatches, it first checks the LCP to predict whether it should use a forwarded value. If the value is already present, then the load can use it right away, otherwise the load stalls until a value arrives. If the LCP indicates

⁴This example uses a LQ size of 6 entries, but a practical implementation would have a power-of-two-sized LQ, thus making the modulo a trivial computation.

that the load will not receive a value from an earlier store, then the load accesses the data cache after it computes its effective address. For a load to consume a forwarded store value, it must go through two levels of “filtering”: a store must predict to forward to this load, and the load must independently predict to make use of the value. It is possible that some earlier store will attempt to forward a value to this LQ entry, but the load can always simply ignore it. If the sourcing store had a load index misprediction, or no earlier store even predicted to forward a value to this load, the load could potentially deadlock in a situation where it is waiting for a value that will never arrive. To guarantee forward progress, a load will issue to the data cache when there are no longer any earlier uncommitted stores. The regular SVW mechanism is still in place to guarantee that the load’s “final answer” is correct. To enable SVW to handle forwarded values, a store includes its SSN with its forwarded value. At commit, a load that uses a forwarded value checks the SSBF. If the SSN read from the SSBF matches the SSN that accompanied its forwarded value, then the load received its value from the correct store and re-execution is not needed.

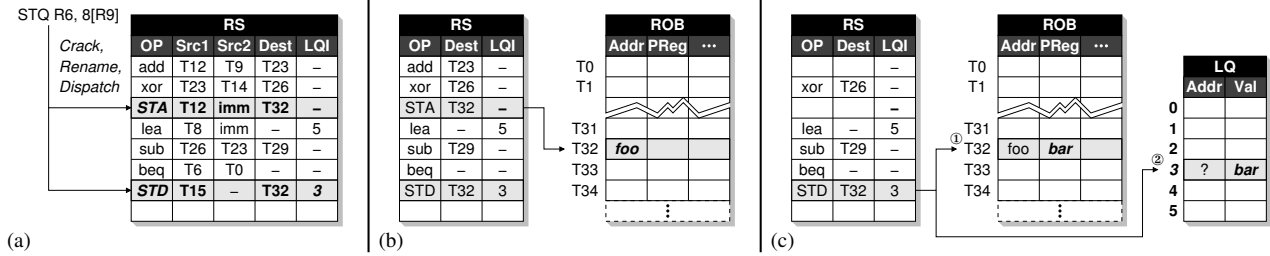


Figure 4. An example of Fire-and-Forget without a store queue showing (a) cracking and dispatch of a store, (b) store address tracking, and (c) store data tracking and speculative forwarding.

Similar to the baseline SVW_{NLQ} and $SQIP$, the three auxiliary tables/predictors used by Fire-and-Forget (SPCT, LDP, LCP) are all accessed off of the critical path of load-store execution. As a result, none of these structures affect the scalability of the core memory scheduling hardware. The SPCT needs one write port for each store that can commit each cycle. A load re-execution that reveals a misforwarded load requires the load to read the SPCT. Even though we may commit multiple loads per cycle, we only process the first detected misforwarding and therefore the SPCT only needs a single read port. Processing only a single misforwarding implies that the LDP only needs a single write port as well. The LDP needs one read port for each store dispatched per cycle, although one port can be combined with the write port because a write and read will not happen in the same cycle.⁵ The LCP requires as many read ports as loads dispatched per cycle, and as many write ports as loads committed per cycle. However, this slightly higher port count corresponds to the smallest of our three tables. Compared to $SQIP$, our FnF table management is simpler because all updates occur at commit and therefore never need any sort of recovery. $SQIP$ updates its SAT in the rename stage which requires extra ports and storage for checkpointing to support recovery after pipeline flushes.

3.2. Complete Store Queue Elimination

The previous discussion explained why FnF removes the need for SQ access circuitry for loads. At this point, we question what benefit the remaining logic in the store queue actually provides. Load re-execution guarantees that loads eventually receive the correct value, and FnF provides a means of forwarding values from stores to loads, and therefore the SQ only buffers the stores in program order for in-order writeback to the cache. There is another structure in the processor that already tracks all instructions in program order: the reorder buffer (ROB). Since the ROB maintains a list of *all* instructions in program order, it necessarily in-

⁵A write to the LDP only occurs after a misforwarding, which implies that the processor has been flushed and several cycles must elapse before any new stores have been fetched and are ready to read the LDP.

cludes all stores in program order. We still need to buffer the results of the stores. In some microarchitectures (e.g., Pentium-Pro/Pentium-M [8, 23]), each ROB entry contains a physical register to store the result of that instruction. Normally, this physical register goes unused for store instructions because the store's value is kept in the corresponding store queue entry. In FnF, we propose to make use of this otherwise neglected resource to record the store's value.⁶ After this change, *all* of the SQ's functions have been outsourced to other mechanisms, and so we can completely do away with the store queue!

To properly execute store instructions, some hardware resources are still required. Similar to a store's address-and-data (STA-STD) μop decomposition in modern x86 microarchitectures, we allocate two separate scheduler (RS) entries for each component of the store instruction, as shown in Figure 4(a).⁷ In this example, we assume that an instruction's destination physical register is integrated into its ROB entry. When the STA μop executes, it computes its effective address and saves it in its ROB entry for its eventual cache update at commit (b). When the STD μop executes, it also writes its value to the ROB entry for eventual writeback (c:①). In addition, if the store has a predicted load to forward to, it uses its load queue index and performs a direct write (RAM) to the corresponding load queue entry (②). This speculative forwarding occurs in a completely blind fashion: the destination LQ entry may not have predicted to wait for a forwarded value. This is the one and only chance a store has to directly forward a value to a load and afterward makes no other effort for proper forwarding.

In the FnF mechanism without a store queue when a store-data issues it forwards a value to the predicted LQ index and then forgets about it. This value needs to be stored somewhere. In designs that assume that the LQ includes storage for forwarded values [19] FnF does not incur any

⁶For a microarchitecture that uses a separate physical register file such as the Intel® Pentium 4 or the Alpha 21264, we could allocate a physical register to the store without updating any RAT entries.

⁷The Pentium-M's μop fusion could be used to pack both STA and STD into the same RS entry, but we do not explore this optimization in this work.

