Filter Caching for Free: The Untapped Potential of the Store-Buffer

Ricardo Alves
Uppsala University
ricardo.alves@it.uu.se

Alberto Ros
Universidad de Murcia
aros@ditec.um.es

David Black-Schaffer
Uppsala University
david.black-schaffer@it.uu.se

Stefanos Kaxiras
Uppsala University
stefanos.kaxiras@it.uu.se

ABSTRACT

Modern processors contain store-buffers to allow stores to retire under a miss, thus hiding store-miss latency. The store-buffer needs to be large (for performance) and searched on every load (for correctness), thereby making it a costly structure in both area and energy. Yet on every load, the store-buffer is probed in parallel with the L1 and TLB, with no concern for the store-buffer's intrinsic hit rate or whether a store-buffer hit can be predicted to save energy by disabling the L1 and TLB probes.

In this work we cache data that have been written back to memory in a unified store-queue/buffer/cache, and predict hits to avoid L1/TLB probes and save energy. By dynamically adjusting the allocation of entries between the store-queue/buffer/cache, we can achieve nearly optimal reuse, without causing stalls. We are able to do this efficiently and cheaply by recognizing key properties of stores: free caching (since they must be written into the store-buffer for correctness we need no additional data movement), cheap coherence (since we only need to track state changes of the local, dirty data in the store-buffer), and free and accurate hit prediction (since the memory dependence predictor already does this for scheduling).

As a result, we are able to increase the store-buffer hit rate and reduce store-buffer/TLB/L1 dynamic energy by 11.8% (up to 26.4%) on SPEC2006 without hurting performance (average IPC improvements of 1.5%, up to 4.7%). The cost for these improvements is a 0.2% increase in L1 cache capacity (1 bit per line) and one additional tail pointer in the store-buffer.

CCS CONCEPTS

• **Computer systems organization → Superscalar architectures**; **Pipeline computing**; **Multicore architectures**.

KEYWORDS

store-buffer, filter-cache, single thread performance, memory architecture, energy efficient architecture

ACM Reference Format:
Ricardo Alves, Alberto Ros, David Black-Schaffer, and Stefanos Kaxiras. 2019. Filter Caching for Free: The Untapped Potential of the Store-Buffer. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3307650.3322269

1 INTRODUCTION

The store-buffer (SB) is a sine qua non for high-performance processor implementations that allow stores to retire under cache misses. It is so important that all prevailing memory models today, including Total Store Order (TSO), relax the store→load order for the express purpose of accommodating the store-buffer. Under TSO (and consequently any weaker memory model that relaxes the same order), performance-critical loads are allowed to bypass committed stores that are waiting in the store-buffer to be inserted in the memory order, i.e., written to the L1. On the other hand, the store-buffer is intentionally sized and *managed* in a way that keeps its occupancy low, so that it does not induce processor stalls from being full on future cache misses.

To maximize the use of available resources, the store-buffer (SB), which holds stores between commit and when they are written to memory, is often unified with the store-queue (SQ), which holds stores from dispatch to commit. The resulting unified SQ/SB allows for better overall utilization of the expensive CAM-FIFO needed to support searches by load address and enforce store→store ordering, and eliminates the cost of moving entries between separate queues.

To enforce sequential execution semantics, load instructions must probe the SQ/SB searching for the youngest store (but older than the load) to the same address that has not been written to memory yet. If found, the value of the store takes precedence over any value that may be in the L1 or elsewhere in the memory hierarchy. However, delaying the access to the L1 to perform this search would unduly delay all L1 accesses, even when there is no such store found in the SQ/SB. To avoid a performance loss on all loads, the SQ/SB is commonly probed *in parallel* with the L1/TLB access.

A hit in the SQ/SB, makes the L1/TLB access irrelevant and the value found in the SQ/SB is forwarded to the load (store-to-load forwarding). Conceivably, the access to the L1 could be discarded as soon as a hit in the SQ/SB is detected, but the damage in performance (interference in cache ports) and energy (TLB and L1 tag access) would already have been done.

For efficiency, we would like to predict whether we are likely to find the correct value in the SQ/SB or the L1, and thereby only probe one structure. If a reliable prediction indicates that the correct

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06.
https://doi.org/10.1145/3307650.3322269

value is to be found in the SQ/SB, we can serialize the access to the L1/TLB and save the energy penalty of an irrelevant access, without paying the latency penalty of the serialization when we do not hit in the SQ/SB. Yet today's designs try to empty the SQ/SB as rapidly as possible, which explicitly reduces the chance of hitting in the SQ/SB. This approach reduces the potential benefit of predicting SQ/SB hits and using the prediction to avoid irrelevant L1/TLB probes. In this paper we address the conflict between emptying the the SQ/SB quickly to avoid increasing latency on write-misses and keeping it full to use hits in the SQ/SB to reduce irrelevant L1/TLB probe energy.

This work makes three main observations. First, although expensive CAM-based SQ/SBs are made as large as possible to prevent processor stalls when full (e.g., the Intel Skylake processor employs a combined 56 entry SQ/SB), they typically remain underutilized. Second, since the SQ/SB has to be probed on every load instruction (to allow store-to-load forwarding), the SQ/SB acts as a filter or L0 cache [4, 14, 19]. Third, the hit ratio of existing SQ/SB's is kept intentionally low (8%) due to aggressive write back policies that lead to under-utilization (first observation).

From these observations, we explore the potential benefits of keeping data in the SQ/SB, and determine that an ideal approach could reduce L1/TLB probes by 14% on average. We then develop an essentially zero-cost approach (a second dirty bit per L1 cache line is all we require) that allows us to keep the SQ/SB full to obtain these benefits. We do so in two steps: First, by keeping SB entries that have been written to cache around as long as the SQ/SB is not full, thereby increasing its hit ratio, and, second, by efficiently predicting when the data will be found in the SQ/SB so that the L1/TLB probes can be avoid to save energy.

Increasing SQ/SB hits. Keeping stores that have already been written to memory in the SQ/SB is free in terms of storage, as we simply leverage the unused portion of the SQ/SB, and free in terms of data movement, as the data has already been installed in the SQ/SB for correctness. We name the part of the SQ/SB that keeps already performed stores the *Store-Buffer-Cache* (SBC) and the new shared structure SQ/SB/SBC, *S/QBC* (Figure 1). This allows us to maximize the use of the expensive CAM storage by keeping it full with a combination of entries from S/QBC, and only requires adding a head pointer to track the SBC portion. As a result, we can improve its hit ratio without increasing the likelihood of processor stalls, since all stores in the SBC part have already been written to memory, and can therefore be immediately removed when more space is needed. Our key observation here is that the SQ/SB is already paying the data movement and capacity overheads of a filter cache, but without any energy benefit on hits. (Section 4.1)

S/QBC coherence. The data in the SBC must be coherent with the data in memory as they have already been written to memory and another processor can modify them, resulting in a stale copy in the SBC. A central contribution of our work is proposing a highly-effective, low-cost mechanisms for achieving such coherence. (Section 4.3)

Avoiding L1/TLB probes. While leveraging the unused portion of the SQ/SB as a store-buffer-cache increases the hit ratio, to achieve energy benefits we need to avoid the parallel L1/TLB probe on S/QBC hits. For correctness we must always check the S/QBC, but we can afford to serialize the access to the L1/TLB if we expect to hit in the S/QBC. In contrast, when we expect to miss in the S/QBC, we can start the parallel access to the L1/TLB, as in current practice, so as to not penalize performance. This requires an early prediction of the chances of hitting in the S/QBC. Our key observation here is that such hardware already exists in out-of-order cores in the *memory dependence predictor* [10, 25, 40], and we simply use it (without loss of generality) to select between serial or parallel access to the S/QBC and the L1/TLB. (Section 4.4)

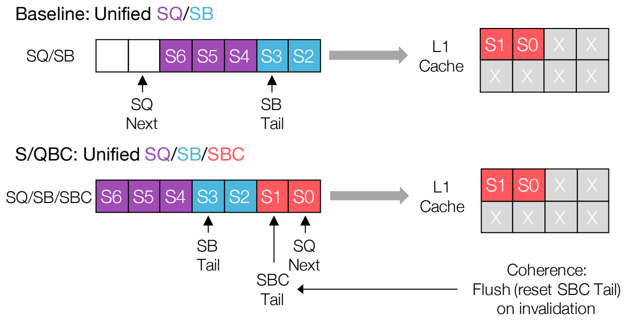


Figure 1: The Store Buffer Cache extends a unified SQ/SB with a third logical cache partition (SBC) that holds copies of data that has already been written to the L1 to increase store buffer hits. This data can be immediately and silently evicted when space is needed (so it does not increase stalls) but needs to participate in coherence.

Our results show that by using the empty portion of the SQ/SB as a store-buffer-cache, we can keep stores around for long enough to improve the hit ratio from 8.1% to 18.1%. Using the the CPU memory dependence predictor to choose between serial or parallel probing of the S/QBC and L1/TLB, the S/QBC/TLB/L1 cache dynamic energy can be reduced by 11.8%. Our design achieves this essentially for free: We leverage the existing memory dependence predictor and SQ/SB capacity and add only 1 bit per L1 cache line (0.2% storage increase) and one additional SQ/SB head pointer. Moreover, we achieve this energy reduction without impacting ability of the SQ/SB to reduce latency, and deliver 1.5% average IPC improvement on SPEC2006. Our contributions are:

- We identify that writes in the SQ/SB are paying the energy and area costs of a filter cache, but are not seeing savings on hits.
- We identify that L1/TLB accesses can be avoided on SQ/SB hits using the existing memory dependence predictor, but that current SQ/SB hit ratios are too low to benefit from this.
- We determine the potential reuse available through the SB/SQ and propose a third logical cache partition in the SB/SQ, the Store-Buffer-Cache, that can obtain 99% of the reuse.
- We identify that copies of writes in the SQ/SB can be kept coherent very cheaply by tracking epochs of dirty data in the L1, and develop an extremely-low overhead multi-dirty-bit coherence implementation for the Store-Buffer-Cache.
- We combine these insights to develop the Store-Buffer-Cache, a nearly zero-cost design that saves 11.8% of SQ/SB + L1/TLB energy with no performance loss (actually having an modest improvement of 1.5%).

2 BACKGROUND

2.1 The Store-Queue/Store-Buffer (SQ/SB)

CPUs with an out-of-order execution pipeline implement a several structures to keep track of the original program order. The *store-queue* (SQ), in particular, is responsible for keeping track of the original order of store instructions. Its purpose is twofold: (1) to keep track of the stores' original order so that they are committed to memory in that same order, and, (2) to forward data to load instructions that address the same memory location of an uncommitted store, thus guaranteeing that a load always accesses the most recent value.

A common challenge is that stores that are ready to commit may be stalled due to cache misses or contention. Such delays block the ROB and may stall the pipeline. To allow stores to retire in these conditions, a *store-buffer* (SB) is used to track stores that have committed but have not yet been written back to memory. When entries in the SQ have been committed, they are moved to the SB until they are written back to memory (typically to the L1).

The store-queue (SQ) and store-buffer (SB) are generally implemented in a unified physical structure called the SQ/SB. The unified approach means that the distinction between entries in the SQ and SB is purely logical: stores that are not yet committed are in the SQ portion, and stores that are committed but not yet written to memory are in the SB portion. This allows for a more efficient implementation, as there is no need to copy between separate buffers on commit (moves simply require changing head/tail pointers as all moves are in store order) and either the SQ or the SB size can increase up to the maximum capacity. The higher utilization by sharing capacity between the SQ/SB is important as the structure is implemented as a FIFO (to support writing to memory in-order as required by widely-supported memory models such as Total Store Order —TSO), but requires CAM access (to allow searches by address for later loads). As a result, the cost of this structures is high, but it must be large enough to handle bursts of write misses that would otherwise stall the processor.

Too avoid increasing load latency, loads probe the SQ/SB and L1 cache in parallel. If the address matches a store in the SQ/SB (i.e. a SQ/SB hit), the data is forwarded from the youngest store that matches the address, and the in-flight L1 cache request is ignored[1]. In addition, since L1 caches are generally physically tagged, the load address has to be translated, requiring a parallel access to the TLB as well.

2.2 SQ/SB Utilization and Hit Ratio

The relatively small size of the SQ/SB, combined with its aggressive eviction policy (designed to keep it as empty as possible to avoid stalls) results in a low utilization and a low hit ratio. Figure 2 demonstrates this low-utilization for a 56-entry SQ/SB across the SPEC2006 benchmarks. While the SQ/SB is highly-utilized (>80% full) at some point in all benchmarks, the majority of the time the buffer remains largely un-utilized (<40% full). Indeed, the average benchmark uses 20% or less of the SQ/SB for 62% of its execution, and 40% or less for 85% of its execution.

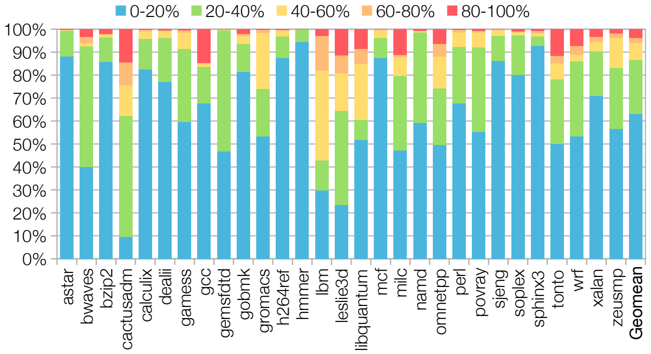


Figure 2: SQ/SB occupancy across benchmarks. All benchmarks have high SQ/SB utilization (>80%) at some point, while the majority of the time it experiences low utilization (<40%). Simulation methodology is explained in Section 5.1.

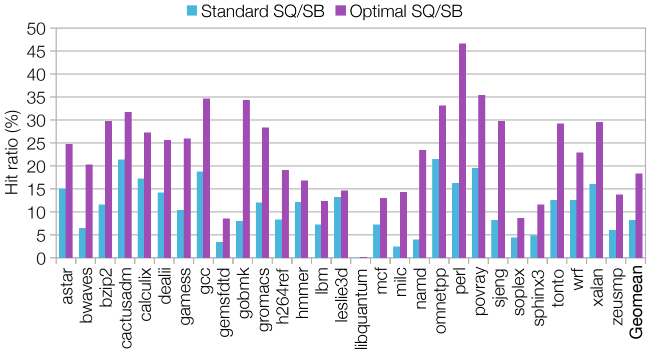


Figure 3: Percentage of loads that hit in the SQ/SB for: 1) A Standard SQ/SB that aggressively writes out to memory, and 2) an Optimal SQ/SB that maximizes hits by keeping values around exactly as long as possible without stalling the processor.

Figure 3 shows the percentage of loads (hit ratio) that receive their data from the Standard SQ/SB (aggressively writing back data) for SPEC2006 with a Skylake-like 56-entry unified SQ/SB. (Configuration details in Section 5.1.) While *cactusadm*, *omnetpp*, and *povray* have significant SQ/SB hit ratios (21.4%, 21.5% and 19.6%, respectively), most applications have hit ratios around or below 10%, and the overall SPEC2006 average is only 8.1%. This is not surprising given the utilization, but is very low for a typical cache.

The low hit ratio suggests that (1) programs are unlikely to benefit from the lower latency of data forwarded by the SQ/SB as the majority of the loads experience the longer L1 latency on SQ/SB misses, and, that (2) today's approach of probing both the SQ/SB and the L1/TLB in parallel is reasonable, as most data requests will miss in the SQ/SB and have to access the L1/TLB anyway. With this SQ/SB hit ratio, serializing SQ/SB→L1/TLB accesses would increase the latency for the 92% of the accesses that miss in the SQ/SB and only provide an energy benefit for 8% that hit. Thus, although the SQ/SB could provide energy benefits by filtering accesses to the L1/TLB, its poor hit ratio justifies today's approach of parallel accesses.

[1] As mentioned in the introduction, it might be possible to squash the L1 request at this point, but the cost of port contention and tag/TLB access has already been paid.

2.3 Filter Caches and the SQ/SB

Filter caches [4, 14, 19] add a very small cache between the CPU and the L1, typically in the range from a few cache lines with high associativity up to a few dozen direct-mapped lines. Because of their small size, a hit in a filter cache is inherently faster and more energy efficient than a hit in the L1. While their lower latency is unlikely to translate into performance gains, as it will typically be covered by OoO execution, the reduced access energy can still deliver efficiency benefits by filtering accesses to the L1. Unfortunately, the small capacities of filter caches often result in extremely low hit rates, which incur the additional energy and latency of probing the filter cache and probing and copying from the L1. For low hit rates, this overhead can be worse than directly accessing the L1 and actually increase the memory access energy and latency [3].

Intriguingly, if we consider using the SQ/SB as a filter cache *for writes*, the low hit rate does not incur an additional energy cost, as all stores must be installed in the SQ/SB and it must always be probed for correctness. That is, *the SQ/SB is already paying the probe and copy energy overheads of a filter cache*, but by choosing a policy that empties it as aggressively as possible, we are reducing the chance of hits. This differs from addressing loads in the load queue, as it does not store the load data, using it as a cache requires both additional storage and data movement energy [27].

3 MOTIVATION AND POTENTIAL

To understand the potential of the store-buffer as a cache, we need to identify how much locality it can deliver, both as a function of its size and its write back policy.

3.1 Maximizing the SQ/SB Hit Ratio

We implemented an Optimal SQ/SB that delays writes from the SB to the L1 as long as possible without hurting performance. The Optimal SQ/SB models an instantaneous write back to the L1 from the SB that is triggered as soon as new entries in the SQ/SB are needed. This allows us to see the potential for hits in the SQ/SB.

Figure 3 shows that with the Optimal SQ/SB, *perl*, *povray* and *gobmk* now have the highest hit ratios of 46.6%, 35.4% and 34.4%, respectively, increases of 2.9x, 1.8x and 4.3x over the Standard SQ/SB, with its aggressive write back policy. On average, the hit ratio increase to 18.4% from 8.2%, a 2.3x improvement over the Standard SQ/SB.

Figure 4 shows the potential of the Optimal SQ/SB to reduce L1/TLB accesses (assuming perfect hit prediction) and the resulting SQ/SB+L1/TLB dynamic energy savings. Taking perfect advantage of maximal SQ/SB locality would filter an average of 15.5% of the L1/TLB accesses (up to 31.5% on *perl*) and save an average of 13% of the dynamic energy (up to 28.7% on *perl*).

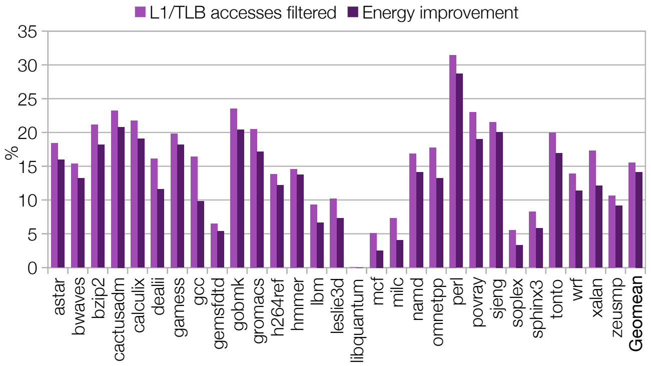


Figure 4: Potential accesses to the L1/TLB that can be filtered with an optimal SQ/SB write policy and the resulting SQ/SB+L1/TLB dynamic energy improvement. (Higher is better.)

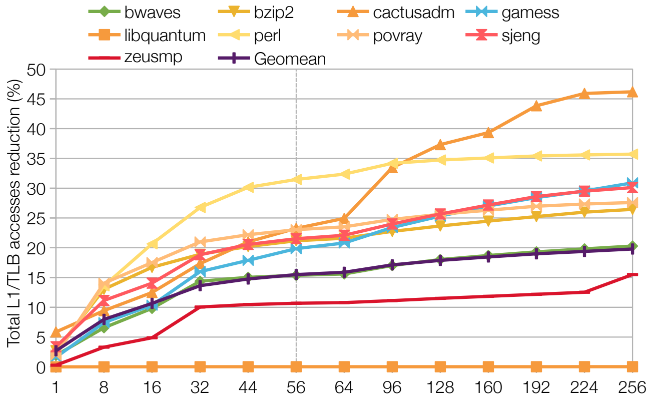


Figure 5: Percentage of memory accesses filtered by a SQ/SB with an optimal (delayed write back) policy as a function of size. Note the non-linear X-axis that highlights sizes 44 and 56, which are AMD Zen's and Intel Skylake's SQ/SB sizes, respectively. Only selected benchmarks are shown for clarity, while the mean includes all SPEC2006 benchmarks.

3.2 Sensitivity to Store-Buffer Size

Figure 5 explores the *total percentage of memory accesses the SQ/SB can filter* as a function of size for an Optimal SQ/SB. This metric includes the effect of load/store mix in the application, as only loads can hit in the SQ/SB. The average percentage of filtered accesses across SPEC2006 (purple line, Geomean) increases only slightly from 15.5% at our baseline size of 56-entries to 19.7% at 256-entries, despite the 4.5x increase in queue size. *Catusadm* stands out as the exception, with a significant increase for sizes of 96 and up. However, designing FIFO-CAMs of that size is a challenge, and has been shown to incur performance overheads [35]. As most of the locality appears to be captured by size 56, and it is typical of modern processors, we choose it for the remainder of our experiments.

4 THE STORE-BUFFER-CACHE

To use the SQ/SB to reduce L1/TLB accesses we need to (1) improve its hit ratio without increasing CPU stalls from running out of capacity during store-misses, and, (2) avoid accessing the L1/TLB on SQ/SB hits.

4.1 The Cache Portion of the SQ/SB

The first step in making the SQ/SB into an effective cache is to improve its hit ratio. Simply delaying write-backs from the SQ/SB is not ideal as it could increase CPU stalls due to the decreased capacity available to the SQ to handle bursts of writes.

To delay writing stores in the SQ/SB to memory without increasing CPU stalls, one would need to predict when more capacity will be needed in the SQ/SB in time to write out enough entries to free that capacity. This requires: (1) accurately predicting store-misses, (2) doing so sufficiently early to perform enough writes to the L1 to free enough entries for the new stores, and (3) predicting how many entries are required to hide the store-miss latency, to avoid writing back too many entries. Building such a predictor would be a challenge. Instead, we propose using a portion of the unified SQ/SB storage as a *Store-Buffer-Cache* (SBC), and we refer to this unified structure as the S/QBC and the *logical portion* that holds copies of written out data as the SBC.

Instead of delaying stores in the SB to increase hits, the S/QBC writes stores to the L1 as soon as possible, as in a traditional SB. However, a *copy* of the store is kept in the SBC. The S/QBC uses the same storage as a traditional unified SQ/SB, but now with the three logical partitions: the SQ holds not yet committed stores, the SB holds committed, but not yet written to L1 stores, and the SBC holds copies of committed stores that have been written to the L1. We implemented the S/QBC storage as a circular FIFO queue, where the head of one queue precedes the tail of the next. Thus the movement of stores between the different queues is simply a pointer increment, and no physical copying is required. This is possible because writes to the L1 are performed in FIFO order, and therefore every new entry added to the SBC will have been the oldest entry in the SB. As a result, the SBC is effectively free, as moving data from the SB to the SBC does note require a copy and all stores are already written into this shared structure when they were installed in the SQ for correctness.

The S/QBC is able to keep the results of stores as long as possible as evictions only happen when a new entry is needed. Since the stores in the SBC were already written back to memory, evictions can be done silently and immediately whenever space is needed for new writes in the SQ. This maximizes hits in the S/QBC, by keeping data around as long as there is space, without causing extra CPU stalls due to lack of available space for new stores, but means that the copies in the S/QBC need to address synonyms and be kept coherent.

4.2 Store-Buffer-Cache Synonyms

A translation from virtual to physical address is required to detect possible incorrect forwardings due to synonyms, even on S/QBC hits. We can elide the traditional TLB access on S/QBC hits since the load queue (LQ), SQ, SB and SBC hold both virtual and physical addresses [21]. A load that matches a virtual address from the SQ, SB or SBC, can copy the same physical address of the matching store entry as well: virtual-to-physical mapping is one-to-one. One-to-many mappings (homonyms) are avoided by the operating system.

A load hit on an SBC entry whose physical address has not yet been retrieved requires only a single TLB access to translate both the earlier store and the later load. This eliminates the need for a second TLB access for the later load. Moreover, as noted by Lustig et al. [21], some loads can get their data from a store using only virtual addresses, if it can be guaranteed that no synonym exists in between them in program order. The S/QBC can therefore ensure that all load hits will have correct physical addresses, even though they do not need separate TLB accesses.

4.3 Store-Buffer-Cache Coherence

Keeping clean copies of the data in the S/QBC creates a coherence problem: As the store has already written its data out to the L1, any other core can access the data block and modify it unbeknownst to the S/QBC. Hits in the S/QBC in such cases return incoherent values. Lack of coherence is devastating for a consistency model such as TSO, but also needs to be addressed in weaker models in relation to memory ordering fences. In this work we address the coherence problem for TSO and we discuss its handling on weaker models. We assume a MESI invalidation protocol, but our approach can be easily adapted to more complex protocols (MOESI/MESIF).

A naïve solution is to forward *any* invalidation that reaches the L1 and *any* L1 eviction to the S/QBC. In this way, we could selectively invalidate individual entries in the SBC portion of the S/QBC. This is already done for the load queue: invalidations and evictions search the load queue for speculative loads that violate consistency ordering and squash them. Selectively invalidating individual entries in the S/QBC would be energy-expensive, as it would require additional CAM ports for searching, and complex, as it would require compacting the entries in the SBC to recover the capacity of invalidated entries. At the other extreme, the simplest approach is to bulk-flush the SBC on *any* invalidation or eviction from the L1. Such an approach does not require any associative searches, but wipes out all SBC entries, and would thereby reduce the S/QBC hit ratio.

We can simplify how we handle coherence by noting that stores are treated differently from loads in the coherence domain. Specifically, a store that is written to the L1 implies that the local L1 has *ownership* of the data block (in a MESI protocol the cache line is in state `MODIFIED`) because its data are *dirty*. As a result, L1 invalidations or evictions of `EXCLUSIVE`/`SHARED`, *clean* cache lines are *irrelevant* to the SBC, and can be ignored. However, if we lose ownership of a cache line, either through an invalidation, an eviction, or simply because of a read request from another core that forces the local cache line to downgrade to `SHARED` (and become clean), then we are no longer able to detect that the coherence actions on that line affect our SBC. In this case one of our own prior stores has been affected by a coherence action, and *if a corresponding clean copy exists in the store-buffer-cache, it must be stale.*

More specifically: (1) An invalidation reaching a `MODIFIED`, *dirty* cache line, means that another core is writing the cache line and therefore a copy of the data in the SBC is now stale. (2) An eviction of a `MODIFIED` cache line does not necessarily mean that a copy in the SBC is stale, but we lose the ability to track any future changes to the data block (we will not get an invalidation if it is written in the future) and therefore we should also remove the copy from the SBC. (3) If we receive a read request from another core and downgrade to `SHARED` (writing back the dirty data and going to a clean state), there is the potential again to lose track of any future changes to the data block, as a `SHARED`, clean copy can be silently conflict-evicted. Across these cases, the key property that governs the validity of the copies in the SBC is the *local ownership* of the cache line, or, equivalently, *holding the cache line in a dirty state in the private L1.*

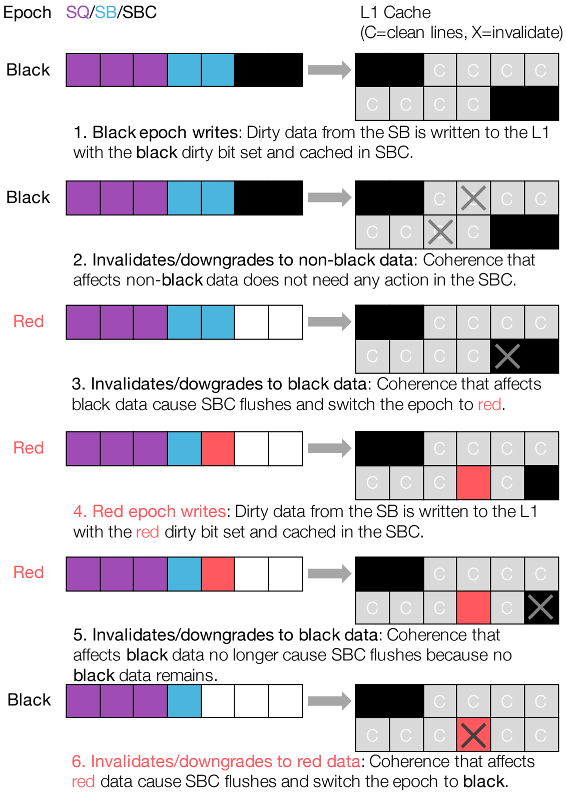
Any action that effectively downgrades the ownership or cleans the cache line is a cause to invalidate the corresponding data in the SBC. (Note that copies in the SBC portion of the S/QBC are always "clean" with respect to the L1, as they can only be cached in the SBC portion once they have been written to the L1.)

Since we now have restricted the cases where we must react, we can relax the the specificity of our reaction: instead of invalidating specific data in the SBC (which would require an associative search and compaction) we opt to bulk-flush all cached data in the SBC, but only under these more restricted circumstances. In our unified S/QBC, such a bulk-flush simply requires moving the head SBC pointer to coincide with the head SB pointer. This is the cheapest method to enforce coherence in the store-buffer-cache: it requires no change in the L1, and just a lone signal from the L1 cache controller to the S/QBC to reset the SBC pointer when the L1 cache loses local ownership of a cache line[2].

Although restricting bulk-flushes to coherence changes to cache lines that are locally owned provides correctness at a reasonable performance, it is overly conservative as cache lines tend to live in the cache much longer than cached data in the S/QBC. It might very well be the case that a downgrade of a locally owned cache line corresponds to a very old store that left the S/QBC a long time ago. As a result of this difference in the recency of data in the SBC and L1, a bulk-flush of the current cached data in the SBC is extreme. On the other hand, associatively searching the cached data in the S/QBC for a specific address requires additional CAM probe ports and compaction or loss of capacity. To address this, we need to incorporate a notion of *recency* in our design in an effective and low-cost manner.

Multicolored dirty bits: The problem we seek to address is that coherence events on much older, locally owned cache lines in the L1 force flushes of the much more recent data in the SBC. We can address this by coarsely tracking the age of dirty data with the use of a dirty bit *of a different color*. On L1 coherence events that require SBC flushes, we need only flush the SBC entries that have the same color dirty bit. By periodically switching the currently active dirty bit color for new writes, we can avoid having to flush the writes in the SBC that are using the current color when lines with an older color are downgraded.

The simplest example is to assume two dirty bits (per cache line) of different colors: a red dirty bit and a black dirty bit. Only one of them can be set at any point in time, and either means that the cache line is dirty. The SB operates in red periods and in black periods. When it is in a red period it sets the red dirty bit when writing in the cache, and vice versa for black periods. The SB changes from a red period to a black period and back on any bulk-flush caused by a coherence action. In addition, the bulk-flush signal from the cache controller indicates whether it is a red or black dirty line that is experiencing a downgrade. We then institute the following policy: when a downgrade happens in the cache for a cache-line of a specific color we cause a bulk-flush in the SBC only if it is in a period of the same color, otherwise we ignore it. An example of the operation with two dirty bits is given in Figure 6. Two colors give us a restricted sense of recency in our actions, and reduces the number of SBC entries that need to be evicted.



1. Black epoch writes: Dirty data from the SB is written to the L1 with the black dirty bit set and cached in SBC.

2. Invalidates/downgrades to non-black data: Coherence that affects non-black data does not need any action in the SBC.

3. Invalidates/dowgrades to black data: Coherence that affects black data cause SBC flushes and switch the epoch to red.

4. Red epoch writes: Dirty data from the SB is written to the L1 with the red dirty bit set and cached in the SBC.

5. Invalidates/dowgrades to black data: Coherence that affects black data no longer cause SBC flushes because no black data remains.

6. Invalidates/downgrades to red data: Coherence that affects red data cause SBC flushes and switch the epoch to black.

Figure 6: Operation of the S/QBC with red/black dirty bits. Only coherence actions on L1 data that affect dirty data from the current epoch result in flushes of the SBC and a color change.

More specifically: if the SB is in a black period, a downgrade of a black cache line from the L1 that is related to the writes in the SBC will be black. This ensures correctness. However, a downgrade to a black line in the L1 may also be much older, in which case there is no correctness concern, but we unnecessarily flush the buffer-cache. If the downgraded L1 cache line is of a red color, then it certainly corresponds to a store of the previous period (or any other older red period) that was bulk-flushed in the most recent switch to the black period. This means that we can safely ignore all red downgrades.

The two-color example can be generalized to any number of "colors" (or *epochs*) to trade-off the overhead of tracking and the risk of unnecessary flushing. Note that with two dirty bits we can encode the clean state (e.g., {0,0}) and three colors. In the steady state, after many switches from color to color, we expect that the cache will contain a mixture of dirty blocks in all three colors and, on average, we expect to be able to ignore two-thirds of the downgrades that are of different color than the current period. With n bits we can encode the clean state and $(2^n) - 1$ colors, thereby expecting to reducing the SBC flushes to $1/((2^n) - 1)$ of the naïve approach with only a single dirty bit. We show in Section 5 that in practice just three colors captures most of the the potential.

[2] To avoid vulnerability windows the bulk-flush, it must be acknowledged by the L1 before the handling of the coherence action on the part of the cache controller.

Alternatively, we can approximate unlimited number of colors with only two dirty bits if our cache architecture supports selective flush-reset of the dirty bits. In this case, we only need two colors: Red, indicates that the line is dirty and it *might* be in the SBC, and Black, indicates that the line is dirty and *cannot* be in the SBC. All writes leaving the SB are marked as red, to indicate that they may still be in the SBC. When we get an invalidation/eviction on a red line from the L1, we flush the SBC and *selectively flash-reset all red lines in the L1 to black*. Since we flushed the SBC, none of the previously red lines in the L1 can still be in the SBC. Now, we are susceptible to evictions/invalidations only from the current period. The trade-off of this approach is that it approximates infinite colors with no additional dirty bit overhead, but requires support to selectively flash-reset. The circuit for doing this needs to be applied independently to each dirty bit pair, but it consists of only one AND gate. Designers can select the approach that best fits their L1 architecture.

Coherence for weaker memory models: The above approach can be used to provide a correct coherence substrate for any memory model weaker than TSO. However, in weaker models some incoherence may be tolerated by the model itself, as other examples of coherence protocols such as DeNovo [9] or VIPS [31] have shown, in relation to Data-Race-Free (DRF) software. In such cases, the coherence of the clean data in the SBC can be tied to memory ordering fences specified by the consistency model. For example, in Release Consistency (RC) there is no strict requirement to see the latest value written by another core, unless we cross an Acquire fence. In such a case, the clean data of the SBC should be invalidated by the fence. In this work we tackle the (harder) problem of the stronger TSO model, and leave the evaluation of weaker models for future work.

4.4 Predicting S/QBC Hits

The most straightforward solution to improve energy efficiency is to serialize the access to S/QBC and the L1/TLB. However, even taking advantage of all locality available in the S/QBC, such an approach would increase the latency of >80% of the loads, while saving L1/TLB energy on the remainder that hit in the S/QBC. (See Section 3.) To avoid this loss of performance, while retaining the energy savings, we need to predict whether a load will hit in the S/QBC so that we can choose to disable the L1/TLB probe, without incurring a latency penalty for loads that do not hit.

Fortunately, determining in which cache level a load is expected to hit is a well-studied problem [20, 22, 40], as knowing the latency of memory operations is essential for scheduling of load dependent instructions [2, 12, 23, 29]. For the S/QBC, the prediction we need is whether a load will hit in the S/QBC. This problem is simplified by the memory dependence predictor, which already exists to predict if a load-dependent instruction will have its data forwarded from the SQ/SB, and inform the instruction scheduler in an effort to avoid instruction replays.

For our study we select an established memory dependence predictor technique based on *store-distances* [40] using a 1K table. The predictor is able to correctly predict S/QBC hits and misses in 93.6% of the cases (95.4% for the Standard SQ/SB), with worst application being *gobmk* at 89.7%. (See Figure 7.)

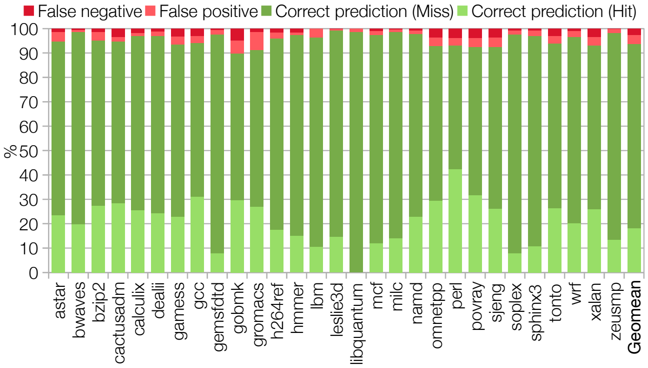


Figure 7: Memory dependency predictor accuracy using dynamic store-distances. Overall, it is able to predict 93.6% of S/QBC accesses correctly.

It is important to note that only correctly predicted *hits* will deliver energy benefits, while correctly predicted misses are important to minimize load latency by probing the S/QBC and L1/TLB in parallel. This means that false negatives (predicting misses for hits) reduces the energy benefit and false positives (predicting hits for misses) increase load latency. The breakdown of false positives and false negatives are shown in Figure 7, and their impact is discussed further in Section 5.

5 EVALUATION

5.1 Simulation and Modeling

We use 10 uniformly-distributed checkpoints for each SPEC-2006 [11] benchmark and a single checkpoint of the area of interest for PARSEC [5] (excluding *freqmine* due to OpenMP/simulator issues). Checkpoints are warmed for 100M instructions and results extracted from 10M instructions of detailed simulation. We use gem5 [6] to simulate a large out-of-order X86_64 CPU (Intel Skylake-like 8-wide, 224 entry ROB, 56 entry unified SQ/SB). The first-level cache is dual-ported with pipelined loads and stores. Each core has private L1/L2 caches and 4 cores share an L3. (See Table 1.) For energy evaluations we use CACTI [26] with a 22nm technology node [3].

We evaluate the performance and energy for several configurations:

- **Standard SQ/SB (aggressive write):** the baseline configuration with a standard SQ/SB write back policy.
- **Optimal SQ/SB (delayed write):** the optimal SB/SQ write-back policy where stores are delayed until space for new entries is required, thereby maximizing SQ/SB hits.
- **Store Buffer Cache, S/QBC (3, 7, 15 colors):** Our unified S/QBC with 3, 7, or 15 colors (2, 3, or 4 dirty bits) for tracking write epochs.

[3] For *energy modeling*, we model a 64 entry, 4-way set-associative dTLB to match the first-level TLB of the Intel Skylake architecture. For *performance modeling* we simulate a 512 entry fully-associative TLB. This models the low energy of hitting in a small first-level TLB without the unrealistic performance from not having a larger second-level TLB. We chose this approach to overcome gem5's inability to model multi-level TLBs.

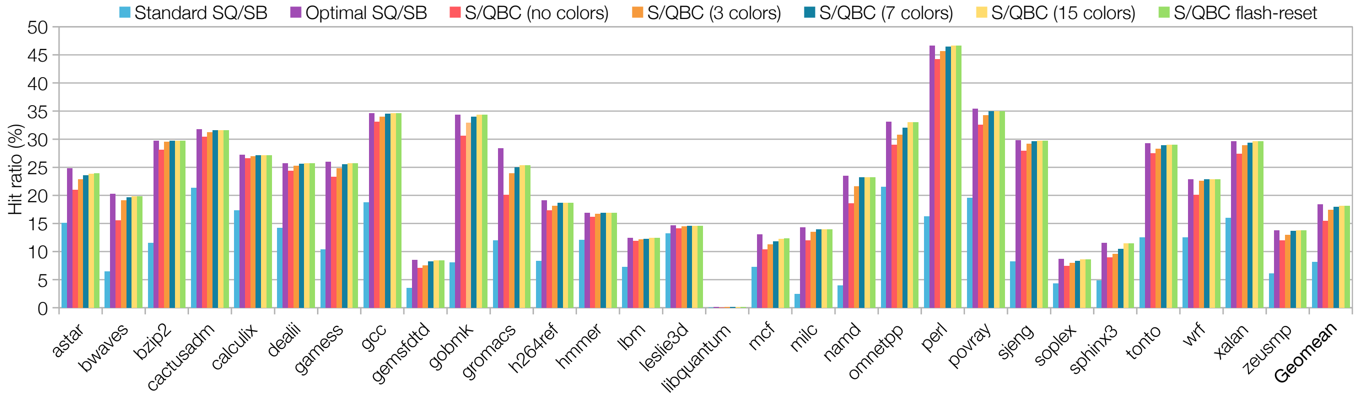


Figure 8: Hit ratio for a Standard SQ/SB (aggressive write back) and Optimal SQ/SB (delayed write back), and our store-buffer-cache (S/QBC), with no extra dirty bits to avoid flushing (no colors), 2, 3, and 4 dirty bits (3, 7, 15 colors), and using the flash-reset strategy (2 dirty bits, but equivalent to infinite colors). (higher is better)

Frequency	3.6GHz
IssueWidth/Ld,St Units	8/2,2
SQ/LQ/IQ/ROB	56/72/50/224
iTLB/dTLB	512/512 fully-assoc
Caches	L1I/L1D/L2/L3
Size	32KB/32KB/256KB/8MB
Latency	1c/4c/12c/38c
Associativity	8w/8w/8w/16w
DRAM	DDR3, 1600MHz, 64bits

Table 1: Gem5 simulator configuration. 4 cores share an L3 for the multi-threaded simulations.

- **Store Buffer Cache, S/QBC flash-reset:** Our unified S/QBC that flash resets red cache lines to black on SBC flushes. This is equivalent to an infinite number of colors, while just requiring 2 dirty bits in the L1.

5.2 Hit Ratio

Figure 8 compares the load hit ratios for all configurations. The S/QBC hit ratio increases with the number of colors used to differentiate write epochs as they avoid extraneous flushes. For no colors, e.g., flushing on every L1 eviction or downgrade, the hit ratio is 84% of the optimal, while with only 3 colors (2 dirty bits per line total) we obtain 95% of the optimal hit ratio.

Additional bits further improve the hit ratio, with the flash-reset strategy (using 2 dirty bits total, but equivalent to infinite colors) being almost identical to the optimal solution (18.2% vs 18.4%). 3 and 4 dirty bits (7 and 15 colors) preserve 98% and 99% of the optimal hit ratio respectively.

The effectiveness of the coloring strategies in avoiding SBC flushes can be seen in Figure 9. Using two dirty bits (3 colors) is enough to prevent 78% of the SBC flushes seen with no colors. Using more colors (7 and 15) reduces the flushes even further (preventing 95% and 99% of the flushes compared to S/QBC (no colors)). This shows that a simple 2 dirty bit strategy is enough to significantly reduce unduly SBC flushes.

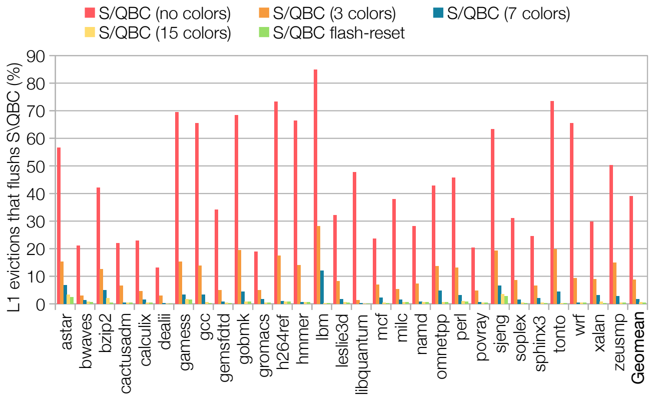


Figure 9: Percent of cache line evictions from L1 that cause a S/QBC flush with, 2, 3, 4 dirty bits (3, 7, 15 colors) and using flash-reset. (lower is better)

For the rest of the paper we evaluate only the 3-color (2 dirty bits) configuration, our least accurate coloring strategy; and flash-reset (2 dirty bits, with a flash reset circuit), our most accurate coloring strategy.

5.3 Hit Prediction

While the S/QBC obtains essentially the same hit ratio as the Optimal SQ/SB, the hit ratio alone is insufficient. In addition, we need to be able to predict S/QBC hits accurately enough avoid L1/TLB probes (to save energy) without causing unnecessary serialization of the L1/TLB accesses (and hurting performance).

Figure 10 shows how much the memory dependence predictor reduces the effective S/QBC hit ratio for the 3-color and flash-reset S/QBC. This includes the false negatives, which are hits in the S/QBC that deliver no energy benefit since they were incorrectly predicted to be S/QBC misses. Overall, the use of the predictor reduces the 3-color solution's filter rate from 17.4% (perfect) to 16.5% (with predictor) and the flash-reset solution's filter rate from 18.2% (perfect) to 17.2% (with predictor).

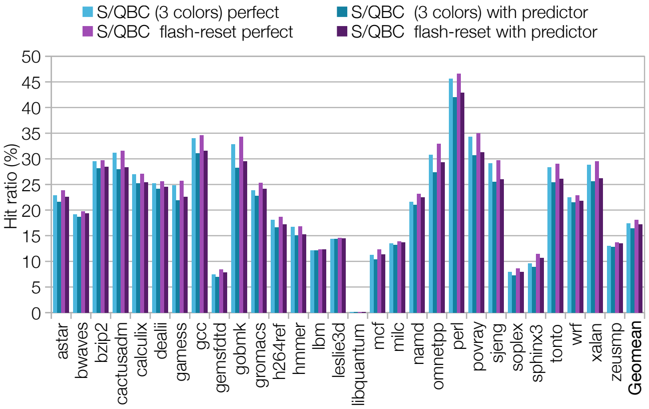


Figure 10: S/QBC hit ratio when factoring in the memory dependence predictor accuracy. This is the effective ratio of loads that hit in the S/QBC that are filtered, i.e. total hit ratio minus the false negative errors. (higher is better)

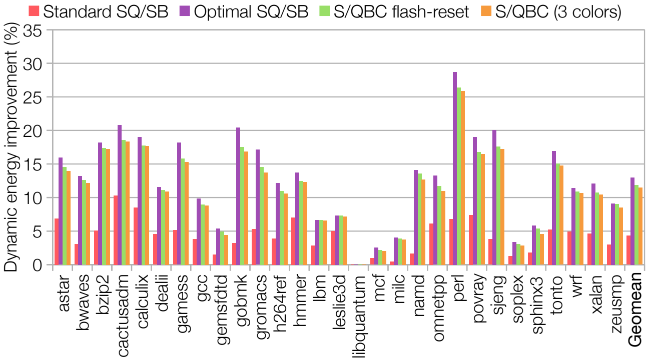


Figure 11: Dynamic energy savings (S/QBC and L1/TLB) of disabling L1/TLB probes on predicted SQ/SB hits, comparing the Standard SQ/SB (aggressive write) and Optimal SQ/SB (delayed write) with perfect hit predictors to the S/QBC with the memory dependence predictor. (higher is better)

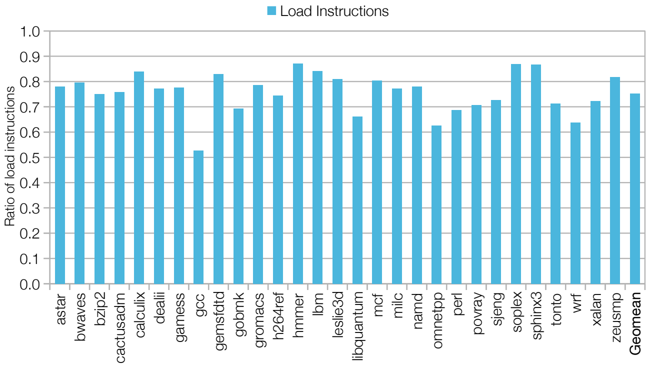


Figure 12: Percentage of load instructions in each benchmark. As the S/QBC can only hit on load instructions accessing previous stores, the more store instructions an application has, the fewer hits are possible.

5.4 Energy

Figure 11 shows the dynamic energy reduction (S/QBC and L1/TLB data accesses) normalized to the Standard SQ/SB with parallel L1/TLB probes. The configurations evaluated are the Standard SQ/SB and Optimal SQ/SB, with perfect avoidance of L1/TLB accesses on hits, and the S/QBC with 3-colors and the flash-reset, with the memory dependence predictor[4]. The Optimal SQ/SB, by improving the store-buffer hit ratio and perfectly selecting between serial/parallel access, gives an upper-bound on the potential energy savings of 13% on average. The benchmarks with the highest hit ratios are the ones that show the best improvement: *perl*, *povray* and *gobmk*, with a hit ratios of 46.6%, 35.4% and 34.4%, improve dynamic energy by 28.7%, 19% and 20.4%, respectively.

The S/QBC flash-reset reduces dynamic energy by 11.8% on average, achieving 91% of the energy improvement of the Optimal SB/SQ. The most improvement is seen in *perl*, *cactusadm* and *calculix* with an improvement of 26.4%, 18.6% and 17.8% respectively. *Povray* and *gobmk* show lower benefits than with the Optimal SQ/SB due to inaccuracies of the memory dependence predictor (Figure 7).

The S/QBC (3 color) configuration is predictably worse than the S/QBC flash-reset configuration due to the lower S/QBC hit ratio caused by more frequent flushes. Despite this, the S/QBC (3 color) is able to improve energy over the baseline by 11.5%, only 0.3 percentage point bellow the S/QBC flash-reset results. This suggests that the largest cause of inefficiency is the accuracy of the memory dependence predictor and not the S/QBC hit ratio: the simplest (2 dirty bit) extension to L1 is sufficient to extract most of the energy benefit using the memory dependence predictor as a S/QBC hit-predictor.

Note that there is not a one-to-one relationship between benchmarks with the highest hit ratios and the highest energy improvements. The difference comes from two sources: (1) the ratio of load instructions (which can hit in the S/QBC) to store instructions (which cannot), and, (2) the predictor accuracy. Figure 12 shows the percentage of load accesses per benchmark. *gcc* and *omnetpp* stand out as having some of the highest S/QBC hit ratios (34.1% and 33.1%,) but having energy improvements (Figure 11) comparable to *zeusmp*, which has a low hit ratio (Figure 10), due to the low percentages of loads in their memory accesses (53% and 63% respectively).

The largest effect of predictor accuracy is seen in *gobmk*, where the energy improvement for the optimal policy would be 20.4%, while the S/QBC flash-reset obtains only 17.6%. This directly correlates with having the largest number of false negative errors from the predictor (4.8%, see Figure 7). Even though *gobmk* shows the largest deviance from optimal, the S/QBC is still able to achieve 91% of the dynamic energy benefit that the optimal implementation would.

To complete the study, we also evaluated the potential energy benefits of the Standard SQ/SB policy using a perfect serial/parallel access predictor Figure 11. We see that the default aggressive write policy severely limits the ability of the SQ/SB to reduce L1/TLB energy. On average, the Standard SQ/SB is only able to improve energy by 4.3%, with *cactusadm* delivering the highest improvement

[4] Since the dTLB consumes only 1.3% of energy of a dL1 cache per access, the energy graph does not discriminate between the two.

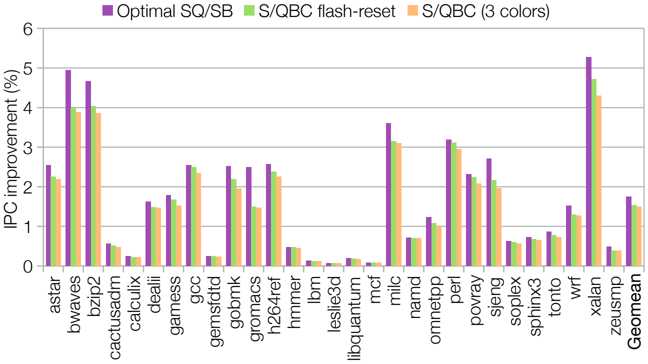


Figure 13: IPC improvement (%) of the Optimal SQ/SB and S/QBC (3 colors and flash-reset) over the baseline Standard SQ/SB. (higher is better)

of 10.3%. This demonstrates that even with accurate hit prediction, increasing the hit ratio is essential for improving energy efficiency.

5.5 Performance

Figure 13 shows the IPC impact of the Optimal SQ/SB and the S/QBC configurations compared to the Standard SQ/SB. The Optimal SQ/SB improves IPC by 1.7% on average and *xalan* sees the biggest improvement of 5.3%. This small performance improvement is expected for two reasons: (1) the latency difference between the SQ/SB and L1 cache is small (data access of 1 cycle vs. 4 cycles), and, (2) the aggressive OoO cpu core is able to hide a significant amount of this latency difference, reducing the impact of the lower latency accesses. It is reasonable to expect that a smaller core would see more performance benefit, but such cores also tend to have smaller SQ/SBs, and would therefore have less potential to hit in a unified S/QBC.

The S/QBC flash-reset is able to improve performance by 1.5% on average with the largest improvement of 4.7% for *xalan*. Some applications are hurt by false positive errors that serialize the S/QBC and L1/TLB accesses, thereby increasing hit latency. This is most evident in the *gromacs* and *bzip2* benchmarks, which have two of the highest false positive error rates (5.5% and 3.5% respectively) and are sensitive to load latency, causing the largest drop in IPC compared to the optimal configuration. S/QBC (3 colors) has similar results, and deviates from the optimal solution for the same reasons as the S/QBC flash-reset. The further decrease in IPC of the S/QBC (3 color) compared to the S/QBC flash-reset is due to the decreased overall hit-ratio. S/QBC (3 color) improves IPC by 1.4%, only 0.1 percentage points behind S/QBC flash-reset.

Overall, S/QBC achieves the goal of improving L1/TLB access energy without hurting performance. This demonstrates that we are able to keep stores in the S/QBC without increasing processor stalls, and even benefit modestly (1.5% on S/QBC flash-reset) from the reduced latency of the increased hits.

5.6 Instruction Scheduling Implications

Pipelines in an aggressive OoO processors have several cycles of delay between the issue and execution stages, which means that dependent instructions have to be scheduled speculatively to be able to execute back to back. Variable load latency (e.g., from hit-mispredictions) will therefore force instruction replays of dependent instructions and hurt performance and energy [29]. For this study we assume zero issue-to-execute delay of instructions (the default in gem5). This choice is consistent with the baseline, which uses the same predictor, and as such will have a similar prediction accuracy. As a result, the number of replays in both cases will be the same.

The only difference in scheduling prediction between the baseline and the S/QBC is that when a SQ/SB hit is mispredicted it takes one additional cycle to access the data from the L1 in the S/QBC configuration, due to the serialization of the L1/TLB access. However, in both designs the unexpected difference in latency from such a misprediction will cause dependent instructions to be flushed and replayed. The overhead of the flush-and-replay is significantly longer than the single cycle difference in returning the data. As a result, both designs will see very similar performance impacts from mispredictions, and the impact of not modeling replays in detail is unlikely to significantly change the relative performance of the designs.

5.7 Parallel Workloads

As the targeted energy reductions are in the private L1/TLB, and the vast majority of memory accesses are to private data, we do not expect parallel workloads to behave significantly differently from single-threaded applications. However, since the S/QBC does have to participate in coherence, there could be an increase in flushes, and hence a reduction in effectiveness, with multiple cores. To examine this, we simulated the PARSEC benchmarks.

Figure 14 shows the energy and IPC improvement[5] of the S/QBC flash-reset over the baseline for these parallel applications. In keeping with the single-threaded results, we see an average IPC improved of 0.4% (vs. 1.5% for single-thread) with *swaptions* showing the largest improvement of 3.2%. S/QBC and L1/TLB energy improved by 11.2% (vs. 11.8% for single-threaded) with *swaptions* again having the best improvement of 32.1%. The results are similar to those of the single-threaded benchmarks for similar reasons: hit ratio, load instruction ratio, and memory dependence predictor accuracy.

The only difference between the parallel and serial workloads was the potential increase in number of SBC flushes caused by coherence traffic. While flush requests on single-threaded benchmarks were caused exclusively by dirty cache line evictions, on multi-threaded benchmarks, invalidations and downgrade request from other cores can also cause flushes. While coherence traffic indeed caused extra SBC flushes, they were far fewer than the flushes caused by dirty cache lines evictions (at least one order of magnitude fewer), thus did not significantly affect the S/QBC hit ratio.

[5] As the effect of the S/QBC on locks will be seen through increased flushes, which will hurt energy savings but not change inter-core synchronization latency, we do not expect to change the number of instructions spent in locks.

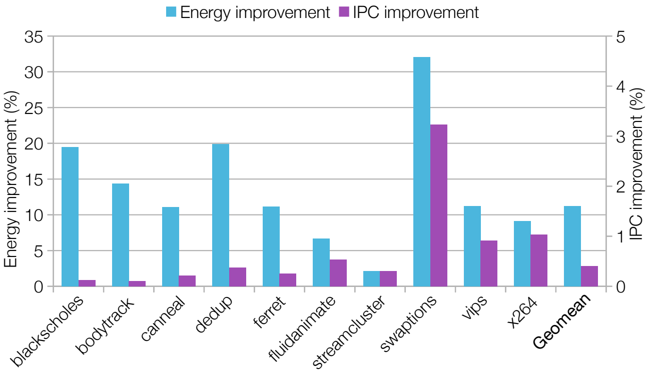


Figure 14: Performance and energy improvements for the parallel applications from PARSEC. (higher is better)

5.8 S/QBC Worst Case Scenarios

There are two ways in which the S/QBC can fail: if the application has little *read locality* or if the predictor is *inaccurate*. For applications with little read locality, the overhead of our extra L1 bit (0.2% area) will be very small compared to the energy/latency of searching and missing in the L1 in the first place. (Even “low-locality" applications show >30% L1 read hits [1].) The impact of predictor inaccuracies will not hurt energy vs. a standard SQ/SB as both need to check the SQ/SB, but it may cause serialization of the S/QBC and L1 accesses. The four scenarios are:

- **Locality+Accurate:** If the application has L1 read locality and the predictor is accurate, the S/QBC will **improve energy** (by avoiding L1 probes) and **improve performance** (by returning data from S/QBC).
- **Locality+Inaccurate:** If the application has L1 read locality but the predictor is inaccurate, the S/QBC will have the **same energy** (probes both S/QBC and L1) and the **same performance** (both S/QBC, L1 probed in parallel).
- **No-locality+Accurate:** If the application has no L1 read locality and the predictor is accurate, the S/QBC will have the **same energy** (probes both S/QBC and L1) and the **same performance** (both S/QBC, L1 probed in parallel).
- **No-locality+Inaccurate:** If the application has no L1 read locality and the predictor is inaccurate, the S/QBC will have the **same energy** (probes both S/QBC and L1) but **may degrade performance** (due to false positives causing serialization of the S/QBC search and L1 search).

The only situation in which the S/QBC might be worse than the baseline is if the memory dependence predictor is inaccurate when there is no locality. Other cases may not save energy, but do not hurt performance. For this worst-case to occur, the store-load instruction pairs must regularly change behavior to cause incorrect predictions. In applications with little L1 locality, the predictor learns that there is no load-store dependency, delivering the same performance as the baseline. This can be seen by looking at *libquantum*, *mcf* and *milc* in Figures 4 (poor locality) and 7 (but accurate prediction). Further, if the predictor was terribly inaccurate, bad instruction scheduling would likely outweigh additional load latency.

Parallel applications with significant sharing can also see lower energy benefits as the S/QBC will be flushed more due to invalidations from coherence requests. However, this should not cause any additional overhead compared to the baseline since prediction is based on store distance, so flushes will also update the predictor and thereby avoid useless searches. Cores that are spinning on locks will have little opportunity to benefit from the S/QBC (no stores to put it in the SBC) but they will also not cause frequent flushes in other non-spinning cores (the data will only be in the other cores after a write).

6 RELATED WORK

Filter or L0 caches. Filter caches [4, 14, 19] improve energy efficiency and latency of memory accesses by decreasing the access energy and latency compared to the L1 due to their small capacity. This strategy is successful when there is enough locality to overcome the energy and latency overheads of probing and copying data to/from the filter cache and the increase in latency of filter cache misses. Unfortunately, for heavily out-of-order processors, the performance benefit of the slightly lower latency is often minimal, and the energy cost of moving data for a low hit ratio is often high [3]. Our solution differs in the sense that the data installation and probing is necessary for correctness in the unified SQ/SB, so there is no additional overhead of using the same structure as a filter cache. These characteristics come with the downside that only loads can benefit, while filter caches can improve both loads and stores energy.

Energy efficient caches. Other solutions improve energy efficiency not by reducing the number of cache accesses but by improving efficiency of the accesses themselves. Way-predictors [7, 15, 18, 30, 38] sacrifice some access latency due to mispredictions, but reduce the cost (number of ways probed). Way-estimators [13, 41] have no mispredictions, but can increase the number of bits read over a way-predictor. Other techniques trade-off hit ratio [17] to improve access energy efficiency. These techniques could be used on top of our proposal for further benefits.

Delaying writes. Policies for delaying writes have been proposed for single-thread [16, 37] and multi-thread applications [32] with the aim of coalescing stores to increase the effective SB size and reduce the number of write transactions to execute and track [33] or with the aim of avoiding L1 accesses in processors with non-associative LQs [34]. These approaches delay writes and start writing back when a high water mark is reached to avoid stalling. In our design we need no high water mark as evictions are performed immediately and silently. This is possible thanks to the fact that we do not delay the writes, but perform the writes immediately and only keep a copy.

Filtering L1 accesses. The cached load store queue transforms a unified load/store queue into a more traditional filter cache to reduce L1 accesses [27]. As it targets loads, it requires additional storage and data movement to hold the load data that would otherwise not be present in a load queue, and because it combines loads and stores in a single cache, all entries must fully participate in coherence traffic, as well as the CAM-accesses. The design serializes LSQ and L1 accesses to save energy. Carazo, et. al. [8] proposed combining two different predictors to switch between parallel and serial L1 accesses on hits in the cached load store queue. By only targeting stores we are able to deliver a far simpler design. The S/QBC is able to filter L1 accesses without hurting performance, increasing data movement, adding storage, requiring complex coherence, or adding additional predictors. The downside is that the S/QBC only targets load instructions.

Store-buffer optimization. Existing work on store-buffers aims to reduce their cost by reducing the frequency of accesses [28], the number of entries probed [35], or removing the structure completely [36, 39]. These techniques focus on making the access to the buffer itself more efficient, but do not target reducing the energy of the L1/TLB.

Memory dependence prediction. Memory dependence prediction started with the work of Moshovos et al. [24] who were the first to show that dependencies between loads and stores are very stable and can be predicted with high accuracy. Subsequent work by Chrysos et al. expanded the idea to predict the dependence of a load not to a single store but to a set of stores, thereby expanding its reach [10]. Retaining the identity of the stores in the predictor was deemed unnecessary (since we only care to predict when we see the program counter of the load) and the notion of *store distance* was introduced to give a sense of how far in the instruction stream a load's dependence is expected to be encountered [40]. The high accuracy of such predictors, coupled with measure of distance that can be correlated to the chance of finding a store in the store buffer, is a compelling argument to use them in this work. Memory dependence prediction has also been used to completely eliminate the store queue by predicting store-load forwardings at the stores and bypassing store values to loads without ever using an intermediary storage area (store queue) [36]. Instead of going to such an extreme, we make the case that the intermediary storage area of the SQ and the SB can be put to very good use by increasing the number of store-load forwardings using our techniques.

7 CONCLUSION

Store-queues and store-buffers are ubiquitous parts of modern out-of-order microprocessors to ensure that bursts of writes do not stall the pipeline while waiting to be committed and written back to the cache. This requires that all writes be installed in the SQ/SB and that all loads probe it for data, which essentially costs the energy overhead of a filter cache. However, the low hit ratio, due to aggressively writing back entries to avoid stalls, and accessing it in parallel with the L1/TLB, to avoid increased latency, means that hits do not deliver any energy or performance benefits.

In this paper we introduced a unified S/QBC store-buffer-cache, which adds a third logical partition to the SQ/SB that keeps copies of data that has already been written back. As a result we are able to increase the S/QBC hit ratio to 18.2% (just 0.2 percentage points shy of an Optimal SQ/SB). By leveraging the existing memory dependence predictor, we are also able to accurately predict hits/misses in our unified S/QBC 93.6% of the time, which allows us to avoid 17.2% L1/TLB probes. However, by keeping copies of data, we need to include the S/QBC in coherence. To achieve this we take advantage of only needing to update the S/QBC on changes locally owned (dirty) data in the L1. This allows us to cheaply track whether dirty data in the L1 could be in the S/QBC by adding one additional dirty bit, and flushing the S/QBC copies only when such data is affected.

The overall design has essentially no overhead (0.2% additional L1 storage for one additional dirty bit per line and one additional S/QBC tail pointer) and does not increase data movement energy (moving entries from the SB to the SBC is only a logical pointer update). With this work we are now able to take advantage of the existing storage capacity of the SQ/SB and accesses to reduce dynamic L1 and TLB energy by 11.8% with no performance impact (indeed, a marginal 1.5% improvement).

ACKNOWLEDGMENTS

This work was supported by: the Knut and Alice Wallenberg Foundation through the Wallenberg Academy Fellows Program; the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant No 715283); the SSF Strategic Mobility 2017 (grant SM17-0064); the Spanish Ministerio de Economía, Industria y Competitividad — Agencia Estatal de Investigación (grant ERC2018-092826); and EU Horizon 2020 EPEEC Project (www.epeec-project.eu) (grant No 801051).

REFERENCES

[1] Sam Ainsworth and Timothy M. Jones. 2016. Graph Prefetching Using Data Structure Knowledge. In *International Conference on Supercomputing (ICS)*. ACM, 39:1–39:11.

[2] Ricardo Alves, Stefanos Kaxiras, and David Black-Schaffer. 2018. Dynamically Disabling Way-prediction to Reduce Instruction Replay. In *International Conference on Computer Design (ICCD)*. IEEE, 140–143.

[3] Ricardo Alves, Nikos Nikoleris, Stefanos Kaxiras, and David Black-Schaffer. 2017. Addressing Energy Challenges in Filter Caches. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 49–56.

[4] Nikolaos Bellas, Ibrahim Hajj, and Constantine Polychronopoulos. 1999. Using dynamic cache management techniques to reduce energy in a high-performance processor. In *International Symposium on Low Power Electronics and Design (ISLPED)*. ACM/IEEE, 64–69.

[5] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (2011), 1–7.

[7] Brad Calder and Dirk Grunwald. 1996. Predictive Sequential Associative Cache. In *International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 244–253.

[8] Pablo Carazo, Rubén Apolloni, Fernando Castro, Daniel Chaver, Luis Pinuel, and Francisco Tirado. 2010. L1 data cache power reduction using a forwarding predictor. In *International Workshop on Power and Timing Modeling, Optimization and Simulation*. Springer, 116–125.

[9] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. 2011. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 155–166.

[10] George Z Chrysos and Joel S Emer. 1998. Memory dependence prediction using store sets. In *International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, 142–153.

[11] Standard Performance Evaluation Corporation. 2006. SPEC CPU2006. http://www.spec.org/cpu20066

[12] Dan Ernst, Andrew Hamel, and Todd Austin. 2003. Cyclone: A Broadcast-free Dynamic Instruction Scheduler with Selective Replay. In *International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, 253–263.

[13] Mrinmoy Ghosh, Emre Özer, Simon Ford, Stuart Biles, and Hsien-Hsin S. Lee. 2009. Way Guard: A Segmented Counting Bloom Filter Approach to Reducing Energy for Set-Associative Caches. In *International Symposium on Low Power Electronics and Design (ISLPED)*. ACM/IEEE, 165–170.

[14] Roberto Giorgi and Paolo Bennati. 2007. Reducing leakage in power-saving capable caches for embedded systems by using a filter cache. In *Workshop on*

MEmory performance: DEaling with Applications, systems and architecture. ACM, 97–104.

[15] Koji Inoue, Tohru Ishihara, and Kazuaki Murakami. 1999. Way-predicting set-associative cache for high performance and low energy consumption. In *International Symposium on Low Power Electronics and Design (ISLPED)*. ACM/IEEE, 273–275.

[16] Norman P. Jouppi. 1993. Cache Write Policies and Performance. In *International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, 191–201.

[17] Georgios Keramidas, Polychronis Xekalakis, and Stefanos Kaxiras. 2007. Applying decay to reduce dynamic power in set-associative caches. In *International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC)*. Springer, 38–53.

[18] Richard E Kessler, Richard Jooss, Alvin Lebeck, and Mark D Hill. 1989. Inexpensive implementations of set-associativity. In *International Symposium on Computer Architecture (ISCA)*. IEEE, 131–139.

[19] Johnson Kin, Munish Gupta, and William H Mangione-Smith. 1997. The filter cache: an energy efficient memory structure. In *International symposium on Microarchitecture (MICRO)*. IEEE, 184–193.

[20] Yongxiang Liu, Anahita Shayesteh, Gokhan Memik, and Glenn Reinman. 2004. Scaling the issue window with look-ahead latency prediction. In *International Conference on Supercomputing (ICS)*. ACM, 217–226.

[21] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. 2016. COATCheck: Verifying memory ordering at the hardware-OS interface. *ACM SIGOPS Operating Systems Review* 50, 2 (2016), 233–247.

[22] Gokhan Memik, Glenn Reinman, and William H Mangione-Smith. 2005. Precise instruction scheduling. *Journal of Instruction-Level Parallelism* 7 (2005), 1–29.

[23] Pierre Michaud and André Seznec. 2001. Data-flow prescheduling for large instruction windows in out-of-order processors. In *International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 27–36.

[24] Andreas Moshovos, Scott E Breach, Terani N Vijaykumar, and Gurindar S Sohi. 1997. Dynamic speculation and synchronization of data dependences. In *International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, 181–193.

[25] Andreas Moshovos and Gurindar S Sohi. 1997. Streamlining inter-operation memory communication via data dependence prediction. In *International Symposium on Microarchitecture (MICRO)*. ACM/IEEE, 235–245.

[26] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. *CACTI 6.0*. Technical Report HPL-2009-85. HP Labs.

[27] Dan Nicolaescu, Alex Veidenbaum, and Alex Nicolau. 2003. Reducing data cache energy consumption via cached load/store queue. In *International Symposium on Low Power Electronics and Design (ISLPED)*. ACM/IEEE, 252–257.

[28] Il Park, Chong Liang Ooi, and T. N. Vijaykumar. 2003. Reducing Design Complexity of the Load/Store Queue. In *International Symposium on Microarchitecture (MICRO)*. ACM/IEEE, 411–422.

[29] Arthur Perais, André Seznec, Pierre Michaud, Andreas Sembrant, and Erik Hagersten. 2015. Cost-effective speculative scheduling in high performance processors. In *International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, 247–259.

[30] Michael D Powell, Amit Agarwal, TN Vijaykumar, Babak Falsafi, and Kaushik Roy. 2001. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *International Symposium on Microarchitecture (MICRO)*. ACM/IEEE, 54–65.

[31] Alberto Ros and Stefanos Kaxiras. 2012. Complexity-Effective Multicore Coherence. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 241–252.

[32] Alberto Ros and Stefanos Kaxiras. 2016. Racer: TSO Consistency via Race Detection. In *49th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*.

[33] Alberto Ros and Stefanos Kaxiras. 2018. Non-speculative store coalescing in total store order. In *International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, 221–234.

[34] Alberto Ros and Stefanos Kaxiras. 2018. The Superfluous Load Queue. In *International Symposium on Microarchitecture (MICRO)*. ACM/IEEE, 95–107.

[35] Tingting Sha, Milo MK Martin, and Amir Roth. 2005. Scalable store-load forwarding via store queue index prediction. In *International Symposium on Microarchitecture (MICRO)*. ACM/IEEE, 159–170.

[36] Tingting Sha, Milo MK Martin, and Amir Roth. 2006. Nosq: Store-load communication without a store queue. In *International Symposium on Microarchitecture (MICRO)*. ACM/IEEE, 285–296.

[37] Kevin Skadron and Douglas W. Clark. 1997. Design Issues and Tradeoffs for Write Buffers. In *International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE.

[38] Kimming So and Rudolph N. Rechtschaffen. 1988. Cache Operations by MRU Change. *IEEE Trans. Comput.* 37, 6 (1988), 700–709.

[39] Samantika Subramaniam and Gabriel H Loh. 2006. Fire-and-forget: Load/store scheduling with no store queue at all. In *International Symposium on Microarchitecture (MICRO)*. ACM/IEEE, 273–284.

[40] Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan. 1999. Speculation techniques for improving load related instruction scheduling. In *International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, 42–53.

[41] Chuanjun Zhang, Frank Vahid, Jun Yang, and Walid Najjar. 2005. A Way-Halting Cache for Low-Energy High-Performance Systems. *Transactions on Architecture and Code Optimization (TACO)* 2, 1 (2005), 34–54.