Regional Out-of-Order Writes in Total Store Order

Sawan Singh
singh.sawan@um.es
University of Murcia
Murcia, Spain

Alexandra Jimborean
alexandra.jimborean@um.es
University of Murcia
Murcia, Spain

Alberto Ros
aros@ditec.um.es
University of Murcia
Murcia, Spain

ABSTRACT

The store buffer, an essential component in today's processors, is designed to hide memory latency by moving stores off the processor's critical path. Furthermore, under the Total Store Order (TSO) memory model, the store buffer ensures the in-order retirement of stores. Problems arise when the store buffer is full or, under TSO, when the leading store encounters a cache miss, which blocks all subsequent stores and incurs severe performance bottlenecks.

This work presents a software-hardware co-designed approach to cope with this bottleneck for processors with strong consistency guarantees. Our proposal is driven by the insight that store operations can be reordered if their reordering does not change the observable program behavior. The compiler delineates *safe* regions within which stores can be shuffled while still delivering the same observable behavior as if they performed in program order and *unsafe* regions within which stores must be kept in program order. This is leveraged by a novel dual-mode store buffer that switches between the *out-of-order* and *in-order* execution of stores within the safe and respectively unsafe regions. Correctness is preserved through well-placed fences inserted by the compiler, which impede the execution of stores from the following regions until all stores of the current region complete. Our dual-mode store buffer only requires one extra bit per entry, significantly decreases processor stall cycles, and brings 8.13% performance improvements compared to a mainstream store buffer.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**; *Superscalar architectures*; • **Software and its engineering →** *Multithreading*; *Consistency*.

KEYWORDS

Memory Consistency Models; Total Store Order; Store Buffer

ACM Reference Format:
Sawan Singh, Alexandra Jimborean, and Alberto Ros. 2020. Regional Out-of-Order Writes in Total Store Order. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3410463.3414645

1 INTRODUCTION

Strong memory consistency models such as Sequential Consistency (SC) [20] offer intuitive semantics to programmers by preserving the program order of all memory accesses. However, a high memory level parallelism is key for performance, but relies on reordering memory instructions to hide long memory latency.

This role is assumed by the store buffer (SB). The store buffer is an essential component of modern-day processors and it is designed to hide the long latency of store operations by allowing processors to perform the store operations out of the processor's critical path. Store operations therefore first retire from the processor's pipeline to the store buffer and then perform the memory write. This causes load instructions to be effectively reordered with respect to store instructions, thus relaxing the consistency model semantics. The resulting consistency model, supported by Intel and AMD processors, is known as Total Store Order (TSO) [32].

TSO semantics preserve the order of store operations with respect to other stores and of load operations with respect to other loads. However, to achieve memory level parallelism, in practice, loads are speculatively reordered with respect to other loads [14]. Stores, on the other hand, perform in-order based on the observation that stores are not on the processor's critical path. Unfortunately, when a store operation misses in the cache, all subsequent stores block until the miss is resolved and the store performs. This creates severe bottlenecks in current out-of-order (OoO) processors.

Previous proposals alleviate the store buffer bottleneck by performing the stores out-of-order while still complying with TSO [8, 15, 28, 33, 34]. However, they either require complex hardware structures and speculative writes [34], or they only enable limited reordering [28, 33]. For example, the compile-time analysis by Singh *et al.* [33] only enables reordering of ≈75% of the stores, on average, while Ros and Kaxiras [28] simply force a different (lexicographical) order for all stores.

Our goal is to provide virtually full store re-ordering (98.8% of the stores, on average) while maintaining a simple hardware design. To achieve this, we propose regional out-of-order writes (ROOW), a software-hardware co-designed solution that consists in a compiler that delineates safe regions of code where stores can be reordered without affecting the consistency guarantees, a technique to transfer the information from the compiler to the hardware, and a simple dual-mode store buffer architecture that enables store reordering within safe regions, while forcing all the stores outside safe regions to write to memory in-order (Section 4).

Our main insights are: i) by exploiting the properties of the executing program (guaranteed by the consistency model of the programming language), the compiler can perform a more accurate analysis of the memory accesses, in contrast to conventional compile-time analyses that are conservative by nature and therefore less accurate; ii) by shifting complexity from hardware to software

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

ACM ISBN 978-1-4503-8075-1/20/10.
https://doi.org/10.1145/3410463.3414645

for classifying accesses and conveying this information from the compiler to the store buffer, we maintain hardware simplicity; and iii) designing a *single* dual-mode store buffer ensures correctness for multiple accesses to the same data (more precisely, prohibiting store-store reordering and handling store-to-load forwarding, as described in detail in Section 2).

Given the ultra-concise information to be transferred to the store buffer, the hardware overhead is absolutely minimal and consists in only one extra bit to signal the region type and another bit per store buffer entry to indicate the type of each store. Together with the simple structure of the store buffer itself, ROOW requires minimal modifications in the core, which yields our solution highly applicable and can be easily integrated in mainstream store buffer implementations.

Furthermore, by allowing the stores to reside in the store buffer even after being performed (as long as the store buffer is not full), ROOW provides a *zero-cost* solution for the use of the store buffer as a cache [4].

Results show that ROOW can perform 98.8% stores, on average, out-of-order, thus improving system performance by 8.13% compared to Intel's mainstream store buffer implementation [17] (Section 7). Furthermore, we conducted a sensitivity study which shows that thanks to reordering, we can reduce the store buffer size to just half of its entries without significantly degrading performance, and to one forth of its size while still improving performance with respect to the baseline configuration. Reducing the store buffer size could be desirable in future processors design, since it is searched associatively on every load and due to its significant energy consumption.

In particular, the main contributions of this paper are:

- A software-hardware co-design to provide precise information from the compiler with minimum overhead, enabling 98.8% of stores to be reordered within their region.
- A dual-mode store buffer (with regional in-order vs out-of-order processing of stores) implemented in a single buffer rather than two, creates a more efficient design leading to a better utilization of resources. Moreover, the hardware overhead is only one extra *flag* bit and one *mode* bit per store buffer entry compared to mainstream store buffers.
- A zero-cost solution to use the store buffer as a cache by keeping stores in the store buffer as long as they do not cause pipeline stalls.
- A sensitivity analysis of the performance variation with respect to the store buffer size that reveals that the store buffer can be reduced to half without degrading performance.

To the best of our knowledge, this is the simplest design of an out-of-order store buffer that preserves a strong consistency model such as TSO, thanks to a region-based classification to increase the compiler accuracy. This simple design makes our proposal suitable to be implemented in real processors with minimal engineering efforts.

2 BACKGROUND

2.1 Sequential semantics

Sequential semantics restrict reordering of accesses to the same memory location (dependencies), within a single thread, and need to be preserved by any program (sequential or parallel) to offer an intuitive behaviour.

The impact of the store buffer regarding memory consistency guarantees is that it allows stores to perform after a load that follows the store in program order. However, if the subsequent load targets the same memory location as the store, the load should read the value generated by the store in-order to preserve sequential semantics. That is, loads and stores targeting the same address cannot be reordered.

More generally, loads must retrieve the value written by the *latest* previous store to the same address. This is known as store-to-load forwarding and it is commonly implemented by requiring every load to snoop the store buffer in parallel to performing the memory access.

Reordering stores to the same address can cause loads reading an old value and therefore breaking the program's sequential semantics. This is not a problem when stores perform in-order, but it appears when the store-store order is relaxed, as it happens in our safe regions.

The aforementioned problem does not only occur when a load must retrieve the value from a previous store, but also when the stores targeting the same address write to memory, as it could result in the memory being updated with an old value.

ROOW proposes a simple solution to address these two challenges, i.e., correct store-to-load forwarding and memory updates, preserving sequential semantics even when stores perform out-of-order.

2.2 The SC-for-DRF consistency model

SC is globally accepted as the most intuitive memory model for shared memory multi-threaded programming model as it provides the illusion of one unique global ordering of all memory accesses within a program. However, SC is also the most restrictive model for shared memory, with an immense impact on performance compared to more relaxed memory models. Modern-day programming languages such as Java, C, C++ provide SC only for data-race-free (DRF) programs [1, 13]. In contrast, for racy programs, most programming languages offer no guarantees (e.g., C++) or a very relaxed consistency model at best.

This led to the advent of the SC-for-DRF consistency model [1]. Weaker than SC, as it provides the SC guarantees only for DRF software, SC-for-DRF enables powerful compile-time software optimizations.

SC-for-DRF requires racy accesses to be confined within synchronization operations (e.g. locks, critical sections). The regions of code delimited by synchronization operations (see Section 4, denoted here as *DRF regions*, offer the guarantee that different threads executing concurrently multiple DRF regions do not access the same location if at least one performs a write. Consequently, the writes performed by a thread during a DRF region remain "invisible" to the other threads until the end of the DRF region, since if one of the threads writes to a memory location, no other thread will attempt to read or write the same location until the end of the DRF region. This guarantee opens up the possibility of performing memory accesses in any order during DRF regions (as long as intra-thread dependencies are fulfilled as mandated by sequential semantics).

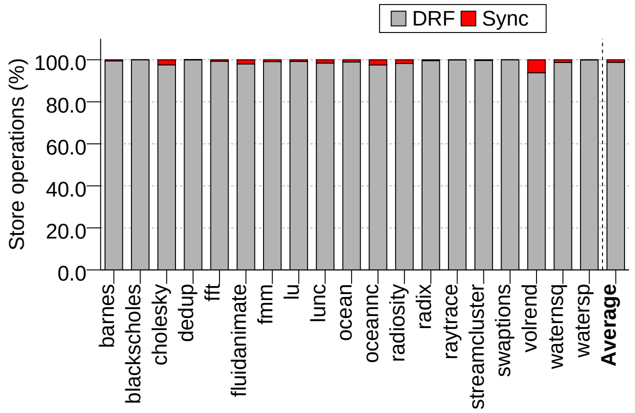


Figure 1: Percentage of store operations found outside synchronization code (DRF) and inside synchronization code (sync)

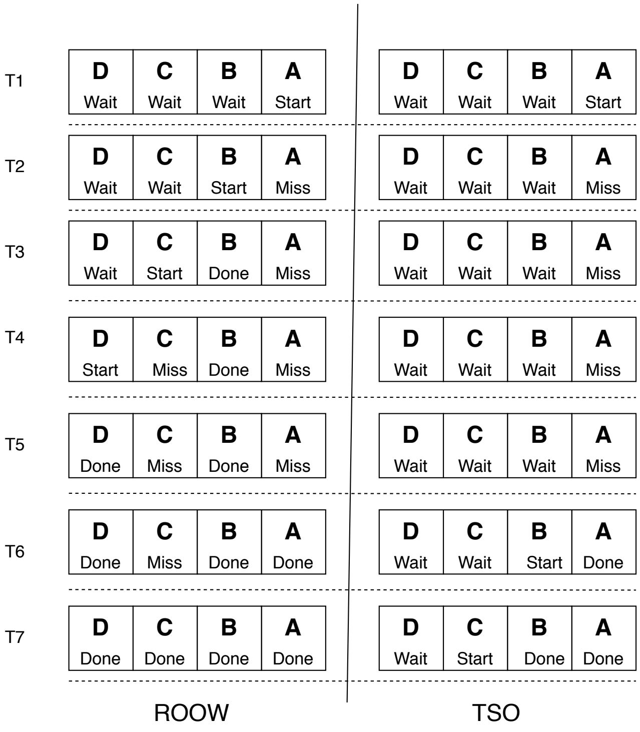
DRF region boundaries, i.e., language specific synchronization operations, include fences that ensure that memory accesses perform and become visible to the other threads at the end of the region. That is, fences forbid the compiler to reorder memory operations across different regions.

3 MOTIVATION

Although the DRF property is present in a large number of applications, it is currently not exploited at the hardware level and many processors (e.g. Intel and AMD) blindly implement a stronger consistency model, missing out enormous optimization potential.

Figure 1 shows the percentage of store operations found outside synchronization code (DRF) and inside synchronization code (sync) for the applications evaluated in this proposal (see Section 6 for details). The application with less stores in DRF regions of code is volrend with 93.8%. That percentage rises up to 99.9% in watersp while reaching 100% in blackscholes and swaptions. The average across all the evaluated benchmarks is 98.8%. Given the immense percentage of DRF store operations, performing them in-order is a very expensive luxury in TSO processors. This work is the first to explore how SC-for-DRF semantics can be exploited by the store buffer of TSO processors.

If processors had information about the nature of the stores residing in the store buffer (stemming from either DRF or sync regions), they could orchestrate the in-order vs. out-of-order execution of store operations, without relaxing the consistency guarantees. Figure 2 walks the reader through an example illustrating the benefits of performing stores out-of-order (as we propose in this work – ROOW) in contrast to enforcing their order across the entire store buffer (as implemented in standard TSO). Each row represents the content of the store buffer and the status of each store. All stores *A, B, C,* and *D* belong to the same DRF region, hence can be performed out-of-order as they have exclusive access to the target memory location. In ROOW, they all start as soon as they enter the store buffer (on the right), which means *B* can complete before *A* (which encounters a cache miss) and similarly *D* can complete before *C*. This parallelism hides the miss latency as the processor does not wait for the miss to be resolved, as in the standard TSO store buffer implementation. At the end of the DRF region the other cores observe that all stores completed, but not the order in which they have been performed.



	D	C	B	A	D	C	B	A
T1	Wait	Wait	Wait	Start	Wait	Wait	Wait	Start
T2	Wait	Wait	Start	Miss	Wait	Wait	Wait	Miss
T3	Wait	Start	Done	Miss	Wait	Wait	Wait	Miss
T4	Start	Miss	Done	Miss	Wait	Wait	Wait	Miss
T5	Done	Miss	Done	Miss	Wait	Wait	Wait	Miss
T6	Done	Miss	Done	Done	Wait	Wait	Start	Done
T7	Done	Done	Done	Done	Wait	Start	Done	Done
	ROOW				TSO			

Figure 2: Benefits of out-of-order execution of store operation

4 ROOW: REGIONAL OUT-OF-ORDER WRITES

While SC-for-DRF becomes the norm in most modern programming languages, the guarantees offered by DRF software are not exploited at the hardware-level. To leverage this valuable information, we propose a dual-mode store buffer architecture that enables regional out-of-order writes (ROOW). Namely, ROOW allows reordering the stores within safe (DRF) regions while forcing all the stores in unsafe (synchronization) regions to write to memory in-order. As reordering of stores happen only in *DRF regions* where data races are not possible, store-to-store order guarantees are still respected. This way the efficiency of the store buffer is improved by performing stores out-of-order without affecting the consistency model provided by the system.

To this end, the ROOW compiler performs a *region-based classification of accesses*, rather than data-based classification. The compiler delineates *synchronization regions* (denoted as sync regions) and *synchronization-free regions* (denoted as DRF regions). The collected information is transmitted to the hardware through a new dedicated instruction. To increase memory level parallelism, our store

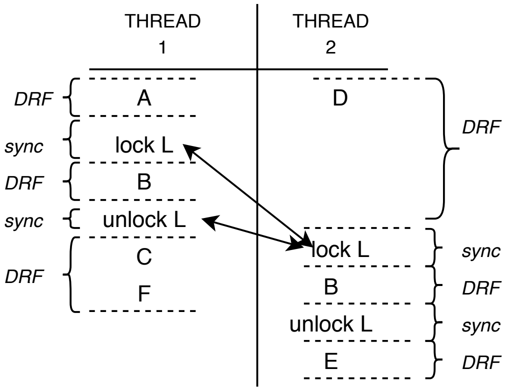


Figure 3: Example code showing a parallel SC-for-DRF program. Stores A, C, F, D, and E are part of DRF regions that run concurrently, while store B belongs to a DRF region that runs sequentially due to synchronization.

buffer implementation allow stores stores to perform out-of-order when they suffer a cache miss. Next sections detail the behavior of ROOW.

4.1 Compile-time delineation of regions

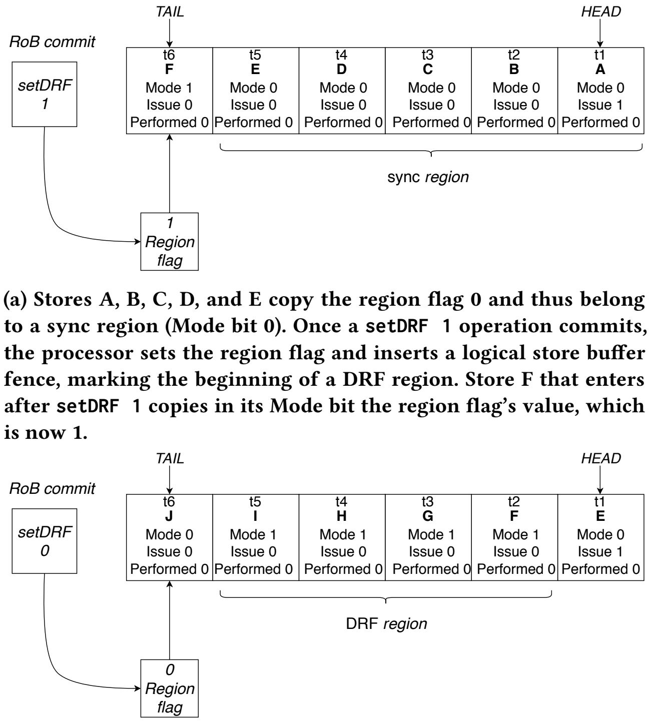
We implement a compiler pass in LLVM [21] to mark the data race free regions and delineate synchronization operations. In this work we assume parallel code to be data race free (as required by most modern programming languages, such as C++ and Java). Under these circumstances, the compiler marks all synchronization operations as *sync regions* and the resulting regions delimited by synchronization as *DRF regions*. Figure 3 shows an example of a SC-for-DRF program with delineated DRF and sync regions.

Synchronization operations are identified by analyzing the corresponding libraries[1] and represent the regions in which stores must be executed in order to ensure correctness under the TSO guarantees. DRF regions on the other hand allow stores to be performed out-of-order while preserving the TSO guarantees since no concurrent loads or stores to the same address can take place in other threads.

We distinguish between two types of *DRF regions*: (1) parallel code, that is DRF by our initial assumption (stores to A, C, F, D, E in Figure 3 and (2) sequential code (e.g., store B, executed by one thread at a time due to the synchronization operations) that is DRF by definition.

To delineate the regions, the compiler inserts a dedicated instruction, `setDRF val`, where val is a one-bit operand. A value of 1 indicates the beginning of a DRF region while 0 indicates a sync region.

Furthermore, the compiler performs an alias analysis to determine if DRF and sync regions may access the same location and inserts fences in between the corresponding regions [19].


TAIL
HEAD
RoB commit
setDRF 1
t6 F Mode 1 Issue 0 Performed 0
t5 E Mode 0 Issue 0 Performed 0
t4 D Mode 0 Issue 0 Performed 0
t3 C Mode 0 Issue 0 Performed 0
t2 B Mode 0 Issue 0 Performed 0
t1 A Mode 0 Issue 1 Performed 0
sync region
1 Region flag
TAIL
HEAD
RoB commit
setDRF 0
t6 J Mode 0 Issue 0 Performed 0
t5 I Mode 1 Issue 0 Performed 0
t4 H Mode 1 Issue 0 Performed 0
t3 G Mode 1 Issue 0 Performed 0
t2 F Mode 1 Issue 0 Performed 0
t1 E Mode 0 Issue 1 Performed 0
DRF region
0 Region flag

(a) Stores A, B, C, D, and E copy the region flag 0 and thus belong to a sync region (Mode bit 0). Once a `setDRF 1` operation commits, the processor sets the region flag and inserts a logical store buffer fence, marking the beginning of a DRF region. Store F that enters after `setDRF 1` copies in its Mode bit the region flag's value, which is now 1.

(b) Operation `setDRF 0` marks the end of a DRF region: it resets the region flag and triggers the insertion of an store buffer fence. As seen before, store J copies the current value of the region flag in its Mode bit. (Stores F, G, H and I copied the value of the region flag at the moment the stores entered the store buffer, marking them as DRF.)

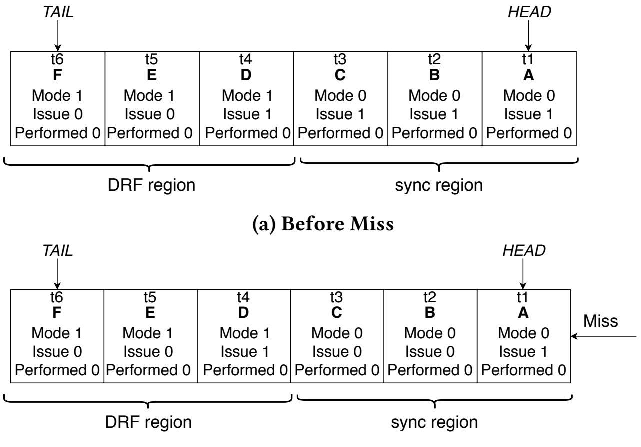
Figure 4: Conveying static information to the store buffer

4.2 Conveying static information to the store buffer

The processor requires minor modifications to execute the `setDRF` instruction. The instruction can be basically considered as a *nop* operation except at commit time. When the `setDRF` instruction commits, a dedicated processor flag called *region flag* (1 bit) changes its mode according to the operand value of the `setDRF` instruction. The update of the flag is performed at commit time since all instructions commit in-order. A value of the flag equal to 0 indicates that the processor is committing sync stores from that point on. A value of 1 indicates that the next stores are DRF. The region flag is set by default to 0. This way, applications in which the DRF regions have not been delineated (e.g. legacy code) still preserve the TSO semantics.

When a store commits, it reads the current value of the region flag to detect whether it belongs to a DRF region or a sync region. Stores keep this information in a per-entry bit added to the store buffer, called the *mode* bit. This bit is required because both in-order and out-of-order stores can co-exist in the store buffer, so they must indicate their nature on an individual basis. Figure 4 shows an example of how the compiler information is conveyed to the store buffer.

[1] Our compiler provides support for pthreads [25] and OpenMP libraries [26], but it can be easily extended to recognize any other synchronization library.


TAIL
HEAD
t6 F Mode 1 Issue 0 Performed 0
t5 E Mode 1 Issue 0 Performed 0
t4 D Mode 1 Issue 1 Performed 0
t3 C Mode 0 Issue 1 Performed 0
t2 B Mode 0 Issue 1 Performed 0
t1 A Mode 0 Issue 1 Performed 0
DRF region
sync region
TAIL
HEAD
t6 F Mode 1 Issue 0 Performed 0
t5 E Mode 1 Issue 0 Performed 0
t4 D Mode 1 Issue 1 Performed 0
t3 C Mode 0 Issue 0 Performed 0
t2 B Mode 0 Issue 0 Performed 0
t1 A Mode 0 Issue 1 Performed 0
Miss
DRF region
sync region

(a) Before Miss

(b) After Miss

Figure 5: Store buffer behaviour upon a cache miss

4.3 Dual-mode store buffer

ROOW implements a dual-mode store buffer. That is, a store buffer than can simultaneously perform stores in-order or out-of-order depending to the type of store (mode bit): DRF or sync. Store buffers are content-addressable memories (CAM) that implement a circular buffer with a head and a tail pointer. Stores are inserted through the tail and removed from the head.

Sync stores behave as in a standard TSO store buffer. They are initiated in-order (from head to tail) and they are inserted in-order in the cache pipeline.[2] If the store hits in the cache, the write will be performed in-order. However, in case of a cache miss, subsequent stores in the cache pipeline would be reordered if they hit in the cache or have a sorter miss latency. To prevent this behaviour, when a store encounters a cache miss, the subsequent stores in the cache pipeline are squashed and they will have to be initiated again, in-order, once the missing store completes.

On the other hand, DRF stores may perform out-of-order. DRF stores are also initiated in-order and inserted in the cache pipeline. However, DRF stores are never flushed from the cache pipeline and do not need to be re-initiated. On a cache miss of a DRF store, no action is taken regarding subsequent stores, thus increasing memory level parallelism as more store misses will be in flight. DRF stores may therefore perform out-of-order and are initiated one after the other until the first sync store is encountered.

Note that on a cache miss for a sync store, DRF stores in the cache pipeline do not need to be flushed. Figure 5 shows an example of this scenario. Assume that the state of the store buffer is the one showed in Figure 5a. Stores A, B, C and D are issued in the cache pipeline to perform the write operation. Stores D, E, and F are DRF stores. Once store A, the store at the head, suffers a cache miss, the processor needs to re-issue all the issued sync stores except itself, as shown in Figure 5b. These stores, B and C, are flushed from the cache pipeline. However, store D can safely continue, being DRF, and therefore is not re-issued.

Store operations are retired from the store buffer in-order once they complete the write, regardless if the store is DRF or sync. That is, the head pointer of the store buffer simply moves to the next store. Removing DRF stores out-of-order, would improve store buffer utilization but in contrast would add complexity to the store buffer and make sequential semantics harder to fulfill, as we explain in the next section.

4.4 Guaranteeing sequential semantics

DRF stores to different locations can be reordered as per SC-for-DRF guarantees. However, stores to the same location have to adhere to sequential semantics even during DRF regions. That is, two stores to the same or overlapping addresses have to write to their memory location in-order. Similarly, a load following these two stores has to read the value written by the latter. Reordering stores however can break this simple rule if care is not taken.

The solution for correct store-to-load forwarding is to insert and retire stores from the store buffer in program order. This way, loads will always read from the latest store, as the store buffer mirrors the program order. Furthermore, despite allowing DRF stores to write to cache out-of-order, the store buffer is always coherent with the value in memory. This is thanks to the DRF guarantees that DRF stores do not race with accesses in other threads.

The solution to guarantee sequential semantics with respect to memory writes is a bit more elaborated. As stores are inserted in the cache pipeline in-order, in case of cache hits, the stores will respect program order. In case the first store misses in the cache, it will issue a request for write permission. Subsequent stores targeting the same address will also experience a cache miss, as they belong to the same cache block, and the subsequent store will be coalesced in the miss status holding register (MSHR) with the previous store, waiting for the miss to be resolved. As a consequence, the value of the first store will be updated with the value of the second store and the writes will appear to be performed in-order.

On the other hand, both DRF and sync stores can co-exist in the store buffer, and stores to the same address can appear in both regions. If we allow DRF stores to initiate the write before sync stores (as sync stores may be waiting for a previous store that missed in cache), and sync and DRF stores target the same address, then sequential semantics may be broken.

Our solution to this problem is to forbid reordering of DRF stores across sync stores, when the same address can appear in both the DRF and the sync region. Our compiler detects such regions through an alias analysis and inserts fences in between DRF and sync regions with aliasing stores. Fences inform the processor when it cannot reorder stores across regions of code. That means a DRF store cannot initiate the write to cache if a previous sync store has not completed and there is a fence in between. The fence enforces the order of aliasing stores from different regions and guarantees sequential semantics. These fences entail a very low performance cost even when they are inserted on every region boundary, since the transitions from DRF to sync are not frequent. This is shown in Section 7.

Figure 6 shows an example of a region boundary accompanied by a fence, due to store A (in sync) and store A' (in a DRF region) alias. In this case, store A' cannot initiate the write to cache until all

[2] Without loss of generality, this work assumes a four-stage cache pipeline and a single cache port for stores, so on each cycle a single store is initiated.

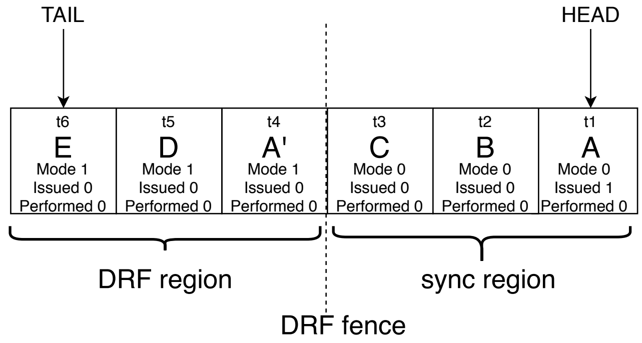


Figure 6: Store buffer behaviour upon fences

previous sync stores complete. In short, only when the head of the store buffer points to store A' can all the DRF stores initiate their write to cache.

4.5 Enabling the store buffer as a cache

Having exclusive access to memory during a DRF region provides the guarantee that the data written by the stores will not be modified by other cores. Therefore there is no need for removing the DRF stores (and the data) from the store buffer until the resources occupied by these stores are necessary, i.e., the store buffer is full.

Keeping DRF stores in the store buffer even after they complete makes the store buffer behave as a cache, as the number of loads forwarded from stores increases to a great extent, as previously noticed by Alves *et al.* [4]. The main advantage of our approach is that the use of the store buffer as a cache comes at zero-cost since it does not require any additional hardware to keep coherence between the data in the DRF stores of the store buffer and in memory. This is always respected by definition in DRF regions. The only modification required with respect to a standard store buffer is that the head of the store buffer is not updated when a DRF store completes, but when a new store needs to enter the store buffer and the latter is full.

Sync stores on the other hand need to be removed from the store buffer when they complete as the written data gets exposed to the coherence protocol and to reads and writes performed by other cores. Keeping them in the store buffer could cause inconsistency problems, unless extra hardware is added to expose coherence messages to the store buffer [4].

5 DISCUSSION

5.1 Non SC-for-DRF programs

Our model is well defined and can correctly execute legacy code and non SC-for-DRF programs. Legacy software would be automatically executed in the sync mode, thus providing TSO, since it does not contain any DRF annotations inserted by the compiler. Tools such as data race detectors (e.g., ThreadSanitizer [31], Fast&Furious [27]) that operate either at the compiler level or directly on the binary may be employed to analyze and annotate legacy code. Alternatively, one can extend our proposal with an interface for the programmer to hint the compiler and enable/disable our optimizations for selected parts of the code. This could also be beneficial for handling legacy code that âĂIJacceptsâĂİ data races, by disabling our optimizations for the racy accesses (marking them as single-access, in-order regions.) We leave such extensions for future work.

5.2 Support for Sequential Consistency

ROOW can efficiently support a Sequential Consistency (SC) memory model. The only requirement is that during sync regions, loads are not allowed to commit until the store buffer drains completely[3]. The impact of this restriction on execution time will be minimal as the sync regions in DRF programs contain a very small number of memory accesses (see Figure 1).

5.3 Store buffers for weak memory models

Apart from the simple integration in an x86-TSO store buffer (described before), *ROOW* can easily be applicable to store buffers designed for weaker memory models such as ARM [5] or Power [23], providing cost-effective store ordering (TSO) when required by the program. When executing a DRF region, stores would be naturally reordered, as this is the default mode in these processors. In contrast, during the execution of sync regions, store order can be enforced by stalling writes to memory blocks different than the one at the tail of the store buffer (allowing just coalescing of consecutive blocks). Although stalling writes may reduce the utilization of the store buffer, the impact of this action on performance would be minimal for SC-for-DRF programs since sync regions represent a negligible fraction of the code.

5.4 Debugging

Debugging parallel programs (before they become DRF) may seem extra challenging with our non-TSO hardware, as one can expect our out-of-order SB would reveal data races masked by the TSO memory model (and thus invisible on mainstream hardware).

Nevertheless, it is not only the underlying hardware that can hide or reveal such races. Conventional compilers perform instruction reordering invisible to the programmer as part of the standard optimizations (O2, O3), such as code motion, hoisting loop invariants, grouping memory instructions to increase memory-level-parallelism, etc. As long as the programming language memory model and the hardwareâĂŹs memory model do not provide guarantees for racy code, any such transformation can reveal the race. Hence, ROOW does not introduce a new problem as the compiler or other software optimization tools may have a similar effect for incorrect (racy) programs.

One simple debugging approach would be to (1) debug the sequential code; (2) debug the parallel code by compiling without the pass that delimits the regions (i.e equivalent to executing on the baseline TSO hardware) and fix data races; (3) enable our technique by re-compiling the code with our pass. If bugs surface only when ROOW is enabled but not on the baseline TSO hardware, this can actually indicate to the programmer that some data races are still present and could even help them hone in on the problem to isolate the data race.

[3] In our implementation, we opt for allowing loads to commit with pending stores in the store buffer, as SC is not required by existing x86 software.

Table 1: System parameters

Processor	
Processor Model	Intel Skylake
Fetch Width	5 instructions
Issue Width	8 ports
Allocation Queue	97 entries
Reorder Buffer	224 entries
Load Queue	72 entries
Store Queue + Store Buffer	56 entries
Memory	
Private L1 I&D caches	32KB, 8 ways, 4 hit cycles, pipelined
L1 prefetcher	Stride, degree 3
Private L2 cache	256KB, 8 ways, 12 hit cycles
Shared L3 cache	1MB per bank, 8 ways, 35 hit cycles
Directory	8 ways, 200% coverage of L2
Memory access time	160 cycles
Network Topology	2D Mesh

6 SIMULATION ENVIRONMENT

Our simulation infrastructure employs modified versions of Sniper [9] and the cycle-accurate GEMS simulator [24]. We simulate a multi-core processor providing a TSO consistency model and consisting of 8 out-of-order cores. Our processor mimics an Intel Skylake micro-architecture employing macro-op and micro-op fusion. Our processor employs a single circular queue for both the store queue and store buffer that utilizes better the resources as done in Intel architectures [17]. GARNET [2] is used to model the interconnect. We use CACTI [22] to model the store buffer and L1 cache using a 22nm technology in order to compute their energy consumption. Both the L1 and the SB use the high-performance model (hp) provided by CACTI and the store buffer is modelled as a CAM. The most relevant characteristics of the system used in our simulations are displayed in Table 1.

We explore four different configurations in the evaluation: (1) Our baseline is a TSO-like store buffer that writes in-order (*TSO*). (2) An oracle version (DRF_All) that sets the *Mode bit* to true permanently. That means, all stores belong to a DRF region and can be performed fully out-of-order. (3) Our ROOW proposal where the compiler inserts fences on each region boundary (*ROOW*). (4) Finally, our ROOW proposal with fences inserted only on region boundaries with aliasing stores residing in consecutive regions (similar to the xDRF analysis [18, 19]).

We run both parallel applications from the Splash-3 benchmark suite [30], which is a data-race-free version of the original Splash-2 [35] benchmark suite released before the C/pthreads memory model was updated to enforce SC-for-DRF, and the PARSEC 3.0 benchmark suite [7], which is a popular modern suit that complies with the C++ standard and therefore enforces SC-for-DRF. We use the *simmedium* inputs for barnes, blackscholes, cholesky, dedup, fft, fluidanimate, lu_cb, and lu_ncb and the *simsmall* inputs for fmm, ocean_cp, oceanncp, radiosity, radix, raytrace, streamcluster, swaptions, volrend, water_nsquared, and water_spatial. Results correspond to parallel region of the applications. The simulated benchmarks include a large variety of access patterns.

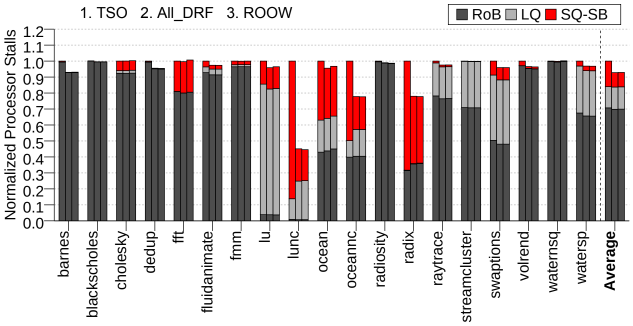


Figure 7: Processor stalls

7 RESULTS

In the motivation of this work, we already showed that the number of stores executed in *DRF regions* are dominating (Figure 1). In what follows, we analyze the performance benefits of performing the DRF stores out-of-order, focusing on the processor stalls and the ratio of store-to-load forwarding. We present the performance improvements considering several store buffer sizes and report the energy savings on the L1 cache and the store buffer achieved by ROOW.

7.1 Processor Stalls

Figure 7 provides a breakdown of the processor stalls classified by the reason of the stall reorder buffer (RoB) full, load queue (LQ) full, and store queue/store buffer (SQ/SB) full. SB/SQ stalls are significantly reduced in both *ROOW* and *All_DRF* for programs such as *LU-nc*, *ocean-nc* and *radix*, while in other cases the bottleneck shifts from the SB/SQ to either the RoB or the LQ. For example, in *ocean-nc*, *fluidanimate* and *LU-nc*, despite reducing the SB/SQ stalls, we observe higher LQ stalls stemming from speculative loads due to branch missprediction [29]. In contrast, in *radix* and *water-nsq*, RoB becomes the bottleneck (due to a higher number of instruction squashes, 5.4% more squashes for *radix* and 0.0012% less squashes in *water-nsq*). Other applications, such as *barnes*, *radiosity*, *raytrace* and *volrend* show less RoB stalls (due to fewer instructions being squashed, 2.08%, 1.1%, 1.79% and 2.38% respectively) along with less SB/SQ stalls. *FMM*, *cholesky*, *blackscholes*, *streamcluster* and *water-nsq* perform similarly in terms of stalls in all three versions. *FFT* is the only program that suffers more stalls than TSO, which as we will see translates to a slight performance loss. Overall *ROOW* achieves 7.11% less processor stalls compared to TSO and is almost on par with the oracle configuration, with 7.11% less stalls in *ROOW* and 7.21% in *All_DRF*.

7.2 Loads forwarded from stores

Keeping the stores in the store buffer even after completion increases the number of loads-forwarded-from-store, making the

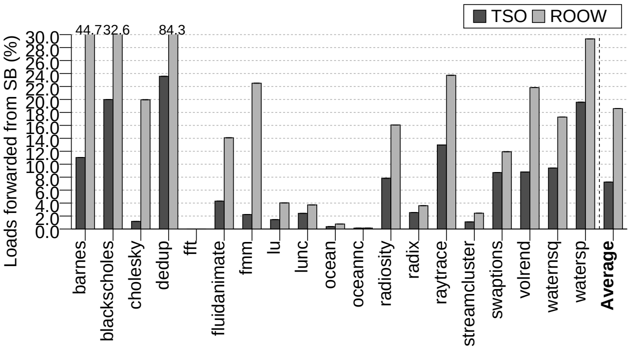


Figure 8: Percentage of loads forwarded from stores

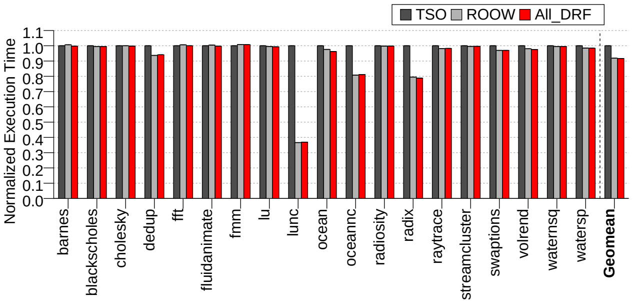


Figure 9: Normalized execution time with respect to an store buffer with 56 entries that implements TSO

store buffer act as a cache. Figure 8 reveals significant improvements in the number of loads-forwarded-from-store in ROOW compared to the baseline. *ROOW* provides outstanding increments in the number of loads-forwarded-from-store, namely 18.58% loads-forwarded-from-store compared to 7.24% in the baseline. The highest percentage is observed in *dedup* from 23.57% to 84.34% while lowest in *FFT* from 0.0143% to 0.0144%. This finding can be employed in reducing the energy consumption, as showed in the work done by Alves *et al.* [4] where their solution avoids the parallel search in both store buffer and the cache. We leave evaluation of this optimization for future work.

7.3 Execution time

Processor stalls degrade performance considerably and are therefore an important target for speeding up applications. The more stalls the applications encounter running on the baseline 56-entry store buffer, the higher the benefits of applying our technique. Figure 9 shows the applications' execution time normalized with respect to our baseline. We can see that programs like *LU-nc*, *ocean-nc* and *radix* show improvements in execution time (by 63.36%, 19.16% and 20.40%, respectively) because of less overall stalls as shown in Figure 7.

In particular, *LU-nc* shows an impressive improvement due to 19.70% less stalls in RoB and 77.48% less in the SB/SQ as shown in Figure 7. ROOW shows higher benefits for the non-contiguous applications, such as *LU-nc* and *ocean-nc*, as the number of different cache lines accessed and present in the SB at a particular time is larger. With contiguous applications, subsequent stores match to the same cache line, and stores will not be effectively reordered. Applications such as *blackscholes*, despite having all the instructions in DRF regions, do not show much improvement due to the fact that the baseline stalls very little (251 SB stalls, which is reduced to 10 in ROOW). In general, ROOW excels when the reordering opportunities are greater. All programs except *Barnes, FMM, Fluidanimate* and *FFT* show performance improvements. *FFT* suffers from high processor stalls compared to the baseline – 6% more SB/SQ stalls.

Overall, ROOW achieves 8.13% speed up compared to our baseline 56-entry store buffer that performs store operations in order.

7.4 Store buffer size impact on performance

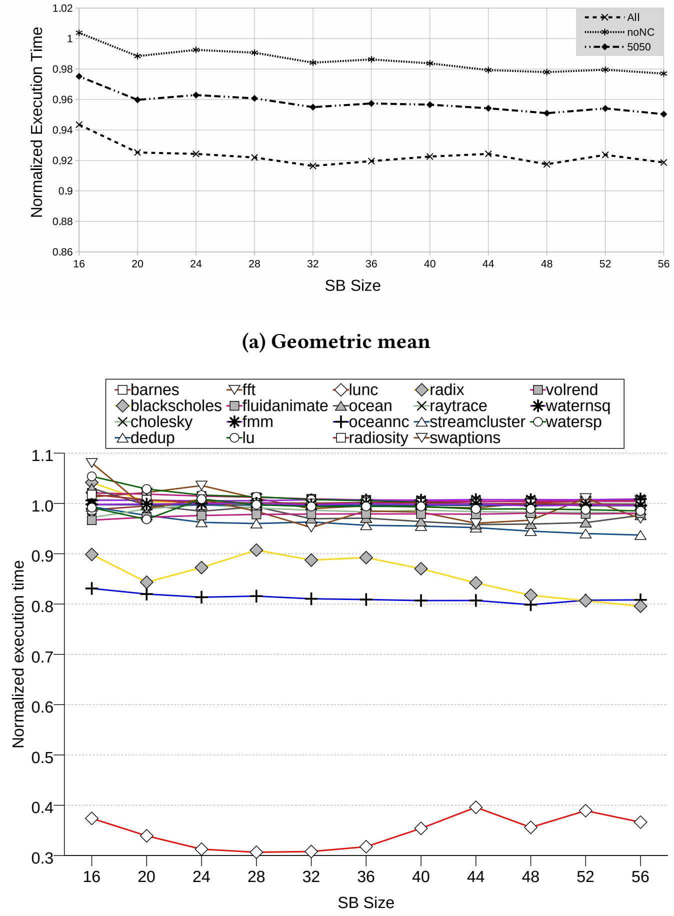
Larger store buffers do not automatically guarantee performance improvements as performing the search for data forwarding may takes more cycles. If the trend towards higher processor clock, wider pipelines, more execution units and large programs continues, the latency of search in the store buffer will become critical for performance. Along with increasing the latency searching the store buffer, larger store buffers also increase the energy expenditure.

In contrast, a small store buffer brings numerous benefits such as low power consumption and less hardware overhead, but it affects the overall performance as it increases the SB/SQ stalls. All modern day processor are a result of a trade-off between energy expenditure and speed. Our analysis shows that we can actually achieve low energy expenditure without degrading performance, by boosting performance even with a small store buffer.

Figure 10a shows the normalized execution time (geometric mean of all applications — label *All*) when varying the store buffer size. The baseline is a store buffer of 56 entries implementing TSO (as before). In addition, since most of the average performance gains come from non-contiguous (NC) applications, we first explain why they represent a interesting scenario and then we present also the average execution time with different weights for such applications in order to demonstrate that not all gains come from NC applications.

NC versions are interesting as they offer different memory access patterns increasing the pressure on the store buffer, as shown in Figure 7. Ocean-nc has the highest number of shared memory accesses in SPLASH (46%) out of which 1/3 are write operations [6]. Such versions could resemble the behaviour of database and web applications, which stress more the need for efficient store buffer designs [3].

When ignoring the NC applications (label *noNC* in Figure 10a), ROOW delivers the same average performance for a 16-entry store-buffer as the 56-entry TSO store-buffer. This results in about 50% reduction in the energy consumption of the store buffer and L1 cache (Figure 11). Without NC applications, the 56-entry ROOW store buffer achieves 2.3% performance improvement, on average, over the baseline configuration. When adding a 0.5 weighted average for LU and Ocean, since they are basically two versions of



(a) Geometric mean

(b) Details per application

Figure 10: Execution time for various store buffer sizes

the same application (contiguous and non-contiguous), ROOW still delivers improvements of 5% (label *5050* in Figure 10a).

Figure 10b shows the performance of each individual benchmark. Even with a store buffer as small as 16 entries, ROOW leads to performance improvements of 5.64%. Overall, a store buffer with 32 entries gives the highest performance of 8.35%

ROOW becomes particularly attractive in the context of a growing demand for low power and high performance, especially since it requires minimal modifications of the mainstream store buffer implementations.

7.5 Energy Consumption

Figure 11 shows the energy consumption of the store buffer and the L1 cache normalized with respect to a TSO store buffer with 56 entries. Results show the energy consumption of ROOW as the store buffer size decreases from 56 entries to 16 entries.

Overall, *ROOW* with 56 entries store buffer provides a small energy improvement of 6.01% with respect to a TSO with 56 entries store buffer. This improvement stems from a lower number of executed loads, as with ROOW misspeculation is detected earlier since the pipeline suffers less stalls. Executing less loads results in less store buffer snoops. Applications such as *barnes*, *LU*, *raytrace*, *volrend*, *water-nsq*, *blackscholes*, *dedup*, *fluidanimate* and *swaptions* show less store buffer snoop energy.

Table 2: Performance benefits and energy savings in ROOW

SB Size	Performance	Energy (SB and L1 cache)
16 entries	5.64%	45.16%
20 entries	7.47%	40.21%
24 entries	7.56%	35.62%
28 entries	7.80%	31.33%
32 entries	8.35%	27.25%
36 entries	8.04%	23.44%
40 entries	7.74%	19.96%
44 entries	7.57%	16.36%
48 entries	8.24%	13.2%
52 entries	7.63%	9.63%
56 entries	8.13%	6.01%

ROOW provides a design which facilitates the use of small store buffer in order to save energy (as shown in Figure 11) while still showing better performance than our baseline configuration (as shown in Figure 10). Figure 11 shows that reducing the store buffer size provides important energy improvements mainly due to reduction in number of store buffer snoops. With energy reductions of 45.16% compared to TSO with a 56-entry store buffer, *ROOW* with a 16-entry store buffer still provides 5.64% performance improvement. Similarly, Table 2 shows the overall improvement including both performance and energy (normalized with TSO 56 entries store buffer) for the different evaluated ROOW configurations.

7.6 Compiler analysis to remove superfluous fences

We employed a more powerful compile-time analysis to explore the possibility of reducing the number of fences that separate code regions, as mentioned in Section 4. We briefly recall that this analysis removes superfluous fences inserted at region boundaries (if stores from consecutive regions do not alias), without affecting the guarantees offered by ROOW.

On average, the advanced analysis removes 50% of fences. In some applications *(Barnes, Fluidanimate, Radiosity, Raytrace)* most fences are removed. However, in others *(FFT, LU, Ocean, Streamcluster)* fences cannot be removed because their barrier synchronization implies aliasing memory accesses across regions.

Despite the large reduction in the number of fences, the average performance improvement is around 1% compared to the ROOW version that inserts fences on each region boundary. The reasons are that (i) fences are not very frequent (scalable applications do not synchronize frequently) and (ii) DRF fences are not so critical in ROOW, as ROOW just resorts to TSO in their presence.

This study demonstrates that more complex compiler support add some performance benefits with minimal or zero hardware cost and that fences do not add a significant performance penalty, even when used in abundance.

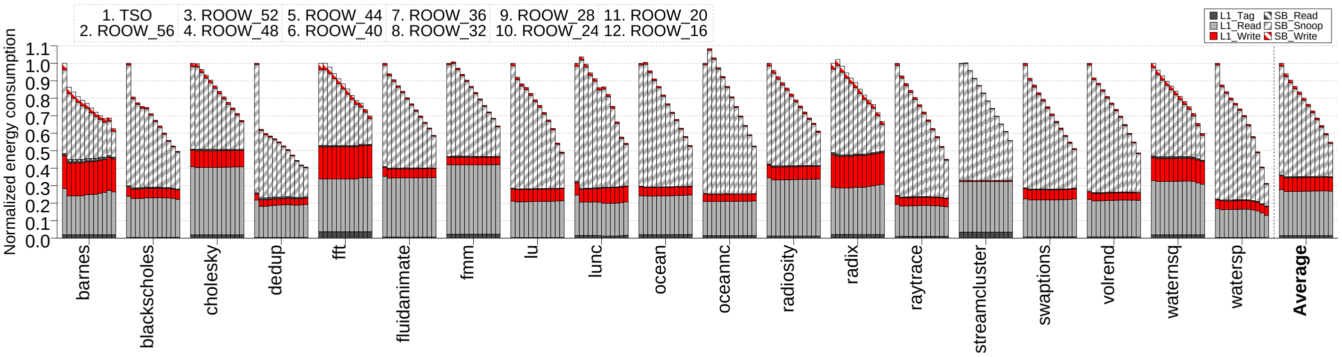


Figure 11: Energy consumption of the store buffer and L1 cache

8 RELATED WORK

A plethora of techniques have been proposed for efficient hardware designs that provide Sequential Consistency guarantees [3, 8, 10, 11, 14, 16, 33, 34]. Some of them rely on speculative mechanisms that require support to recover upon SC violations [8, 10, 14, 34]. Speculation can be managed while instructions are in the reorder buffer requiring moderate complexity [14], as implemented in commercial processors [17], or it can take place after instructions retire, aiming for higher performance [8, 10, 34]. The latter are more aggressive speculation techniques and often require major hardware changes. The more aggressive the speculation, the more costly becomes to check and recover from SC violations. In consequence, aggressive speculation techniques are not implemented in any real processor because of their complexity. In contrast, other techniques employ software-hardware co-designs and allow reordering of memory operations in a non speculative manner [3, 11, 33], either at compile-time [3], at the cache coherence protocol level [11], or at the store buffer level [33]. The latter one is therefore the most relevant to our proposal, as we do not perform speculation and target store buffer efficiency. We detail in what follows a selection of the most relevant techniques.

Singh *et al.* [33] propose a software-hardware co-design in which the compiler provides information about which accesses can be reordered in a dedicated store buffer. Their design implements two store buffers (out-of-order for safe stores –unshared– and in-order for unsafe ones –shared–, respectively) and store operations are allocated in one or the other depending on their type. This design with two store buffers increases the complexity of the solution and adds a very high hardware overhead (e.g. for the logic unit that selects the store buffer to snoop in order to forward data to load instructions). Also, if a program consists of numerous shared stores or vice-versa unshared stores, this design can lead to underutilization of the hardware along with generating numerous stalls in the corresponding store buffer. Since in our design the processor uses a single store buffer, the hardware is always utilized effectively and more efficiently. More importantly, our design has considerably less overhead in terms of added hardware, yielding ROOW ready to be easily integrated in current processors' design, a key feature of our solution.

Moreover, the compiler support employed by Singh *et al.* [33] performs a data-based classification. This means their design forces all accesses to the same memory address to be classified as having the same type and to be placed in the same store buffer in order to solve the notorious store-store reordering and store-to-load forwarding problems. Nevertheless, this represents a serious drawback, limiting the reordering to only 70% of the stores, on average (and down to 50% of the stores in Barnes). In contrast, ROOW enables the reordering of 99% of the stores (similarly for Barnes), thanks to exploiting the DRF semantics of the code and performing a region-based classification of the memory accesses. Thus, our per-access classification is more fine-grain and more precise than the data-based classification employed by Singh *et al.* [33]. Furthermore, our region-based classification requires minimum information to be conveyed to the hardware –one bit (flag) per region, enough to signal the type of the current region (in-order vs. out-of-order) compared to one bit per access [33]. The more accurate classification, simpler design, and minimum hardware extensions relative to mainstream implementations make our proposal a more appealing solution to enabling store reordering at the store buffer level compared to its predecessors.

BulkSC [10] dynamically group sets of consecutive instructions into chunks and allow them to execute in isolation. This design requires modifications in the memory hierarchy such as the addition of arbiters. Communication between cores increases with BulkSC, which in turn increases network traffic by 3-13%. ROOW on the other hand does not require modifications in the memory hierarchy and it does not incur extra network traffic.

Store-wait-free [34] seek for the same goal as ROOW, remove stalls due to the store buffer. To this end, it implements a large store buffer (named SSB). It also requires adding sub-block valid bits to the performance-critical L1 cache, and stores write in cache speculatively, which requires undo support. ROOW entails minimal hardware extensions, never undoes stores, and does not require any L1 cache modification.

Ros and Kaxiras [28] propose coalescing stores in the store buffer and avoid breaking the store order by performing stores in atomic groups. In contrast to our approach where DRF stores can perform completely out-of-order, stores in an atomic group perform following a globally defined order. On the other hand coalescing stores can be applied to ROOW (either for DRF stores or sync stores). This can further improve our performance as coalescing may reduce the occupancy of the store buffer. The counter part of this approach is that it requires separate Store Queue and store buffer to fully get the benefits of coalescing.

Duan *et al.* [12] propose WeeFence. WeeFence uses lightweight fences (WFences) that allow post fence memory operations to perform before the fence commits and adds extra hardware to ensure correctness. In their evaluation, the authors increase the number of fences in the programs to provide SC behaviour when running on a more relaxed hardware, and then replace the inserted fences with WFences. A similar strategy could be used instead of ROOW, that is, to completely relax the order of the stores in the store buffer and enforcing the order with software fences. However, WFences increase execution time and add complexity to the system for checking correctness, which makes WeeFence a less appealing solution. ROOW only uses DRF fences to separate the DRF and sync regions, which is not a frequent scenario in scalable applications. On average, ROOW encounters only 0.0079 fences per 1K instructions, while WeeFence encounters 23.9 fences per 1K instructions (applications *LU-nc, LU* are only simulated in ROOW while *canneal* and *pbzip2* are only simulated in WeeFence).

Recently, Alves *et al.* [4] have proposed to increase the hit ratio of the store buffer by keeping performed writes in the store buffer (store buffer cache). This significantly increases the number of loads forwarded from stores. The resulting store buffer requires an external signal (from the memory system) or snoop port to inform the store buffer about L1 cache invalidations and evictions. Additionally, the authors propose to perform the access to the store buffer and to the L1 cache sequentially, and predict in advance whether the store buffer will hit, thus reducing the number of L1 accesses and resulting in less energy consumption. ROOW leverages the use of the store buffer as a cache for DRF code, with the key insight that DRF code does not require that the store buffer receives external signals, since invalidations to DRF data cannot occur during DRF regions. In our design, we increase the number of loads forwarded from stores significantly by storing the stores in the store buffer as long as possible (until the store buffer is full). One simple extension to predict the hit can be used to reduce the power consumption along with a performance boost.

9 CONCLUSIONS

The store buffer is an essential component of modern day out-of-order processors, as it allows the store instructions to retire before performing, taking the long latency of stores out of the critical path. In this paper we introduce ROOW, a software-hardware co-design technique that uses compiler support and requires minor modifications in the design of the store buffer in order to reduce processor stalls due to the store buffer capacity. ROOW performs store operations out-of-order within the code regions indicated as safe by the compiler. ROOW leads to a speed up of 8.13% on average (+1% when performing alias analysis to remove fences) and reduces processor stalls by 7.11% compared to a mainstream store buffer. Our design also allows the use of the store buffer as a cache for free (without adding or changing any hardware) which gives us 18.54% loads-forwarded-from-stores compared to 7.24% in the baseline. We have also conducted a sensitivity analysis which shows that we can reduce the size of the store buffer from 56 entries to 16, while still improving performance by 5.64% compared to the baseline configuration.

Overall the design has close to zero hardware overhead, as ROOW entails a negligible memory overhead bits (N+1 bits, where N is the number of entries in the store buffer), so it can be easily adopted by modern processors.

ACKNOWLEDGMENTS

This work was supported by the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No 819134) and by the Ramón y Cajal Research Contract (RYC2018-025200-I).

REFERENCES

[1] Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering – A New Definition. In *17th Int'l Symp. on Computer Architecture (ISCA)*. 2–14.

[2] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. 2009. GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator. In *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*. 33–42.

[3] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and David Wong. 2009. BulkCompiler: High-performance Sequential Consistency through Cooperative Compiler and Hardware Support. In *42nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*. 133–144.

[4] Ricardo Alves, Alberto Ros, David Black-Schaffer, and Stefanos Kaxiras. 2019. Filter Caching for Free: The Untapped Potential of the Store Buffer. In *46th Int'l Symp. on Computer Architecture (ISCA)*. 436–448.

[5] ARM. 2015. *ARM Architecture Reference Manual ARMv8-A*.

[6] Christian Bienia, Sanjeev Kumar, and Kai Li. 2008. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on Chip-Multiprocessors. In *112 Proceedings of the IEEE International Symposium on Workload Characterization (IISWC âĂŽ08)*. 47–56.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*. 72–81.

[8] Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch. 2009. InvisiFence: Performance-transparent Memory Ordering in Conventional Multiprocessors. In *36th Int'l Symp. on Computer Architecture (ISCA)*. 233–244.

[9] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations. In *Conf. on Supercomputing (SC)*. 52:1–52:12.

[10] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. 2007. BulkSC: Bulk Enforcement of Sequential Consistency. In *34th Int'l Symp. on Computer Architecture (ISCA)*. 278–289.

[11] 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 *20th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*. 155–166.

[12] Yuelu Duan, Abdullah Muzahid, and Josep Torrellas. 2013. WeeFence: Toward Making Fences Free in TSO. In *40th Int'l Symp. on Computer Architecture (ISCA)*. 213–224.

[13] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *2009 Conf. on Programming Language Design and Implementation (PLDI)*. 121–133.

[14] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1991. Two Techniques to Enhance the Performance of Memory Consistency Models. In *20th Int'l Conf. on Parallel Processing (ICPP)*. 355–364.

[15] Chris Gniady and Babak Falsafi. 2002. Speculative sequential consistency with little custom storage. In *11th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*. 179–188.

[16] Mark D. Hill. 1998. Multiprocessors Should Support Simple Memory-Consistency Models. *IEEE Computer* 31, 8 (Aug. 1998), 28–34.

[17] Intel. 2016. Intel® 64 and IA-32 Architectures Optimization Reference Manual. www.intel.com.

[18] Alexandra Jimborean, Per Ekemark, Jonatan Waern, Stefanos Kaxiras, and Alberto Ros. 2018. Automatic Detection of Large Extended Data-Race-Free Regions with

Conflict Isolation. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 29, 3 (March 2018), 527–541.

[19] Alexandra Jimborean, Jonatan Waern, Per Ekemark, Stefanos Kaxiras, and Alberto Ros. 2017. Automatic Detection of Extended Data-Race-Free Regions. In *15th Int'l Symp. on Code Generation and Optimization (CGO)*. 14–26.

[20] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers (TC)* 28, 9 (Sept. 1979), 690–691.

[21] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd Int'l Symp. on Code Generation and Optimization (CGO)*. 75–88.

[22] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *2011 Int'l Conf. on Computer-Aided Design (ICCAD)*. 694–701.

[23] Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. *A Tutorial Introduction to the ARM and POWER Relaxed Memory Models*. Technical Report.

[24] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. 2005. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *ACM SIGARCH Computer Architecture News* 33, 4 (Sept. 2005), 92–99.

[25] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. 1996. *Pthreads Programming*. O'Reilly & Associates, Inc.

[26] OpenMP Architecture Review Board. 2008. OpenMP Application Program Interface Version 3.0. http://www.openmp.org/mp-documents/spec30.pdf

[27] Alberto Ros and Stefanos Kaxiras. 2015. Fast&Furious: A Tool for Detecting Covert Racing. In *6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA) and 4th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (DITAM)*. 1–6.

[28] Alberto Ros and Stefanos Kaxiras. 2018. Non-Speculative Store Coalescing in Total Store Order. In *45th Int'l Symp. on Computer Architecture (ISCA)*. 221–234.

[29] Alberto Ros and Stefanos Kaxiras. 2018. The Superfluous Load Queue. In *51st IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*. 95–107.

[30] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. 2016. Splash-3: A Properly Synchronized Benchmark Suite for Contemporary Research. In *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*. 101–111.

[31] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Workshop on Binary Instrumentation and Applications (WBIA)*. 62–71.

[32] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97.

[33] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. 2012. End-to-End Sequential Consistency. In *39th Int'l Symp. on Computer Architecture (ISCA)*. 524–535.

[34] Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2007. Mechanisms for Store-Wait-Free Multiprocessors. In *34th Int'l Symp. on Computer Architecture (ISCA)*. 266–277.

[35] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *22nd Int'l Symp. on Computer Architecture (ISCA)*. 24–36.