CELLO: Compiler-Assisted Efficient Load-Load Ordering in Data-Race-Free Regions

Sawan Singh[*], Josue Feliu[†], Manuel E. Acacio[*], Alexandra Jimborean[*], Alberto Ros[*]

[*]Computer Engineering Department, University of Murcia, Murcia, Spain
[†]Computer Engineering Department, Universitat Politècnica de València, València, Spain
Email: singh.sawan@um.es, jfeliu@disca.upv.es, meacacio@um.es, alexandra.jimborean@um.es, aros@ditec.um.es

***Abstract*—Efficient Total Store Order (TSO) implementations allow loads to execute speculatively out-of-order. To detect order violations, the load queue (LQ) holds all the in-flight loads and is searched on every invalidation and cache eviction. Moreover, in a simultaneous multithreading processor (SMT), stores also search the LQ when writing to cache. LQ searches entail considerable energy consumption. Furthermore, the processor stalls upon encountering the LQ full or when its ports are busy. Hence, the LQ is a critical structure in terms of both energy and performance.**

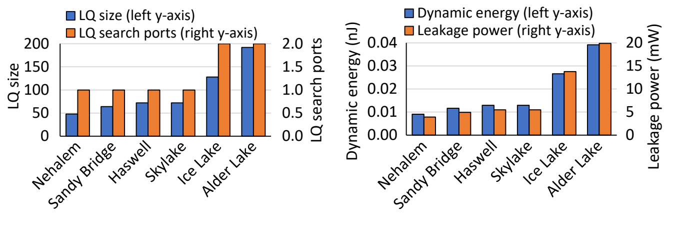
In this work, we observe that the use of the LQ could be dramatically optimized under the guarantees of the data-race-free (DRF) property imposed by modern programming languages. To leverage this observation, we propose CELLO, a software-hardware co-design in which the compiler detects memory operations in DRF regions and the hardware optimizes their execution by safely skipping LQ searches without violating the TSO consistency model. Furthermore, CELLO allows removing DRF loads from the LQ earlier, as they do not need to be searched to detect consistency violations.

With minimal hardware overhead, we show that an 8-core 2-way SMT processor with CELLO avoids almost all conservative searches to the LQ and significantly reduces its occupancy. CELLO allows i) to reduce the LQ energy expenditure by 33% on average (up to 53%) while performing 2.8% better on average (up to 18.6%) than the baseline system, and ii) to shrink the LQ size from 192 to only 80 entries, reducing the LQ energy expenditure as much as 69% while performing on par with a mainstream LQ implementation.

I. Introduction

Sequential Consistency (SC) [29] is one of the strongest memory consistency models. It preserves the program order of all memory accesses and thus offers intuitive semantics to programmers. On the other hand, exploiting memory level parallelism (MLP), which relies on reordering instructions to hide long-latency memory operations, is key for performance. The Total Store Order (TSO) memory consistency model [51], supported by Intel and AMD processors, achieves a good balance between programmability and performance by allowing load instructions to be effectively reordered with respect to store instructions. This way, latency of store operations is hidden by allowing them to perform out of the processor's critical path, at the cost of relaxing the consistency model semantics.

Despite TSO preserves the load-load order, in hardware, loads are speculatively reordered with respect to each other to improve MLP [20]. This speculative execution requires



(a) LQ size and search ports (b) Dynamic energy per search and leakage power

Fig. 1: Evolution of LQ characteristics across different generations of Intel processors.

all loads to be placed in order in the load queue (LQ), a content-addressable memory (CAM) structure that is searched on certain events to prevent exposing speculative ordering violations and is coupled with a mechanism to recover from misspeculation when detected.

The LQ is one of the most critical structures in a processor, in terms of performance and energy [19]. It needs to keep all in-flight loads in order and support priority searches which, for performance reasons, are done associatively. Furthermore, it is searched frequently: each time a store executes, in order to safeguard sequential semantics [36], and on any invalidation or cache eviction, to enforce the load$\rightarrow$load order [20]. In addition to the high contention on its search (i.e. snoop) ports, the LQ also stalls out-of-order (OoO) processors when it becomes full. Consequently, as depicted in Figure 1a, both LQ size and search ports have increased over the last few years. For example, Intel processors have increased the LQ size from 48 entries in the Nehalem microarchitecture [25] to 192 in the Alder Lake [46]. The number of search ports also grew in Ice Lake [39], adding a second search port to the LQ.

However, the increase in LQ size and search ports comes with a high energy cost. The LQ is a power-hungry and latency-sensitive structure as it needs to support fast associative searches on addresses [19]. Figure 1b shows the rise in energy consumption per search and leakage power as the LQ size and search ports increase. The energy consumption of the LQ is dominated by searches and almost quadrupled from Skylake to Alder Lake. Similarly, leakage power quadrupled across the same microarchitectures.

The simultaneous multithreading (SMT) paradigm, nowadays adopted by Intel, AMD, and IBM in their high-

TABLE I: LQ searches in non-SMT and SMT processors.

Event	non-SMT	SMT
Store computes its address	Yes (D-spec search)	
Store performs	No	Yes (M-spec search core)
Cache Inv. & Evictions	Yes (M-spec search memory system)	

performance processors, further exacerbates the criticality of the LQ. SMT enables a core to execute multiple threads simultaneously, while sharing most of the pipeline resources, including the LQ. Moreover, SMT threads also share the coherent state of the cachelines in the L1, requiring additional LQ searches to prevent an SMT thread from exposing a speculative load$\rightarrow$load reordering to another thread co-running in the same core [16]. Namely, on each store, a thread must search the LQs of the co-running threads in the SMT core when writing from the store queue (SQ) to the L1. This almost doubles the energy consumption of the LQ and the search ports contention since, in an SMT, every store needs to search the LQ twice: when it executes and when it writes to the cache.

Since associative LQ searches are expensive and contention in the LQ search ports can be high, previous proposals try to reduce LQ searches either through software-hardware co-designs [24] or pure hardware solutions [9], [22], [43], [47]. However, these approaches do not target consistency-related searches [24], and increase the number of accesses to the L1 cache [9], [22], [47] or to the SQ [43].

In this work, we aim to reduce pressure on the LQ in SMT processors by leveraging data-race-free (DRF) guarantees. Data race freedom is a property that requires that all data sharing among threads is properly guarded by synchronization mechanisms. Most modern programming languages [1], [6], [26] impose data race freedom and offer no correctness guarantees in the presence of data races. Thus, DRF programming became a standard. Exposing the precise sharing of data to the hardware represented a cornerstone for architectural and micro-architectural optimizations, with a long history of DRF-enabled hardware optimizations [4], [11], [23], [32], [41], [53]. To the best of our knowledge, however, none of these optimizations has tackled consistency searches in the LQ of an SMT processor.

We exploit the DRF property and enable CELLO: Compiler-assisted Efficient Load-Load Ordering in data-race-free regions. CELLO broadens the compiler to delineate the DRF regions of code exposing the synchronization operations (thus, it does not require manual changes in the source code), and transmits this information to the hardware using a dedicated instruction. In hardware, CELLO implements a simple mechanism that filters unnecessary LQ searches, guaranteeing the load-load ordering in a more efficient way. CELLO allows the processor to i) virtually eliminate the LQ searches required to detect consistency violations, and ii) reduce LQ occupancy by removing loads that do not require searches.

Therefore, CELLO enables a more efficient utilization of the LQ when running SC-for-DRF applications, such that:

- The number of LQ searches is almost halved, greatly reducing the energy expenditure in one of the most power-hungry structures of the processor. Simulation results show that for an 8-core 2-way SMT processor, CELLO filters 47% of the LQ searches, reducing LQ energy expenditure by 33% on average (up to 53%) and improving performance by 2.8% on average (up to 18.6%), without affecting consistency guarantees.
- DRF loads can safely exit the LQ earlier, before committing, without exposing consistency violations. This allows shrinking the LQ size from 192 to 80 entries, reducing the LQ energy consumption and area by 69% and 56%, respectively, while still performing on par with a state-of-the-art design.

Overall, CELLO *maintains hardware simplicity by requiring minimal modifications of the processor, while greatly reducing the energy consumption of the LQ and improving performance under the same consistency guarantees.*

II. Background

In this section, we first describe how (non-SMT) processors guarantee sequential semantics and correctness under the speculative out-of-order execution of loads. Next, we briefly introduce the SMT paradigm and discuss the additional LQ searches they require to prevent exposing speculative load-load reorderings. Table I summarizes the LQ searches performed by non-SMT and SMT processors. Finally, we present the SC-for-DRF consistency model.

A. Sequential semantics

Sequential semantics restrict reordering memory accesses to the same memory location (dependencies) within a single thread and must be respected by any program to offer an intuitive behavior. Although TSO allows a store to perform after a younger load in program order, when both of them target the same memory location, in order to preserve sequential semantics, the load should read the value generated by the store if no other thread wrote that location in the interim [12], [36]. To this end, every load searches the SQ[1] in parallel with the memory access. On a hit, the store forwards the value to the load.

Yet, when loads execute, the target address of older stores may be unknown. These loads can execute speculatively (dependency-speculative or simply D-speculative [40]). To detect speculative executions that break sequential semantics, when stores resolve their address, they search the LQ looking for any matching younger D-speculative load. If found, the load and the subsequent instructions are squashed and re-executed. A load executes non-D-speculatively when all older stores have already computed their addresses.

B. Speculative load-load reordering

Executing load instructions in program order severely limits MLP, and hence, performance. To address this limitation, at the same time that the load-load program order appears to be respected, as required by a TSO consistency model,

[1]In this work, we refer to SQ as a unified store queue and store buffer.

out-of-order cores allow loads to execute *speculatively* out-of-order [20]. In this scenario, a load-load re-ordering can occur, for example, when an older load misses in the cache while a younger load hits. The younger load is therefore speculative (M-speculative following the terminology of Duan et al. [14]) when an older load has not yet performed and remains speculative until the older load performs. If the load-load reordering is detected by another core, e.g., a remote store to the same address performs in the interim, the speculative load and subsequent instructions are squashed and re-executed.

The LQ plays an essential role to guarantee correctness under speculatively out-of-order execution by allowing store operations to detect M-speculative loads to the same address in other cores. In particular, when a core receives an invalidation, generated by a remote store operation, for an address that matches a speculative load in the LQ, the load is squashed. This matching is detected by associatively searching the LQ. In addition to invalidation requests, cache evictions also cause loads to be squashed because after evicting a cacheline, subsequent invalidations for that line would not be received. Therefore, on each received invalidation and local cache eviction, the core searches its LQ for any speculative load that retrieved data from the invalidated/evicted cacheline.

C. SMT and extensions for speculative load-load reordering

SMT enables a core to issue instructions from different threads in the same cycle [56]. Exploiting thread-level parallelism makes SMT cores very efficient to hide the stalls of a thread by executing instructions from a different co-running thread. This way, SMT improves the throughput of the core. To minimize the overhead of implementing SMT, the core's resources are shared as much as possible. For instance, the front-end stages typically operate on a single thread and are time-shared among threads. Only resources that are critical for performance are replicated. Back-end resources, including the issue queue, the reorder buffer (ROB), the LQ, and the SQ are shared among threads. Its significant performance benefits, along with its low hardware overhead, have led the major manufacturers to implement SMT in their high-performance processors.

As discussed earlier, a speculative reordering that violates the load$\rightarrow$load order in a thread could be exposed by stores performed by a different thread. In an SMT processor, both threads can run in the same SMT core, sharing the coherent state of the cachelines in the L1. This means that a thread will not receive a coherence invalidation from a store performed by a co-running thread (in the same SMT core). Without such an invalidation, no LQ search is triggered to check for potential load$\rightarrow$load order violations.

To address this issue, in an SMT core, each store that writes from the SQ to the L1 triggers a search in the core's LQ looking for matching speculative loads from the co-running threads. In order to avoid increasing the write latency, this LQ search is typically performed in parallel with the cache write. Nevertheless, this solution exacerbates contention at the LQ search ports. While stores from threads running in different cores are greatly filtered by the coherence protocol (a core only sees these writes when it holds the corresponding cacheline in any of its private cache levels), there is no filtering for cache writes performed by threads co-running in the SMT core. In practice, this means that in an SMT core, each store needs to search the LQ twice: when it executes, to squash D-speculative loads from the same thread, and when it writes to the L1, to squash M-speculative loads from the other threads.

Since many LQ searches are triggered conservatively, Hilton and Roth [22] proposed filtering the unnecessary ones. Their solution adds a bit-vector to each cacheline in the L1, called SMT-directory, to keep track of which threads have read each cacheline. When a store in the SQ is ready to write, it checks the SMT-directory: If a different thread, running in the same SMT core, has read the cacheline, performing the write requires searching the core's LQ; otherwise, the LQ search is skipped. This approach greatly filters the LQ searches when stores write, as many cachelines are not shared among threads. However, it has a hardware overhead of N bits per cacheline, with N equal to the number of SMT threads per core. Furthermore, it complicates the write logic, which is on the critical path. The SMT-directory should be retrieved from the L1 when performing the write and should be checked before the write can actually be performed. If another thread has read the cacheline, an LQ search is needed and the write needs to be squashed and issued again along with the LQ search. Writing to the L1 before the LQ search completes could expose load$\rightarrow$load order violations. Consequently, writes that require an LQ search double their latency.

D. The SC-for-DRF consistency model

SC is a strong memory model that offers the most intuitive behavior for a shared-memory multithreaded program. However, it is also the most restrictive, yielding significant performance penalties compared to more relaxed memory models. Prior work [2] and processor manufacturers [5] have opted for weaker memory models, as they offer greater performance potential, but at the cost of programming complexity. To address this shortcoming, SC-for-DRF [2], [17] redefines the weak ordering providing sequential consistency guarantees for DRF software. Interestingly, widely used programming languages such as Java, C, and C++ adhere to the SC-for-DRF model and demand DRF programs to guarantee SC. In contrast, in the presence of data races, most programming languages have undefined semantics (e.g., C++) or, at best, provide weak guarantees.

The SC-for-DRF consistency model [2] is weaker than SC, as it provides the SC guarantees only for DRF software, but it enables powerful optimizations, both in software at compile-time or in hardware during execution. Additionally, the hardware can be greatly simplified and optimized for software complying with the SC-for-DRF consistency model.

In SC-for-DRF, potentially racy accesses must be guarded by synchronization, entailing that they will execute sequentially. The regions of code delimited by synchronization operations, denoted in this work as *DRF regions*, offer the guarantee

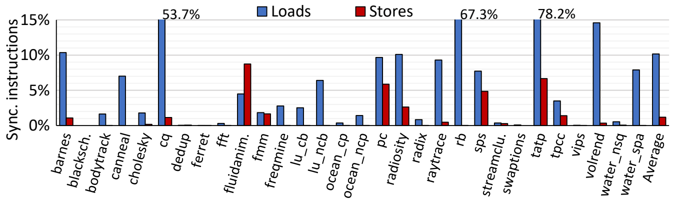


Fig. 2: Percentage of load and store operations found in synchronization code (sync) at runtime.

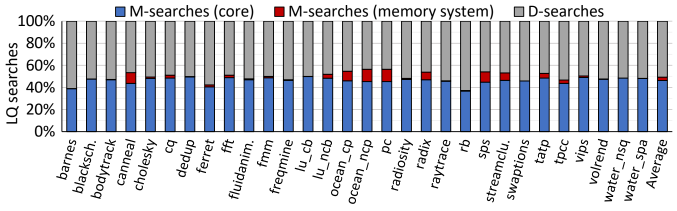


Fig. 3: Percentage of LQ searches due to memory consistency events and memory dependencies.

that writes do not target the same memory location as other concurrent memory accesses (see Fig. 4). Consequently, the memory operations performed by a thread during a DRF region remain "invisible" to the other threads until the end of the DRF region. This guarantee allows performing memory accesses in any order during DRF regions as long as sequential semantics are respected. DRF region boundaries ensure that memory accesses perform and become visible to the other threads at the end of the region. Thus, boundaries include not only synchronization operations but also fences, which prevent the reordering of memory operations across them.

We do not advocate for weakening the TSO model, but for minimally extending the x86 ISA to optimize the LQ usage for improving the performance of DRF programs. To this end, CELLO selectively filters LQ searches for DRF software, and provides the same consistency guarantees as TSO. For legacy software not compiled with our compiler, CELLO simply disables the optimizations and provides TSO guarantees, as described in Section V-B.

III. Motivation

Although many applications are DRF by design, most processors (e.g. Intel and AMD) do not exploit this property and implement a stronger consistency model, such as TSO, leaving significant performance potential untapped when executing DRF applications. Our goal is to exploit DRF properties in the processor design. First, we show that DRF accesses prevail in parallel applications. To identify DRF loads and stores, the compiler delineates DRF and synchronization regions of code, the latter being regions of code that contain synchronization operations (see Section IV-A for details about the delineation). Then, we show that almost half of the LQ searches are triggered to prevent consistency violations. Exploiting the DRF property of both loads and stores, the vast majority of these LQ searches can be avoided, greatly reducing the LQ energy consumption without negatively impacting performance.

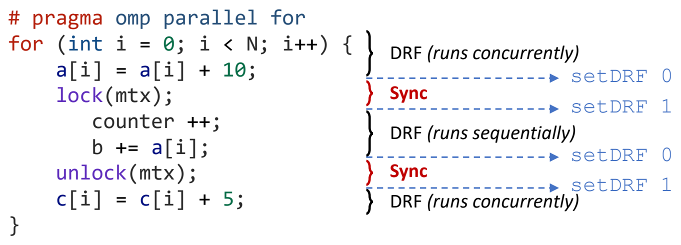
The dynamic percentage of load and store operations that are part of synchronization code (non-DRF) for the applications evaluated in this work on an 8-core 2-way SMT processor (see Section VI for details about the methodology) is shown in Figure 2. Generally, the percentage of sync stores is virtually 0. Only in some applications, such as *fluidanimate* and *radiosity* and the synchronization-intensive *pc*, *sps*, and *tatp*, more than 2% of the stores belong to sync regions. On average, only 1.2% of the stores belong to sync regions, yielding almost all LQ searches when stores write unnecessary. Similarly, the percentage of sync loads is small in most applications. Only three synchronization-intensive workloads (*cq*, *rb*, and *tatp*) show a much larger percentage because threads frequently spin on locks trying to acquire them. Despite these three workloads, the percentage of loads that belong to sync regions is 10.2% (and only 3.9% without these three workloads).

Figure 3 shows the percentage of LQ searches that are required either to maintain the correct load-load ordering (labeled as *M-searches*, as they target M-speculative loads) or to enforce memory dependencies (labeled as *D-searches*, as they target D-speculative loads). M-searches are further divided depending on their source: *M-searches (Memory system)* are triggered after invalidations or evictions that reach the private caches, while *M-searches (Core)* are triggered on the core's LQ when stores write from the SQ to the L1. M-searches range from 37% (*rb*) to 57% (*ocean_ncp*) depending on the application and, on average, represent 49% of the LQ searches. Since i) most stores are DRF and ii) frequently the LQ contains just DRF loads, an overwhelming majority of the LQ M-searches are superfluous and can be avoided by conveying the DRF information to the processor. Alleviating these searches turns to lower LQ energy consumption and search port contention.

IV. CELLO: Efficient Load-Load Ordering

Given that most modern programming languages demand DRF programs to guarantee SC, nowadays software usually adheres to the SC-for-DRF consistency model. Most processors, however, do not leverage this property, missing out on enormous optimization potential. To exploit this property, we propose CELLO, a software-hardware co-designed approach to efficiently guarantee the load$\rightarrow$load order in data-race-free regions. The CELLO compiler classifies memory accesses within *synchronization regions* (denoted as *sync* regions) and *synchronization-free regions* (denoted as *DRF* regions), performing a *region-based classification of accesses* [27] rather than a data-based classification [54]. The collected information is transmitted to the hardware through a new dedicated instruction. With (minimal) hardware extensions and based on the DRF status of the memory operations involved in each LQ search, CELLO allows loads to i) execute out of order while safely skipping superfluous LQ searches, and ii) exit the LQ early when they do not need to be checked against consistency violations.

Next, we explain how the compiler delineates DRF regions, detail the hardware modifications required by CELLO, and



```
# pragma omp parallel for
for (int i = 0; i < N; i++) {
    a[i] = a[i] + 10;
    lock(mtx);
       counter ++;
       b += a[i];
    unlock(mtx);
    c[i] = c[i] + 5;
}
```

Fig. 4: Example of code showing a parallel SC-for-DRF program and the delineated DRF regions and sync operations.

describe how the compiler conveys the DRF information to the hardware. Finally, we discuss how CELLO enables a more efficient load-load ordering during DRF regions.

A. *Compiler support to delineate DRF regions*

The CELLO compiler support is implemented as an LLVM pass [30] to expose the synchronization operations as sync regions, which effectively delineates the DRF code regions [27]. It expects as input code that meets the SC-for-DRF semantics, as required by most modern programming languages standards. Figure 4 shows an example of an SC-for-DRF program with the delineated DRF and sync regions. DRF regions can either be part of the parallel code regions and thus run concurrently, or part of the synchronized code regions and thus run sequentially. Note that the synchronized code regions, in contrast to synchronization operations, are DRF by definition, since they are executed by one thread at a time.

The compiler marks the delineated regions with a dedicated instruction (`setDRF val`), as shown in Figure 4. The operand `val` is a single bit that indicates the beginning of a DRF region (`val` $= 1$) or a sync region (`val` $= 0$) for the thread executing the instruction. Processors that do not support the `setDRF` instruction can simply treat it as a *nop* operation.

The compiler identifies synchronization operations that correspond to the standard mechanisms supported in widely used libraries. In particular, we offer support for pthreads [37] and OpenMP libraries [38], but CELLO, as an LLVM pass, can be easily extended to recognize new synchronization mechanisms. The memory operations residing in the sync regions must be executed in-order to ensure correctness and to preserve the TSO guarantees. During the execution of DRF regions, in contrast, memory operations can be performed out of order while inherently preserving the TSO guarantees since no other threads can concurrently perform loads or stores to the same address if at least one thread performs a write (by the DRF definition). Thus, thanks to the SC-for-DRF guarantees, both private and shared variables (i.e. local and global accesses) are treated equally, which greatly simplifies the static analysis and increases its accuracy.

B. *CELLO hardware overview*

Besides the compiler, CELLO also requires (small) changes within the processor core. Figure 5 shows an overview of the CELLO hardware on top of an SMT core, with the additional structures and flags marked in yellow. Each hardware thread has a *region flag* and a *num-sync* counter. The *region flag* is set by the dedicated `setDRF` instruction. Memory instructions read the flag to keep it in the *mode* (M) bit augmented to the SQ and LQ entries, and loads increase the *num-sync* counter of its thread if the mode is *sync*. When committing sync loads, the counter is decreased. Store writes, and cache invalidations and evictions check the *store-DRF* and *load-DRF* filters, respectively, and only search the LQ when it is essential to guarantee that no load→load ordering violation is exposed. Overall, in a 2-way SMT core with a 192-entry LQ and a 128-entry SQ, CELLO hardware overhead amounts to 40 bytes (320 mode bits and two 1-bit *region flags*), two 8-bit counters, and simple additional logic per core.

C. *The `setDRF` instruction implementation*

The processor requires minor modifications to support the `setDRF` instruction. When the `setDRF` instruction is allocated, a dedicated per-thread processor flag, called *region flag* (1 bit), is set or unset respectively, to mirror the value of the `setDRF` instruction's operand. An SMT core requires as many *region flags* as hardware threads it supports. The *region flag* is updated at allocation time, leveraging that instructions are allocated in order, and hence preserving the compile-time delineation of regions. If a thread's flag is 0, all the following memory instructions of that thread are considered to be part of a sync region. Similarly, if the flag is 1, the next memory instructions of the thread are DRF. The *region flag* is set by default to 0 (sync mode), which boils down to handling all memory operations conservatively, disabling our optimizations. Consequently, applications that have not been compiled with the CELLO compiler and do not include the regions delineations (e.g., legacy code) naturally preserve the TSO semantics and guarantees.

Stores and loads keep the region information individually in a *Mode* bit added to each entry of the SQ and LQ. The *Mode* bit is required because both DRF and sync memory operations can co-exist in the LQ and SQ. A series of instructions from a thread with the *Mode* bit set to 1 represents a DRF region, while the Mode bit set to 0 indicates a sync region. All the stores and loads entering the SQ and LQ, respectively, copy its thread *region flag* to their *Mode* bit. This step classifies the memory operation either as a DRF or a sync memory access.

Upon a misspeculation, a squash of the in-flight instructions is initiated. If the *region flag* has been modified by any squashed instruction, it needs to be restored. To enable fast recovery, we set the value of the *region flag* to the mode of the older instruction that is squashed, which restores the correct DRF/sync state.

D. *Efficient out-of-order execution of loads*

CELLO enables more efficient execution of DRF stores and loads by leveraging precise information from the compiler with two simple hardware optimizations.

1) Reducing LQ searches: Since DRF stores are guaranteed to be executed in the absence of concurrent loads to the same address in other threads, they cannot expose consistency violations (e.g., load→load). Therefore, it is not necessary

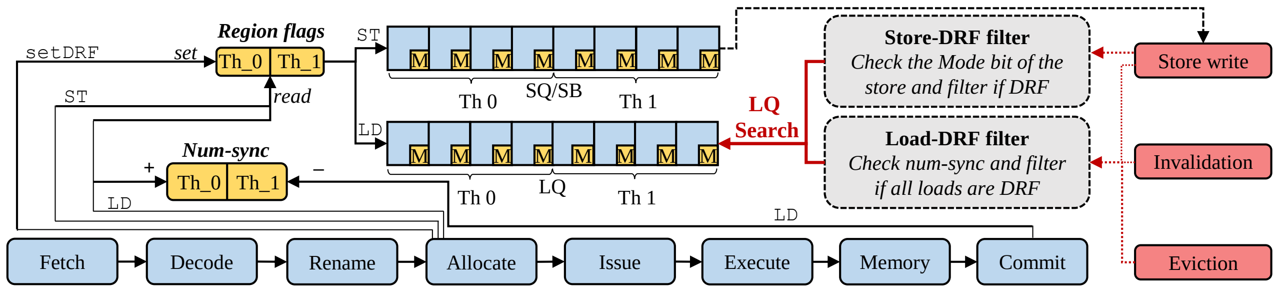


Fig. 5: Pipeline overview for CELLO on top of an SMT core. The structures / flags in yellow and the DRF-based filters are the additions required to support CELLO.

to perform any search in the same core LQ (to squash M-spec loads from other SMT threads) when a DRF store writes. Store writes can also cause invalidations that reach other cores private caches. An invalidation request caused by the write of a DRF store cannot expose a consistency violation either. Thus, at a first glance, we could skip its corresponding LQ search. However, after the cacheline is invalidated, the core would not detect forthcoming store writes to the same cacheline, potentially from non-DRF stores, which could expose a load→load reordering. Therefore, despite only sync stores can expose consistency violations in other threads, we can only skip safely the LQ searches associated to DRF store writes when they are performed by a thread running in the same SMT core.

Likewise, DRF loads are also guaranteed to be executed in the absence of concurrent stores to the same address in other threads and thus, they cannot expose consistency violations while being executed out of order. Therefore, no LQ search is required on store writes, and cache invalidations and evictions when the LQ contains only DRF loads, as they execute non-M-speculatively by definition. On the contrary, consistency violations are possible for racy loads, which may execute M-speculatively and, therefore, based solely on the DRF status of loads, LQ searches are required to prevent exposing any consistency violation.

Summing up, CELLO proposes to avoid the LQ search when i) it is caused by the write of a DRF store from a thread running in the same SMT core, or ii) there are no sync loads in the LQ. Since DRF memory operations are far more frequent than non-DRF ones, most LQ searches can be safely avoided, reducing energy consumption in the LQ and contention in its search ports. The implementation for reducing LQ searches is fairly simple. We implement two DRF-based filters: *Store-DRF filter* and *load-DRF filter*.

The *store-DRF filter* is based on the DRF status of stores. When the core performs a write, it checks the *Mode* bit on the corresponding SQ entry. A *Mode* bit set to 1 implies that the store belongs to a DRF region. Therefore, it cannot conflict with concurrent same-address loads in other threads and, consequently, it does not require any LQ search in the same core.

The *load-DRF filter* is based on the DRF status of the loads. We use a counter per SMT thread in a core, called *num-sync*, that holds the number of sync loads for a particular thread in the LQ. The counter is set to 0 by default indicating that there are no sync loads in the LQ, is incremented by 1 once a sync load from the thread enters the LQ, and is decremented by 1 when a sync load from the thread leaves the LQ. When performing a cache invalidation or eviction, the memory system first checks the *num-sync* counters of the core. If all the counters are 0, the LQ search does not take place, as the pipeline contains no M-speculative loads. When one of the counters is not 0, it indicates the presence of a sync load in the LQ, and thus, the memory system performs the LQ search. Similarly, when the core performs a write, the load-DRF filter checks the *num-sync* counters of the other threads co-running in the core. (It does not need to check the *num-sync* counter of the thread performing the write since this LQ search targets M-spec loads from the other threads running in the SMT core.) If all these counters are 0, the LQ search does not take place.

2) Early removal of loads from the LQ: Having an unresolved load at the head of the LQ is common, as long latency loads are the main culprits for processor stalls. While loads are not resolved, they are the source of M-speculation for subsequent loads [40]. In a standard TSO implementation, loads remain at the LQ head until they commit and are squashed if there is a match in an LQ search.

In CELLO, we aim to remove the loads from the LQ as soon as they do not need to be searched. Removing loads from the LQ helps reduce LQ occupancy, which either i) eliminates LQ-induced stalls and thus improves performance or ii) allows shrinking the LQ and thus reduces its energy consumption and area. Three conditions need to be satisfied in order to remove a load from the LQ:

1) *The load is at the head of the LQ.* Since the LQ is a circular buffer, occupancy reduction is only effective when removing the load at the head.
2) *The load is DRF.* The compiler guarantees that DRF loads do not conflict with stores from other threads and therefore they are non M-speculative by default.
3) *All stores older than the DRF load at the head have already resolved their address and searched the LQ.* The load becomes, then, non D-speculative.

The first two conditions are checked using the head pointer in the LQ and the *Mode* bit attached to each load in the

LQ. For the D-speculative condition, CELLO leverages a bit per entry in the SQ that indicates if the store has executed (commonly found in current implementations). This bit tracks resolved/unresolved store addresses. The default value is 0, indicating that the store address is still unresolved. When the store executes, the bit is set. The load simply performs a bitwise *OR* operation between the *Execute* bits and a bitmask that identifies the stores older than the load in the SQ with the corresponding bit set to 0. Such a bitmask can be generated with a range decoder, which sets to 0 and 1 the bit corresponding to each store depending on whether it is older or younger than the load, respectively. An *AND* operation is performed on all the resulting bits. If the result is 1, it means that no unresolved older store addresses exist, and the load can be considered non-D-speculative. Note that since this operation only acts on a single bit per entry in the SQ, it is simpler than an SQ search, which requires a priority decoder and full address comparison.

V. Discussion

A. Context switching

Context switches take place whenever one task (application, thread, etc.) waives its control of the processor to the next task in the system. Traditionally, when the OS dictates a context switch, the processor's registers and flags of the current process are saved before the OS kernel code is executed. With CELLO, on a context switch, the *region flag* is saved along with the other processor flags. Unless the OS code is annotated to mark DRF and sync regions, all instructions from the OS kernel are conservatively handled as sync instructions. Once the process resumes, its context is restored together with the previously saved *region flag*.

B. Non SC-for-DRF programs and debugging

In CELLO, legacy software executes all loads and stores in sync mode since it does not contain DRF annotations inserted by the compiler. Alternatively, tools such as data race detectors (e.g., ThreadSanitizer [49], Fast&Furious [42]) that operate either at the compiler level or directly on the binary could be used to analyze legacy code and annotate safe regions.

Still, although nowadays most programming languages require data race freedom, buggy code (which includes data races) may create issues both on CELLO and on conventional hardware. For debugging purposes, CELLO can simply turn off the DRF markings, thus executing the code as if it was running in a baseline TSO. This enables the programmer to debug the code with traditional methods. We argue that CELLO, in fact, can help programmers expose bugs, and then, track them. While it is the programmer's responsibility to track and fix data races, should the input code contain any such races, the "undefined" behavior can appear on CELLO the same as on platforms that provide TSO consistency.

VI. Simulation Environment

Our simulation infrastructure consists of a detailed in-house, out-of-order, simultaneous multithreading processor model. It employs Sniper [10] to feed the processor model with the instructions to execute, GEMS [35] to model the memory hierarchy and cache coherence, with a standard invalidation-based directory protocol, and GARNET [3] to model the interconnect. We model the LQ and SQ faithfully, including the searches for the speculative support for memory ordering. We simulate a multi-core processor, with eight 2-way SMT cores, providing a TSO consistency model. The processor parameters, shown in Table II, are chosen after nowadays processors and resemble the Intel Alder Lake micro-architecture [46]. Following Intel's SMT implementations [13], the ROB, LQ, and SQ are statically partitioned among the threads, assigning a fraction of the structures to each thread, while the execution units are dynamically shared. Since we observed that CELLO reduces the number of misspeculated loads we conservatively replicate the memory dependence predictor to avoid that thread interference in this resource skews our results. We use CACTI-P [31] to model the energy consumption of the LQ using a 22nm technology, the minimum technology node allowed by the latest CACTI version. The LQ is modeled as a CAM and uses the high-performance (hp) model. The energy consumption per search, write, and read is 0.0391nJ, 0.0079nJ, and 0.0017nJ, respectively.

TABLE II: System parameters

Processor	
Cores and threads	8-core 2-way SMT
Fetch Width	6 instructions
Issue/Commit Width	12 instructions
Reorder Buffer	512 entries
Load queue	192 entries, 3 write, 2 search ports
Store queue	128 entries, 2 write, 3 search ports
Branch predictor	TAGE-SC-L [52]
Mem. dep. predictor	StoreSet [12]
Memory	
Private L1I cache	32KB, 8 ways, 4 hit cycles, pipelined
Private L1D cache	48KB, 12 ways, 5 hit cycles, pipelined, IP-stride prefetcher
Private L2 cache	1MB, 8 ways, 12 hit cycles
Shared L3 cache	4MB per bank, 16 ways, 35 hit cycles
Directory	8 ways, 200% coverage of L2
Memory access time	160 cycles
Network Topology	2D Mesh

We analyze five different configurations: *Base*, *ST–DRF filtering*, *ST+LD–DRF filtering*, *CELLO*, and *SMT-directory*. *Base* is our baseline, a TSO-like SMT system that issues loads speculatively out-of-order, searching the core's LQ on each store write and on each invalidation and cache eviction. *ST–DRF filtering* and *ST+LD–DRF filtering* are baseline-like systems with the addition of the ST-DRF filter and both ST-DRF and LD-DRF filters, respectively. *CELLO* is our final proposal and combines the two DRF filters with the early removal of loads from the LQ. Finally, *SMT-directory* is the state-of-the-art mechanism to filter LQ searches in SMTs [22], discussed in Section II-C. It stores a bit-vector per cacheline to keep track of which SMT threads have read it and determine if an LQ search is needed. Our implementation of the SMT-directory stores the bitvectors only in the L1 cache to reduce

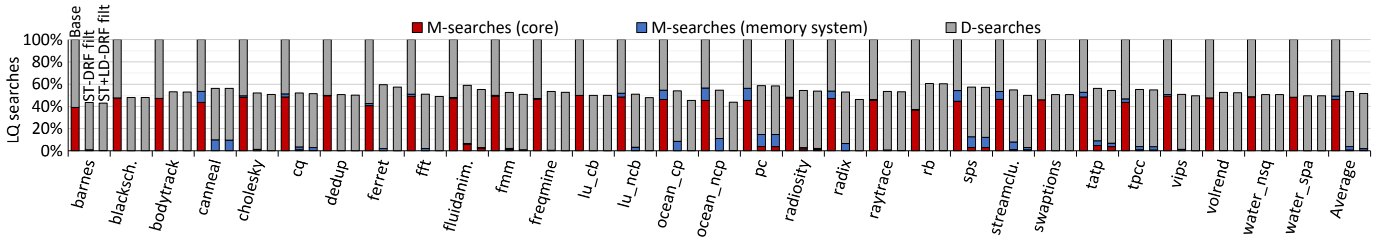


Fig. 6: Percentage of LQ searches performed with the baseline, ST–DRF filtering, and the ST+LD–DRF filtering configurations compared to the baseline. *ST+LD–DRF filtering avoids* 47% *of the LQ searches.*

the hardware overhead. In a 2-way SMT core with a 48-KB L1 data cache, its hardware overhead amounts to 1.5 KB per core. Adding bitvectors to the L2 cache increases the hardware overhead to 33.5 KB per core and provides minor performance benefits (the highest benefit is 0.7% in *cq* but the average across all workloads is only 0.1%).

We evaluate parallel workloads from the Splash-3 [48] and PARSEC 3.0 [7] benchmark suites, as well as, six fine-grain synchronization-intensive benchmarks [21], [28]. All the applications comply with the C/C++ standard, and thus, enforce SC-for-DRF. Results correspond to the parallel regions of the applications.

VII. Experimental Evaluation

The number of memory operations executed in *DRF regions* is dominating (Figure 2). In what follows, we analyze the benefits of leveraging this property in hardware when executing DRF software. First, we analyze how CELLO reduces the number of LQ searches and how that impacts energy expenditure and performance. After that, we present a sensitivity study on how CELLO allows reducing the LQ size without negatively impacting performance.

A. LQ search filtering

Figure 6 presents the LQ searches in the baseline 8-core 2-way SMT processor and in the same processor with only the ST-DRF filter and combining the ST-DRF and LD-DRF filters, normalized to the number of LQ searches in the baseline.[2] *M-searches (core)* and *M-Searches (memory system)* represent the LQ M-searches triggered when stores write and after invalidations and evictions, respectively, to prevent consistency violations. Our DRF-based filters aim at filtering, from these searches, all the ones where the DRF properties guarantee that are not required to prevent exposing a consistency violation. *D-Search* represents LQ searches triggered when stores compute their address to squash matching misspeculated loads. Our DRF-based filters do not target these LQ searches.

M-searches (Core) represent 46% of the LQ searches in the baseline system. However, since very few stores are non-DRF, most of them are not required to prevent consistency violations. Consequently, the ST-DRF filter eliminates virtually all these LQ M-searches, reducing them to only 0.8%. Adding

[2]Compared to the ST+LD–DRF filtering, CELLO adds the early removal of loads from the LQ. However, this feature does not affect the LQ search filtering. Thus, for the sake of clarity, we omit the CELLO setup in this figure.

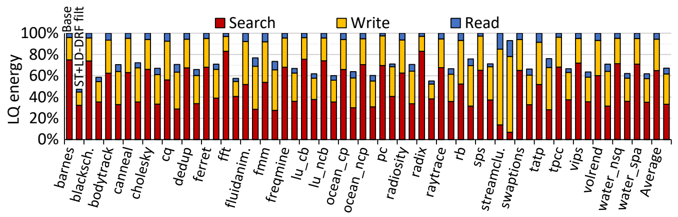


Fig. 7: LQ energy expenditure of the baseline and ST+LD–DRF filtering configurations normalized to the baseline.

the LD-DRF filter on top of the ST-DRF one cannot greatly improve the filtering since, basically, almost all searches are already filtered. Nevertheless, it has a slightly higher impact in a few benchmarks such as *fluidanimate* or *tatp*, where it is more frequent that non-DRF stores write while the co-running thread has only DRF loads in the LQ. In these two benchmarks, the LD-DRF filter helps skip, respectively, an additional 3.4% and 1.0% of LQ searches on top of the ST-DRF filter.

M-searches (memory system) represent a significantly lower fraction of the LQ searches (only 3% on average). This is not surprising as store writes are far more frequent than invalidations and evictions. Still, they exceed 10% of the LQ searches in benchmarks such as *canneal*, *ocean_ncp*, and *pc*. The LD-DRF filter avoids M-searches to the LQ caused by cache invalidations or evictions when the LQ only contains DRF loads, which reduces the LQ M-searches by 10.7% in *ocean_ncp*, 8.5% in *ocean_cp*, and 6.7% in *radix*. However, since i) evictions are not frequent (thanks to the large caches of modern systems), and ii) the LQ often contains non-DRF loads, adding the LD-DRF filter does not have a big impact on most workloads. On average, LQ M-searches are reduced from 3.0% to 1.5% when adding the LD-DRF filter.

Overall, the ST+LD–DRF filtering eliminates 47% of the LQ searches (virtually all M-searches) performed by the baseline system. Although the LD-DRF filter does not significantly reduce the LQ searches on average, we recommend enabling it as it provides non-negligible filtering in several workloads and the hardware overhead is minimal (only a *num-sync* counter per hardware thread).

B. Impact on energy

Filtering LQ searches directly turns into energy savings in the LQ. Figure 7 shows the energy consumed by the LQ for the baseline and ST+LD–DRF filtering setups normalized to the baseline processor.[3] We compute energy consumption as the sum of the energy spent on an LQ search, write, and read multiplied by the number of accesses of each type to the LQ. The energy expenditure of the LQ is clearly dominated by the LQ searches, which represent 65% of the energy consumed by the LQ. Even though LQ searches are less frequent than reads and writes, their high energy consumption outweighs their lower count.

The ST+LD–DRF filtering setup filters unnecessary LQ searches, reducing the LQ energy consumption, on average, by 33%. The energy savings are relatively similar across the workloads, ranging from 23% in *fluidanimate* to 45% in *radix*, with the only exceptions of *barnes* (53%) and *streamcluster* (7%). Nevertheless, as expected, workloads with a smaller ratio of stores per load search the LQ less frequently, with *streamcluster* as the most obvious example, which results in smaller energy savings.

C. Impact on performance

Filtering LQ searches does not only provide important energy savings in the LQ but also has a positive impact on performance. Figure 8 reports performance normalized to the baseline system for the SMT-directory, ST+LD–DRF filtering, and CELLO setups. By filtering almost all M-searches to the LQ, the ST+LD–DRF filtering setup improves the performance of the baseline by 2.8%, on average, with minimal hardware overhead. Furthermore, it is particularly high in workloads such as *barnes* (18.6%), *water_nsq* (11.9%), *blackscholes* (10.5%), and *lu_ncb* (7.2%). The CELLO setup adds the early removal of loads from the LQ to the two DRF-based filters. It only provides a small performance benefit in workloads such as *cq* or *streamcluster*, which clearly indicates that the LQ size is not a major performance bottleneck in this design. On the contrary, we observe that it degrades performance in *lu_ncb* and *volrend*. In both cases, the reason is a combination of two effects. First, a slightly higher memory contention, since the early removal of loads allows younger loads to enter the LQ and execute earlier. Second, barrier-based synchronization, which requires all threads to wait for the slowest thread to reach the barrier and hides the performance benefit that some of them achieve with CELLO.

The first reason why filtering LQ searches improves performance is because it allows stores to execute earlier, avoiding virtually all stalls suffered by them when they require searching the LQ but no LQ search port is available. Many of these stalls are on the critical path, particularly when they are related to synchronization operations. However, avoiding other stalls does not reduce execution time if, for example, it immediately exposes stalls in other resources or if, despite the LQ-search ports stalls, stores can still be executed under the shadow of the execution of older instructions. Furthermore, SMT can also effectively hide the stalls suffered by one thread

[3]Since the LQ energy consumption LQ is dominated by searches, the energy savings difference between CELLO and the ST+LD–DRF filtering setup is negligible. Thus, for the sake of clarity, we omit CELLO in this figure.

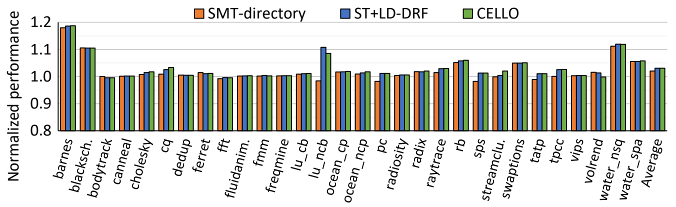


Fig. 8: Normalized performance for the SMT-directory, ST+LD–DRF filtering, and CELLO setups compared to the baseline system.

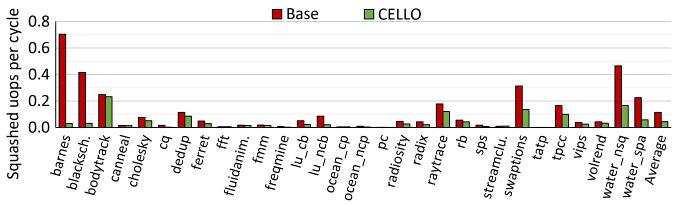


Fig. 9: Squashed uops per cycle due to load misspeculations in the baseline and CELLO setups.

executing instructions from the other threads running in the core. Consequently, it is hard to establish a direct connection between the avoided stalls and the performance benefit.

Instead, we observed that executing stores earlier allows for reducing load misspeculations significantly, particularly, in the workloads that achieve the highest performance benefit. When a load misspeculation is detected, the load and all younger instructions from the thread are squashed and re-executed. Such an event, therefore, causes a significant performance penalty. Figure 9 shows the number of uops squashed due to load misspeculations per cycle for the baseline and CELLO setups. On average, CELLO reduces the number of squashed uops per cycle from 0.11 in the baseline to 0.04. More importantly, the applications on which CELLO achieves the highest performance benefit (i.e., *barnes*, *blackscholes*, *lu_ncb*, *swaptions*, *water_nsq*, and *water_spa*) are the applications on which the number of squashed uops per cycle reduces the most. This observation demonstrates that load misspeculations have a direct impact on performance. CELLO filters most M-searches to the LQ, reducing LQ search port contention, and allowing stores to execute earlier. This way, CELLO reduces the processor dependence on the memory dependence predictor to avoid load misspeculations, resulting in fewer squashes and, consequently, superior performance.

Finally, Figure 8 also compares CELLO against SMT-directory [22]. The SMT-directory also filters virtually all M-searches to the LQ.[4] However, it suffers double the write latency when the LQ search is needed. Consequently, it performs worse than CELLO in synchronization-intensive workloads (e.g., *cq*, *pc*, *sps*, *tatp*, and *tpcc*), where sharing cachelines, particularly in synchronization operations, is more frequent. As synchronization is on the critical execution path of the applications (e.g., a thread cannot progress until it

[4]Due to space constraints, we omitted the SMT-directory setup in Figure 6, but on average, it filters 46% of the LQ searches.

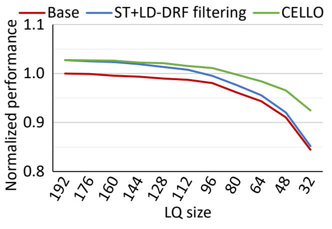
acquires a lock), the longer write latency easily turns into performance loss. In addition to these benchmarks, CELLO also widely outperforms the SMT-directory in *lu_ncb*. The reason that explains this performance difference is twofold. First, in *lu_ncb*, the DRF-based filters are effectively avoiding M-searches to the LQ caused by the memory system. However, the SMT-directory finds that there is at least a thread that read the cacheline. This makes the LQ search required, even if the load that read the cacheline already committed and left the LQ. Second, these LQ searches cause unnecessary load re-executions due to false sharing because invalidations and evictions search the LQ for the entire cacheline. Compared to the baseline, Sthe MT-directory improves performance on average by 2.1%, while CELLO achieves a 2.8% performance benefit with lower hardware overhead.

D. LQ size sensitivity analysis

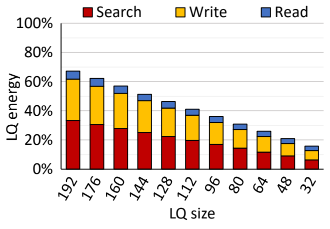
Larger LQs do not automatically guarantee performance improvements as searches become more costly. If the trends towards a higher processor clock, wider pipelines, and more execution units continue, the latency of searching in the LQ will become critical for performance. Along with increasing the search latency, a larger LQ also increases energy expenditure and area. In contrast, smaller LQ structures bring numerous benefits, such as lower energy expenditure and area, but they can affect the overall performance due to the LQ stalls. In what follows, we present a sensitivity study on the LQ sizes and their impact on performance, energy, and area.

Our analysis shows that CELLO can actually achieve low energy expenditure without degrading performance, when running DRF software, thanks to a better utilization of smaller LQs. Figure 10a shows the performance of the baseline, ST+LD–DRF filtering, and CELLO setups when reducing the LQ size from 192 to 32 entries compared to the baseline system with a 192-entry LQ. The performance of the baseline system starts to shrink faster from a 96-entry LQ (2% performance degradation) and grows up to 16% when the LQ size drops to 32 entries. The ST+LD–DRF filtering setup starts with a performance benefit of 2.8% with a 192-entry LQ compared to the baseline with the same LQ size, but its benefits start to diminish quickly from a 112-entry LQ, performing only marginally better than the baseline with LQs of 64 and 32 entries. On the contrary, the early removal of loads from the LQ allows CELLO to make more efficient use of the LQ entries and clearly outperforms the previous two setups. With LQs of 64 and 32 entries, CELLO performs 4.3% and 9.5% better than the baseline with the same LQ size. More importantly, with an LQ of only 80 entries, CELLO performs on par with the baseline system with a 192-entry LQ, providing huge energy and area savings in the LQ, as we discuss next.

Reducing the size of the LQ greatly reduces its energy expenditure. Figure 10b shows the average energy consumption of the LQ with CELLO when reducing the LQ size normalized to the energy consumption of the baseline system with a 192-entry LQ. With the same LQ size, thanks to filtering almost all M-searches, CELLO reduces the LQ energy consumption



(a) Performance



(b) Energy

Fig. 10: Sensitivity study when reducing the LQ size. (a) Performance of the baseline, ST+LD–DRF filtering, and CELLO setups. (b) Energy consumption of CELLO. All results are normalized with respect to the baseline with a 192-entry LQ.

by 33%. When reducing the number of entries in the LQ, searches, writes, and to a lesser extent reads, require less energy to be performed. Therefore, the energy expenditure of the LQ reduces. Focusing on an 80-entry LQ, where CELLO performs on par with the baseline system, the LQ energy consumption with CELLO reduces by 69% compared to the baseline. Finally, shrinking the number of entries of the LQ from 192 to 80 also reduces its area by 56%.

In the context of a growing demand for low power and high performance, CELLO becomes particularly attractive for systems dedicated to run mostly modern DRF software, as it allows to significantly reduce the LQ energy and area expenditures, with minimal modifications to the baseline system and without negatively impacting the performance nor the memory consistency guarantees.

VIII. Related Work

Relaxed consistency models [2], [5], [34] or speculative support to strong models [15], [20], [44], [57] are widely adopted to reach high performance. The LQ is a key structure to enforce not only the load→load order required by strong consistency implementations [20] but also sequential semantics enforced by weak consistency models [5], [36]. This section discusses first hardware-only optimizations for the LQ, and then, software-hardware co-designs.

Hardware-only solutions. Previous work has proposed value-based speculation to eliminate LQ searches by delaying the detection of conflicts between a store and a load until the load commits. At commit, the load is replayed and the new loaded value is rechecked. This idea was first proposed by Gharachorloo *et al.* [20] but was put aside as the replays added tremendous pressure on L1. Cain and Lipasti [9] addressed this issue by filtering the loads that do not need to be replayed, if there are no unresolved stores or any invalidation between their issue and commit time. Roth *et al.* [47] introduced a Store Vulnerability Window (SVW) for more accurately filtering out the loads. Despite these improvements, there remains a sizeable fraction of loads that need to be re-executed at commit. Ros and Kaxiras [43] recently proposed to eliminate the LQ and the replays accessing the coherent memory system (e.g., the L1 cache) completely. Loads instead replay on the SQ, where stores are delayed until the loads that bypassed them replay and commit. Their solution requires however non-obvious modifications to the cache coherence protocol and adds pressure to the SQ ports.

Leaving aside value-based speculation, Sethumadhavan *et al.* [50] propose to use a Bloom filter in order to reduce the frequency and number of LQ searches of local stores. Bloom filters, however, are not precise and add extra latency to the LQ search. Ros *et al.* [40] allow the loads to execute speculatively out-of-order and hide the load-load reordering by the coherence protocol. This allows them to commit the reordered loads out-of-order without waiting for the loads to become non-speculative. However, the proposed design requires important changes in the coherence protocol. Finally, Garg *et al.* [18] target the store$\rightarrow$load order and propose replacing age-ordered LQs with address-indexed hash tables, allowing late allocation into the LQ. However, this mechanism adds an extra number of replays mostly due to different data access widths and table pollution. In contrast, CELLO targets a more efficient load$\rightarrow$load order and does not change the behavior of out-of-order loads nor the coherence protocol.

Related work focusing on SMT processors is scarce. Only, Hilton and Roth [22], discussed above, proposed a mechanism seeking to filter the LQ searches using a bit-vector to determine which threads have read each cacheline. Recently, Feliu et al. [16] proposed inter-thread store-to-load forwarding in SMTs and combine the LQ searches when stores execute and write into a single one, triggered when the store becomes non-speculative. The proposal, however, does not filter LQ searches after cache invalidations and evictions, and it is significantly more complex than CELLO, since it requires tracking the speculative state of stores.

Software-hardware co-designs. Huang et al. [24] propose a software-hardware co-design such that selected memory operations can bypass the hardware memory disambiguation. The selection is based on a binary analysis that identifies loads guaranteed not to overlap with older in-flight stores. The static analysis with binary parsing is limited to identifying read-only loads and the relation between loads and stores within loops with regular array-based accesses. Thus, it is complemented by a dynamic approach to identify other safe loads with hardware support. In contrast, using a region-based classification, our compiler is more accurate and does not require additional hardware support for regions disambiguation.

Focusing on stores, Singh *et al.* [54] propose a design in which the compiler marks the stores that can be reordered. Stores to the same memory address are forced to be placed in a so-called *unsafe* SQ in order, while *safe* stores are placed in a different unordered SQ. The solution adds extra hardware overhead and can lead to under-utilization and stalls in the SQs. In addition, it does not exploit the DRF semantics and the data-based classification limits the number of stores that can be reordered. Addressing both issues, ROOW [55] proposes a compiler that identifies regions within which stores can be shuffled while still delivering the same observable behavior as if they performed in program order and a store buffer design that can switch between out-of-order and in-order execution modes within the safe and unsafe regions, respectively. Unlike these proposals, CELLO focuses on efficiently maintaining the load-load ordering by filtering unnecessary LQ searches and is the only proposal that leverages the DRF guarantees to filter the LQ searches triggered in SMT processors when stores write. Furthermore, its simpler design and the minimum hardware extensions relative to mainstream implementations make CELLO a more appealing solution to enable efficient load reordering.

DRF interface. Conflict Exceptions (CE) [33] proposes to treat data races as precise hardware exceptions, thus simplifying debugging parallel programs. Although the employed software-hardware interface is similar to CELLO, they leverage that interface for optimizing the cache coherence protocol. Biswas et al. [8] propose a practical implementation of CE, called CE+, that reduces the number of memory accesses by means of an on-chip metadata cache. Finally, Ros et al. [45] introduce forward self-invalidation (FSI), a technique to improve the performance of cache coherence protocols based on self-invalidation [11], [41] at the end of DRF regions. All these proposals target cache coherence protocols.

IX. Conclusion

In this paper we introduce CELLO, a software-hardware co-designed approach that uses compiler support to reduce the LQ pressure with minor hardware modifications. CELLO filters LQ searches, triggered to detect potential consistency violations, for regions that are indicated as safe by the compiler. Since these regions are predominant, CELLO skips 47% of the LQ searches, reducing the total LQ energy expenditure by 33% on average (up to 53%) compared to a baseline SMT system. At the same time, CELLO improves the baseline performance by 2.8% on average (up to 18.6%), a noticeable benefit given CELLO's minimal hardware overhead: 40 bytes (a 1-bit flag in the LQ and SQ entries), two 8-bit counters, and simple logic to check the flags and counters. Furthermore, by allowing the non-speculative loads to exit the LQ early, CELLO enables shrinking the LQ size from 192 to 80, achieving energy and area savings of 69% and 56%, respectively, without performance penalties compared to the 192-entry LQ baseline system.

Acknowledgment

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 819134), from the MCIN/AEI/10.13039/501100011033/ and the "ERDF A way of making Europe", EU (grant PID2022-136315OB-I00), from the European Union's Horizon 2021 research and innovation program (grant agreement No 101070374 under HORIZON-CL4-2021-DIGITAL-EMERGING-01) and RYC2018-025200-I, and from the MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/PRTR (grants TED2021-130233B-C33/C32 and RYC2021-030862-I).

References

[1] S. V. Adve and H.-J. Boehm, "Memory models: A case for rethinking parallel languages and hardware," *Communications of the ACM*, vol. 53, no. 8, pp. 90–101, Aug. 2010.

[2] S. V. Adve and M. D. Hill, "Weak ordering – a new definition," in *17th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1990, pp. 2–14.

[3] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 33–42.

[4] J. Alsop, M. D. Sinclair, and S. V. Adve, "Spandex: A flexible interface for efficient heterogeneous coherence," in *45th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2018, pp. 261–274.

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

[6] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language*. Addison Wesley Professional, 2005.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, pp. 72–81.

[8] S. Biswas, R. Zhang, M. D. Bond, and B. Lucia, "Rethinking support for region conflict exceptions," in *33rd Int'l Parallel and Distributed Processing Symp. (IPDPS)*, May 2019, pp. 1095–1106.

[9] H. W. Cain and M. H. Lipasti, "Memory ordering: A value-based approach," in *31st Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2004, pp. 90–101.

[10] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *Conf. on Supercomputing (SC)*, Nov. 2011, pp. 52:1–52:12.

[11] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the memory hierarchy for disciplined parallelism," in *20th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2011, pp. 155–166.

[12] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *25th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1998, pp. 142–153.

[13] M. Dixon, P. Hammarlund, S. Jourdan, and R. Singhal, "The next-generation Intel core microarchitecture," *Intel Technology Journal*, vol. 14, no. 3, pp. 8–28, Mar. 2010.

[14] Y. Duan, D. Koufaty, and J. Torrellas, "Scsafe: Logging sequential consistency violations continuously and precisely," in *22nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Mar. 2016, pp. 249–260.

[15] Y. Duan, A. Muzahid, and J. Torrellas, "Weefence: Toward making fences free in tso," in *41st ACM SIGARCH Computer Architecture News*, Jun. 2013, p. 213–224.

[16] J. Feliu, A. Ros, M. E. Acacio, and S. Kaxiras, "ITSLF: Inter-Thread Store-to-Load Forwarding in Simultaneous Multithreading," in *54th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2021, pp. 1296–1308.

[17] C. Flanagan and S. N. Freund, "Fasttrack: Efficient and precise dynamic race detection," in *2009 Conf. on Programming Language Design and Implementation (PLDI)*, Jun. 2009, pp. 121–133.

[18] A. Garg, F. Castro, M. Huang, D. Chaver, L. Piñuel, and M. Prieto, "Substituting associative load queue with simple hash tables in out-of-order microprocessors," in *Proceedings of the 2006 International Symposium on Low Power Electronics and Design*, ser. ISLPED '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 268–273.

[19] A. Garg, C. Fernando, H. Michael, D. Chaver, L. Pinuel, and M. Prieto, "Substituting associative load queue with simple hash tables in out-of-order microprocessors," in *Proceedings of the 2006 international symposium on Low power electronics and design*, Oct. 2006, pp. 268–273.

[20] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *20th Int'l Conf. on Parallel Processing (ICPP)*, Aug. 1991, pp. 355–364.

[21] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," in *45th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2018, pp. 46–61.

[22] A. D. Hilton and A. Roth, "SMT-directory: Efficient load-load ordering for SMT," *IEEE Computer Architecture Letters*, vol. 9, no. 1, pp. 25–28, Jan. 2010.

[23] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-race-free memory models," in *14th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2014, pp. 427–440.

[24] R. Huang, A. Garg, and M. C. Huang, "Software-hardware cooperative memory disambiguation," in *12th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2006, pp. 244–253.

[25] *First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem)*, Intel Corporation, White paper, Apr. 2009.

[26] ISO, *ISO/IEC 14882:2015 Information technology — Programming languages — C++*. International Organization for Standardization, 2015.

[27] A. Jimborean, J. Waern, P. Ekemark, S. Kaxiras, and A. Ros, "Automatic detection of extended data-race-free regions," in *15th Int'l Symp. on Code Generation and Optimization (CGO)*, Feb. 2017, pp. 14–26.

[28] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *44th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2017, pp. 481–493.

[29] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers (TC)*, vol. 28, no. 9, pp. 690–691, Sep. 1979.

[30] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd Int'l Symp. on Code Generation and Optimization (CGO)*, Mar. 2004, pp. 75–88.

[31] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *2011 Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov. 2011, pp. 694–701.

[32] Y. Li, R. G. Melhem, and A. K. Jones, "Practically private: Enabling high performance cmps through compiler-assisted data classification," in *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 231–240.

[33] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm, "Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races," in *37th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2010, pp. 210–221.

[34] L. Maranget, S. Sarkar, and P. Sewell, "A tutorial introduction to the arm and power relaxed memory models," INRIA and University of Cambridge, Tech. Rep., Oct. 2012.

[35] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[36] A. Moshovos and G. S. Sohi, "Streamlining inter-operation memory communication via data dependence prediction," in *30th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 1997, pp. 235–245.

[37] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[38] OpenMP Architecture Review Board, "OpenMP application program interface version 3.0," May 2008. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[39] I. E. Papazian, "New 3rd gen Intel Xeon Scalable processor (codename: Ice Lake-SP)," in *32nd HotChips Symp.*, Aug. 2020, pp. 1–22.

[40] A. Ros, T. E. Carlson, M. Alipour, and S. Kaxiras, "Non-speculative load-load reordering in tso," in *44th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2017, pp. 187–200.

[41] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 241–252.

[42] ——, "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)*, Jan. 2015, pp. 1–6.

[43] ——, "The superfluous load queue," in *51st IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2018, pp. 95–107.

[44] ——, "Speculative enforcement of store atomicity," in *53rd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2020, pp. 555–567.

[45] A. Ros, C. Leonardsson, C. Sakalis, and S. Kaxiras, “Efficient self-invalidation/self-downgrade for critical sections with relaxed semantics,” *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 12, pp. 3413–3425, Dec. 2017.

[46] E. Rotem, A. Yoaz, L. Rappoport, S. J. Robinson, J. Y. Mandelblat, A. Gihon, E. Weissmann, R. Chabukswar, V. Basin, R. Fenger, M. Gupta, and A. Yasin, “Intel Alder Lake CPU architectures,” *IEEE Micro*, vol. 42, no. 3, pp. 13–19, Mar. 2022.

[47] A. Roth, “Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization,” in *32nd Int’l Symp. on Computer Architecture (ISCA)*, Jun. 2005, pp. 458–468.

[48] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, “Splash-3: A properly synchronized benchmark suite for contemporary research,” in *Int’l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2016, pp. 101–111.

[49] K. Serebryany and T. Iskhodzhanov, “Threadsanitizer: Data race detection in practice,” in *Workshop on Binary Instrumentation and Applications (WBIA)*, Dec. 2009, pp. 62–71.

[50] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler, “Scalable hardware memory disambiguation for high ilp processors,” in *36thIEEE/ACM Int’l Symp. on Microarchitecture (MICRO)*, ser. MICRO 36. USA: IEEE Computer Society, 2003, p. 399.

[51] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, “x86-TSO: A rigorous and usable programmer’s model for x86 multiprocessors,” *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010.

[52] A. Seznec, “TAGE-SC-L branch predictors,” in *JILP - Championship Branch Prediction*, Jun. 2014, pp. 1–8.

[53] M. D. Sinclair, J. Alsop, and S. V. Adve, “Chasing away rats: Semantics and evaluation for relaxed atomics on heterogeneous systems,” in *44th Int’l Symp. on Computer Architecture (ISCA)*, Jun. 2017, pp. 161–174.

[54] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi, “End-to-end sequential consistency,” in *39th Int’l Symp. on Computer Architecture (ISCA)*, Jun. 2012, pp. 524–535.

[55] S. Singh, A. Jimborean, and A. Ros, “Regional out-of-order writes in total store order,” in *29th Int’l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2020, pp. 205–216.

[56] D. M. Tullsen, S. J. Eggers, and H. M. Levy, “Simultaneous multithreading: Maximizing on-chip parallelism,” in *22nd Int’l Symp. on Computer Architecture (ISCA)*, Jun. 1995, pp. 392–403.

[57] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, “Mechanisms for store-wait-free multiprocessors,” in *34th Int’l Symp. on Computer Architecture (ISCA)*, Jun. 2007, pp. 266–277.