Received: 30 January 2021 | Revised: 23 May 2021 | Accepted: 21 June 2021
DOI: 10.1002/cpe.6483

WILEY

SPECIAL ISSUE PAPER

sRSP: An efficient and scalable implementation of remote scope promotion

Ayse Yilmazer-Metin

Computer Engineering Department, Istanbul Technical University, Istanbul, Turkey

Correspondence
Ayse Yilmazer-Metin, Computer Engineering Department, Istanbul Technical University, Istanbul, Turkey.
Email: yilmazerayse@itu.edu.tr

Abstract
GPUs only support simple coherence operations. For data integrity, heavyweight synchronization operations must be explicitly used. Scoped synchronization Hower et al. (2014) and Gaster et al. (2015) has been introduced to facilitate low overhead communication, when the communication is local across a subset of threads. Scoped synchronization can be used if the sharing is static. However, it is not possible to avoid using heavyweight global-scoped synchronization when the sharing is dynamic. Asymmetric sharing is one of the dynamic sharing patterns where a shared data is frequently accessed by a single agent while being rarely accessed by remote agents. It requires using global-scoped synchronization to encompass all possible synchronization agents on a GPU device without any special support. Remote scope promotion (RSP) Orr et al. (2015) allows use of lightweight local-scoped synchronization for frequent accesses and defers the most synchronization overhead to the rare remote accesses. We propose sRSP which is an efficient and scalable implementation of RSP semantics. sRSP tracks local-scoped synchronizations on local caches. When performing remote-scoped synchronization, it applies the costly cache operations *selectively*. We thoroughly evaluate our work and show that it reduces remote synchronization overhead significantly and scales better. On the average, it improves the performance around 25% for a GPU device with 64 compute units.

KEYWORDS
asymmetric synchronization, graphic processing units, remote scope promotion, work stealing

1 | INTRODUCTION

GPUs have found extensive use in accelerating many general-purpose data-parallel applications. Most of these applications are characterized with *regular* control flow and memory access patterns. Now, they are also finding wide-spread use in accelerating data-parallel *irregular* applications. Nevertheless, some of these applications demand efficient synchronization support for good performance improvements and energy-efficiency.

Memory systems on GPUs are designed to support thousands of threads running concurrently. Higher bandwidth is preferred over latency hiding techniques. When performing regular load and store operations, sophisticated coherence operations are avoided. Instead, they use simple write-through or write-combining mechanisms. The operations to maintain the data consistency are deferred to the synchronization points at which heavyweight cache operations (e.g., cache-flush and cache-invalidation operations combined with atomic operations) must be used. If the global communication is rare and the synchronizations are not frequent, this solution works fine. However, many widely used irregular applications use global communication and fine-grained synchronization and their performance heavily relies on synchronization support. These applications suffer from significant performance overhead due to lack of support for efficient synchronization operations.

Recently, *scoped synchronization*[1,2] has been introduced to limit the scope of synchronization across a subset of threads (e.g., work-group), and therefore the synchronization overhead. For example, when the communication is across a subset of the threads and the data is shared on the same local cache, it is possible to use *local-scoped* synchronization and avoid heavyweight cache operations (i.e., cache flush and/or cache-invalidation operations). This model works well when the applications possess well structured, regular sharing patterns. However, there are many general purpose applications that contain irregular and dynamic sharing patterns and cannot benefit from scoped synchronization.

An important class of dynamic sharing pattern is known as *lob-sided* or *asymmetric* sharing in which a shared data is regularly accessed by an agent and rarely accessed by other agents. In this article, we refer to the frequently accessing agent in asymmetric sharing as *local-modifier* and we refer to the other rarely accessing agents as *remote-modifiers*. While the shared data is mostly accessed by the same agent (i.e., the local-modifier), the existence of rare accesses from other agents (i.e., remote-modifiers) prevents from using lightweight local-scoped synchronization operations. A good example for asymmetric sharing exists in work-stealing[3] and some networking applications.[4] In implementation of work-stealing on GPUs, each work-group owns a statically assigned work-queue. When the threads in a workgroup run out of work by emptying their own work-queue, they try to steal work from other groups' work-queues. Due to these possible accesses from nonowners, both owner's and nonowners' accesses to a work-queue must be guarded with heavyweight global-scoped synchronization operations.[*]

Remote scope promotion (RSP) has been proposed[5] to support asymmetric sharing on GPUs. RSP allows using local-scoped synchronization for frequent accesses by the local-modifier, while global-scoped remote synchronization must be used for rare accesses by remote-modifiers. Despite the existence of dynamic sharing, consistency of shared data is preserved by promoting the local-scoped synchronization to global-scope (or to a larger target scope) when performing remote synchronization. The first implementation of RSP[5] works with *broadcast* cache-flush and cache-invalidation operations that are applied to *all* local caches when performing remote-scoped synchronization. Unfortunately, the heavy use of cache-flush and cache-invalidation operations disrupts the effectiveness of local caches and this severely limits the scalability and therefore the applicability of RSP. A more efficient and scalable implementation of RSP is needed.

In this work, we propose a new mechanism to implement RSP semantics that avoids performing heavyweight cache-flush and cache-invalidation operations on *all* local caches. Our mechanism which is called *sRSP*, tracks all local-scoped synchronization operations on local caches. It applies the costly cache operations *selectively* when performing remote-scoped synchronization. We thoroughly evaluate our work and show that, with an efficient implementation, RSP can be employed on GPU devices with large number of compute units (CUs) and can be used with applications utilizing fine-grained synchronization.

2 | BRIEF OVERVIEW OF GPU EXECUTION MODEL AND MEMORY SYSTEM

In this section, we provide an overview of GPU execution model in the context of general purpose GPU computing (GPGPU).[†]

A GPGPU application consists of one or more CPU threads (i.e., host threads) and thousands of GPU *workitems* to run in parallel. When a kernel is launched for execution, a hierarchy of GPU workitems are created. At the top of this hierarchy, all host threads and GPU workitems form a *system* group. Workitems executing on the same device (e.g., a GPU device) form a *component* group. Workitems are divided into *workgroups* to have some degree of sharing and cooperation within the group.[‡] The hierarchical organization of threads facilitates a more efficient allocation of resources on GPU CUs and efficient management of large number of threads. In Figure 1, we show this abstraction that allows programmers to hierarchically organize GPU threads (i.e., work-items) in GPGPU computing.

A GPU device is composed of a set of CUs and each CU is composed of multiple functional units (i.e., single instruction multiple data (SIMD) units) that execute in lockstep. Workitems in a workgroup are scheduled onto the same CU. At run-time, workitems in a workgroup are further divided into SIMD groups known as *wavefronts*. Wavefronts execute on SIMD units. On a GPU device, each CU includes a private L1 data cache and all CUs on a GPU device share a common L2 cache. [§]Typically an L1 data cache facilitates for local communication within a workgroup and the L2 cache facilitates for global communication across all workgroups.

2.1 | GPU memory consistency and scoped synchronization

GPUs favor weak memory models. Recently, a weak memory model that is based on Acquire and Release synchronization semantics[6] is adopted by OpenCL. OpenCL extends acquire and release semantics with scopes while adopting it. In this section, we first explain how basic acquire and release synchronization semantics work, and then, we describe how OpenCL extends it with scopes.[1,2]

[*] In scoped synchronization, the synchronization scope must be chosen to include all possible agents.
[†] In our work, we use OpenCL programming model and therefore, we make use of OpenCL terminology throughout this article.
[‡] The number of workitems in a workgroup and the number of workgroups are statically determined at the kernel launch time.
[§] L1 data cache will be referred as *local* cache throughout the article.

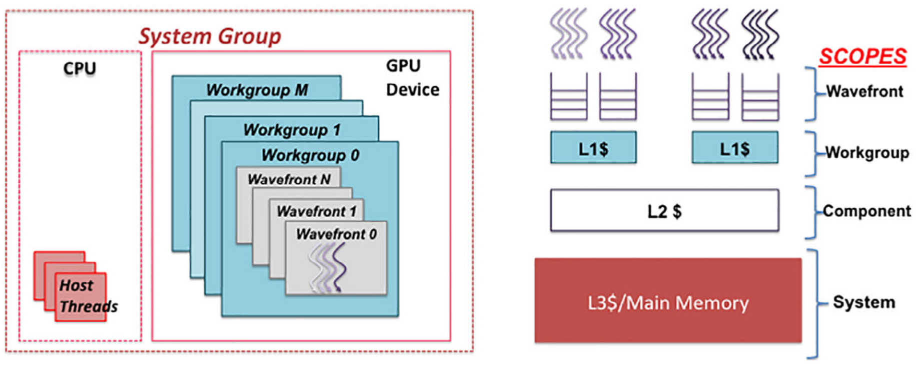

FIGURE 1 Hierarchy of work-items on a GPU device and corresponding synchronization scopes for each level of this hierarchy. All workgroups (i.e., the *component* group) can synchronize at L2 cache (the synchronization point for global-scope) and the workitems of a workgroup can synchronize at their local L1 data cache (the synchronization point for local- scope)

In acquire-release semantics, threads must synchronize with acquire and release synchronization operations to maintain data consistency. Typically, a synchronization operation is paired with an atomic memory operation—usually an acquire with an atomic load operation and a release with an atomic store memory operation. Acquire and release operations act as memory fences at the synchronization points for a robust communication.

Hardware and system software (i.e., compiler) must provide at least two things to support acquire and release semantics: ① The ordering of memory operations around the synchronization operations (i.e., acquire and release) must be preserved. Any memory operations after an acquire in the program order must be prevented from moving upward (i.e., the memory operations after an acquire must not execute before the acquire). Likewise, any memory operations before a release in program order must be prevented from moving downward (i.e., memory operations before a release must not execute after the release). ② Hardware must ensure that any updates made prior to a release operation must be returned for any load operation performed after the acquire operation. [¶]Considering hardware caching mechanism and many levels of cache hierarchy in a memory system, with a release, any updates prior to the release must be pushed to a common synchronization point (e.g., main memory) and this up-to-date data must be pulled from that common synchronization point with an acquire.

Using the acquire and release semantics, the overhead of coherence actions to provide memory consistency is deferred to the synchronization points. However, synchronization overhead could become very significant when large number of threads are involved in the synchronization. To limit the penalties of synchronizations for a subset of threads, scoped acquire and release operations have been introduced.[1,2] OpenCL[7] specifies following five synchronization scopes: *workitem (wi)*, *wavefront (wv)*, *workgroup (wg)*, *component (cmp)*, and *system (sys)* scopes. For example, a *wg*-scoped synchronization operation limits the synchronization scope to the workitems that exist in the same workgroup. Each of these five synchronization scopes resembles to a level in hierarchical GPU execution model that is shown in Figure 1. Since this study focuses on memory system, hereon, we will be only discussing *wg*-scoped, *cmp*-scoped, and *sys*-scoped synchronizations. [#]

Scoped-synchronization also enables localized synchronization in a hierarchical memory design. For example, wg-scoped acquire and release operations can be served at the L1 data cache level. Since all workitems of a workgroup are scheduled on the same CU, the L1 data cache serves as a common communication and synchronization point for them. In Figure 1, we also indicate the synchronization points for each scope on a GPU memory system.

2.2 | Typical implementation of scoped acquire and release synchronization operations in GPU memory system

The fundamental task of a cache protocol is to maintain the coherence and consistency of shared data.[8] GPUs use simple coherence actions with regular memory operations to scale for thousands of threads running concurrently. [‖]When serving regular read and write memory operations, only simple caching mechanisms are used with write-through or write-combining caches. Write-combining allows combining multiple writes for a cache block until it gets evicted from the cache. To implement write-combining, dirty bytes are marked with a mask for each cache block. When a dirty cache line is flushed or evicted from the cache, only the dirty bytes are written back to the lower levels of memory hierarchy.

While GPUs use simple caching mechanisms, data consistency is managed with explicit use of acquire and release synchronization operations paired with an atomic operation and a scope identifier. The synchronization point for each scope varies. While L1 data cache serves as a

[¶]The acquire operation that is synchronized with the release operation.
[#]In this article, *wg*-scoped synchronization and *local-scoped* synchronization are interchangeably used. Similarly, *cmp*-scoped synchronization and *global-scoped* synchronization are interchangeably used.
[‖]Even a simple coherence protocol with a few main states is quite complicated in its real hardware implementation due to unavoidable many transient states.[8]

synchronization point for a local-scoped synchronization operation, typically the L2 cache serves as the synchronization point for a global-scoped synchronization operation.

When performing a scoped release operation, all updates must be pushed to the specified scope's synchronization point and the paired atomic operation must be performed at the specified scope. GPU caches are equipped with cache-flush operations and they can be used for pushing updates on shared data (i.e., dirty lines). When all dirty data is reached to the specified scope's synchronization point then the cache-flush operation completes. For example, when performing a local-scoped release operation, there is no need to perform any cache-flush operation, *only* the paired atomic operation must be performed at L1 data cache. Since all workitems of a workgroup execute on the same CU and share the same L1 data cache, any updates performed by a workitem are already visible within the workgroup. On the other hand, when performing a global-scoped release operation, all dirty data must be written back to the L2 cache and the paired atomic operation must be performed at the L2 cache. In the same way, when performing a system-level (sys-scoped) release operation, first the L1 data cache, and next the L2 cache must be flushed and the atomic operation must be performed at the L3 cache or at the system memory.

Cache-flush operations can be implemented efficiently using an sFIFO structure.[9] sFIFO is a FIFO structure that tracks all dirty cache blocks on a GPU cache. Whenever a cache block is modified locally, its address is inserted into sFIFO. When the sFIFO is full, first address at the front of sFIFO is removed and the related cache line is written back to the next level of memory hierarchy (if the block is still dirty). When performing a release operation, all addresses in sFIFO must be removed in the FIFO order and each related cache line must be written back to the next level in the memory system. A cache-flush operation completes once all writes complete with acknowledgments. In this work, in our baseline GPU architecture, L1 data cache and L2 cache include sFIFO structures and make use of sFIFO based flushing.

Scoped acquire operations are performed to ensure that all shared data must be pulled from the specified scope. To make sure up-to-date data is pulled down from a synchronization point, any locally stored, possibly stale data must be invalidated. A cache-invalidation operation requires invalidating all cached data. During invalidation, if a cache block is dirty, it must be first flushed, then it can be marked as invalid. When performing a local-scoped acquire operation, *only* the paired atomic operation must be performed at L1data cache and there is no need to invalidate any caches. While performing a global-scoped acquire operation, L1 data cache must be invalidated and the associated atomic operation must be performed at the L2 cache. Furthermore, for a system level acquire, both L1 and L2 caches must be invalidated and atomic operations must be performed at L3 cache or system memory level. When performing a scoped-acquire operation, first, all dirty cache lines can be flushed using sFIFO and following the flush, a single-cycle flash-invalidate can be performed on the cache.

We explain how a global-scoped synchronization with acquire-release semantics works with an example shown in Figure 2. In this example, a communication between two threads—*t0* which is running on CU0 and *t1* which is running on CU1, is depicted. The communication happens through shared data `Y`. We see that the shared data is protected within a critical section that is guarded with a lock. Accesses to the lock variable are performed with atomic operations paired with acquire and/or release synchronization operations to provide consistency of data. We can see how the global-scoped acquire and release synchronization operations are realized by the hardware to provide integrity of the shared data. First, at step ①, a store operation happens when *t0* updates the value of `Y` to `3`. To finish the critical section, *t0* performs a global-scoped release synchronization operation while making the lock available to others. To make the new value of `Y` globally visible, the release operation causes a flush operation on L1 data cache on CU0 (step ②). Once the flush completes then the atomic store operation happens at L2 cache (at step ③), which updates the value of lock variable `L` to `0` on L2 cache. Next, *t1* executes a `atomic_cas` paired with acquire synchronization operation to enter the critical section and updates the value of shared variable `Y`. While acquiring the lock, first, the `atomic_cas` operation is performed at L2 cache, changing the value of `L` to `1` (step ④). Following the `atomic_cas`, a cache-invalidate operation is performed on CU1's local L1 data cache (step ⑤). Once the acquire completes and the thread *t1* enters to the critical section, the up-to-date value of `Y` is loaded from L2 cache and updated on CU1's L1 data cache (step ⑥).

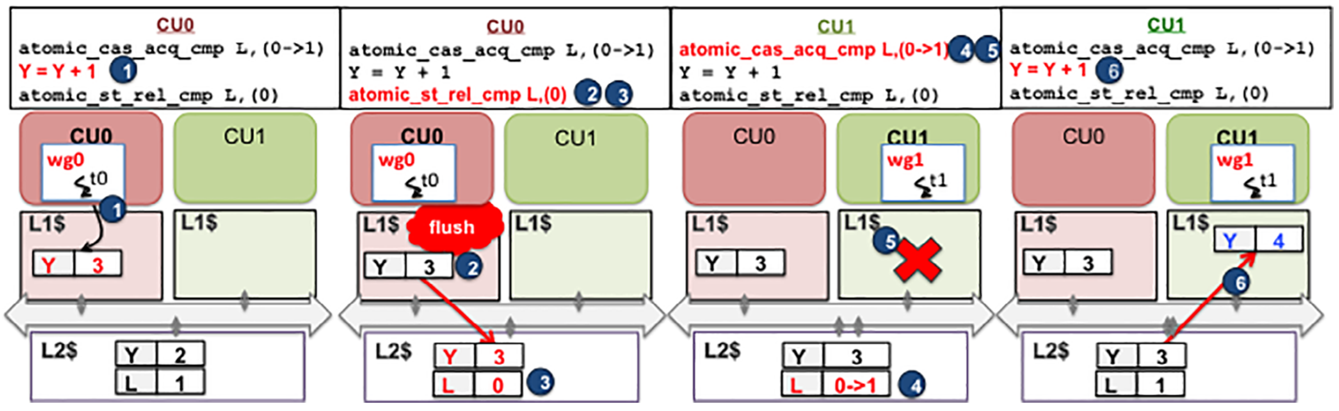

FIGURE 2 Hardware implementation of global-scoped acquire and release synchronization operations

As the example clearly shows, heavyweight cache invalidation and/or flush operations must be performed on local data caches and the atomic operation must be performed at the L2 cache to implement a global-scoped synchronization operation. So, *it is highly recommended to use local-scoped synchronization whenever possible, since local-scoped synchronization significantly reduces the synchronization overhead.*

3 | ASYMMETRIC SHARING

Scoped-synchronization is not sufficient for asymmetric sharing. Without any special support, asymmetric sharing requires using heavyweight global-scoped synchronization to encompass all possible synchronization agents. RSP has been proposed[5] to support asymmetric sharing on GPUs.

3.1 | Remote scope promotion

RSP[5] allows use of lightweight local-scoped synchronization for frequent accesses by local-modifier and defers the most synchronization overhead to the rare remote accesses in asymmetric sharing. When performing a remote synchronization by a remote-modifier, the prior and subsequent local-scoped synchronization operations must be promoted to global-scope. This facilitates pushing all prior local updates to global scope and pulling up-to-date shared data from the global-scope during the execution of critical section. RSP introduces three new remote synchronization operations. These new operations and their semantics are as follows:

- *Remote acquire (rem_acq):* Remote acquire synchronization operation works as an upward fence for remote-synchronization agent (i.e., remote-modifier). First, it promotes last local-scoped release operation (performed by local-modifier) to the global-scope. And then, it carries out an acquire operation for remote-synchronization agent. Promotion of local-scoped release enables pushing all prior local updates to global-scope and the acquire operation by remote-modifier facilitates for pulling the most up-to-date data from global scope.
- *Remote release (rem_rel):* Remote release synchronization operation works as a downward fence for the remote-synchronization agent. It performs a release operation at the global scope for remote-synchronization agent and ensures the promotion of next local-scoped acquire operation (which is subsequently executed by the local-modifier). Release operation enables pushing all updates that are performed by the remote-modifier to global scope. Promotion of the subsequent local-scoped acquire ensures that the local-modifier pulls the most up-to-date data from global scope.
- *Remote acquire+release (rem_ar):* Remote acquire+release synchronization operation works as both an upward and a downward memory fence for the remote-synchronization agent. It provides the remote-modifier with promotion of last local-scoped acquire and/or release operation (which is performed by local-modifier) and an acquire+release operation which is performed by the remote-modifier at the global scope.

3.1.1 | First implementation of RSP with broadcast cache flush and cache invalidate operations

Hardware support for RSP semantics can be implemented by extending the baseline GPU cache protocol. In RSP work,[5] a naive first implementation of RSP semantics is provided. This implementation is based on a simple broadcasting mechanism that performs heavyweight cache-flush and/or cache-invalidate operations on *all* local caches to promote prior/subsequent local-scoped synchronization operations. From hereon, we refer to the broadcast based first implementation of RSP semantics as *bRSP*.

Unfortunately, bRSP does not include any mechanism to track and locate local-modifiers. Rather, it utilizes a broadcasting based mechanism when promoting a local-scoped synchronization to a target scope (i.e., global scope). As a result, to promote a local-scoped synchronization to global scope, heavyweight cache-flush and cache-invalidate operations are performed on *all* local L1 data caches. In bRSP, all local caches get affected from remote-synchronization operations despite the fact that not all of them are involved in the synchronization. Remote-synchronization operations reduce the effectiveness of local caches and might create pressure on memory system.

3.2 | Lazy release consistency for GPUs

Recently, Alsop et al. proposed heterogeneous lazy release consistency (hLRC) which provides support for asymmetric synchronization without making use of scopes. Instead, it utilizes atomic registration to track ownership of synchronization variables. hLRC performs the required coherence actions such as cache-invalidation and/or cache-flush operations at the time of registration transfer (i.e., when a synchronization variable changes its location). With registration tracking mechanism, it performs heavyweight cache-flush and cache-invalidation operations only on the L1 caches that are involved in the synchronization (i.e., the L1 cache that is the current owner of the synchronization variable and the L1 cache that is requesting the ownership of the synchronization variable).

hLRC does not distinguish between the scopes and the coherence actions are triggered by the atomic accesses. Accesses to any atomics require having the registration for them. An atomic variable can be registered to one of the L1 data caches or it could be available at L2 cache. L2 cache keeps track of the owners of the atomic variables and registration transfers are coordinated by L2 cache. There are three cases that must be considered on an atomic access. These three cases are as follows: ① If the accessing L1 data cache has the registration, atomic accesses are simply performed locally. ② If the registration is on L2 cache, first atomic registration must be requested from L2 cache. Once the registration is transferred from L2 cache then, the atomic access is performed locally. Following the atomic access, a cache-invalidate operation is performed. ③ If another L1D has the registration then the registration must be transferred from the current L1D to the requesting L1D. To perform the registration transfer, first the current owner performs a cache-flush operation and once the cache-flush completes, then the registration is passed to the requesting L1D cache. When the new owner gets the registration, atomic operation is performed locally. Following the atomic operation, a cache-invalidation is performed locally on the new owner.

We provide the coherence actions in hLRC implementation and a comparison of sRSP and hLRC in Appendix A.

4 | AN EFFICIENT IMPLEMENTATION OF RSP SEMANTICS: SELECTIVE CACHE-FLUSH AND CACHE-INVALIDATION OPERATIONS FOR RSP (SRSP)

In this work, we propose a novel mechanism for implementation of RSP semantics. Our new mechanism is based on identifying the cache affinity of the local-modifier when performing a remote synchronization. Once we identify the local-modifier's cache, we perform the cache flush and cache invalidate operations *selectively*—only on the local-modifier's cache. With selective-flush and selective-invalidate operations, our new mechanism reduces the unnecessary cache-flush and cache-invalidation operations, and therefore it reduces the overhead of remote synchronization operations.

Our mechanism to identify the local modifier's cache involves tracking all local-scoped (i.e., wg-scoped) synchronization operations. To implement RSP semantics with *selective* cache-flush and cache-invalidate operations, we add two new hardware structures and modify the baseline GPU cache protocol. Our implementation also makes use of sFIFO based cache flushing and single cycle flush-invalidation operations. Two new hardware structures that we introduce are as follows:

- *Local release table (LR-TBL):* We use *LR-TBL* to track all local-scoped release operations. LR-TBL can be considered as a small-sized CAM structure. For each local-scoped release operation, we associate an entry in the LR-TBL and an sFIFO entry for the paired atomic operation. Each LR-TBL entry includes an address (i.e., memory access address) and a pointer to the sFIFO entry associated with the synchronization operation. As we will explain later, we use this sFIFO pointer as a marker to be used when performing a cache-flush operation that is initiated by a remote-synchronization.

  We preferred to keep LR-TBL and sFIFO structures separate to eliminate false sharing and for performance reasons. This way we can perform the sFIFO push operations when performing regular *store* operations without added extra delay.
- *Promoted acquire table (PA-TBL):* To avoid unnecessary (false) cache-invalidation operations when performing local-scoped acquire operations, we track addresses that require scope promotion. We perform a cache-invalidation only when the address of a local-scoped acquire operation matches with an address in the PA-TBL or when the PA-TBL is full.

Similar to original RSP implementation, our work supports the three new RSP operations. In the remaining of this section, we explain the modifications to the baseline GPU cache protocol to implement RSP semantics with *selective* cache-flush and cache-invalidate operations.

We explain our mechanism with a running example which is shown in Figures 3–5. In this example, we demonstrate a critical section protected by a lock which is most of the time accessed by workgroup (*wg0*, *the local-modifier*), while being rarely accessed by other workgroups (*the remote-modifiers*). The local-modifier (wg0) is assumed to be running on CU0 and the accesses by local-modifier are accompanied with lightweight local-scoped synchronization operations. A remote-modifier is assumed to be workgroup *wg1* and running on CU1. The remote-modifier uses *remote acquire (rem_acq)* and *remote release (rem_rel)* synchronization operations when accessing to the shared data that is local for wg0 (i.e., the shared data owned by wg0). With this running example, we explain how a remote-modifier synchronizes with local-scoped release and acquire operations.

4.1 | Implementation of a local-scoped release operation in sRSP

As we illustrate in Figure 3(A), our example starts with workitem *t0* from *wg0* performing an update on shared data`Y`. The line address of `(Y)` is inserted into the sFIFO at step ①.

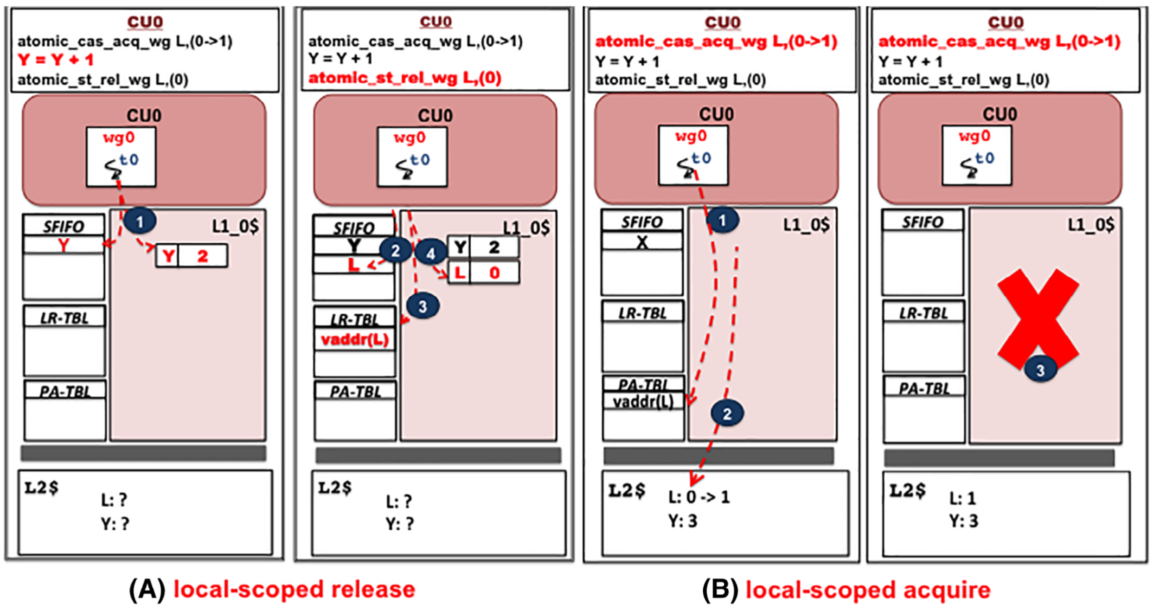

FIGURE 3 Implementations of local-scoped release and acquire operations in sRSP. (A) *t0* in workgroup *wg0* updates `Y` and then performs a local-scoped release operation by executing `atomic_st_rel_wg(L ← 0)`, (B) To enter the critical section again, *t0* from *wg0* executes `atomic_cas_acq_wg(L, 0 → 1)`

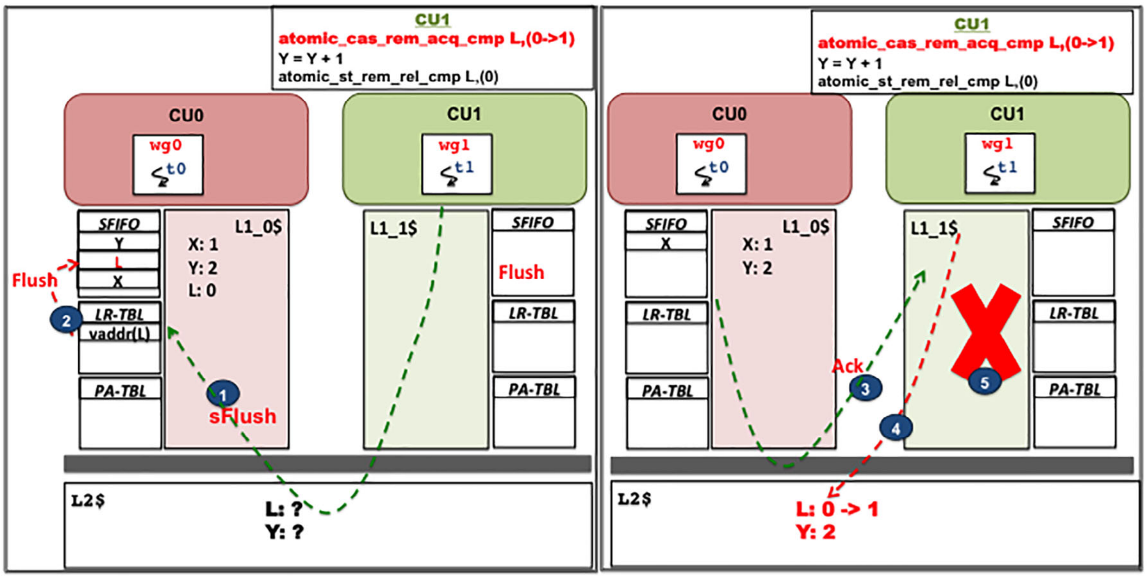

FIGURE 4 Implementation of global-scoped remote acquire operation in sRSP. Remote-agent *t1* from *wg1* executes `atomic_cas_rem_acq_cmp(L, 0->1)` to enter to the critical section

Next, *t0* executes `atomic_st_rel_wg(L ← 0)` synchronization operation. When the `atomic_st_rel_wg(L ← 0)` executes, first the line address of `(L)` is inserted into sFIFO and a pointer (i.e., sFIFO entry id) for the sFIFO entry is collected (at step ②). Following the sFIFO insertion, *LR-TBL* is looked up to find if there is any entry associated with the address of `(L)`. If there is already an entry in the *LR-TBL*, the sFIFO pointer in this entry is updated with the current pointer. Please note that, this sFIFO pointer works as a flush marker for a possible remote-acquire operation associated with address `(L)`. If there is no entry exists for address `(L)` in *LR-TBL*, then we create a new entry in *LR-TBL* for address `(L)` with the collected sFIFO pointer (at step ③). Finally, the `atomic_st` is performed locally on L1 cache and the result is returned to *t0* in wg0 (at step ④).

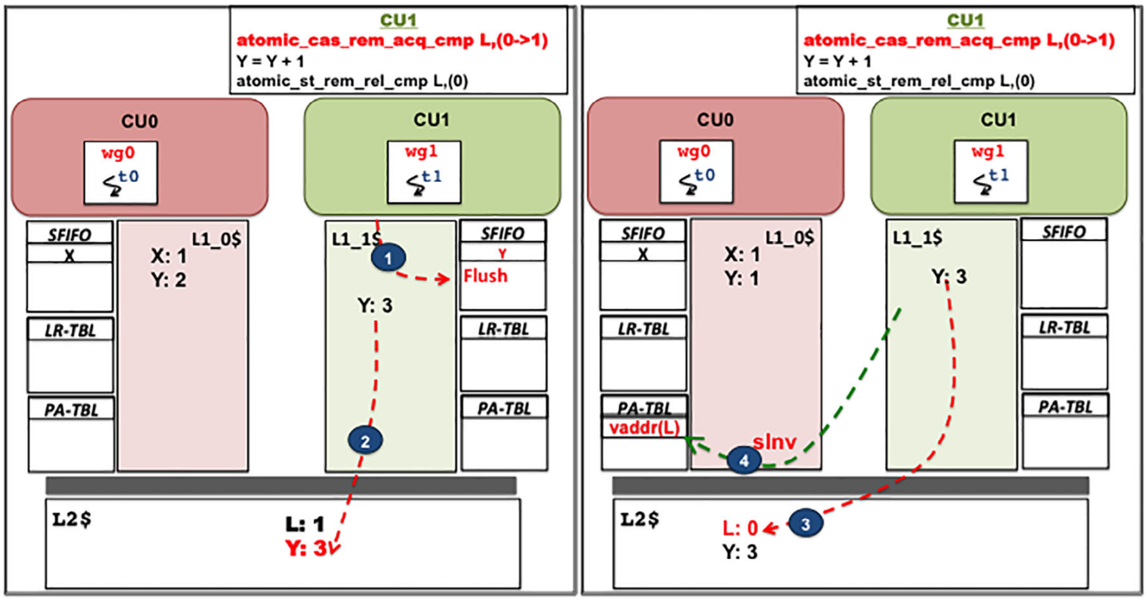

FIGURE 5 Implementation of global-scoped remote release operation in sRSP. To exit the critical section, remote agent *t1* from *wg1* executes `atomic_st_rem_rel_cmp(L, 0)` instruction

4.2 | Implementation of a remote-acquire operation in sRSP

As shown in Figure 4, workitem *t1* from *wg1* (which is running on CU1) executes the `atomic_cas_rem_acq_cmp(L, 0 → 1)` operation at step ① with an intention to enter the critical section. Remote-acquire operation guarantees to pull the current data from local-modifier's cache (i.e., $L1_0 on CU0). $L1_1 sends a *selective-flush* request to all L1 caches at step ②. Upon receiving selective-flush request, L1 caches lookup their *LR-TBLs* for address `(L)`. This is to decide if there is any prior local-scoped release operation on address `(L)` that must be promoted to the global-scope. Since the data is statically mapped to a single agent (i.e., the local-modifier) we expect only a single hit. If *LR-TBL* lookup misses on an L1 cache, then that L1 cache invalidates the cache line with address*(L)* (if the line exits), and immediately sends an acknowledgement to the remote-requester.

In our running example, we observe a *LR-TBL* lookup hit on CU0's L1 cache($L1_0), since the *wg0* is local-modifier. From the LR-TBL entry that hits, we get an sFIFO flush marker which is a pointer to the atomic_st operation paired with the last local-scoped release operation associated with the address `(L)`. Now, on $L1_0, the cache controller initiates a cache-flush operation at step ② and flushes all addresses up to sFIFO flush marker (flush marker is the last address to be flushed). Once the cache-flush completes, $L1_0 invalidates the cache line with address*(L)* (if the line exits). Next, an acknowledgment is sent to $L1_1 step ③. After that, $L_1 sends the `atomic_cas` operation to L2 cache at step ④. In the end, a cache-invalidate operation is performed on $L1_1 (at step ⑤) and the result of atomic_cas is returned to the workitem *t1* from *wg1* to finish the synchronization operation.

While performing the remote-acquire operation, L2 cache must lock the address (L) so that no load operation is allowed on that cache line from any L1 caches. In addition, $L1_1 must be blocked to serve any other requests from CU1.

4.3 | Implementation of a remote-release operation in sRSP

Once the remote-agent *t1* from *wg1* successfully synchronizes, it can then proceed with critical section. Following the critical section, the `atomic_st_rem_rel_cmp(L → 0)` operation is executed by *t1* as shown in Figure 5. This operation initiates a cache-flush on $L1_1 (at step ① in Figure 5). As a result of cache-flush, the updated value of `Y` is written back to L2 cache (step ② in Figure 5. Following the cache-flush, the `atomic_st` operation is performed at L2 cache (step ③). Finally, to promote the next local-scoped acquire operation on address `(L)`, a *selective-invalidate* request is sent out to all L1 caches. Upon receiving the selective-invalidate request with address `(L)`, L1 caches insert the address `(L)` into their *PA-TBLs* (step ④ in Figure 5) and they invalidate the cache block that the address (L) maps. The cache block that the address (L) maps is also invalidated on the remote-modifier's cache. The result of `atomic_st` is returned to CU1 and the remote-release operation completes.

4.4 | Implementation of a local-scoped acquire operation in sRSP

When performing a local-scoped acquire in sRSP, first it must be checked if the scope promotion is needed. As seen in Figure 3(B), the local-modifier, *t0* from *wg0* in our running example, executes the `atomic_cas_ acq_wg(L, 1 ← 0)` operation. At step ①, the *PA-TBL* is looked up for address `(L)` on \$L1_0. If there is no hit in *PA-TBL*, then the `atomic_cas` operation is performed locally and the synchronization completes. If the lookup hits as it happens in our running example, then the `atomic_cas` operation is sent to L2 cache to be performed at global scope (step ② in Figure 5). When the atomic operation result is returned from L2 cache, a cache-invalidation is performed on \$L1_0 (step ③ in Figure 5). When performing cache invalidation, *PA-TBL* and *LR-TBL* are emptied on \$L1_0.

4.5 | Implementation of a local-scoped acquire+release operation in sRSP

Implementation of a local-scoped acquire+release operation in sRSP starts similar to a local-scoped acquire operation. First, it is checked that if there is a need for scope promotion by checking the *PA-TBL* for a hit. If there is no hit in *PA-TBL*, then the paired atomic operation is performed locally and the synchronization completes. Otherwise (if there is a hit), first, a cache-flush operation is performed, next the paired atomic operation is sent to L2 cache and in the end, the synchronization completes with a cache invalidation.

4.6 | Implementation of a remote acquire+release operation in sRSP

Implementation of a remote-acquire operation starts with a cache-flush operation that is performed on remote-modifier's cache. While cache-flush is being performed on remote-modifier's cache, a *selective-flush* request is sent out to all L1D caches. Depending on the *LR-TBL* hit, a cache-flush operation is triggered on a L1D cache. If there is no hit in *LR-TBL*, then there is no need for cache-flush operation. All L1D caches send an acknowledgment to the requesting L1D cache (the remote-modifier's cache). Once the remote-modifier's cache collects all acknowledgments for selective-flush, then the paired atomic operation is sent to L2 cache. When the atomic operation result is returned from L2 cache, a *selective-invalidate* request is sent to all L1D caches. All receiving L1D caches insert the associated synchronization address into their *PA-TBL*. A cache-invalidate operation is performed on remote-modifier's cache and that completes this synchronization.

4.7 | Managing LR-TBL and PA-TBL

LR-TBL and PA-TBL are finite-sized CAM structures and they can become full as time passes. If a PA-TBL is full, a cache-invalidation operation is performed and both PA-TBL and LR-TBL are emptied. We choose the size of LR-TBL as same size as the sFIFO. As the entries from sFIFO are removed when it is full, this will result in removing entries from the LR-TBL. In our design, when an sFIFO entry that is associated with local-scoped release is being removed, then the associated LR-TBL is also removed.

4.8 | Implementing sRSP in a hierarchical memory system

Even though we explained our design to perform RSP from local (i.e., workgroup) to global (i.e., component) scope, it is not limited to that. It is easily extendable to work with multiple levels of memory hierarchy and to perform a scope promotion from local to system scope.

4.9 | sRSP optimizations

When performing a remote synchronization operation, the local-modifier could exist on the same CU. In that case, since the local-modifier and the remote-modifier share the same L1 cache, there is no need to perform scope promotion. In our implementation, as an optimization, we first do a *LR-TBL* lookup locally. If we find the address in local LR-TBL then we perform the remote-acquire or remote-release operation as local-scoped acquire and release operations. Selective-flush or selective-invalidation requests are sent out to other L1 caches only when the LR-TBL misses.

We also implemented another optimization to eliminate some of the broadcasting messages when performing remote acquire and/or release operations. For this optimization, we extended our work further, first, to differentiate between the L1 caches responding to remote-flush operations with a positive acknowledgement and a negative acknowledgement, then, we implemented a caching mechanism to follow the L1 caches responding with a positive acknowledgement and finally, we made use of this information for only sending selective-flush request to a single L1 cache (i.e., the local-modifier's cache).

5 | EVALUATION

5.1 | Methodology

For this study, we used Gem5 APU simulator[10] which is capable of emulating OpenCL run-time API and executing OpenCL[7] kernels compiled into HSAIL instructions.[11]

Our simulated GPU devices are modeled with the configuration parameters listed in Table 1. Each GPU device that we simulate has 64 CUs. Each CU consists of four SIMD units. There exists a scheduler on each CU to select and execute a wavefront from a maximum of 40 wavefronts using an oldest-job-first algorithm. A wavefront is composed of 64 workitems.

Each CU includes a local L1 data cache. There are L1 instruction caches, each of them is shared by four CUs. There is also an L2 cache that is shared by all CUs. All L1 data caches and L1 instruction caches are connected to the L2 cache. Then the L2 cache is connected to the system memory.

All local data caches and L2 cache are equipped with sFIFO structures. sFIFO structures on L1 data caches include 16 entries, and sFIFO on L2 cache includes 24 entries. In our simulations, L1 and L2 caches implement a write-combining with no-allocate cache protocol. In our baseline GPU model, release operations trigger cache-flush(es) (depending on scope of release) using sFIFO. Acquire operations initiate 1 cycle cache-invalidation (also depending on scope of acquire). We extended and modified the L1 data cache and L2 cache controllers to support selective-invalidation and selective-flush operations that we explain in Section 4.

5.1.1 | Workloads

We selected four graph applications from Pannotia[12] benchmark suite. Pannotia implements a set of graph applications in OpenCL. These applications experience significant load imbalance depending on input graph. Load-imbalance happens when workitems process graph nodes with diverse numbers of edges. We run our graph benchmarks for a diverse set of inputs chosen from 9th and 10th DIMACS Implementation Challenge[13] and from Reference 14.

We modified implementations of these benchmarks to use work-stealing[3] for dynamic load-balancing. Similar to the original RSP work, our implementation uses a lock-free implementation of work-stealing as described in Reference 15. In our implementation each work-queue is initially mapped to a workgroup. When a workgroup's local work-queue is empty, then the workgroup attempts to steal work from other workgroups' work-queue. The kernel will exit when all of the work-queues are empty. Dequeuing from a local work-queue happens at the tail of the queue, while stealing from another workgroup's queue happens at the head. Collision might happen while dequeuing and stealing. For that reason, accesses to work-queues must be guarded with synchronization. We implement our benchmarks to have 64 work-queues.

The graph benchmarks and their input sets that we used are as follows:

Graph coloring (CLR)
It is an iterative algorithm in which, each node's color is determined by evaluating adjacent nodes. *CLR.1*uses *dip20090126_MAX*[13] and *CLR.2* uses *ecology1*.[13]

TABLE 1 Configuration parameters for simulated GPU devices

`Num. of CUs`	**64**
`Clock`	1 GHz, four SIMD units
`Wavefronts per Scheduler`	40
`Data cache`	16-way set-associative 16 kB with 64B cache line and four cycles latency
`L2 Cache`	16-way set-associative 512 kB with 64B line and 24 cycles latency
`Instruction cache`	8-way set-associative 32 kB with 64B line and four cycles latency
`DRAM`	DDR3, eight Channels, 500 MHz
`Protocol`	Write-combining, write-no-allocate

Abbreviations: CU, compute unit; SIMD, single instruction multiple data.

Maximal independent set (MIS)

This benchmark finds the largest subset for a given graph such that there are no two nodes neighboring. *MIS.1* uses *itdk0304_rlinks_undirected_MAX*[14] and *MIS.2* uses *caidaRouterLevel*.[13]

PageRank (PR)

This is a well-known graph application that tries to estimate the importance of web pages. The importance of a web-page is calculated using the other linked pages. *PRK.1* uses *coAuthorsDBLP*,[14] *PRK.2* uses *cond-mat-2003*,[13] and *PRK.3* uses *smallWorld*.[13]

Single-source shortest path (SSSP)

It finds the shortest distances from a given source vertex to all other vertexes. It uses an approach that gradually expands frontiers. We use three input graphs when running this benchmark. *SSSP.1* uses *USA-road-BAY*,[13] *SSSP.2* uses *USA-road-COL*,[13] and *SSSP.3* uses *USA-road-NY*.[13]

5.1.2 | Evaluation scenarios

We perform our evaluations by running our graph benchmarks for five execution scenarios. Four of the execution scenarios are also used for original RSP[5] study. Our evaluation scenarios are as follows:

In the Baseline scenario, stealing is disabled and global-scoped synchronization is used when dequeuing from work-queues. In this scenario, synchronization is not necessary because each work-queue is only accessed by the owner and there is no enqueuing. But, with global-scoped synchronization, this scenario provides a baseline for us.

Steal-only scenario uses stealing with global-scoped synchronization. Benefits come from load-balancing.

Scope-only scenario avoids global-scoped synchronization and uses local-scoped synchronization. But since scoped-synchronization is not sufficient for dynamic sharing, no stealing is allowed. Benefits come from using lightweight local-scoped synchronization.

bRSP uses the initial implementation of RSP with broadcast cache-flush and cache-invalidation operations. Benefits come from both stealing and local-scoped synchronization with frequent accesses by the owner of the work-queue.

sRSP uses our new implementation for RSP with selective-flush and selective-invalidation operations. Benefits come from both stealing and local-scoped synchronization with frequent accesses by the owner of the work-queue.

5.2 | Results

5.2.1 | Speedup

In Figure 6, we compare the speedups of five configurations (*Baseline*, *Steal-only*, *Scope-only*, *bRSP*, and *sRSP*) relative to *Baseline* configuration for 64 CU GPUs.

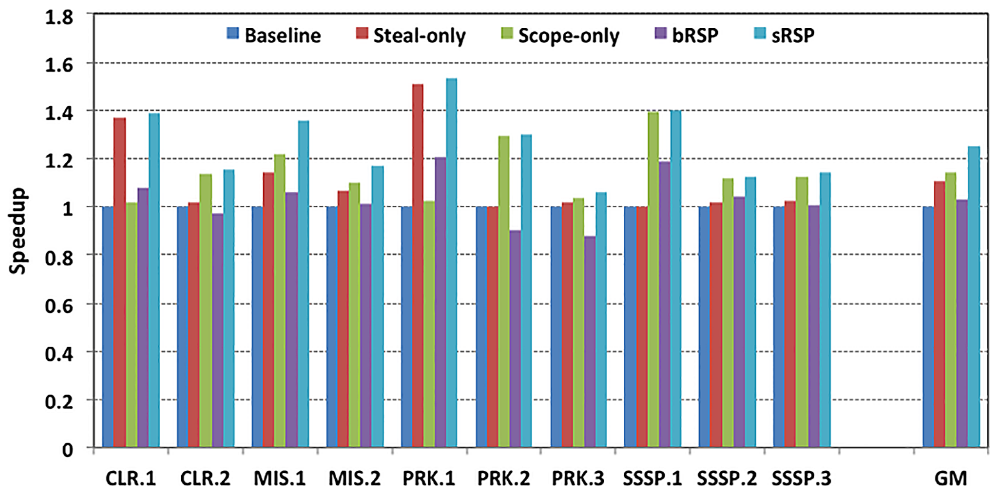

FIGURE 6 Speedups with our benchmarks (CLR.1, CLR.2, MIS.1, MIS.2, PRK.1, PRK.2, PRK.3, SSSP.1, SSSP.2, and SSSP.3) for five scenarios (*Baseline*, *Steal-only*, *Scope-only*, *bRSP*, and *sRSP*) on a 64 CU GPU device

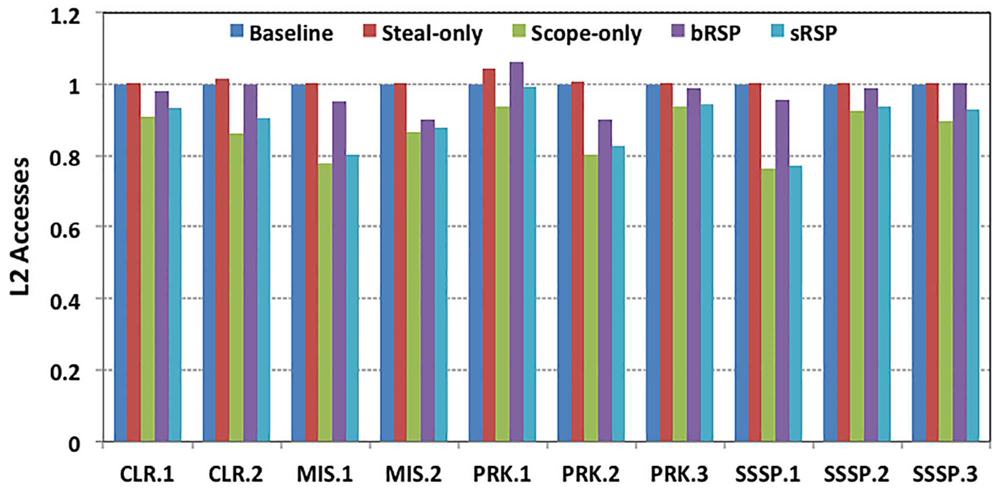

FIGURE 7 Bandwidth usage with five scenarios (*Baseline*, *Steal-only*, *Scope-only*, *bRSP*, and *sRSP*) using our benchmarks (*CLR.1*, *CLR.2*, *MIS.1*, *MIS.2*, *PRK.1*, *PRK.2*, *PRK.3*, *SSSP.1*, *SSSP.2*, and *SSSP.3*) on a 64 CU GPU device. Values are represented relative to *Baseline* scenario

The *Scope-only* configuration shows good performance improvement in general. On the average, it shows 14% performance improvement on a 64 CU GPU device. At the highest, the Scope-only configuration improves the performance of SSSP.1 and PRK.2 benchmarks 39% and 29%, respectively. With use of lightweight local-scoped synchronization, the Scope-only configuration always shows some amount of performance improvement. Depending on the amount of synchronization and the locality in the benchmark, the amount of performance improvement would vary.

While the *Steal-only* configuration improves the performance by 11% on the average, we observe not much benefits for some of the benchmarks. The performance benefits of work-stealing totally depend on the load imbalance that the benchmark experiences. When the load imbalance is high as it happens in PRK.1 and CLR.1, we observe very high performance improvement with Steal-only configuration.

The *bRSP* configuration uses initial implementation of RSP and therefore it benefits from both work-stealing and use of scopes. On the other hand, heavy use of cache-invalidation and cache-flush operations with remote synchronization could harm the performance. At each remote-synchronization bRSP performs 64 cache-flush and/or cache-invalidation operations. On the average, we observe very insignificant (3%) performance improvement with bRSP configuration on a 64 CU GPU device. At the highest, bRSP configuration shows 20% performance improvement when running PRK.1 benchmark and it shows a 14% slowdown when running PRK.3 on a 64 CU GPU device.

The *sRSP* configuration uses our implementation of RSP semantics. It benefits from work-stealing and also from use of scopes. Since it makes use of selective cache-invalidation and cache-flush operations, the synchronization overhead does not affect the performance. Therefore, in general we observe sRSP configuration improves the performance. On the average, sRSP configuration improves the performance up to 25%. At most, it improves the performance 53% with PRK.1 benchmark on a 64 CU GPU device.

5.2.2 | Bandwidth usage

In Figure 7, we illustrate the L2 accesses for five configurations relative to *Baseline* configuration. As we can see from the figure, use of scopes in *Scope-only*, *bRSP*, and *sRSP* configurations helps with bandwidth usage. We observe up to 24% reduction in L2 accesses with Scope-only scenario, 11% reduction in L2 accesses with bRSP configuration, and 23% reduction in L2 accesses with sRSP implementation. The reduction in L2 accesses positively affect the performance.

5.2.3 | Remote synchronization overhead

In Figure 8, we compare the remote synchronization overhead for *bRSP* and *sRSP* configurations. As seen from Figure 8, *sRSP* configuration reduces the remote synchronization overhead significantly. On the average, we observe 37% reduction in synchronization overhead compared with bRSP. We observe up to 57% improvement when running PRK.1 benchmark. [**]

[**]We also performed experiments running on 8 CU, 16 CU, and 32 CU GPU devices with a subset of our benchmarks and observed that performance overhead of remote synchronization is mostly tolerable for small GPU devices (8 CU, 16 CU, and 32 CU GPU devices) it becomes more significant for 64 CU GPU devices.

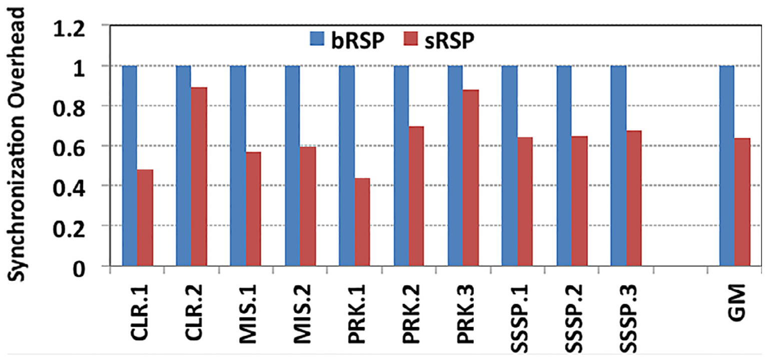

FIGURE 8 Synchronization overhead of bRSP and sRSP relative to bRSP with our benchmarks (*CLR.1*, *CLR.2*, *MIS.1*, *MIS.2*, *PRK.1*, *PRK.2*, *PRK.3*, *SSSP.1*, *SSSP.2*, and *SSSP.3*) on a 64 CU GPU device

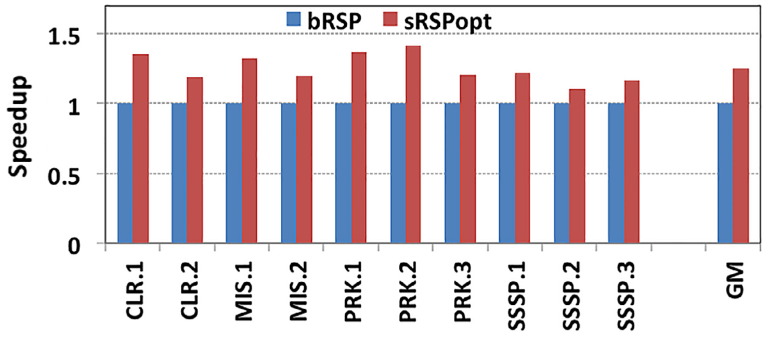

FIGURE 9 Performance of bRSP and optimized sRSP implementation (sRSPopt) relative to bRSP when running modified version of our benchmarks (*CLR.1*, *CLR.2*, *MIS.1*, *MIS.2*, *PRK.1*, *PRK.2*, *PRK.3*, *SSSP.1*, *SSSP.2*, and *SSSP.3*) on a 64 CU GPU device

5.3 | Performance of sRSP with optimizations

To evaluate the performance effect of optimizations on sRSP that we explain in Section 4.9, we modified our benchmark implementations to have a single wave-front in a workgroup and four workgroups mapped to a CU. This modification allows sRSP to detect the locality of synchronization when performing remote-synchronization and avoid unnecessary cache-flush and/or cache-invalidation operations.

In Figure 9, we show the performance results for bRSP and *optimized sRSP implementation (sRSPopt)* normalized to bRSP configuration when running with modified versions of our benchmarks. As we see from the figure, we observe up to 41% of performance improvement with sRSPopt compared with bRSP configuration. On the average sRSPopt achieves 25% performance improvement over bRSP implementation.

6 | HARDWARE COST AND POWER CONSUMPTION

As we mention in Section 4, we modify the L1D and L2 cache protocols to implement the remote synchronization operations and we add two new structures to local caches for sRSP implementation. We believe that sRSP implementation does not increase the area significantly. sRSP helps with dynamic power consumption with reduced execution time and reduced bandwidth use. However, we should also consider increased static power consumption due to two newly added hardware structures (LR-TBL and PA-TBL) in sRSP implementation. We leave the detailed study of area estimation and power consumption for sRSP as a future work.

7 | RELATED WORK

Entry consistency[16] and Scope consistency[17] are two examples using scopes in the context of memory consistency on general purpose multiprocessor systems.

There have been several research focusing on GPU coherence and GPU transactional memory (TM). *TC-Weak*[18] proposed a physical time-stamp based cache coherence protocol while *RCC*[19] and *G-TSC*[20] proposed logical time-stamp based cache coherence protocols. Sinclair et al.[21] extended the Denovo's[22] cache coherence protocol to GPUs. Although, it is possible to support asymmetric synchronization with coherent L1 caches, neither of these work provides the benefits of scoped synchronization in asymmetric synchronization. The studies in References 23-27, and 28 presented hardware or software based TMs for GPUs. Hardware TMs incur high hardware overhead and Software TMs incur high performance overhead. Any of these GPU TM studies did not consider scopes.

There have also been studies to improve the support for fine grained synchronization on GPUs. HQL[29] proposed a hardware based locking mechanism to address the fine-grained synchronization on GPU devices. Li et al.[30] presented a locking mechanism using scratchpad-memory for interthread producer-consumer style communication. Xu et al.[31] proposed a new locking mechanism that is based on lock-stealing to address the concurrency bugs on GPUs. Wang et al.[32] proposed a software approach for efficient fine-grained synchronization on GPUs. In their approach, programmers have to implement two separate kernels, one kernel for handling critical sections (synchronization server) and another one for implementing the noncritical section work (synchronization client). They replaced the global lock operations with scratchpad-memory lock operations. Client threads communicate with server threads using a message passing system that makes use of global memory. Anand et al.[33] proposed a deadlock free lock-based synchronization approach for GPUs. *Quick Release*[9] improved the GPU cache performance at the synchronization points.[34] showed an efficient implementation of Sequential Consistency for GPUs. Unfortunately, none of these studies considers the scopes and/or asymmetric synchronization.

Quickly Reacquirable Locks[35] and Simple and Fast Biased Locks[4] target asymmetric synchronization on CPUs. Both work rely on sophisticated synchronization support that CPUs provide. Yi and Yao[36] proposed a delegation based scalable lock for NUMA architectures. Unfortunately, GPUs use simple caching mechanism and these work cannot be simply extended to GPUs.

Two studies most relevant to ours are RSP[5] and hLRC. In RSP[5] work, Orr et al. introduced RSP semantics with an initial hardware implementation which is shown to be not scalable. sRSP implements RSP semantics and provides a scalable hardware implementation. hLRC is recently proposed by Alsop et al.[37] hLRC combines RSP with DeNovo's[22] mechanism for tracking ownership of synchronization variables. DeNovo's mechanism for tracking ownership uses dynamic registration of synchronization variables. Each synchronization variable must be dynamically registered and this registration is managed at L2 cache similar to HQL locks.[29] Although hLRC provides scalability while supporting asymmetric synchronization on GPUs, unfortunately, as explained before in Section 3.2, it comes with several shortcomings. Our work also supports asymmetric synchronization while providing a scalable and efficient implementation of RSP while making use of scopes.

8 | CONCLUSIONS

GPUs utilize simple coherence actions and synchronization becomes quite expensive to maintain data consistency. Recently, scoped synchronization has been introduced to limit overheads of synchronization when the sharing is local. Unfortunately, asymmetric sharing like other dynamic sharing patterns cannot benefit from scoped synchronization. RSP work[5] introduced RSP to allow the use of lightweight synchronization in asymmetric sharing patterns. However, initial implementation of RSP applies a simple broadcasting approach and applies heavyweight cache-invalidation and cache-flush operations on *all* local caches when performing remote synchronization. This could quite disturb the effectiveness of local caches on GPUs.

In this work, we introduced a novel mechanism, sRSP, to track local-scoped synchronizations and apply cache-invalidation and cache-flush operations *selectively* when performing remote synchronization. sRSP significantly reduces remote synchronization overhead, therefore scales better and improves performance for applications that use asymmetric sharing. On the overage 25% performance improvement is achieved with sRSP when running on 64 CU GPUs.

ACKNOWLEDGMENTS

This research is supported by The Scientific and Technological Research Council of Turkey (TUBITAK) under the Career Development Program (CAREER), grant no 117E053. A preliminary version of this work was presented at the sixth High Performance Computing Conference (BASARIM 2020).[38]

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

ORCID

Ayse Yilmazer-Metin https://orcid.org/0000-0003-4502-7365

REFERENCES

1. Hower DR, Hechtman BA, Beckmann BM, et al. Heterogeneous-race-free memory models. Paper presented at: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS '14; 2014:427-440; ACM, New York, NY.

2. Gaster BR, Hower D, Howes L. HRF-relaxed: adapting HRF to the complexities of industrial heterogeneous memory models. *ACM Trans Archit Code Optim*. 2015;12(1):7:1-7:26.
3. Blumofe RD, Leiserson CE. Scheduling multithreaded computations by work stealing. *J ACM*. 1999;46(5):720-748.
4. Vasudevan N, Namjoshi KS, Edwards SA. Simple and fast biased locks. Paper presented at: Proceedings of the . 19th International Conference on Parallel Architectures and Compilation Techniques PACT '10; 2010:65-74; ACM, New York, NY.
5. Orr MS, Che S, Yilmazer A, Beckmann BM, Hill MD, Wood DA. Synchronization using remote-scope promotion. *SIGPLAN Not*. 2015;50(4):73-86.
6. International organization for standardization . working draft, standard for programming language C++; 2017. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4713.pdf. Accessed January 21, 2021.
7. The Khronos Group Inc Khronos OpenCL registry; 2018. https://www.khronos.org/registry/OpenCL/. Accessed January 21, 2021.
8. Sorin DJ, Hill MD, Wood DA. *A Primer on Memory Consistency and Cache Coherence*. 1st ed. San Rafael, CA: Morgan & Claypool Publishers; 2011.
9. Hechtman BA, Che S, Hower DR, et al. QuickRelease: a throughput-oriented approach to release consistency on GPUs. Paper presented at: Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), Orlando, FL; 2014:189-200.
10. AMD The AMD gem5 APU simulator: modeling heterogeneous systems in gem5; 2015. http://www.m5sim.org/wiki/images/f/fd/AMD_gem5_APU_simulator_micro_2015 _final.pptx. Accessed January 21, 2021.
11. HSA Foundation HSA programmer's reference manual: HSAIL virtual ISA and programming model, compiler writer, and object format (BRIG); 2018. http://www.hsafoundation.com/standards/. Accessed January 21, 2021.
12. Che S, Beckmann BM, Reinhardt SK, Skadron K. Pannotia: understanding irregular GPGPU graph applications. Paper presented at: Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC), Portland, OR; 2013: 185-195.
13. DIMACS Implementation challenges; 2021. http://archive.dimacs.rutgers.edu/Challenges/. Accessed January 21, 2021.
14. Sommer C. http://www.sommer.jp/graphs/. Accessed January 21,2021.
15. Cederman D, Tsigas P. Dynamic load balancing using work-stealing. Hwu Wen-mei W., *GPU Computing Gems Jade Edition*. In Applications of GPU Computing Series Burlington, MA: Morgan Kaufmann; 2012;485-499.
16. Bershad BN, Zekauskas MJ, Sawdon WA. The Midway distributed shared memory system. Digest of Papers Compcon Spring; 1993:528-537; San Francisco, CA.
17. Iftode L, Singh JP, Li K. Scope consistency: a bridge between release consistency and entry consistency. Paper presented at: Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA '96; 1996:277-287; ACM, New York, NY.
18. Singh I, Shriraman A, Fung WWL, O'Connor M, Aamodt TM. Cache coherence for GPU architectures. Paper presented at: Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), Shenzhen; 2013:578-590.
19. Ren X, Lis M. Efficient sequential consistency in GPUs via relativistic cache coherence. Paper presented at: Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), Austin, TX; 2017:625-636.
20. Tabbakh A, Qian X, Annavaram M. G-TSC: timestamp based coherence for GPUs. Paper presented at: Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), Vienna, Austria; 2018:403-415.
21. Sinclair MD, Alsop J, Adve SV. Efficient GPU Synchronization without scopes: saying no to complex consistency models. Paper presented at: Proceedings of the 48th International Symposium on Microarchitecture MICRO-48; 2015:647-659; ACM, New York, NY.
22. Choi B, Komuravelli R, Sung H, et al. DeNovo: rethinking the memory hierarchy for disciplined parallelism. Paper presented at: Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques, Galveston, TX; 2011:155-166.
23. Fung WWL, Singh I, Brownsword A, Aamodt TM. Hardware transactional memory for GPU architectures. Paper presented at: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture MICRO-44; 2011:296-307; New York, NY.
24. Xu Y, Wang R, Goswami N, Li T, Qian D. Software transactional memory for GPU architectures. *IEEE Comput Archit Lett*. 2014;13(1):49-52.
25. Cederman D, Tsigas P, Chaudhry MT. Towards a software transactional memory for graphics processors. Paper presented at: Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization EG PGV'10; 2010:121-129; Eurographics Association, Aire-la-Ville, Switzerland.
26. Villegas A, Asenjo R, Navarro A, Plata O, Kaeli D. Lightweight hardware transactional memory for GPU scratchpad memory. *IEEE Trans Comput*. 2018;67(6):816-829.
27. Villegas A, Navarro A, Asenjo R, Plata O. Toward a software transactional memory for heterogeneous CPU–GPU processors. *J Supercomput*. 2019;75:4177–4192.
28. Ren X, Lis M. High-performance GPU transactional memory via eager conflict detection. Paper presented at: Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), Vienna, Austria; 2018; 235-246.
29. Yilmazer A, Kaeli D. HQL: a scalable synchronization mechanism for GPUs. Paper presented at: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, Boston, MA; 2013:475-486.
30. Li A, van den Braak GJ, Corporaal H, Kumar A. Fine-grained synchronizations and dataflow programming on GPUs. Paper presented at: Proceedings of the 29th ACM on International Conference on Supercomputing ICS '15; 2015:109-118; Association for Computing Machinery, New York, NY.
31. Xu Y, Gao L, Wang R, Luan Z, Wu W, Qian D. Lock-based synchronization for GPU architectures. Paper presented at: Proceedings of the ACM International Conference on Computing Frontiers CF '16; 2016:205-213; Association for Computing Machinery, New York, NY.
32. Wang K, Fussell D, Lin C. Fast fine-grained global synchronization on GPUs. Paper presented at: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS '19; 2019:793-806; Association for Computing Machinery, New York, NY.
33. Anand AS, Srivastava A, Shyamasundar R. A deadlock-free lock-based synchronization for GPUs. *Concurr Comput Pract Exper*. 2019;31(7):e4991.
34. Singh A, Aga S, Narayanasamy S. Efficiently enforcing strong memory ordering in GPUs. Paper presented at: Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Waikiki, HI; 2015:699-712.
35. Dice D, Moir M, Scherer W. Quickly reacquirable locks; 2010.
36. Yi Z, Yao Y. A scalable lock on NUMA multicore. *Concurr Comput Pract Exper*. 2020;32(24):e5964.
37. Alsop J, Orr MS, Beckmann BM, Wood DA. Lazy release consistency for GPUs. Paper presented at: Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, Taiwan; 2016:1-14.
38. Yilmazer-Metin A. sRSP: GPUlarda Asimetrik Senkronizasyon Icin Yeni Olceklenebilir Bir Cozum. Paper presented at: Proceedings of the 6th High Performance Computing Conference; 2020; Ankara.

How to cite this article: Yilmazer-Metin A. sRSP: An efficient and scalable implementation of remote scope promotion. *Concurrency Computat Pract Exper*. 2022;34(9):e6483. doi: 10.1002/cpe.6483

APPENDIX A. COMPARISON OF SRSP WITH BRSP AND HLRC

A.1 sRPS versus bRSP

bRSP follows a simple broadcasting approach and applies heavyweight cache operations to all local caches when performing RSP. Below, we briefly review the details of bRSP implementation.

- *Implementation of remote acquire:* A remote acquire operation combines three suboperations: ① Last local-scoped release synchronization is promoted to global scope. To realize this, a cache-flush request is broadcast to and performed on *all* local data caches. A cache-flush operation on remote-modifier's local cache is also carried out. ② Once the flush-acknowledgments are collected from all local caches, the associated atomic operation (e.g., atomic LD) is performed at the target (i.e., global) scope. ③ Finally, cache-invalidation operations on remote-modifier's local cache and all other local caches are performed.

  In order to provide correct ordering of operations, no L1 caches must be allowed to serve any locally requested memory operations during any remote-acquire operations.
- *Implementation of remote release:* A remote release operation requires performing following three suboperations: ① First, a cache-flush operation is performed on remote-modifier's local cache to push all updates down to global scope. ② Next, the associated atomic operation (e.g., atomic ST) is performed at the global scope. ③ After the completion of atomic operation, cache-invalidation operations are performed on *all* local caches.
- *Implementation of remote acquire+release:* A remote acquire+release operation incorporates three suboperations: ① To promote the last acquire and/or release operation to global scope, a cache-flush request is broadcast to all L1 caches. All L1 caches perform cache-flush operations. A cache-flush operation is also performed on remote-modifier's local cache. ② Once the cache-flush acknowledgments are collected from all L1 caches, the paired atomic operation is performed at the target (i.e., global) scope. ③ In the end, a cache-invalidate request is broadcast to all local caches and cache-invalidate operations are performed on all local caches. A cache-invalidation operation on remote-modifier's local cache is also performed.

  In order to provide correct ordering of operations, no L1 caches must be allowed to serve any locally requested memory operations during any remote-acquire operations.

We summarize and compare the coherence actions of sRSP to bRSP in Table A1. With selective cache-flush and cache-invalidation instructions, sRSP implementation greatly reduces the remote synchronization overhead. When performing a remote-synchronization on a `N`-Compute-Unit GPU device with *bRSP* implementation, *N* L1D caches are affected from the synchronization. On the other hand, at most, only *two* L1D caches (the local modifier's cache and the remote modifier's cache) are affected from the remote-synchronization with sRSP implementation.

A.2 sRSP versus hLRC

With registration tracking, hLRC provides support for asymmetric synchronization without implementing scopes and RSP semantics and it also greatly reduces the synchronization overhead. We summarize the coherence actions in hLRC implementation in Table A2.

While hLRC provides a scalable solution for asymmetric synchronization it comes with several shortcomings. In hLRC, L2 cache is the central point for registration tracking and the registration transfer. This centralized approach makes the L2 cache a hot spot and makes the system wide coherence protocol very prone to deadlock problems.

To obtain good performances with hLRC, the programmer/compiler must be aware of the L2 cache line size to wisely allocate the synchronization variables such that no other variables (including data and other synchronization variables) map to the same cache line. That means any time a synchronization variable changes its location due to false sharing that will trigger heavyweight cache operations. Consequently, mapping more than one synchronization variable to the same cache line and data collocation will result in high synchronization overhead. Programmer or the compiler can use padding to avoid false sharing and data collocation but this could easily harm the effectiveness of L2 cache.

hLRC does registration tracking for all atomics. Without scopes, it performs registration transfer and heavyweight cache operations whenever a synchronization variable changes its location. Even though there is no global communication, there will be registration overhead if the atomic variables miss locally (this could be cold miss or coherence miss due to false sharing).

L2 must be inclusive of registration variables. Whenever a registered line is evicted from L2 cache that triggers unnecessary heavyweight cache operations. The replacement policy must be aware of registered lines, otherwise the performance of hLRC will suffer from high synchronization overhead. This could be a big problem especially for irregular applications with large working sets (e.g., graph applications).

TABLE A1 Coherence actions for implementing RSP semantics in bRSP and sRSP

Synchronization instruction	bRSP	sRSP
atomic_ld_acq_wg	(1) load from local L1D	*PA-TBL hit*
		(1) Load from L2
		(2) invalidate local L1D
		PA-TBL miss
		(1) load from local L1D
atomic_ld_acq_cmp	(1) load from L2	(1) Load from L2
	(2) invalidate local L1D	(2) invalidate local L1D
atomic_st_rel_wg	(1) store at local L1D	(1) store at local L1D
atomic_st_rel_cmp	(1) flush local L1D	(1) flush local L1D
	(2) store at local L2	(2) store at local L1D
atomic_rmw_ar_wg	(1) RMW at local L1D	*PA-TBL hit*
		(1) flush local L1D
		(2) RMW at L2
		(3) invalidate local L1D (*selective*-invalidation happens here)
		PA-TBL miss
		(1) RMW at local L1D
atomic_rmw_ar_cmp	(1) flush local L1D	(1) flush local L1D
	(2) RMW at L2	(2) RMW at L2
	(3) invalidate local L1D	(3) invalidate local L1D
atomic_ld_rem_acq_cmp	(1) flush and lock remote modifier's and all other *all* L1D caches	(1) *selectively* flush local modifier's L1D
	(2) load from L2	(2) load from L2
	(3) invalidate and unlock remote modifier's and *all* other L1D caches	(3) invalidate remote modifier's local L1D
atomic_st_rem_rel_cmp	(1) flush remote modifier's local L1D	(1) flush remote modifier's local L1D
	(2) store at L2	(2) store at L2
	(3) invalidate *all* L1D caches	
atomic_rmw_rem_ar_cmp	(1) flush and lock remote modifier's and *all* other L1D caches	(1) flush remote modifier's local L1D and *selectively* flush local modifier's L1D cache
	(2) RMW at L2	(2) rmw at L2
	(3) invalidate remote modifier's and *all* other L1D caches	(3) invalidate remote modifier's local L1D

Last but not least, the programming becomes very difficult with hLRC due to cache line aware data and synchronization variable mapping. The code becomes not portable and the cache protocol itself becomes very complicated and hard to verify with deadlock conditions.

While hLRC compares to the sRSP in terms of number of L1D caches that is affected by remote-synchronizations, it comes with some critical disadvantages as we explain in Section 3.2. Only advantage that hLRC has over sRSP is observed when a remote agent performs subsequent synchronizations (this is a remote-synchronization in RSP semantics) without any other intervening synchronizations (a local synchronization or another remote-synchronization in RSP semantics). Since hLRC transfers the registration when the remote-agent performs an atomic access, the subsequent atomic accesses will be local in hLRC implementation until the registration is transferred back to the local-modifier's cache. In such a case, sRSP can partially eliminate the remote-synchronization overhead (subsequent cache-flush /cache-invalidation operations on local-modifier's cache will not happen). As a comparison, we list the coherence actions of hLRC and sRSR in Table A3 for a list of important events that affect the performance and complexity.

TABLE A2 Coherence actions for implementing heterogeneous lazy release consistency

Synchronization instruction	Current location of registered atomic	Coherence actions
atomic_ld_acq_wg, atomic_ld_acq_cmp, atomic_st_rel_wg, atomic_st_rel_cmp, atomic_rmw_ar_wg, atomic_rmw_ar_cmp atomic_ld_rem_acq_cmp, atomic_st_rem_rel_cmp, atomic_rmw_rem_ar_mp	local L1D cache	(1) load\store\at local L1D cache
	L2 cache	(1) register at L2 cache and fill data from L2 cache
		(2) load\store\rmw at local L1D cache
		(3) invalidate local L1D cache
	remote L1D cache	(1) flush remote L1D cache
		(2) register at L2 cache and fill data from L2 cache
		(3) lload\store\rmw at local L1D cache
		(4) invalidate local L1D cache

TABLE A3 sRSP and hLRC coherence actions during a list of critical events

	hLRC		sRSP
L1 Eviction	(1) Flush local L1D cache		Evict the line regular way
	(2) Data from L1D cache to L2 cache		
	(3) UnRegister at L2 cache		
L2 Eviction	(1) Flush local L1		Evict the line regular way
	(2) Data from registered L1 to L2		
	(3) UnRegister @ L2		
Load/store to a colocated nonatomic data	load/store at L2 cache		Load/store the data regular way
Atomic access to a colocated atomic data	registration at local L1D	(1) Register&fill data from L2	Perform atomic access regular way
		(2) LD/ST/RMW at local L1D	
	registration at L2	(1) Register&fill data from L2	
		(2) LD/ST/RMW at local L1D	
		(3)invalidate local L1D	
	registration at remote L1D	(1) flush remote L1D	
		(2) Register&fill data from L2	
		(3) LD/ST/RMW at local L1D	
		(4) invalidate local L1D	

Abbreviation: hLRC, heterogeneous lazy release consistency.