Systems-on-Chip with Strong Ordering

SOORAJ PUTHOOR, University of Wisconsin—Madison, AMD Research
MIKKO H. LIPASTI, University of Wisconsin—Madison

Sequential consistency (SC) is the most intuitive memory consistency model and the easiest for programmers and hardware designers to reason about. However, the strict memory ordering restrictions imposed by SC make it less attractive from a performance standpoint. Additionally, prior high-performance SC implementations required complex hardware structures to support speculation and recovery.

In this article, we introduce the lockstep SC consistency model (LSC), a new memory model based on SC but carefully defined to accommodate the data parallel lockstep execution paradigm of GPUs. We also describe an efficient LSC implementation for an APU system-on-chip (SoC) and show that our implementation performs close to the baseline relaxed model. Evaluation of our implementation shows that the geometric mean performance cost for lockstep SC is just 0.76% for GPU execution and 6.11% for the entire APU SoC compared to a baseline with a weaker memory consistency model. Adoption of LSC in future APU and SoC designs will reduce the burden on programmers trying to write correct parallel programs, while also simplifying the implementation and verification of systems with heterogeneous processing elements and complex memory hierarchies.[1]

CCS Concepts: • **Computer systems organization** → *Single instruction, multiple data*; *Heterogeneous (hybrid) systems*;

Additional Key Words and Phrases: Consistency model, GPU, lockstep execution

ACM Reference format:
Sooraj Puthoor and Mikko H. Lipasti. 2021. Systems-on-Chip with Strong Ordering. *ACM Trans. Archit. Code Optim.* 18, 1, Article 15 (January 2021), 27 pages.
https://doi.org/10.1145/3428153

1 INTRODUCTION

Memory consistency models define the order in which loads and stores should be visible to the memory. A strong consistency model like sequential consistency (SC) [3, 48, 75] tightly couples this memory order to the program order of loads and stores whereas weaker consistency models relax this program order constraint. As such, weaker consistency models provide better

[1]New paper, not an extension of a conference paper.

This work was supported in part by NSF award CCF-1628384, CCF-1813434, and CCF-2010830, and AFRL award FA9550-18-1-0166.
Authors' addresses: S. Puthoor, University of Wisconsin—Madison, Advanced Micro Devices, 7171 Southwest Pkwy, Austin, TX 78735; email: puthoor@wisc.edu; M. H. Lipasti, University of Wisconsin Madison, Electrical and Computer Engineering, 1415 Engineering Dr, Madison, WI 53706; email: mikko@engr.wisc.edu.

1544-3566/2021/01-ART15
https://doi.org/10.1145/3428153

opportunity for reordering memory operations and enable higher memory level parallelism, leading to better performance. This has resulted in both CPUs and GPUs adopting weaker models. For example, many modern CPUs adhere to the total store order (TSO) memory model that allows younger loads to bypass pending stores [68, 76], and many commercially available GPUs and APUs adhere to the release consistency (RC) or heterogeneous-race-free (HRF) models that require explicit synchronization operations to communicate between threads [24, 33, 38, 47].

However, there is a plethora of work showing weak models are notoriously difficult to program and are extremely challenging to verify [8, 9, 21, 25–27, 45, 69, 80, 81]. The reordering permitted by the weak models results in a non-intuitive sequence of memory operations, often resulting in programming bugs that are hard to detect. Additionally, there has been a series of prior work attempting to formalize the weak models of the POWER and ARM architectures, only to get it wrong in their initial attempts [10, 20, 32, 56, 62, 67]. Alglave et al. observed that some implementations of weak models violate programming guides and vendor documentation guarantees [7].

The complexity with weak models is even higher for APU programming where a programmer is forced to reason about two different memory models of the CPU and GPU. This is further amplified with APU/GPU programming moving toward single source languages like C++ AMP [58] and HC [14]; forcing a programmer to target different memory models with the same language constructs. One prevalent memory model for APUs, HRF memory model of Heterogeneous System Architecture (HSA) [46], requires a programmer to (a) specify the scope of communication and (b) explicitly initiate the communication. That means HRF goes one step further than RC, demanding the programmer for explicitly managing the communication scope whereas RC only demands for explicit synchronization operations. While it is true that most of these complexities can be abstracted away with synchronization routines, the low level software developer who writes these synchronization routines will still have to deal with these intricacies. A strong model like SC neither asks the programmer for visibility scopes nor demand the programmer to insert explicit synchronization primitives.

Despite the clear programmability advantages with strong models, the major impediment for their adoption has been their perceived performance and area overheads. Researchers have proposed various techniques to implement SC on CPU cores without significant performance penalty [22, 28, 29, 39–41, 51, 52, 64, 72, 78]. Recent academic research has also shown that enforcing SC on a GPU has minimum hardware complexity and SC can match the performance of weaker models [44, 65, 71]. For example, Hechtman et al. [44] studied the characteristics of heavily threaded applications and demonstrated that a simple SC implementation achieves performance comparable to weaker models.

In this article, we ask the question: can a strong model be implemented on an APU SoC with low overhead? While answering this question, we make the observation that a well-defined and formal strong model does not exist for lockstep executing threads in a GPU. So, we first define and formalize lockstep SC (LSC), a consistency model with strict ordering requirements for lockstep executing threads. Then, we implement LSC across the entire APU SoC with low performance and area overhead. The contributions of this article are as follows:

- We define and formalize the LSC memory model.
- We efficiently implement LSC on an APU SoC. The performance cost of our implementation is just 0.76% for the GPU execution phase and 6.11% for the entire SoC. Our implementation only adds simple direct mapped hardware components and requires minimum changes to the baseline. As such, our implementation does not require expensive associative look ups, does not need compiler support and does not interfere with the underlying cache coherence implementation.

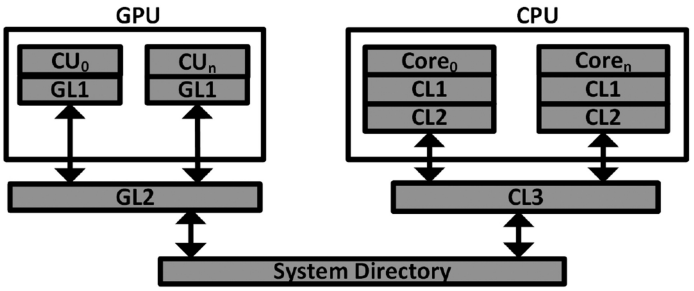


Fig. 1. Baseline APU system.

- We show that a strong model performs better than the baseline RC model for GPU benchmarks with fine-grained data sharing.

2 BACKGROUND

2.1 Baseline APU System

Figure 1 shows our baseline APU system with integrated CPU and GPU. Each GPU core is called a compute unit (CU) and hosts four 16-wide vector ALU units (VALUs) and one 16-wide address generation unit (AGU). VALUs execute ALU instructions and AGU executes memory instructions. Each CU can have up to 40 wavefronts in flight (10 per VALU unit) and a 64-wide wavefront takes 4 cycles to execute on 16-wide VALU or AGU unit. This GPU core is similar to the Graphics Core Next architecture [16] of many commercially available GPUs and APUs [18, 24, 47]. The dynamically scheduled out-of-order (OoO) CPU core has a split load/store queue design with separate load queue and store queue per thread. The reorder buffer (ROB) commits instructions in program order and the committed stores are taken out of store queue and moved into a store buffer. As such, the CPU core adheres to TSO memory model. The core supports speculation and relies on invalidation messages forwarded from the cache subsystem for detecting and correcting mis-speculated memory accesses. When a cacheline gets invalidated, the invalidation notification is forwarded to the core and the core replays any speculative load that has read that cacheline. For the same reason, an eviction notification is also forwarded to the core. This technique proposed by Gharachorloo et al. [37] allows the core to do aggressive memory access speculation.

The CUs and CPU cores communicate over a single shared address space with the help of coherent caches. The GPU L2 cache (GL2) is shared by all CUs but the L1s (GL1) are private to a CU. CPU cores have private L1 and L2 caches (CL1 and CL2) and a shared L3 (CL3). The private CL2 is inclusive of CL1 but shared caches in both CPU and GPU are non-inclusive. To reduce the number of requests issued to the memory hierarchy, our baseline GPU coalesces memory requests from a wavefront to the same cacheline and only these coalesced requests are issued to the GL1 [16, 50].

The system directory acts as the ordering point between CPU and GPU coherence requests. It also hosts an inclusive probe filter that keeps a 2-bit sharer vector indicating if a cache block is in CPU caches or in GPU L2 or both. The probe filter also keeps track of the cache in ownership states [75]. CPU caches are write-back and implement a read for ownership (RFO) protocol. This RFO protocol preserves the single writer invariant of cache coherence protocol [75]. The GPU caches are write-through and implement a simple valid-invalid coherence protocol. In our baseline implementation, invalidation messages generated by a write request are not forwarded to the GL1. The GL1 caches are made coherent by synchronization operations defined in the HRF memory model [46]. Consequently, the probe filter is inclusive of only GL2 cachelines. This baseline APU system is similar to many commercially available APUs [24, 47] and is used as baseline in many recent APU publications [11, 43, 61].

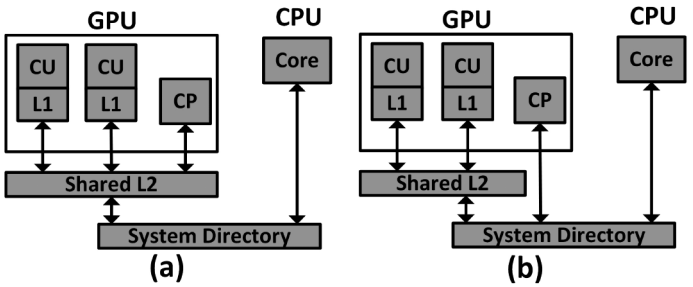


Fig. 2. Command processor hooked up to the GPU cache hierarchy.

2.2 HRF Synchronization Operations

The baseline GPU adheres to HRF. HRF requires explicit synchronization instructions such as load-acquire (ldAcq) and store-release (stRel) to communicate between threads [38, 43, 46]. A formal definition of these synchronization operations can be found elsewhere [38, 46, 75].

GPU caches enforce HRF by invalidating the valid cachelines on a ldAcq operation and write back all outstanding writes on a stRel operation [11, 46]. However, since our baseline GPU caches are write-through, a stRel operation only needs to wait for all write-through completion acknowledgements (cmplAcks) from directory. While a stRel waits for all cmplAcks, the GPU pipeline need not wait for the cmplAck of individual stores, because HRF enforces memory fences only at synchronization operations. Hence, from a GPU pipeline perspective, a store is completed when GL1 sees that store and responds with an acknowledgement. A ldAcq operation invalidates all valid cachelines in the GL1 cache, which can be achieved by resetting their valid bits. A subsequent load request to the GL1 after a ldAcq will find the cache in invalid state and will forward that load to the GL2 for the most recent copy of the data.

GL2s are kept coherent by invalidation messages from system directory. Since GL1s are made coherent with ldAcq operation, the GL2 need not forward invalidation messages from the directory to these GL1s. GL2s perform non-silent evictions to keep it inclusive with probe filter but evictions from GL1s are silent. HSA [33] has adopted this model for HSA compliant GPUs. Several commercially available GPUs also adhere to this memory model [24, 47].

2.3 Limitations of HRF

HRF is a well-defined memory model that allows work-items (threads in CUDA) to synchronize through scopes [46]. A detailed discussion of scoped synchronization can be found elsewhere [46]. HRF defines scopes in terms of the execution hierarchy of GPUs. For example, work-items within the same work-group (threadblock) synchronize through work-group scope, and work-items from different work-groups synchronize through device scope (scopes are present in other models as well [54]). While use of such scopes are well defined for synchronizing between work-items of a GPU, the synchronization between work-items and threads running on other processing elements on the same GPU is not clearly defined. For example, the Graphics Core Next (GCN) GPUs have a command processor (CP) to launch work on the GPU [17] and since CP threads are not part of the GPU execution hierarchy, the synchronization scope between them and work-items are not well defined by HRF.

Figure 2(a) shows the CP and CUs sharing the same shared L2 in a GCN GPU [17]. Recent studies have suggested using the CP for GPU initiated networking [49] and for enabling fine-grained task scheduling [60]. Enabling both these capabilities require CP threads and work-items to synchronize frequently. In the absence of a clearly defined scope for such synchronization in the HRF,

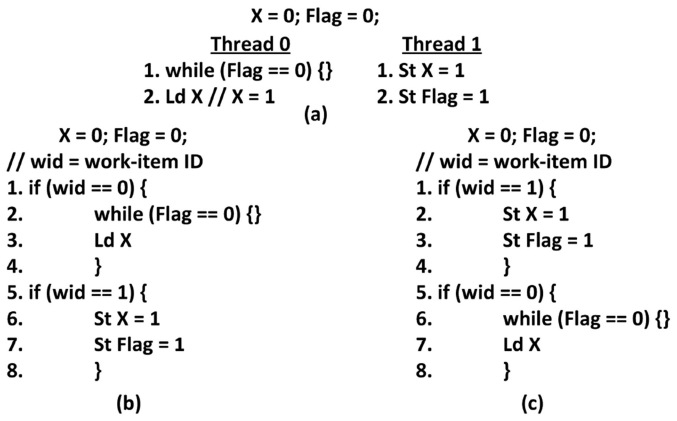


Fig. 3. Program order.

in practice, programmers choose the level of the memory hierarchy where the synchronization should occur as the scope. In this case, the shared L2 is the scope of the synchronization, which is the same as device scoped synchronization. This scenario thus exposes HRF's drawback of forcing a programmer to think about the underlying cache hierarchy design while writing synchronizing programs, an observation made by Sinclair et al. in their work [70] as well.

While the above-mentioned scenario is a programmability drawback of HRF, the implications go far beyond that. Figure 2(b) shows an alternative implementation of hooking the CP into the cache hierarchy of an APU. In this scenario, the CP is hooked to the system directory. Hence, synchronization between CP threads and work-items through device scope is no longer possible. This scenario exposes a portability issue with HRF. While HRF can still guarantee synchronization with system scope in this scenario, the synchronization code has to be manually modified based on this new cache hierarchy.

The *3p* criteria—programmability, performance, and portability—was introduced by Adve [2] to evaluate memory models. As discussed above, HRF has limitations with both programmability and portability. We also show that fine-grained data sharing applications perform poorly with HRF in the result section.

3 LOCKSTEP SC

In this section, we first argue that the program order should be defined for a wavefront in a GPU. Then we formally define lockstep SC. Lockstep SC allows SC's ordering rules to be enforced on a group of threads executing in lockstep.

3.1 GPU Program Order

In a CPU, the program order (PO) is defined for individual threads. However, since a GPU executes work-items of a wavefront in lockstep fashion, it is unclear if the PO should be defined for individual work-items or for the lockstep-executing wavefront. To understand the subtle difference, consider the synchronizing code in Figure 3(a). Here, *thread 0* and *thread 1* have their own PO and they synchronize with the *Flag* synchronization variable enabling the load of *thread 0* to see the value updated by *thread 1*. However, if the same synchronization is attempted between two work-items of the same wavefront, the assumption of them having independent PO can lead a programmer to write the code in Figure 3(b). Since the threads of a wavefront execute in lockstep, the *thread 0* will spin on the *Flag* variable without reaching the release operation by *thread 1* resulting

in a deadlock in current commercially available GPU architectures. The reason for this deadlock is the incorrect assumption of independent PO for work-items within a wavefront. Independent PO based reasoning does not work with lockstep execution. Additionally, the same code may avoid deadlock on some GPU architectures with dynamic wavefront creation capability proposed in the literature [34]. This capability enables some work-items to execute a different basic block than its peers. This may sometimes avoid deadlock, which makes it even harder to find bugs and reason about correctness.

Furthermore, modern GPUs have scalar cores that execute scalar instructions of a wavefront [17, 19, 42]. The scalar instructions perform operations common to a wavefront, such as handling control flow. These scalar instructions are inserted by the compiler and their correct execution relies on a wavefront PO. It is also important to note that the scalar instructions cannot be even defined within the scope of a work-item PO.

Because of these reasons, we argue that the PO should be defined for a wavefront in a GPU. Figure 3(c) shows the code rewritten with the assumption of a per-wavefront PO. This code will not deadlock on the current architectures and the synchronization mechanism is easy to reason about.

ElTantawy and Aamodt also discussed the possibility of deadlock under branch divergence because of the diverged threads sharing the same program order [31]. While their work attempted to mitigate this deadlock by inspecting the application control flow graph, we demonstrated this deadlock to establish that GPUs have a wavefront program order. The lockstep SC model introduced in next section will define ordering rules for wavefront program order.

3.2 Lockstep SC

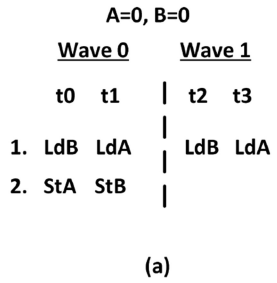
SC ordering rules are defined for individual threads [75]. But GPUs execute work-items in lockstep and ordering memory operations from a lockstep executed instruction is unnecessary. Toward this, we propose LSC that allows threads executing in lockstep in a wavefront to share the SC ordering rules.

For defining LSC, we use the same naming convention used in the Primer [75]: op1 $<_p$ op2 implies that op1 precedes op2 in that core's PO and op1 $<_m$ op2 implies that op1 precedes op2 in global memory order. To formalize LSC, in addition to $<_p$ and $<_m$, we define $=_p$ and $<_m>$. op1 $=_p$ op2 implies that op1 and op2 are logically executed simultaneously in the program order and op1 $<_m>$ op2 implies op1 and op2 are unordered in global memory order. MOp(X) denotes a memory operation that reads and/or modify the state of address X in memory. Any execution is LSC if it adheres to the following conditions:

- If MOp(a) $<_p$ MOp(b) $\Rightarrow$ MOp(a) $<_m$ Mop(b); $a == b$ or $a \neq b$
- If MOp(a) $=_p$ MOp(b) $\Rightarrow$ MOp(a) $<_m>$ Mop(b); $a == b$ or $a \neq b$
- Ld(a) = Value of MAX_m {St(a) | St(a) $<_m$ Ld(a)}

The first condition states that all older memory operations to same or different addresses ($a == b$ or $a \neq b$) in PO should be made visible to the memory before any younger memory operation(s). The third condition specifies that a load to a memory location a should see the value of the last store in memory order to that location. These two conditions are same as SC but LSC relaxes the ordering constraints on operations that are logically executed simultaneously by adding another condition (second condition). As such, memory requests originating from the same wavefront instruction and logically executed simultaneously need not be ordered in LSC.

LSC is a seemingly straightforward extension of SC. But there is clearly value in formalizing the model to prevent undefined behaviors, and the model is useful as it makes a strong memory model practical in GPUs. We demonstrate that a seemingly intuitive idea of *extending SC to GPUs* is really



(a)

models/ outcomes	(t0,t1)	(t2,t3)
SC	(A=0,B=0) (A=1,B=0) (A=0,B=1)	(B=0,A=0) (B=1,A=0) (B=0,A=1) (B=1,A=1)
LSC	(A=0,B=0)	(B=0,A=0) (B=1,A=0) (B=0,A=1) (B=1,A=1)
SLSC	(A=0,B=0)	(B=0,A=0) (B=1,A=1)

(b)

Fig. 4. (a) Program order of wavefront 0 and wavefront 1. (b) The valid outcomes for different models.

subtle and can result in many complex, non-intuitive behaviors. To explain this, we introduce a strictly lockstep SC (SLSC) model by changing the second condition added by LSC to have a strict ordering among lockstep memory requests as follows:

- If MOp(a) $=_p$ MOp(b) $\Rightarrow$ MOp(a) $=_m$ Mop(b); $a == b$ or $a \neq b$

With this change, we have three candidate models for SC on GPUs: (a) SC that treats each GPU thread as a separate independent entity, (b) LSC that completely relaxes the memory ordering among lockstep requests, and (c) SLSC that requires lockstep requests to be visible to memory atomically. To understand the subtleties, let us consider the execution of two wavefronts shown in Figure 4(a). Threads t0 and t1 belong to wavefront 0 and threads t2 and t3 belong to wavefront 1. Figure 4(b) shows the valid outcomes of SC, LSC, and SLSC for the PO shown in Figure 4(a). Since SC treats GPU threads as independent entities, SC allows the outcomes (0,0), (1,0), and (0,1) for threads t0 and t1. However, LSC and SLSC only allow (0,0) outcome, because they enforce ordering among lockstep executing instructions. Both SC and LSC have the same valid outcomes for threads t2 and t3. However, SLSC presents the stores from a lockstep execution as an atomic update to the memory and hence restricts the valid outcome to just (0,0) or (1,1).

We highlight few non-intuitive behaviors from the previous example. Although LSC relaxed the ordering constraints for lockstep executing instructions, the valid outcomes are fewer with LSC than SC. LSC assigns program order to an entire wavefront as opposed to individual work-items and with this wavefront PO, MOp(a) $<_p$ MOp(b) means that the memory operations from all lockstep executing threads are ordered by the PO thus reducing the number of valid outcomes. Additionally, naively extending SC to GPUs with the notion that *a wavefront in a GPU is equivalent to a thread in a CPU* will result in SLSC. Clearly, all these subtleties cannot be captured without a formal definition. In fact, the phrase *extending SC to GPUs* is too vague for one to even understand which among the three models (and possibly others not shown here) is actually implemented.

Figure 4(b) shows that SC does not provide a simple intuitive ordering for threads within a wavefront. Although SLSC provides a stricter global ordering both within and across wavefronts, implementing SLSC requires all wavefront memory accesses to complete atomically at the memory (or appear to complete atomically at the memory to an outside observer). This is prohibitively expensive. For example, if a lockstep execution is writing to multiple cachelines, then the individual writes cannot be made visible until all writes are completed. Such an implementation will require locking the modified cachelines to prevent an individual write from being visible to other cores until all writes from that lockstep execution are completed. A global ordering can also be achieved

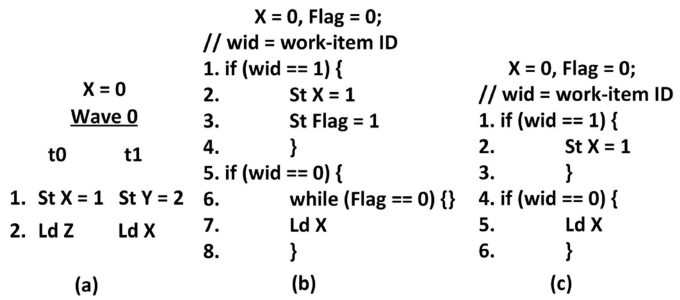


Fig. 5. LSC and synchronization flag.

by forcing instructions executing in lockstep be consecutive within the per-wavefront PO and the same order to be exposed to all threads in the system creating a total order. This is more difficult to achieve than even SLSC. For example, this will require the lockstep executing operations to complete in some order, that is in some serial order (if grouped together, then we will have to make the grouped operations appear atomically, which is SLSC), which in turn will force serialization of lockstep memory operations, taking away all the benefits of lockstep execution model of GPU. Hence, we chose LSC for our implementation. LSC is simple for programmers to reason about, and is agnostic to hardware or memory hierarchy organization unlike HRF. This makes an LSC program more portable than HRF and relieves a programmer from knowing the details of memory hierarchy organization of the targeted system.

3.3 Lockstep SC and SC

Flag Synchronization: Figure 5(a) shows two threads of the same wavefront executing a store followed by a load. Thread t1 is trying to read (Ld X) the store from thread t0 (St X). Since SC treats these two threads as independent entities, the Ld X is not guaranteed to see value 1. However, with LSC, the ordering constraints are applied across all threads in a wavefront and hence Ld X from t1 is guaranteed to see value 1. Figure 5(b) shows the code we discussed earlier in Section 3.1 (Figure 3(c)). In the absence of LSC, the code needed a synchronization variable *Flag* to synchronize between two threads of the same wavefront. However, with LSC, the same code can be rewritten without any synchronization variable as shown in Figure 5(c). Since LSC defines program order to an entire wavefront and consequently its ordering rules are applied to all threads executing in lockstep, LSC provides synchronization without explicit synchronization flags for threads in the same wavefront.

4 IMPLICATIONS

While SC defines the ordering rules for an individual CPU thread, LSC treats an entire wavefront as a single entity and defines the ordering rules for the entire wavefront. Although this difference may seem trivial, it has both programming and hardware design implications. The programming implications are further divided to programming implications for the application developer and for the low-level system software developer.

4.1 Implications for Application Development

We encourage the application developer to rely on underlying low-level software implementation to abstract out hardware specific details. For example, AMD GCN cross-lane operations [53] and CUDA warp-level intrinsics [66] expose hardware details like wavefront (or warp) size to the programmers. While an application programmer can write code with these intrinsics, those

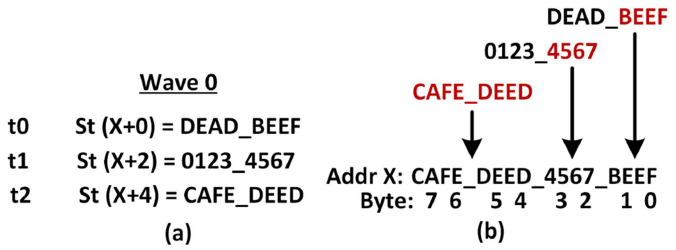


Fig. 6. Coalescer adhering to SC.

applications will not be portable across platforms with a different assumption about the wavefront size. However, with the low-level software abstracting out hardware details, the application writer can develop code without burdening themselves with the hardware specifics. With this envisioned model, synchronization primitives are also provided by the low-level software and hence the application developers need not concern themselves with the underlying consistency model. Thus LSC has limited impact on the application developer.

4.2 Implications for Low-level Software Development

Low-level software acts as the interface between application and hardware by providing hardware independent API calls and converting them to hardware dependent operations. For example, this layer can be used to implement common algorithms like shuffle or reduction using hardware dependent CUDA warp-level intrinsics [53] or AMD GCN cross-lane operations [66]. This layer also provides synchronization APIs that abstract out the behavior of the underlying hardware consistency model. As such, LSC formalization impacts the programmability of this layer of software. For example, if synchronization is needed between two threads, with LSC, this layer can take advantage of the implicit synchronization flag if these threads are from the same wavefront (discussed earlier in Section 3.3). Naturally, this implementation needs to know the wavefront size of the underlying hardware to provide the correct implementation.

4.3 Hardware Design Implications

A GPU coalescer coalesces accesses to the same cacheline from a wavefront instruction into a single access [63]. Suppose a wavefront instruction is generating multiple stores but to the same address and the coalescer is coalescing these requests. If the stores are storing different values, then the coalescer will have to select one store over the other to write to that address. The selection logic will be extremely complex for writes when one store partially overlaps with another. Since SC treats these threads as independent threads, a coalescer design adhering to SC will be forced to mimic an interleaving such that these stores are issued in some sequential order to the memory subsystem.

Figure 6(a) shows an example in which three partially overlapping four-byte stores from the same wavefront instruction are issued by threads t0, t1 and t2 to byte offsets 0, 2, and 4, respectively. Figure 6(b) shows a valid SC interleaving in which the thread with a higher thread ID is treated as the younger store and thus overwrite the bytes written by older stores. From that figure, it can be seen that the final value comprises of bytes from all three stores. That means a coalescer adhering to SC will have to be implemented in such a way that there should be selection logic for each byte that is written. Prior work has shown that even address comparison to identify coalescable accesses is resource intensive [63]. The byte selection logic to determine the correct store values imposed

by SC significantly adds to this overhead. LSC, however, allows these accesses to be unordered and thus does not impose such strict byte selection restriction on the coalescer design.

Irrespective of the actual order, assigning some order to the visibility of lockstep accesses in the memory will complicate the coalescer design. LSC resolves this issue by making these accesses unordered ($<_m>$ memory order), thus facilitating a simple coalescer design. Specifically, if there are two or more stores to overlapping locations from a lockstep execution, a determination as to which one was the last store to that location is not possible and hence the value of the subsequent load to that location is undefined. This is similar to the reasoning provided in prior works where the outcome of racey accesses that "occur at the same time" are undefined [5, 23, 36, 46, 55]. As a result, LSC does not impose any restriction on the value of overlapping stores from a lockstep execution. Thus LSC facilitates simple coalescing logic. Additionally, since LSC implements a strong model, the coalesced requests from the same wavefront are to be issued in program order to the memory hierarchy and hence coalescing requests across wavefront instructions is not allowed. However, prior work has shown that coalescing across wavefront instructions is incredibly expensive and even existing designs do not implement it [63].

5 IMPLEMENTATION CONSIDERATIONS

LSC implementation across an entire APU SoC involves implementing strong ordering between dynamically scheduled OoO CPU cores and simple in-order but lockstep executing GPU cores. Since these diverse processing elements represent two ends of the processor architecture spectrum, arriving at a solution that works for both is challenging. Many existing solutions to support strong ordering rely on the speculation capability of CPU cores [39, 40, 64] or on the underlying cache coherency mechanism [4, 65, 73, 77]. However, the GPU cores lack speculation support making them unimplementable on a GPU, and the CPUs and GPUs have significantly different cache hierarchies making the coherency based methods unattractive as a common solution [11, 33, 43, 61, 74].

Lin et al. [52] made the observation that strong ordering can be enforced by preventing reordering of conflicting accesses. In this article, we leverage this observation for implementing LSC. For preventing the reordering of conflicting access, we leverage the mutex acquisition mechanism proposed by Gope and Lipasti [41]. This mutex mechanism does not require speculation support and does not interfere with the underlying coherence mechanism. Hence, this mechanism can be easily adopted to implement strong ordering on both CPUs and GPUs.

In this section, we first discuss the implementation of strong ordering with mutex mechanism proposed by Gope and Lipasti [41]. Then, we provide a discussion of our proposed mechanisms to implement strong ordering using mutexes on both dynamically scheduled OoO cores and GPU cores.

Strong Ordering with Mutexes: Gope and Lipasti [41] proposed atomic SC (ASC) for implementing sequential consistency on a simple in-order multicore system. ASC allows reordering of memory operations but protects the reordered requests with mutexes making the reordering invisible to other cores. Figure 7 provides a simple demonstration of ASC enforcing SC with reordered memory operations. Figure 7(a) shows the program order of memory operations. We assume the stores missed in the cache and loads hit in the cache. Under these assumptions, if cores C0 and C1 allow reordering of younger loads, the loads bypassing the store will create a memory order as shown in Figure 7(b). The outcome (0,0) is forbidden in SC and violates SC.

ASC avoids SC violations by allowing the reordering of operations *iff* the pending operations have acquired a mutex. So, in the previous example, the loads are allowed to bypass stores *iff* both the older store and younger load has acquired a mutex for the addresses they are trying to access. Forcing the reordered instructions to acquire mutexes create a mutex ordering (<Mu) at the mutex

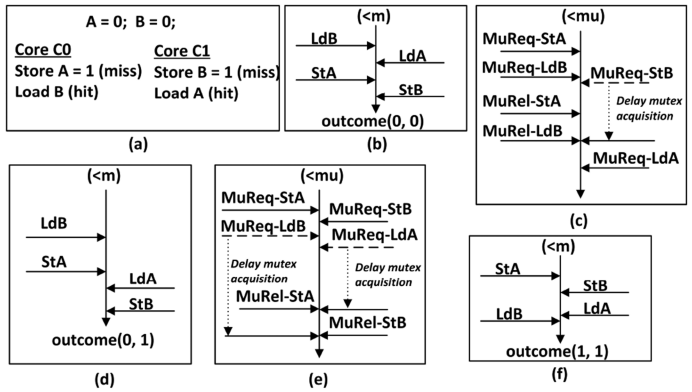


Fig. 7. (a) Program order of memory operations. (b) Memory order of memory operations. (c) Mutex ordering when LdB beats StB. (d) Memory order with ASC for LdB beats StB. (e) Mutex ordering when StB beats LdB. (f) Memory order with ASC for StB beats LdB.

pool as shown in Figure 7(c). In Figure 7(c), the mutex request from C0 to address B (MuReq-LdB) beats the mutex request from C1 to the same location (MuReq-StB) forcing MuReq-StB to acquire the mutex only after LdB releases the mutex (MuRel-LdB). Since C0 has acquired mutexes for both the pending store and the younger load under the shadow of that store miss, C0 can now reorder memory operations. Acquired mutexes are released only after the older store (StA) from the same core is retired (StA). Thus, MuRel-LdB is a guarantee that Store A has retired and the load A (load A from C1 will not be a cache hit, because StA will invalidate all stale copies) from C1 will see that store. The memory order forced by ASC is shown in the Figure 7(d) and the outcome (0, 1) is SC.

Figure 7(e) shows a scenario in which MuReq-StB beats MuReq-LdB to the mutex pool and the MuReq-LdB is delayed at the mutex pool. In this situation, since LdB has not acquired its mutex, it cannot bypass StA creating a memory order given in Figure 7(f). The outcome(1,1) from this execution is still SC. Since ASC allows reordering iff all the participating operations have acquired a mutex, the reordering is not visible to other cores, because all the reordered instructions become visible as if they have executed *atomically*. A formal proof showing that ASC enforces SC can be found elsewhere [41].

Extending Mutex Mechanism to OoO Cores: Dynamically scheduled OoO cores buffer stores that are completed (committed) but waiting for retirement in store buffers. From the memory system perspective, a store is not completed until it is retired. So, for implementing strong ordering with the mutex mechanism, all stores that are waiting in the store buffer and all younger memory operations that are under the shadow of these stores should acquire mutexes before allowing them to be reordered. However, OoO cores issue memory operations out of order to the memory system. But since those operations are considered speculative until they are committed in program order by ROB, enforcing the mutex acquisition requirement at the time of issuing a speculative memory operation is not necessary. The mutex acquisition requirement need to be enforced only for allowing a memory operation to be moved from speculative to non-speculative state, that is at the instruction commit.

As such, when a memory operation reaches the ROB head and there are pending stores in the store buffer, that instruction is allowed to commit only when that instruction and all the pending stores in the store buffer have acquired a mutex. Since mutexes are released only after the store miss is serviced, just verifying that the youngest store in the store buffer has its mutex guarantees that all pending stores in the store buffer have acquired mutexes.

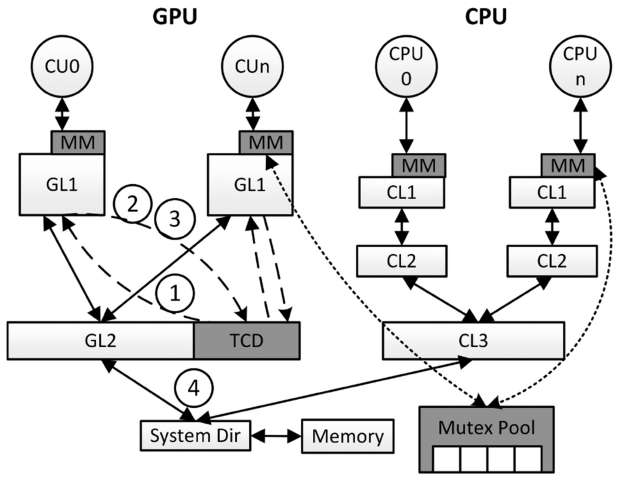


Fig. 8. APU system with changes to support LSC. The blocks added to the baseline are shown in gray. The dotted lines are the extra messages needed for LSC.

Extending Mutex Mechanism to GPU: LSC does not enforce ordering among the requests generated by the same wavefront instruction. With this ordering relaxation, extending mutex mechanism to GPU is straightforward. When there are pending stores from a wavefront, those pending store requests and all younger memory requests from that wavefront should acquire mutexes before issuing these younger requests to the memory subsystem. However, since all modern GPUs coalesce requests to the same cacheline, the mutex requirement is often drastically reduced. In the best case, when all requests can be coalesced to the same cacheline, just one mutex can satisfy this requirement for all memory requests from the same wavefront instruction.

6 IMPLEMENTATION

6.1 Changes to GPU Memory System

The baseline GL1 is made coherent with HRF synchronization operations. Since LSC does not have these synchronization operations, GL2 forwards the invalidation messages from the directory to GL1 to make it coherent ① as shown in Figure 8. To avoid broadcasting these invalidation messages to all GL1s in the GPU, the GL2 keeps track of the sharers with a tagless coherence directory (TCD) [79]. TCD is not required for LSC (any conventional directory will work), but it is an appealing solution for scalability and other reasons [79].

Non-silent Evictions: Since TCD is tracking GL1 sharers, eviction notification should be sent to TCD from GL1s. However, since TCD keep sharer information as hashes stored in sharing vectors, the evicted address alone is not sufficient to reset the appropriate bits in the sharing vector. After each eviction, the hash functions for all remaining blocks in the set are calculated in a typical TCD [79]. This recalculation is relatively simple if the GL2 is inclusive, because an inclusive GL2 can track the entries of a sharer from which the hash can be recalculated [79]. However, since GL2 in our implementation is non-inclusive, the GL1 on evicting a block calculates the hash functions for all remaining blocks in that set and sends the recalculated sharing vectors with the eviction messages. The recalculation of the hash requires reading the entire tag array of a set after an eviction and generating a 320 bit (five 64-bit sharing vectors) eviction message. Since we read the entire tag array of a cache set as part of normal cache operation before evicting a cache line, the tag array read to recalculate hash can be amortized. However, sending 320 bits per eviction message is

expensive. So, to reduce the number of eviction messages transmitted between GL1s and GL2, we only send the recalculated hash once every N evictions from that set. A higher value of N increases the false positives, whereas a lower value of N will result in high eviction message bandwidth. To count the number of evictions since the last hash recalculation, a $\log_2$N-bit counter is added per set to the GL1. This counter is incremented each time a cacheline is evicted from that set and the Nth eviction will generate an eviction message with the recalculated hash. The GL2 on receiving this recalculated hash will update the corresponding TCD entry.

Figure 8 illustrates the operation of this GPU memory system implementation. On receiving a write-through request from GL1 or an invalidation request from SD, GL2 consults the TCD to see if there are any sharers for that cacheline and forwards invalidation message to the sharer GL1s ①. The GL1s invalidate the data and respond to the invalidation message with an invalidation ack ②. In the event of a false positive by TCD, the GL1 will receive the invalidation request in I state but it still responds with an ack. By making the GL1s coherent with invalidation messages, the entire APU system now preserves the single-writer invariant. Finally, when GL1 evicts a block, the eviction message is forwarded to the GL2 ③. If the eviction resulted in removing that block from all GL1s, then the eviction message is forwarded to the SD ④ and the probe filter in SD is made inclusive of all caches in the APU.

6.2 Changes to CPU Pipeline

LSC requires minimal changes to the CPU pipeline. On a store miss, the CPU mutex manager (CMM) notifies the ROB about this store miss. A memory instruction under the shadow of a store miss is allowed to commit only after (a) the missed store has acquired a mutex and (b) the committing memory instruction has acquired a mutex. So, ROB commits an instruction only after all pending stores in the store buffer and the memory instruction at the head of ROB have acquired a mutex. Since mutexes are released only after a pending store is retired, ROB has to only check if the youngest store in the store buffer and the committing instruction have acquired mutexes.

If the store buffer has pending stores, then the ROB issues a mutex request for the memory instruction at the ROB head. The CMM forwards the mutex acquisition ack to ROB. On receiving a mutex ack, the ROB commits that memory instruction if the youngest pending store and the head instruction has acquired mutexes. With this implementation, a younger store under the shadow of a store miss is moved into the store buffer only after acquiring a mutex. The changes to the ROB are trivial. A 2-bit flag to indicate waiting on mutex ack (one bit for the head instruction and one bit for the store) and a check to see if there are pending stores in the store buffer. Next, we discuss the implementation of CMM.

6.3 CPU Mutex Manager

The CMM is responsible for requesting mutexes, storing the acquired mutexes and releasing mutexes. Figure 9(a) shows the components inside a CMM. CMM has a mutex vector to keep track of the requested and acquired mutexes. This mutex vector is indexed by a hash function generated from the cacheline address of memory requests. In our implementation, we use $\log_2$ (mutex pool size) lower bits of the cacheline address as the hash function. Each entry in the mutex vector has 2-bits indicating the status of that mutex. The status of a mutex can be *mutex-requested*, *mutex-acquired* or *mutex-invalid*. A *mutex-requested* state indicates that a request to that mutex is pending. A *mutex-acquired* indicates that this core has exclusive lock on that mutex and *mutex-invalid* is the default state. Each mutex vector entry also has a request count field that keeps track of the number of pending mutex requests to that mutex.

The operation of CMM is shown in Figure 10. The store buffer issues a store to cache ① and the cache notifies if the store is a miss. If the store is a miss and there are no pending store misses,

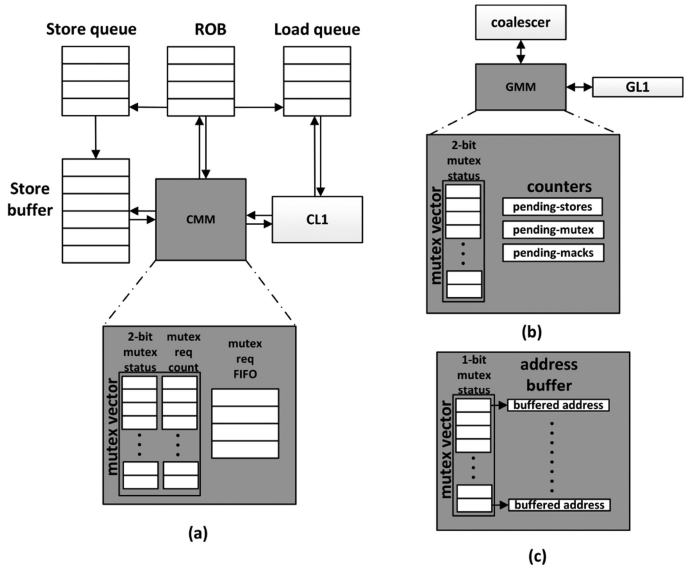


Fig. 9. (a) CMM interacting with other CPU core components. (b) GMM interacting with CPU components. (c) Mutex pool with mutex vector and address buffers.

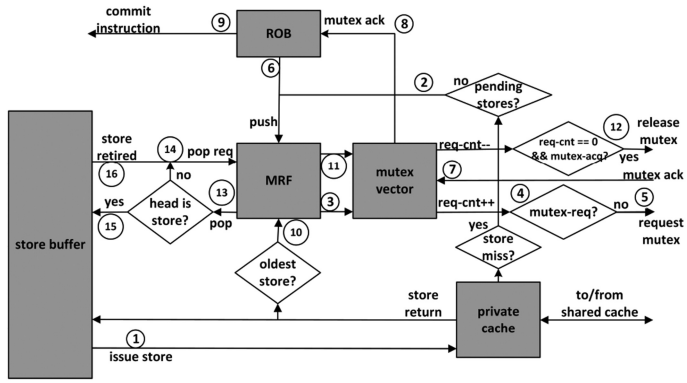


Fig. 10. CMM operation.

then that store is added to the Mutex Request FIFO (MRF) ② and the MRF issues a mutex request to mutex vector ③. The Mutex vector indexes into the mutex entry corresponding to the store address and issues a mutex request to the mutex pool (mutex pool is described in detail in Section 6.5) if the status of that mutex is *mutex-invalid* ④⑤. After issuing the request, this status is changed to *mutex-requested* and the mutex request count is incremented. Since there are stores pending in the store buffer, the ROB now issues a mutex request to MRF for all younger memory requests reaching ROB head ⑥ and does not allow that instruction to commit until it receives an acknowledgement for that mutex request. The MRF forwards this request to mutex vector ③. The requested mutex may be already available in the mutex vector. If not, then a new mutex request is issued following steps ④⑤. If the requested mutex is available in the mutex vector or when the CMM receives the mutex acknowledgement for these mutex request at a later time ⑦, then the mutex acknowledgement is forwarded to ROB ⑧. The mutex status is also set to *mutex-acquired*

when a new mutex acknowledgement is received. If ROB receives mutexes for both the youngest pending store and the instruction at ROB head, then it commits the instruction ⑨.

When the oldest store returns ⑩, the MRF pops its head entry and forwards the mutex release request to mutex vector ⑪. Mutex vector decrements the request count and checks if the status is *mutex-acquired* and request count is 0. If this check passes, then the mutex is released ⑫ after changing the mutex status to *mutex-invalid*. After popping the store ⑬, the MRF checks if the next instruction at FIFO head is a store. If the head is not a store, then the process of popping requests and releasing mutexes continues ⑭. If the head is a store ⑮, then the store buffer is consulted to see if that store is retired. If that store is retired ⑯, then the process of popping requests and releasing mutexes continue until the MRF is completely drained or the head is a store that has not retired yet. When MRF is completely drained, there will be no pending store in the store buffer and ROB resumes its normal operation.

Apart from simple condition checking hardware, CMM just adds a FIFO (MRF) and a direct mapped mutex vector. Thus, this implementation of CMM avoids any expensive associative structures.

6.4 GPU Mutex Manager

Similarly to CMM, GPU Mutex Manager (GMM) manages the mutexes for CUs. However, GMM manages mutexes per wavefront. Figure 9(b) shows the per wavefront structures in a GMM. The GMM has a *pending-store* counter that keeps track of the outstanding store requests from that wavefront. A *pending-mutex* counter keeps track of the outstanding mutex requests from that wavefront. The mutex vector in GMM only has a 2-bit status field. When stores from a wavefront instruction reach the GMM, these stores are issued to the L1 cache. Since the GPU cache is write-through, all stores are considered to be store misses and a mutex request is issued for every store. Both *pending-mutex* and *pending-store* counters are incremented to keep track of the outstanding mutex and stores, respectively. However, before issuing a mutex request, the mutex vector is consulted and a request is issued only when the status is *mutex-invalid*.

Unlike CPU mutex requests, the mutex pool (mutex pool is described in detail in Section 6.5) does not buffer GPU mutex requests because of the limited buffering space available with it. So, the mutex pool can send back a GPU mutex response with a negative acknowledgement (nack) indicating that the request did not acquire a mutex. When *pending-mutex* reaches zero and all these mutex requests have acquired their mutex, the next wavefront instruction is serviced by GMM. The acquired mutexes are released when the *pending-store* counter reaches zero.

Section 4.3 discusses implications of LSC on coalescer design. The coalescer issues coalesced requests from the same wavefront in program order to the L1 cache. The mutex acquisition mechanism discussed above allows younger requests to be issued under pending stores without making reordering of conflicting accesses visible to a wavefront. However, for the implicit flag synchronization discussed in Section 3.3, the younger loads should see the value of older stores from the same wavefront. Unlike CPUs, GPUs do not employ a store buffer. Also, the L1 cache in our implementation blocks a younger load under a pending store to the same address. Thus, a younger load sees the value of older store from the same wavefront and LSC's implicit synchronization flag for lockstep threads is guaranteed by our implementation.

6.5 Mutex Pool

The mutex pool keeps track of the mutexes acquired by a CU or a CPU core with a direct-mapped mutex vector. The direct mapped mutex vector enables the mutex pool to be implemented as an address interleaved, distributed structure making it extremely scalable. It is indexed similar to the mutex vectors in CMM and GMM. It has a single bit to store the mutex status and an address

buffer to buffer pending requests to that mutex as shown in Figure 9(c). If the bit is set, then it indicates that mutex is already acquired by a core in the APU. If a mutex request arrives at mutex pool, then the mutex pool checks if that mutex is available by checking the status bit. If the status bit is not set, then that mutex is available and the mutex request is acknowledged by mutex pool after setting the status bit. If the status bit is set, then the mutex pool buffers the request into a per mutex address buffer only if that mutex request is from a CPU. If the mutex request is from a GPU core, then the mutex pool responds with a negative acknowledgement (nack) indicating the requester did not acquire that mutex. Since a CPU core does not issue multiple outstanding requests to the same mutex, the address buffering needed per mutex vector entry is bounded by the number of CPU cores in the system. However, since GPU cores keep track of requests at wavefront granularity and there are 320 wavefronts (40 wavefronts per CU and eight CUs) in our implementation, the buffering space needed to buffer mutex requests from the GPU makes GPU mutex buffering unattractive. Additionally, CPUs are latency sensitive and this design avoids CPU requests waiting behind GPU requests. Since GPUs are inherently latency tolerant, the nacking of its requests will have little impact on its performance. When a mutex release request reaches the mutex pool, the pending address buffer is consulted and mutex is transferred to the next requester waiting in the pending address buffer. If the pending buffer is empty, then the mutex status bit is unset and that mutex becomes available for future requests.

Scalability of the Mutex Pool: To increase the scalability, the mutex pool can be implemented as an address-interleaved distributed structure. Each address will still have a unique mutex vector entry in this distributed structure and thus the conflicting accesses will map to the same unique entry providing exclusion. Thus distributed mutex pool functions identically to a centralized mutex pool but with better scalability.

7 DISCUSSION

The implementation of LSC has a number of implications on the system, which are discussed here.

Interaction with Explicit Synchronization: Our implementation treats explicit synchronization operations as regular memory operations. For example, a read-modified-write operation from a CPU core or an atomic-compare-and-swap access from GPU is treated as a write operation in our implementation. Since atomic SC allows reordering of instructions only after acquiring mutexes for all instructions involved in reordering under the shadow of a store miss, any reordering requirement imposed by synchronization operations is preserved, because that reordering will not be visible to other cores.

Alternative Design Considerations: All design decisions we made for our implementation were aimed at reducing the hardware cost and complexity. Toward this goal, we avoided using associative hardware structures in our implementation and implemented LSC with minimum modifications to the baseline APU system. However, there are many design choices available for implementing LSC. For example, in a GPU with small number of CUs, an inclusive directory at GL2 to keep track of sharers will not incur a significant overhead. Also, timestamp coherence can be used for reducing coherence traffic overhead [73] and private-shared memory access classification [71, 72] can be used for reducing mutex requirements but we leave these explorations to future work.

LSC and Previous GPU SC Implementations: Singh et al. [71] proposed efficient SC implementation for GPUs by extending the work of Singh et al. [72] for CPUs. This work implemented SC for wavefront instructions (warp instructions) and argued that SC ordering need not be preserved across per-work-item (per-thread) instructions that execute in lockstep fashion. Similar implementations were proposed by other researchers as well but with timestamp based coherence protocols [65, 77]. Some of these implementations were in fact adhering to the LSC consistency model on GPUs [65, 71], not SC.

LSC and Interaction with GPU Execution Model: A wavefront can diverge in a GPU when the work-items diverge in their conditional branch outcome. However, since LSC assigns program order to an entire wavefront, diverged execution can be simply treated as an execution phase in which only a subset of work-items issue memory accesses. These accesses still follow all LSC rules but the number of accesses that are ordered by $<_m>$ memory order will be less than the wavefront width.

GPUs with independent thread scheduling support [1] schedule threads independently. Consequently, there are no lockstep execution assumptions/guarantees in these GPUs [59] and LSC's ordering rule for lockstep execution is not applicable. Since LSC without lockstep execution rule is the same as SC, LSC naturally devolves to enforcing SC at work-item or thread level granularity with independent thread scheduling. Thus, LSC is well defined for these GPU architectures as well. This property of LSC also makes it applicable to future SoCs featuring accelerators without the same lockstep execution model.

Ease of Verification: A strong model has fewer valid outcomes than a weak model. Hence, permissible updates to architectural states are lesser in strong models, making the hardware easier to verify. Program verification also follows similar reasoning [9] and thus, LSC programs are easier to verify than programs running on a weak model. Moreover, since our implementation does not rely on speculation and rollback support to implement strong ordering, the speculative state updates are also avoided further aiding the hardware verification of our implementation.

Deadlocks and Mutex Acquisition: The mutexes are acquired in program order by both the CPU and GPU. For the GPU, the younger memory operations are allowed to request a mutex only after all memory operations from an older wavefront instruction has acquired a mutex. Thus, the mutex acquisition is in program order in a GPU. In a CPU, although the instructions are executed out-of-order, the mutex acquisition itself is delayed to the commit stage and hence the mutex acquisition is in program order. This in-order acquisition of mutexes prevent any younger instruction from holding on to a mutex while an older instruction is waiting for the same mutex. Thus, cyclic mutex dependencies never occur with this design avoiding any possibility of deadlock. Hence, our implementation is deadlock free.

LSC and Disjoint Address Space: The CPU-GPU ordering rules are not applicable to a system where CPUs and GPUs exist in disjoint address spaces (example, discrete GPU system). However, LSC rules are well defined for a GPU operating in its own address space and hence is applicable to discrete GPUs as well.

8 METHODOLOGY

Simulator: We used the AMD gem5 APU simulator [42] that models a GCN GPU architecture [16]. This version of the simulator simulates the GCN3 ISA [17]. The simulator models an OoO cycle level CPU core. The memory subsystem is modeled in Ruby [57].

The simulation parameters are given in Table 1. The mutex managers and the mutex pool modules are added to this baseline APU system to implement LSC. The mutex requests and responses travel through dedicated virtual channels and do not interfere with coherence traffic. Decoupling ordering (consistency) from coherence simplifies the implementation of both and was one of the original arguments made in favor of Atomic SC [41].

The TCD uses five hash functions with 64-bit hashing vectors as suggested by Zebchuk et al. [79]. Like previously discussed in Section 6, we recalculate the hash only after N evictions. We compared the bandwidth increase with N = 4,8,16 and N = 16 generated only 2.7% extra invalidation bandwidth than the best possible case of N = 1. So, we decided to use N = 16 in our implementation.

Table 1. Simulation Parameters

CPU	2 x86 cores @ 3 GHz
ROB Size	192
CPU L1 Data Cache	64 KB (2-way associative)
CPU L1 Instruction Cache	32 KB (2-way associative)
CPU L2 Cache	2 MB (8-way associative)
CPU Shared L3 Cache	8 MB (16-way associative)
GPU	8 GCN CUs @ 1GHz
GPU L1 Data Cache	16 KB (64-way associative)
GPU L1 Instruction Cache	16 KB (16-way associative)
GPU Shared L2 Cache	512 KB (16-way associative)
MRF	100 entries
Mutex Pool	1,024 entries, 30 cycles away
Tagless Directory	5 hash functions, 64-bit sharing vector

Table 2. Benchmarks

AMD Compute App	**Description**	Data shared between CPU and GPU (in MB and in %)
comd	Molecular dynamics simulation	80.14 (11.24%)
lulesh	Hydrodynamics calculation	64.73 (74.84%)
hpgmg	Linear solvers	17.43 (42.24%)
HCC Example App		
array	Streaming write to an array	1.64 (6.36%)
bitonic	Bitonic sort	415.63 (66.68%)
fft	Fast Fourier transform	17.16 (96.69%)
spmv	Sparse matrix-vector mulitply	6.24 (16.32%)
Rodinia		
backprop	Back propogation	3.09 (61.54%)
bfs	Breadth first search	27.26 (33.19%)
nn	Nearest neighbour	1.3 (44.47%)
Hand written		
dgemm	Double precision matrix multiplication	3.13 (26.86%)

Workloads: We used benchmarks from AMD Compute App suite [13], HCC example app suite [15] and Rodinia benchmark suite [30] to evaluate LSC. application that was reported All these benchmarks use a shared virtual address space to share data between CPU and GPU without any explicit data copy. In addition to these benchmarks, we use two benchmarks; *hashtable* and *ATM*; from Fung et al. [35] to demonstrate the performance advantage of a strong model for fine-grained synchronizing and data sharing GPU applications. Each GPU thread in *hashtable* benchmark inserts a key–value pair to a hash table slot necessitating fine-grained synchronization of threads trying to insert value to the same slot and fine-grained data sharing to get the most updated copy of the hash table after a new element is inserted. The *ATM* benchmark simulates GPU threads doing transactions between two accounts and similar to *hashtable*, requires fine-grained synchronization and fine-grained data sharing between GPU threads. We only simulate the GPU phase of these two benchmarks. Table 2 lists these benchmarks. That table also provides

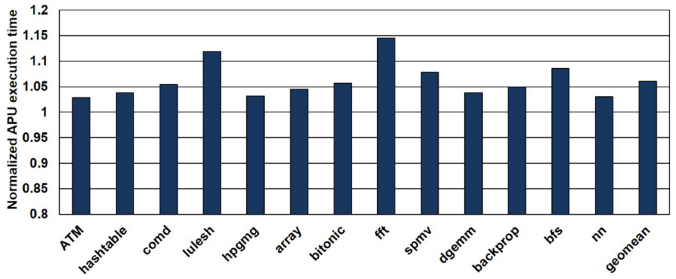


Fig. 11. Normalized application execution time.

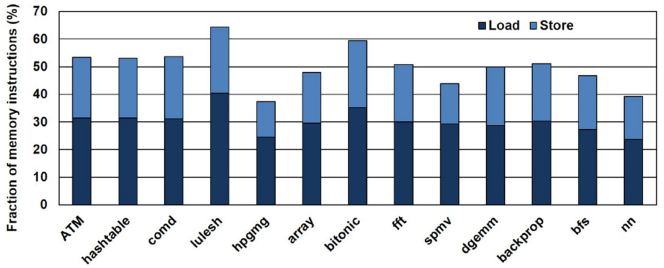


Fig. 12. Fraction of memory instructions committed by each benchmark. The load-store split is also shown.

the amount of data (in MB) coherently shared between CPU and GPU by each benchmark. This coherently shared data as a fraction of the total data accessed by GPU (in %) is also provided. We evaluate LSC with these benchmarks in the next section.

9 RESULTS

9.1 Application Performance Evaluation

We compare the performance of LSC system against the baseline. Since the baseline adheres to a weaker model than LSC, our aim is to show that a strong consistency model like LSC can achieve performance comparable to this baseline.

Figure 11 shows the execution time of LSC compared to baseline for various benchmarks. The LSC was slower than a weaker consistency model by 6.11%. However, from that figure, it can be seen that LSC was performing within the 5% range for more than half of the evaluated benchmarks. The figure shows that two benchmarks *lulesh* and *fft* slowed down by more than 10% with weaker models.

To understand this performance difference, we first looked into the CPU execution of these applications and analyzed their instruction composition. Figure 12 shows the fraction of memory instructions committed by each benchmark. It can be seen that most of these benchmarks have more than 50% memory reference instructions. That figure also shows the percentage of loads and stores. Although LSC adds mutex acquisition latency to the commit stage of a memory instruction, an OoO execution pipeline is usually not affected by it because of its ability to extract ILP over a wide instruction window. As such, LSC will stall the execution pipeline only if the ROB becomes full. However, for memory intensive applications, LSC will frequently stall commit preventing the memory instruction at the head of ROB from committing and freeing up ROB entries and potentially hurting application performance after the ROB gets full.

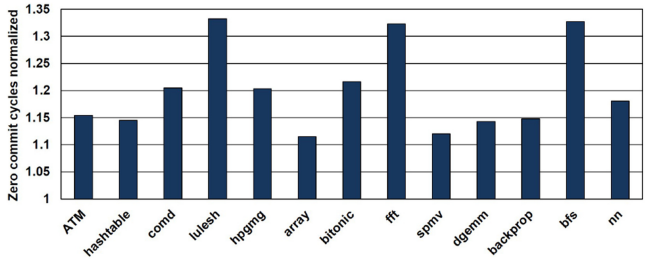


Fig. 13. Increase in zero commit cycles. This measures the commit stage stalls introduced by LSC.

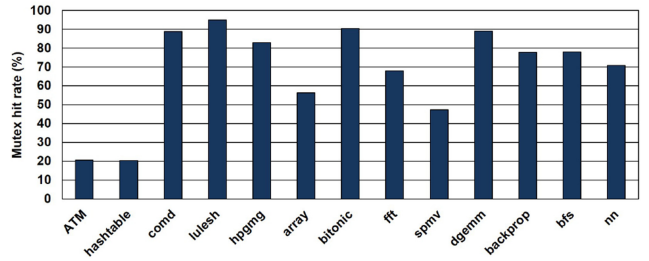


Fig. 14. Mutex hit rate quantify the reuse of mutexes by a CPU core.

Figure 13 shows the increase in number of zero commit cycles with LSC. Zero commit cycle indicates the number of cycles in which no instruction was committed. It is a measure of the commit stage stall cycles. However, it should be noted that the commit stage stalls did not directly translate to performance loss, because the OoO execution pipeline will be stalled only if ROB gets full.

Even if an application is memory intensive, the mutex acquisition latency will affect the critical path in the commit stage only if the head instruction has not acquired a mutex. Typically, the memory instructions of an application have high spatial and temporal locality, so the mutexes acquired by the core can be reused by multiple memory instructions. Figure 14 shows the mutex hit rate of different benchmarks. While most of the applications have high hit rate, *fft* had relatively low hit rate and as such did not benefit from mutex reuse. It can be also seen from that figure that *lulesh* observed a very high mutex hit rate and that helped *lulesh* to perform better than *fft* although it has a larger fraction of memory instructions than *fft*. *hashtable* and *ATM* only have their GPU phases simulated, and the CPU phase hardly had any work to do other than launching kernels; hence the low mutex reuse.

Our implementation is quite conservative: For example, we do not attempt to prefetch mutexes in the OoO core (analogous to exclusive prefetching of write permission [37]). Another possible optimization is caching of mutexes for reducing the mutex acquisition latency overhead. Adding these optimizations would complicate implementation, but is likely to recover these performance losses. We leave exploration of these opportunities to future work.

GPU Performance Comparison: Figure 15 compares GPU execution time of LSC against baseline HRF model. Across the spectrum of benchmarks, LSC performs very close to the HRF baseline with benchmarks slowing down by only 0.76% with LSC. The latency tolerant GPUs can hide the mutex acquisition latency by overlapping execution with context switching between multiple wavefront contexts. Because of this, mutex acquisition latency will not show up on the critical path of GPU execution, and a strong model like LSC performs well on a GPU.

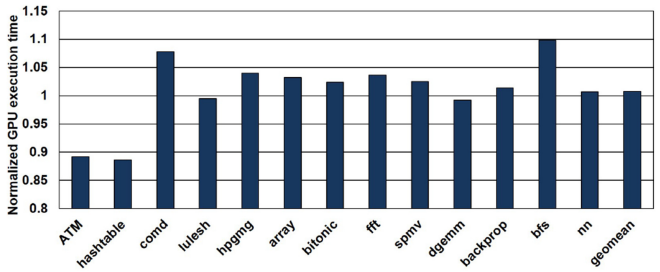


Fig. 15. Normalized GPU execution time.

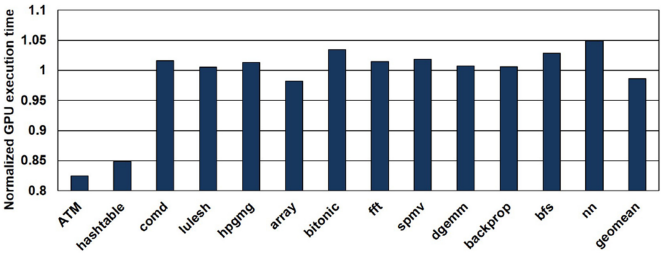


Fig. 16. Performance penalty of implementing HRF with hardware coherence.

Figure 15 also shows that *hashtable* and *ATM* performed better with LSC. Since these benchmarks have fine-grained data sharing, an HRF model does an acquire operation (that is invalidating GPU L1 cache lines) before attempting each insertion (for *hashtable*) or transaction (for *ATM*) to get the latest updates. The LSC model maintains data updates with invalidation messages provided by the underlying hardware coherence mechanism and as such does not encounter the performance penalty of invalidating the entire GPU L1 cache. Consequently, the GPU L1 hit rate of *hashtable* increased from 10.12% to 40.50% and that of *ATM* increased from 40.40% to 66.37% with LSC compared to the baseline. This resulted in *hashtable* and *ATM* achieving a speedup of 12.81% and 12.14%, respectively.

Finally, Figure 16 shows the performance penalty of implementing HRF with hardware coherence mechanism. To measure this overhead, we created an intermediate baseline by modifying the baseline APU system that broadcasts invalidation messages to the GL1 to make it read coherent. Thus, an acquire operation does not need to invalidate the entire GL1 in this implementation. From that figure, it can be seen that making GL1 coherent incurred a performance penalty for all evaluated benchmarks except *hashtable* and *ATM*. For these two benchmarks, the performance improved by 17.82% and 21.20%, respectively.

9.2 Increase in Bandwidth Consumption

LSC modifies the GPU memory subsystem to send invalidation and eviction messages between GL1 and GL2. The bandwidth increase by this extra traffic is given in Figure 17. The mutex acquisition and release traffic is given as *Mutex* in that figure. The traffic generated by eviction messages and its acknowledgement is given as *Eviction*. *Invalidation* measures the bandwidth generated by the invalidation messages and its acknowledgement. The recalculated hash generated every 16th eviction is shown as *Rehash*. For benchmarks like bitonic sort that shares data across CUs, the extra bandwidth is dominated by *Invalidation* traffic. The streaming nature of GPU applications lead to *Eviction* messages and most of the benchmarks contributed to eviction traffic as shown

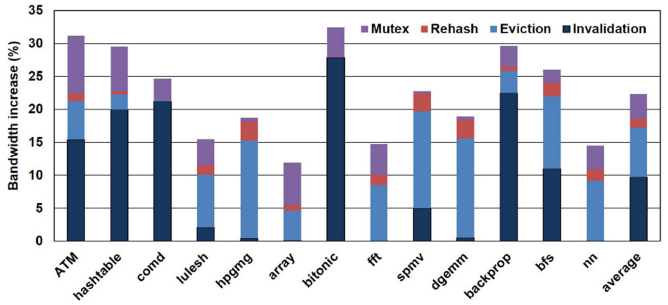


Fig. 17. Bandwidth increase with LSC.

Table 3. Hardware Cost

Mutex vector	1024 entries; 5 bits (CPU core) or 2 bits (CU) per entry
MRF	100 entries per CPU core
Mutex pool	1024 entries; 1-bit and one address buffer per entry
TCD	5 64-bit hash functions per set per core
GPU L1	4 bits per set to count 16 evictions

in Figure 17. The figure also shows that *Rehash* traffic has contributed little to the bandwidth consumption. Overall, the GPU bandwidth requirement increased by an average of 22.35% with eviction, invalidation and mutex traffic contributing 9.70%, 7.51%, and 3.73%, respectively.

LSC implementation did not modify baseline CPU memory subsystem and the only addition to the CPU traffic was the mutex acquisition and release traffic. Since CPU has a very high mutex hit rate (Figure 14), only few memory operations had to request mutexes and the mutex traffic only contributed 0.42% to the CPU bandwidth consumption.

9.3 Sensitivity to Mutex Request FIFO Size

We ran experiments with 400 entry MRF and did not see any noticeable performance improvement indicating that a 100 entry MRF (default MRF size in our experiments) captures all the reordering opportunity for memory accesses.

9.4 Hardware Cost

Table 3 lists the hardware cost of implementing LSC. All structures added by our implementation are direct-mapped and do not require expensive associative lookups, and these structures are extremely small compared to other CPU or CU resources. For example, the direct-mapped mutex vector at CMM is only 640 bytes (5 * 1,024 bits). Compared to our baseline 64 KB set-associative CPU L1, CMM mutex vector is only a fraction of its size and adds negligible area overhead to the baseline.

10 RELATED WORK

Numerous researchers have investigated efficient support for SC such that it matches the performance of weaker models on a CPU [4, 22, 28, 29, 39–41, 51, 52, 64, 72, 78]. These approaches can be classified as speculative approaches and non-speculative approaches. The speculative approaches require expensive hardware mechanisms to speculate beyond consistency model imposed boundary and to rollback when mis-speculated. The non-speculative approaches [4, 41, 52, 72] enforce

SC by ordering only conflicting accesses and significantly reduce the hardware overhead. These non-speculative approaches differ in the way they detect conflicts and order conflicting accesses. For example, Singh et al. [72] rely on complier techniques to identify conflicting accesses whereas Adve et al. [4] rely on underlying cache coherence mechanism to detect and serialize conflicting accesses. Conflict ordering proposed by Lin et al. [52] uses an augmented write buffer to keep track of active memory operations (memory operations retired from the ROB, but whose predecessors are pending in the global memory order) and uses tagged coherence messages to detect and order conflicting active memory operations. Our implementation is an extension of Atomic SC proposed by Gope and Lipasti [41] and relies on mutexes to detect and serialize conflicting accesses. Hence, it does not require expensive associative look ups or compiler support, and does not interfere with the underlying coherence implementation.

Academic researchers have proposed implementing SC on GPUs in the past [44, 65, 71]. While all these implementations produced promising results, they differed in their underlying assumptions. For example, Singh et al. [71] implemented SC on a baseline that has a MESI cache coherence protocol while Ren and Lis [65] and Singh et al. [73] has demonstrated that a CPU like coherence *read-for-ownership* protocol is not suitable for GPUs. To this end, we use a realistic baseline GPU cache coherence and evaluate our implementation against this realistic baseline. We also show the bandwidth overhead of maintaining coherence using invalidation messages in GPU caches.

While most past proposals implemented SC on either the CPU or GPU, the Alsop et al. [12] proposed Spandex that implemented SC-for-DRF [6] on a heterogeneous system. However, SC-for-DRF requires a programmer to insert synchronization primitives similar to HRF, but LSC does not.

11 CONCLUSION

In this article, we introduce lockstep SC, a consistency model that is as strong as SC but formally defined to accommodate the GPU lockstep execution model. We also demonstrate that LSC can be efficiently implemented in an SoC without expensive hardware structures. The performance cost of enforcing LSC was just 0.76% for the GPU and 6.11% for the entire SoC. The proposed LSC is simple for programmers to reason about and is portable across different memory hierarchy organizations because of its memory hierarchy agnostic nature.

REFERENCES

[1] 2017. Inside Volta: The World's Most Advanced Data Center GPU. Retrieved from https://devblogs.nvidia.com/inside-volta/.

[2] Sarita Vikram Adve. 1993. *Designing Memory Consistency Models for Shared-memory Multiprocessors*. Ph.D. dissertation. Madison, WI.

[3] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: A tutorial. *Computer* 29, 12 (Dec. 1996), 66–76. DOI: https://doi.org/10.1109/2.546611

[4] Sarita V. Adve and Mark D. Hill. 1990. Implementing sequential consistency in cache-based systems. In *Proceedings of the 1990 International Conference on Parallel Processing*. 47–50.

[5] S. V. Adve and M. D. Hill. 1990. Weak ordering-a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*. 2–14.

[6] S. V. Adve and M. D. Hill. 1990. Weak ordering-a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*. 2–14. DOI: https://doi.org/10.1109/ISCA.1990.134502

[7] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU concurrency: Weak behaviours and programming assumptions. *SIGPLAN Not.* 50, 4 (March 2015), 577–591. DOI: https://doi.org/10.1145/2775054.2694391

[8] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. 2012. Software verification for weak memory via program transformation. arxiv:1207.7264. Retrieved from http://arxiv.org/abs/1207.7264.

[9] Jade Alglave and Luc Maranget. 2011. Stability in weak memory models. In *Proceedings of the International Conference on Computer Aided Verification (CAV'11)*.

[10] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. DOI: https://doi.org/10.1145/2627752

[11] Johnathan Alsop, Marc S. Orr, Bradford M. Beckmann, and David A. Wood. 2016. Lazy release consistency for GPUs. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, Article 26, 13 pages.

[12] J. Alsop, M. Sinclair, and S. Adve. 2018. Spandex: A flexible interface for efficient heterogeneous coherence. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. 261–274. DOI: https://doi.org/10.1109/ISCA.2018.00031

[13] AMD. [n.d.]. Compute Apps. Retrieved from https://github.com/AMDComputeLibraries/ComputeApps.

[14] AMD. [n.d.]. HC. Retrieved from https://rocm.github.io/languages.html.

[15] AMD. [n.d.]. HCC Example Apps. Retrieved from https://github.com/ROCm-Developer-Tools/HCC-Example-Application.

[16] AMD. 2012. AMD Graphics Cores NEXT (GCN) Architecture. Retrieved from https://goo.gl/GPvy8R.

[17] AMD. 2016. AMD GCN3 ISA Architecture Manual. Retrieved from https://gpuopen.com/compute-product/amd-gcn3-isa-architecture-manual.

[18] AMD. 2016. Dissecting the Polaris Architecture. Retrieved from https://goo.gl/hNrZZo.

[19] AMD. 2019. User Guide for AMDGPU Backend. Retrieved from https://llvm.org/docs/AMDGPUUsage.html.

[20] ARM. [n.d.]. ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile. Retrieved from https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile.

[21] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2010. On the verification problem for weak memory models. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*. ACM, New York, NY, 7–18. DOI: https://doi.org/10.1145/1706299.1706303

[22] Colin Blundell, Milo M. K. Martin, and Thomas F. Wenisch. 2009. InvisiFence: Performance-transparent memory ordering in conventional multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 233–244. DOI: https://doi.org/10.1145/1555754.1555785

[23] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. Association for Computing Machinery, New York, NY, 68–78. DOI: https://doi.org/10.1145/1375581.1375591

[24] D. Bouvier and B. Sander. 2014. Applying AMD's Kaveri APU for heterogeneous computing. In *Proceedings of the 2014 IEEE Hot Chips 26 Symposium (HCS'14)*.

[25] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2007. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, New York, NY, 12–21. DOI: https://doi.org/10.1145/1250734.1250737

[26] Sebastian Burckhardt and Madanlal Musuvathi. 2008. Effective program verification for relaxed memory models. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV'08)*. Springer-Verlag, Berlin, 107–120. DOI: https://doi.org/10.1007/978-3-540-70545-1_12

[27] Jacob Burnim, Koushik Sen, and Christos Stergiou. 2011. Testing concurrent programs on relaxed memory models. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11)*. ACM, New York, NY, 122–132. DOI: https://doi.org/10.1145/2001420.2001436

[28] H. W. Cain and M. H. Lipasti. 2004. Memory ordering: A value-based approach. *IEEE Micro* 24, 6 (Nov. 2004), 110–117. DOI: https://doi.org/10.1109/MM.2004.81

[29] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. 2007. BulkSC: Bulk enforcement of sequential consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. ACM, New York, NY, 278–289. DOI: https://doi.org/10.1145/1250662.1250697

[30] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC'09)*.

[31] A. ElTantawy and T. M. Aamodt. 2016. MIMD synchronization on SIMT architectures. In *Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–14.

[32] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*. ACM, New York, NY, 608–621. DOI: https://doi.org/10.1145/2837614.2837615

[33] HSA Foundation. 2016. HSA Platform System Architecture Specification 1.1. Retrieved from http://www.hsafoundation.com/?ddownload=5114.

[34] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07)*. 407–420. DOI: https://doi.org/10.1109/MICRO.2007.30

[35] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt. 2011. Hardware transactional memory for GPU architectures. In *Proceedings of the 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*. 296–307.

[36] Benedict R. Gaster, Derek Hower, and Lee Howes. 2015. HRF-Relaxed: Adapting HRF to the complexities of industrial heterogeneous memory models. *ACM Trans. Archit. Code Optim.* 12, 1, Article 7 (April 2015), 26 pages. DOI: https://doi.org/10.1145/2701618

[37] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1991. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing*. 355–364.

[38] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*. ACM, New York, NY, 15–26. DOI: https://doi.org/10.1145/325164.325102

[39] C. Gniady and B. Falsafi. 2002. Speculative sequential consistency with little custom storage. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 179–188. DOI: https://doi.org/10.1109/PACT.2002.1106016

[40] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. 1999. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*. IEEE Computer Society, Washington, DC, 162–171. DOI: https://doi.org/10.1145/300979.300993

[41] D. Gope and M. H. Lipasti. 2014. Atomic SC for simple in-order processors. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. 404–415. DOI: https://doi.org/10.1109/HPCA.2014.6835950

[42] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. Rogers. 2018. Lost in abstraction: Pitfalls of analyzing GPUs at the intermediate language level. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. 608–619. DOI: https://doi.org/10.1109/HPCA.2018.00058

[43] B. A. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. 2014. QuickRelease: A throughput-oriented approach to release consistency on GPUs. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. 189–200. DOI: https://doi.org/10.1109/HPCA.2014.6835930

[44] Blake A. Hechtman and Daniel J. Sorin. 2013. Exploring memory consistency for massively-threaded throughput-oriented processors. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 201–212. DOI: https://doi.org/10.1145/2485922.2485940

[45] Mark D. Hill. 1998. Multiprocessors should support simple memory-consistency models. *Computer* 31, 8 (Aug. 1998), 28–34. DOI: https://doi.org/10.1109/2.707614

[46] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-race-free memory models. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, NY, 427–440. DOI: https://doi.org/10.1145/2541940.2541981

[47] G. Krishnan, D. Bouvier, and S. Naffziger. 2016. Energy-efficient graphics and multimedia in 28-nm Carrizo accelerated processing unit. *IEEE Micro* 36, 2 (2016), 22–33. DOI: 10.1109/MM.2016.24

[48] L. Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. C*28, 9 (Sept. 1979), 690–691. DOI: https://doi.org/10.1109/TC.1979.1675439

[49] Michael LeBeane, Khaled Hamidouche, Brad Benton, Mauricio Breternitz, Steven K. Reinhardt, and Lizy K. John. 2018. ComP-net: Command processor networking for efficient intra-kernel communications on GPUs. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT'18)*. ACM, New York, NY, Article 29, 13 pages. DOI: https://doi.org/10.1145/3243176.3243179

[50] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling energy optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*.

[51] C. Lin, V. Nagarajan, and R. Gupta. 2010. Efficient sequential consistency using conditional fences. In *Proceedings of the 2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. 295–306.

[52] Changhui Lin, Vijay Nagarajan, Rajiv Gupta, and Bharghava Rajaram. 2012. Efficient sequential consistency via conflict ordering. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, 273–286. DOI: https://doi.org/10.1145/2150976.2151006

[53] Yuan Lin and Vinod Grover. 2018. Using CUDA Warp-Level Primitives. Retrieved from https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/.

[54] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A formal analysis of the NVIDIA PTX memory consistency model. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. ACM, New York, NY, 257–270. DOI: https://doi.org/10.1145/3297858.3304043

[55] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A formal analysis of the NVIDIA PTX memory consistency model. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. Association for Computing Machinery, New York, NY, 257–270. DOI: https://doi.org/10.1145/3297858.3304043

[56] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. 2012. An axiomatic memory model for POWER multiprocessors. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12)*. Springer-Verlag, Berlin, 495–512. DOI: https://doi.org/10.1007/978-3-642-31424-7_36

[57] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News* 33, 4 (Nov. 2005), 92–99. DOI: https://doi.org/10.1145/1105734.1105747

[58] Microsoft. [n.d.]. C++ AMP : Language and Programming Model. Retrieved from http://download.microsoft.com/download/2/2/9/22972859-15c2-4d96-97ae-93344241d56c/cppampopenspecificationv12.pdf.

[59] NVIDIA. 2020. CUDA C++ Programming Guide. Retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[60] Marc S. Orr, Bradford M. Beckmann, Steven K. Reinhardt, and David A. Wood. 2014. Fine-grain task aggregation and coordination on GPUs. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 181–192. DOI: https://doi.org/10.1145/2678373.2665701

[61] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2013. Heterogeneous system coherence for integrated CPU-GPU systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, 457–467. DOI: https://doi.org/10.1145/2540708.2540747

[62] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article 19 (Dec. 2017), 29 pages. DOI: https://doi.org/10.1145/3158107

[63] Sooraj Puthoor and Mikko H. Lipasti. 2018. Compiler assisted coalescing. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT'18)*. Association for Computing Machinery, New York, NY, Article 11, 11 pages. DOI: https://doi.org/10.1145/3243176.3243203

[64] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. 1997. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'97)*. ACM, New York, NY, 199–210. DOI: https://doi.org/10.1145/258492.258512

[65] X. Ren and M. Lis. 2017. Efficient sequential consistency in GPUs via relativistic cache coherence. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. 625–636. DOI: https://doi.org/10.1109/HPCA.2017.40

[66] Ben Sander. 2016. AMD GCN Assembly: Cross-Lane Operations. Retrieved from https://gpuopen.com/learn/amd-gcn-assembly-cross-lane-operations/.

[67] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, New York, NY, 175–186. DOI: https://doi.org/10.1145/1993498.1993520

[68] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. DOI: https://doi.org/10.1145/1785414.1785443

[69] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2015. Efficient GPU synchronization without scopes: Saying no to complex consistency models. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, 647–659. DOI: https://doi.org/10.1145/2830772.2830821

[70] M. D. Sinclair, J. Alsop, and S. V. Adve. 2015. Efficient GPU synchronization without scopes: Saying no to complex consistency models. In *Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. 647–659. DOI: https://doi.org/10.1145/2830772.2830821

[71] A. Singh, S. Aga, and S. Narayanasamy. 2015. Efficiently enforcing strong memory ordering in GPUs. In *Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. 699–712. DOI: https://doi.org/10.1145/2830772.2830778

[72] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. 2012. End-to-end sequential consistency. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*. IEEE Computer Society, Washington, DC, 524–535.

[73] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. 2013. Cache coherence for GPU architectures. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. 578–590. DOI: https://doi.org/10.1109/HPCA.2013.6522351

[74] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. 2013. Cache coherence for GPU architectures. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. 578–590. DOI: https://doi.org/10.1109/HPCA.2013.6522351

[75] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence* (1st ed.). Morgan & Claypool Publishers.

[76] SPARC International, Inc. 1994. *The SPARC Architecture Manual (Version 9)*. Prentice-Hall, Upper Saddle River, NJ.

[77] Abdulaziz Tabbakh, Xuehai Qian, and Murali Annavaram. 2018. G-TSC: Timestamp based coherence for GPUs. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. IEEE, 403–415.

[78] Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2007. Mechanisms for store-wait-free multiprocessors. *SIGARCH Comput. Archit. News* 35, 2 (June 2007), 266–277. DOI: https://doi.org/10.1145/1273440.1250696

[79] Jason Zebchuk, Vijayalakshmi Srinivasan, Moinuddin K. Qureshi, and Andreas Moshovos. 2009. A tagless coherence directory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, 423–434. DOI: https://doi.org/10.1145/1669112.1669166

[80] Sizhuo Zhang, Arvind, and Muralidaran Vijayaraghavan. 2016. Taming weak memory models. arxiv:1606.05416. Retrieved from http://arxiv.org/abs/1606.05416.

[81] Sizhuo Zhang, Muralidaran Vijayaraghavan, Andrew Wright, Mehdi Alipour, and Arvind. 2018. Constructing a weak memory model. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE Press, Piscataway, NJ, 124–137. DOI: https://doi.org/10.1109/ISCA.2018.00021

Received May 2020; revised September 2020; accepted October 2020