G-TSC: GPU Synchronization Independent Connectivity

Mr.Sakti Charan Panda[1], Mr.Alok Kumar Pattnaik[2]*

*[1]*Assistant Professor,Dept. Of Computer Science and Engineering, NIT , BBSR*
[2]Assistant Professor,Dept. Of Computer Science and Engineering, NIT , BBSR
managesakticharan@thenalanda.com, alokkumar@thenalanda.com*

***Abstract*— In the context of chip multiprocessors, substantial research on cache coherence has been conducted (CMP). It is generally known that when the number of hardware thread contexts rises, conventional directory-based and snooping coherence algorithms generate a significant amount of coherence traffic. Since GPUs may support hundreds or even thousands of threads, using traditional coherence techniques on GPUs will make the bandwidth problems already present on GPUs worse. Previous research has suggested time-based coherence procedures in recognition of this restriction. The main concept is to give the accessed cache block a lease duration, and when the lease ends, the cache block self-invalidates. Yet, global synchronised clocks are necessary for time-based coherence protocols. Furthermore, because threads must wait to retrieve data with an unused timeout, this strategy might result in more execution delays. Recently, the timestamp-based coherence protocol known as Tardis was put forth to do away with the necessity for global clocks in CPUs. This study expands on earlier research and suggests G-TSC, a revolutionary timestamp-based cache coherence mechanism for GPUs. Coherence transactions are carried out by G-TSC in logical time. The difficulties in implementing timestamp coherence for GPUs with unique microarchitecture features and significant thread parallelism are shown in this paper. The following section of this work offers a variety of solutions to problems that are GPU-centric. G-performance TSC's in the GPGPU- Sim simulation framework is evaluated, and it outperforms time-based coherence with release consistency by 38%.**

***Keywords*-GPU; Cache Coherence**

I. Introduction

Graphics processing units (GPUs) have been widely used in high throughput general purpose computing because of their high power efficiency , computational power, and high off-chip memory bandwidth [1], [2]. As the GPU programming languages, such as OpenCL [3] and NVIDIA CUDA [4], enhance their capabilities GPUs are becoming a better computing platform choice for general purpose applications with regular parallelism. Prior study has argued that GPUs can also accelerate applications with irregular parallelism [5]. But porting an irregular parallel application to GPUs is currently hobbled by the lack of efficient hardware cache coherence support. If hardware cache coherence is provided on GPUs, it would enable efficient porting of a broad range of parallel applications. Cache coherence can be used as a building block to design memory consistency models and enable a programmer to reason about possible memory ordering when threads interact.

At the architecture level, most of the GPUs currently achieve cache coherence by disabling private caches and relying on flags [6], [7]) while AMD GPUs support coherent instructionsthat perform memory operations at the shared L2 cache and allow the software to flush the private cache at anytime [8]. Obviously, such approaches provide coherence but at the cost of performance loss stemming from disabling caches. With an ideal coherence mechanism, GPU applications that requires cache coherence can achieve up to 88% performance improvement over disabling L1 cache [9].

Recently, Temporal Coherence (TC) has been proposed for GPUs [9]. TC relies on self-invalidation of expired blocks in the private cache to eliminate coherence traffic due to invalidation requests. TC is inspired by Library Cache Coherence (LCC) [10], a time-based hardware coherence protocol that uses global synchronized counters to track the validity of cache blocks at different levels in the cache hierarchy and delays updates to unexpired blocks until all private copies are self- invalidated.

Unfortunately, TC suffers from several drawbacks. First, the use of *global synchronized counters* in TC to implement coherence raises an issue about the scalability. With the rapid growth in chip size and the increase in clock speed, the global counters can suffer from clock skewness and wiring delay that may affect the correctness of the protocol [11]. Second, *delayed updates* due to unexpired cached copies resultin execution stalls that do not happen in conventional cache coherence protocol. When an update is delayed, all subsequent reads are delayed until the update is performed. Preservingall cache blocks that are unexpired in L2 cache may cause unnecessary cache stalls due to higher hardware resource contention. Third, in TC, the performance can be sensitiveto the *lease period*; a suitable lease period is not always easyto select/predict.

Tardis is a new CPU coherence protocol based on timestamp ordering [12]. It uses a combination of physical time andlogical time to order memory operations. The key difference between Tardis and TC is that Tardis enforces global memory order by logical time rather than physical time. The timestampbased approach can largely eliminate the drawbacks of TC. While Tardis was explored in the context of CPU its applica- bility to a GPU’s unique architecture and execution model are unknown.

In this paper, we propose G-TSC, a timestamp-based cache coherence protocol for GPUs, inspired by the Tardis. We analyze the unique challenges in adopting the logical timestamp ordering approach to the highly threaded GPUs and then present and evaluate solutions. These challenges include controlling the accessibility of the updated data within a streaming multiprocessor (SM), managing the replicated requests from warps in the same SM, and relaxation of the cache inclusion requirement in order to increase the effective cache size. We show how to resolve these challenges in the presence of a large number of concurrent threads in a single SM that can generate a huge number of memory requests in a short time window, and in the absence of the write buffers which are traditionally used to facilitate these interactions in CPUs. We specify the complete operations of G-TSC based on a general GPU memory hierarchy. We consider the implementation of both Release Consistency (RC) and Sequential Consistency (SC) based on G-TSC. We implemented G-TSC in GPGU-Sim [13] and used twelve benchmarks in the evaluation. When using G-TSC to keep coherence between private caches and the shared cache, G-TSC outperforms TC by 38% with release consistency. Moreover, even G-TSC with sequential consistency outperforms TC with release consistency by 26% for benchmarks that require coherence. The memory traffic is reduced by 20% for memory intensive benchmarks.

The rest of this paper is organized as follows. Section II gives a brief background about the GPU architecture, memory system, memory consistency models, and cache coherence protocols. Section III proposes G-TSC. Section IV describes the implementation of G-TSC. Section V presents several GPU-specific challenges. The evaluation results are discussed in Section VI. Some other related works are discussed in Section VII and Section VIII concludes the paper.

II. Background and Motivation

A. Basic GPU Architecture

CPU launches a GPU *kernel* after its input data is transferred to the GPU memory. The kernel consists of 3-dimensional grid of thread blocks, called *Cooperative Thread Array (CTA)*, or *work group* where each thread block in turns consists of a 3-dimensional grid of *threads* or *work items*. Each thread block is assigned to a *Streaming Multiprocessors (SM)* that executes groups of threads (typically 32 threads) using the single instruction multiple thread (SIMT) paradigm. All the threads in a single group form a *warp* or *wavefront* [14]. Typically, a single GPU consists of dozens of SMs.

GPU applications have three memory spaces: *local*, *private*, and *global* memory space. Local memory (also called *shared memory*) is managed by the programmer and used for intra-CTA communication [14]. Private memory is a per-thread memory while the global memory is shared across all threads in the kernel. Local memory is not cached by the multi-level cache hierarchy while private and global memory are stored in the off-chip DRAM and can be cached [7]. Accesses by multiple threads in the same warp are merged into the minimum number of accesses by the coalescing unit in each SM.

The cache hierarchy in GPUs consists of a per-SM L1 private cache and a shared L2 cache. GPU caches adopt non-inclusive, non-exclusive cache policy with no coherence support for private caches [6]. L2 cache is divided into multiple banks and each bank is attached to a GDDR memory partition. The SMs are connected to multiple L2 cache banks over an interconnection network [15]. The cache misses are managed using miss status handling registers (MHRs). The MSHR table holds the information about all outstanding miss requests and allows a single outstanding read request per cache block. Since the interconnection network bandwidth is a performance bottleneck in GPUs all read accesses to the same cache block from different warps are merged in MSHR and a single read request is generated to the lower-level cache.

B. Coherence and Memory Consistency

Coherence is typically defined with the *”single writer multiple reader”* invariant. At any given moment in time, there is either a single writer or multiple readers for any given memory location [16]. The implementation of a cache coherence protocol typically involves three aspects: *1)* propagating the new value to all sharers either by invalidating or updating private copies; *2)* acknowledging the global performance of store operations; *3)* maintaining write atomicity [9] when required (i.e. value from the store operation is atomically seen by all threads at once). Some coherence protocols disregard some of these aspects partially or entirely.

While coherence deals with how values are propagated for a single memory location, it is generally not sufficient to reason about parallel thread interactions where multiple memory locations may be accessed. Memory consistency model defines the valid ordering of memory operations to different locations. In this paper, we consider the implementation of *Sequential Consistency (SC)* and *Release Consistency (RC)* on GPUs built on top of our timestamp-based coherence protocol.

Sequential consistency (SC) [17] requires that the memory operations of a program appear to be executed in some global sequence, as if the threads are multiplexed on a uniprocessor. SC restricts many architecture and compiler optimizations and usually leads to lower performance [18]. Release Consistency (RC), which is a relaxed memory consistency model that allows re-ordering of memory operations to different addresses. RC also relaxes the write atomicity requirements. The programmers can affirm the order between memory operations using *fence*. In summary, SC and RC are considered as two extreme examples as SC is the most restrictive memory model and RC is a more relaxed memory model. There are models in between such as Total-Store-Order (TSO) [18].

C. Invalidation-based Protocols

Conventional invalidation-based coherence protocols designed for multiprocessors (e.g. directory-based or snoopy protocol) are ill-suited for GPUs. They incur extensive coherence traffic and large storage overhead. The traffic overhead incurred by the invalidation-based protocols is due to unnecessary refills for write-once data which is a common access pattern in GPUs. Additionally, invalidation-based protocols incur the recall traffic, when all L1 copies need to be invalidated upon L2 invalidation or directory eviction. The storage overhead of the invalidation-based protocols is mostly due to the need to track outgoing in-flight coherence transactions and incoming coherence requests. If we reserve sufficient storage to handle the worse case scenario, an on-chip buffer as large as 28% of the total GPU L2 cache is needed [19].

D. Time-based Coherence

Temporal coherence (TC) [9] is a time-based cache coherence protocol designed for GPUs. TC uses time-based self-invalidation to reduce the coherence traffic. Like other time-based coherence protocols [10], TC assumes that single chip systems can implement globally synchronized counters. In TC, each cache block in private caches is assigned a *lease,* which indicates the time period that the block can be accessed in the private cache. The synchronized counters are used to count the lease period. A read access to a cache block in L1 cache checks both the tag and expiration time of its lease. A valid tag match but expired lease is considered as a coherence miss, because the block is already self-invalidated. L2 cache keeps track of the expiration time of each cache block. When L2 cache receives a read request, it updates the expiration time of the block's lease, so that the new request could access it. A write request is sent directly to the L2 cache where it can be performed only when the leases of all private copies of the block expire. TC also implements a version that relaxes the write atomicity (TC-Weak) which eliminates write stall and postpones any possible stall to explicit memory synchronization operation (memory fence). Write acknowledgment in TC-Weak returns the time at which the write will become visible to all other SMs. These times are tracked by *Global Write Completion Time (GWCT)* counters for each warp. A memory fence operation uses GWCT to stall warps until all previous writes by that warps are globally visible.

While TC solves some of the challenges in providing coherence to GPUs it suffers from several implementation related challenges.

1) Globally Synchronized Clock: TC uses globally synchronized counters to drive coherence decisions (e.g. self-invalidation) and avoid coherence traffic. Each private cache and shared cache partition maintain its own synchronized counter and all counters are clocked by an independent clock. Relying on synchronized counters in all private and shared caches to make coherence decisions raises scalability concerns. With the growth in GPU chip size and increase of the clock speed, the signal used to clock the synchronized counters can suffer from clock skew and may also lead to extra power consumption for the synchronized clock tree. The clock skew can be aggravated by the increase of clock speed and die area [11], which will in turn affect the correctness of the protocol.

2) Cache Inclusion: Current GPUs do not enforce cache inclusion. TC relies on L2 cache to maintain the lease term of each private L1 cache copy. This approach forces L2 to be inclusive cache. In the absence of cache inclusion one approach to maintain the lease information is to maintain the lease terms in memory. But adding lease information to memory at the granularity of a cache block size is prohibitively expensive, in terms of area. One option to reduce the area cost is to maintain lease expiry information at a coarse-granularity, say at a page level, rather than at the cache block granularity in memory. However, a coarse grained lease counter must be updated to the latest lease expiry time of any cache block within that larger block. Hence, the lease validity times may be unnecessarily increased for all cache blocks in that coarse granular block. The consequence is that when the original block is fetched back the counter (which is modified by some later evictions) can stall the write to the same cache block for a longer period unnecessarily.

To avoid these drawbacks TC assumes inclusive cache, which reduces the effective cache size and could eventually affect cache performance. It is also incompatible with the common assumptions about GPU cache [20], [21], because inclusion is normally not enforced.

3) Lease-Induced Stall and Contention: In TC, when the lease of a cache block is not expired, the writes to the block in L2 need to be delayed until the lease is expired. When a write is delayed, all subsequent reads are delayed until the write is performed. The waiting reads then increase the occupancy of the input queue of the shared cache.

Delayed eviction in L2 caches (due to the inclusion requirement discussed above) can cause similar problem. A cache block with an unexpired lease forces the replacement policy to chose a different victim cache line. If all cache lines in a set have unexpired leases then the replacement process also stalls. Stalls in L2 cache can affect the capability of the GPUs to exploit memory level parallelism which is critical to hide memory latency.

III. G-TSC: GPU Cache Coherence Using Timestamp Ordering

A. Timestamp Ordering

The fundamental reason that TC suffers from the various drawbacks is that the writes need to wait for the unexpired leases. We argue that it is possible to achieve all the benefits of TC without introducing stalls and weakening the semantics.

The key to achieving these desirable properties is *timestamp ordering*. Timestamp ordering is combination of timestamps and physical time used to define the order of memory operations. It is formulated as $Op_{1\longrightarrow} \quad Op_{2\Longrightarrow} \quad (Op_1 <_{ts} Op_2)\ or\ (Op_1 =_{ts} Op_2\ and\ Op_1 <_{time} Op_2)$ where Op_1 and Op_2 are memory operations (load or store), $\longrightarrow$ indicates the order of memory operations, $<_{time}$ means that the operation on the left happened before the operation on the right in physical time, and $<_{ts}$ means that the operation on the left has a timestamp smaller that the operation on the right.

When the timestamps of two memory operations are the same, the physical time is used to order them. It is different from the time-based ordering used by TC, which always uses physical time to order global memory operations: $Op_1 \rightarrow_{mem} Op_2 \Rightarrow Op_1 \rightarrow_{time} Op_2$ where $\rightarrow_{mem}$ indicates global memory ordering while $\rightarrow_{time}$ indicates the order of executing

the operations in time. In timestamp ordering, the global time is only used to order memory operations from the same thread.

The key property of timestamp ordering is the capability to *logically schedule an operation in future by assigning a larger timestamp*. This largely eliminates the lease-induced stalls in TC, as a write could be performed long before the read lease expires but logically it can still happen after the read. Tardis [12] is a previously proposed coherence protocol for CPUs that uses timestamp ordering. In this work, we build on Tardis and design timestamp coherence for GPUs, called G-TSC.

B. Timestamps in GPUs

In G-TSC, each cache block (*C*) in the private and shared caches is associated with two timestamps: a read timestamp (*C.rts*) and a write timestamp (*C.wts*). The timestamps are kept as logical counters. *C.wts* represents the timestamp of the store operation that produces the data in *C*. *C.rts* represents the timestamp through which the data could be correctly read from *C*, after this, the data could be changed. Conceptually, the period between *C.wts* and *C.rts* is a read-only period in which the data in *C* is guaranteed to be valid for the local threads in the SM. We call this period as the *lease*. Each private cache keeps a warp timestamp table (*warp_ts*), where warp *i*'s timestamp is recorded as *warp_ts*$_i$. The timestamp of each warp represents the *conceptual* timestamp of the last memory operation performed by that warp. The shared cache keeps a memory timestamp (*mem_ts*). *mem_ts* keeps track of the maximum *rts* of all cache blocks evicted from the shared cache.

Memory operations can be conceptually ordered using timestamps. It is denoted as OP_{ts} which can be LD_{ts} (for load) or ST_{ts} (for store). All *mem_ts* and *warp_ts* are initially set to 1. *wts* and *rts* are set to (*mem_ts*) and (*mem_ts* + *lease*) when the data is fetched from DRAM.

C. Principles of G-TSC

G-TSC constructs a concurrent system with timestamp ordering such that the load value and write order are consistent with the timestamp order. For example, consider a *load*[*A*] and a *store*[*A*] (produces value 1), assuming that the initial value at *A* is 0. If *load_ts* = 10 and *store_ts* = 8, then the *load* must return 1, because it logically happens after the *store* according to the timestamp, even if according to physical time, the *load* is issued from a warp earlier. If *load_ts* = 8 and *store_ts* = 10, then the *load* must return 0. In essence, G-TSC attempts to *assign* the timestamp to each memory operation, so that the returned values are consistent with the assignments.

Without conflicting memory operations from different warps, each warp monotonically increases its own *warp_ts* and assigns it to each memory operation issued. However, this "default" assignment may not fit into the current state of the system. In order to satisfy coherence, the protocol continuously *adjusts* the assignment to memory operations (LD_{ts} and ST_{ts}) and *warp_ts* as we describe is details in the next section.

IV. G-TSC Implementation

In this section, we discuss the implementation of G-TSC. Our protocol is specified by: *1)* The operations in private L1 after receiving the requests from the SM; *2)* The operations in shared L2 cache. *3)* The operations in private L1 after receiving the response from shared L2;

A. Private Cache Operation

Figure 1a shows the finite state machine of the L1 cache and its transitions. We will explain these states and transitions in the following sections. Note that *PrRd* and *PrWr* are generated by the SM (similar to processor read and processor write in traditional CPU coherence transition diagrams), *BusRd* and *BusWr* are generated by the L1 cache, and *BusFill*, *BusWrAck*, and *BusRnw* are generated by the L2 cache (and delivered through the interconnection network).

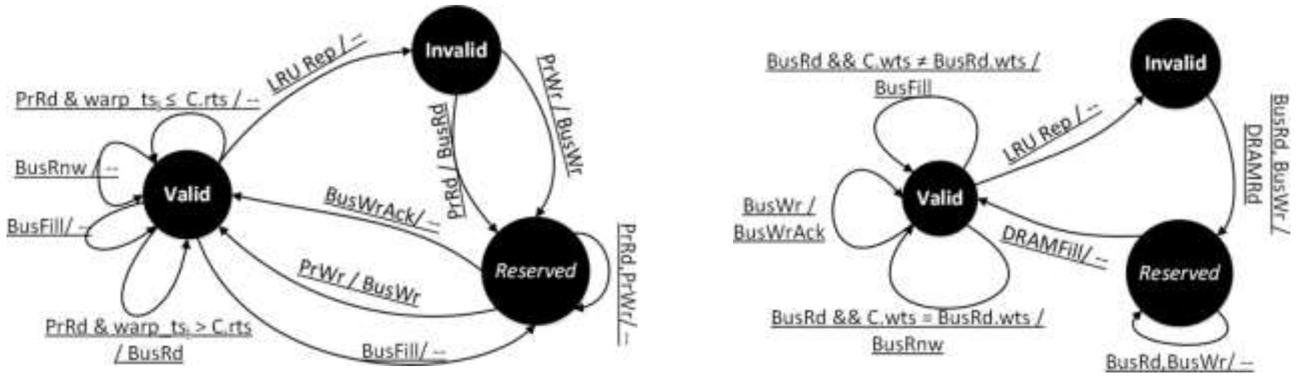
1) Load: Figure 2 shows the flowchart of a load request processing in L1 cache. When a load address has a tag match in cache then the cache line where the tag match occurred is represented by *C* and C.wts and C.rts represent respectively, the write timestamp of the data in that cache line, and the read timestamp assigned when that cache line was fetched previously. The load access is then represented by a tuple *<C, C.DATA, C.wts, C.rts>*.

An access to a cache block in L1 cache results in a hit if it fulfills two conditions: 1) pass the tag check, and 2) the *warp_ts*$_i$ is less than or equal to *C.rts*, where *warp_ts*$_i$ is the timestamp of the warp that generated the load operation. An access that fulfills both conditions results in a *hit* and it may update the *warp_ts*$_i$ to *Max*(*warp_ts*$_i$, *C.wts*). If the access fails to fulfill any of these conditions, a read request *<BusRd, BusRd.wts, BusRd.warp_ts>* is sent to L2 cache. The value of *BusRd.wts* is set to 0 if the requests failed in the tag check, otherwise it is set to *C.wts* if there is a tag match but its lease has expired. The value of *BusRd.warp_ts* is set to *warp_ts*$_i$.

2) Store: Figure 3 shows the flowchart of a store request processing in L1 cache. Since L1 cache is a write-though cache, all store requests (*PrWr*) are processed in the L2 cache. First, if the address hits in the L1 cache the L1 cache block data is updated, but all further accesses to the data from the SM are blocked (further elaboration in section V-A). After that, a write request *<BusWr, BusWr.warp_ ts, BusWr.Data>* is sent to the L2 cache where *BusWr.warp_ts* is set to *warp_ts*$_i$ and *BusWr.Data* holds the store data.

B. Shared Cache Operation

Figure 1b shows the finite state machine of the L2 cache and its transitions. We will explain these states and transitions in the following sections. Note that *BusRd* and *BusWr* are generated by L1 cache and received through the interconnection network, *DRAMFill* is generated by the DRAM, *DRAMRd* is generated by the L2 cache and sent to the DRAM, and *BusFill*,



(a) The FSM Actions in L1 Cache.

(b) The FSM Actions in the shared L2 Cache.

Fig. 1: The Finite State Machine of both L1 and L2 Caches. The prefix *Pr* denotes the messages received from the SM, *DRAM* denotes the messages received from the DRAM and *Bus* denotes the messages exchanged with the NoC.

Fig. 2:

ɔm SM

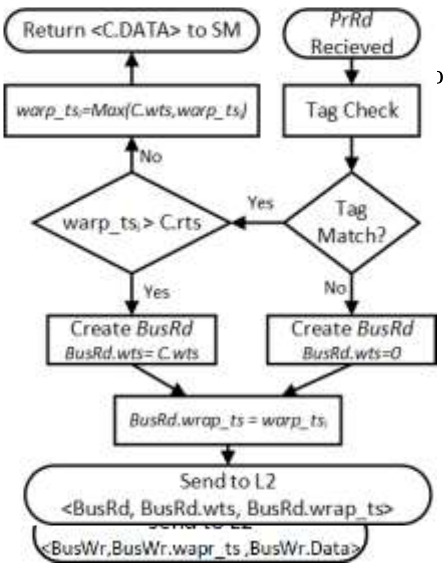


Fig. 3: The Flowchart of the Store Request From SM

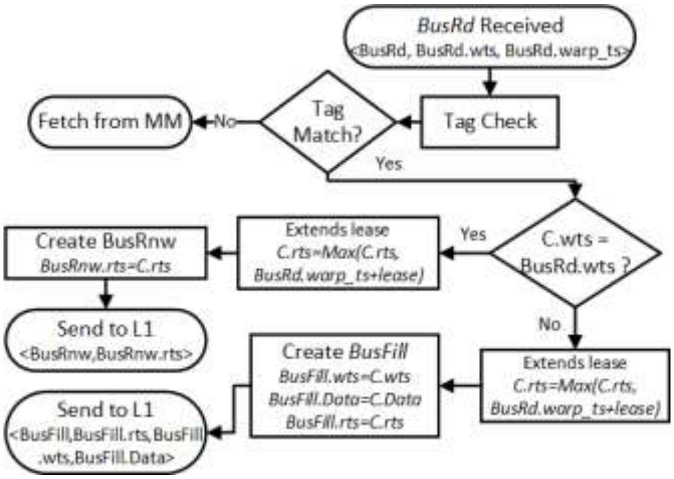


Fig. 4: The Flowchart of the Read Request from L1 Cache

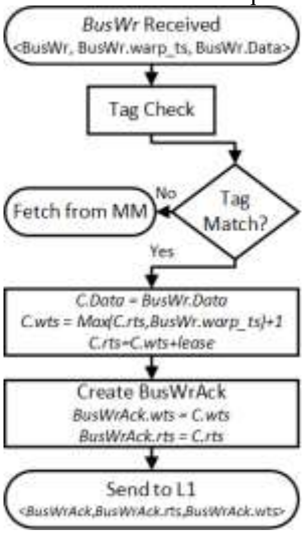


Fig. 5: The Flowchart of the Write Request from L1 Cache

BusRnw, and *BusWrAck* are generated by the L2 cache and sent to L1 cache through the interconnection network.

1) Loads from L1: The flowchart of processing a read request in shared cache is shown in figure 4. If the read address hits in L2 cache block, then the wts in the request (*BusRd.wts*) is checked against the wts in the cache block and if they match then a renewal response is sent back the requester with an updated rts. This is the case when data has not been updated in L2 after the last write that is seen by the private L1. But the lease in L1 has expired and it simply needs

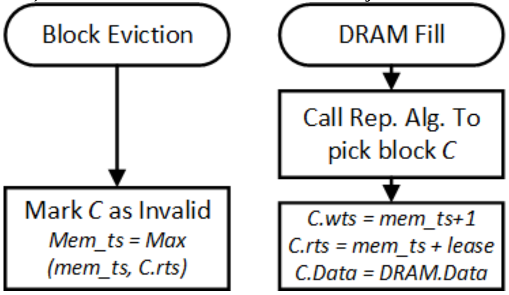


Fig. 6: The Flowchart of DRAM Fill and Eviction

to be renewed.

If the *BusRd.wts* does not match wts in the cache block, it implies that the data is in fact updated by another SM after the requesting SM's lease has expired. Hence, a fill response is sent to the requester including the new data, the wts of the new data, and updated rts as shown in the flow chart.

2) Stores from L1: The processing of a write request (*BusWr*) is described in figure 5. The wts of the new data is calculated based on the stored value of *rts* and the received value of *warp_ts* as shown in the flow chart. After calculating the value of wts, the value of the new value of the rts is also calculated and both timestamps are sent back to the requester with the acknowledgment response.

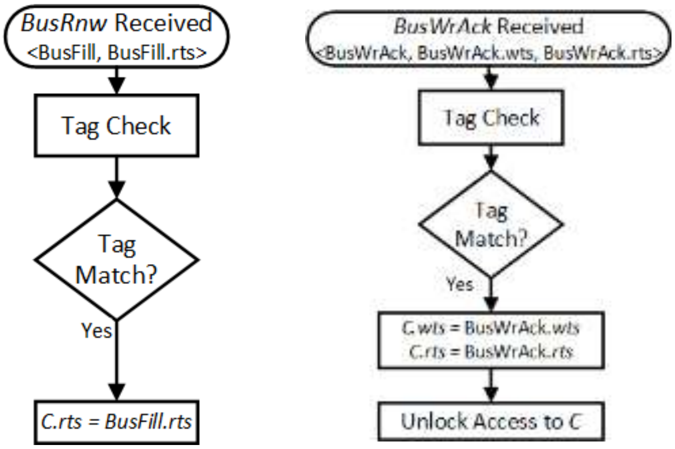
Finally when the L2 receives a request (either *BusRd* or *BusWr*) for a cache block that is not present in the cache, both load and store will trigger a read request (*DRAMRd*) sent to the GDDR DRAM.

C. DRAM Operation

Figure 6 shows how the shared cache handles the DRAM fills and block evictions. When a block is filled from DRAM, *C.wts* and *C.rts* are set based on *mem _ts* and *mem _ts + lease* respectively. On the other hand, when a cache block is evicted from L2, *mem_ts* needs to record the evicted block's expiration time, so that when later the block is fetched back, L2 could assign timestamps to future stores correctly. Upon eviction, the value of *mem_ts* is set to *Max*(*mem* ts_o, C_e.*rts*) where *mem* ts_o is the original value of *mem_ts* and C_e.*rts* is the *rts* of the evicted cache block. As we mentioned, even though all evicted blocks share the same *mem_ts*, it is not an issue for G-TSC, because the timestamp ordering could always logically order stores to a point in future without stall.

D. Private Cache Operation After Response from Shared Cache

Figures 7 and 8 show how the private cache handles the responses from shared cache. The private cache receives a renewal response <*BusRnw, BusRnw.rts*> when it already has the updated version of the data. In this case, it extends the current lease of the block to the rts value in the response. However, a write acknowledgment <*BusWrAck, BusWrAck.rts, BusWrAck.wts*> means that a store operation is completed and a new values of wts and rts has been assigned.



(a) The Flowchart of Renewal Response from L2.

(b) The Flowchart of Write Acknowledgment from L2.

Fig. 7: Flowcharts of Private Cache Operation.

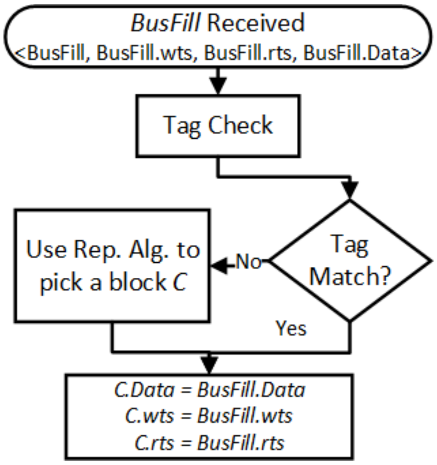
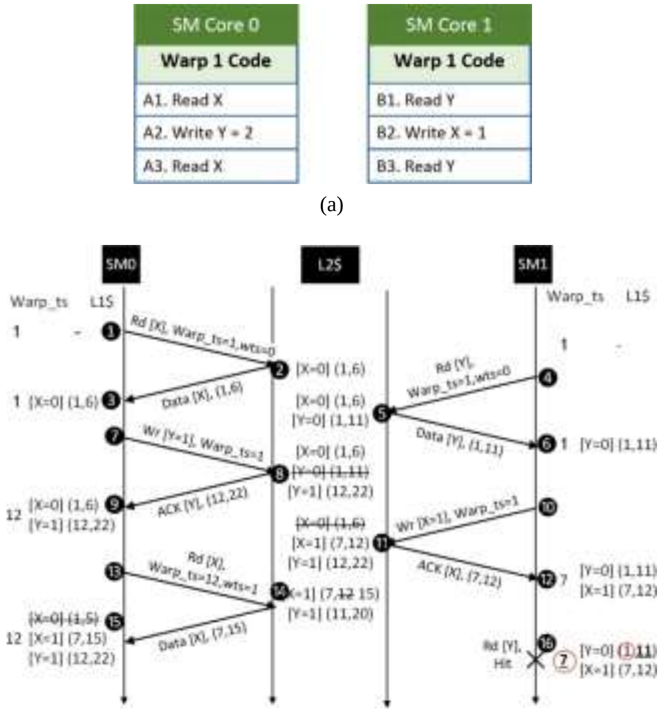


Fig. 8: The Flowchart of Fill Response from LLC

Hence, the private cache needs to update its local information and unlock the block so other warps can access it. A fill response <*BusFill, BusFill.wts, BusFill.rts, BusFill.Data*> can either fill a new block or update an existing block with new data. The private cache should probe the tag array to get the older version of the block, or allocate a new cache block for the incoming block by using the replacement algorithm. The data, rts and wts are copied from the response to the cache block allocated (Figure 8).

We will explain the operations of G-TSC with an example. Assume two warps are being executed in two different SMs where the first one reads some memory location [*X*], writes to another memory location [*Y*] and then reads the [*X*] again. The other warp reads [*Y*], writes to [*X*], and then reads [*Y*] again. The sequence of instructions for both warps are shown in figure 9a. For the sake of this example, we will assume that there is only one warp executed in each SM. The execution sequence for all instructions is shown in figure 9b. The read operation (A1) that tries to read location [*X*] misses in L1 cache and hence the read request is sent to the lower-level



(a)

(b)

Fig. 9: G-TSC Operation Example. The contents of the caches of each SM is shown with the wts and rts of each block in the parenthesis.

cache (1). The request contains the address ($addr = X$), the warp timestamp ($warp\_ts = 1$) and the write timestamp ($wts = 0$) which is set to zero since the block is not present in L1 cache. The block is fetched from the main memory and placed in L2 cache 2 and then is sent to L1 cache with a lease period ([1,6]) as shown in 3 . Instruction (B1) that reads address [Y] follows the same steps as shown in steps 4 , 5 , 6 . We assume a longer lease period for Y for the sake of explanation. The protocol works with any lease value. When SM0 executes the write instruction (A2), the writing operation should be performed at the shared cache. Hence the write is sent to L2 cache with the warp timestamp ($warp\_ts = 1$) 7 . Based on the information in L2 cache, the system knows that the block is valid in some private cache until timestamp 11 (SM1 cache in this case) so it assigns a write timestamp after that lease period ($ST\_ts = 12$) 8 and sends an acknowledgment to L1 cache with the new lease period ($wts = 12$, $rts = 22$) 9 . The timestamp of $warp_1$ that issued the write operation is adjusted to 12 to match the timestamp of the writing operation 9 . Instruction (B2) follows the same steps that are shown in 10 , 11 , 12. After that, SM0 tries to execute instruction (A3) and read X. Even though X is present in the cache but the timestamp of the reading warp ($warp\_ts = 12$) is beyond the lease of address X ([1,6]) 13. So a renewal request is sent to L2 cache containing the write timestamp of X ($wts = 1$) along with the timestamp of the reading warp (($warp\_ts = 12$)) 13.

TABLE I: Contents of Requests and Response Exchanged Between Private and Shared Caches.

Message Type	rts	wts	warp ts	data
Read/Renewal Requests (*BusRd*)		√	√	√
Write Request (*BusWr*)	√	√		√
Fill Response (*BusFill*)	√			
Renewal Response (*BusRnw*)	√	√		
Write Acknowledgment (*BusWrAck*)				

against the actual write timestamp of the address in the cache ($wts = 7$). Since they do not match, it is clear that a new write has occurred after the last value X was seen by SM0. Then the L2 cache sets the new lease of X to be 15 which is greater than the timestamp of the reading warp 14, thereby giving the reading warp an opportunity to read the data. The new data and extended lease period are sent to L1 cache 15 of the requester. When instruction (B3) tries to read Y in SM1, it hits in the cache 16. Note that the timestamp of L2 cache checks the write timestamp in the renewal request the reading warp (*warp ts* = 7) fall within the lease period ([1,11]) hence the read is performed.Based on the timestamp ordering, the order of the executed instructions in this exampleis (*A*1 *B*1 *B*2 *B*3 *A*2 *A*3).

To summarize, Table I shows the contents of different messages exchanged in G-TSC.

V. GPU-Related Considerations

The above state transition description shows how logical time based coherence can be applied within the context of GPUs. In this section we discuss GPU-specific considerations that need to be addressed for achieving good performance.

A. Update Visibility

The L1 cache in GPUs is shared between thousands of threads, and to ensure correctness, an updated data block should not be accessible by other threads until the store is completed and acknowledged. With timestamp ordering, a store operation is not completed until its timestamp is determined.

Figure 10 illustrates this issue with an example. In this example, we will show how poor management of the updated data can affect the correctness of the coherence protocol and cause a coherence violations. Initially, the cache block *A* hasa lease period [1, 5] ([*A.wts, A.rts*]). In step (2), warp 1 attempts to write *A*. According to the validity information available in the private cache, the timestamp of the store operation (ST_{ts}) is set to 6 and the warp timestamp (*warp ts*) and write timestamp (*wts*) are updated accordingly. The write is sent to L2 and L1 waits for the acknowledgement. Beforethe acknowledgement which will contain the lease that L2 assigns to the new data, both *A.wts* and *A.rts* are set to 6. At this point, L1 only knows that the start of the lease will beat least 6. In step (3), warp 2 with *warp ts* = 1 reads *A* and its own *warp ts* is updated to 6 meaning that the timestamp of load operation is set to 6. In step (4), the acknowledgement from L2 for the store operation from warp 1 arrives, and theassigned lease is [11,20]. The start of the lease is greater than 6, and the lease of *A* in L1 is updated to [11,20]. At this point,

_

→ → → →
→

_

_

_

we can see that the timestamp of the read from warp 2 in step (3) is 6, which is less than the lease of warp 1's write in global order ([11,20]). It means that the write is performed at logical time with timestamp 11, but warp 2 already observed it at an earlier logical time with timestamp 6. Essentially, a read observes a value that will be produced a later logical time. It is a violation of coherence.

Intuitively, there are two ways to resolve this problem: *1)* delay all accesses to the updated data until the store operation is globally performed and acknowledged; or *2)* keep the old copy along with the new one and allow accessing the old copy until the store is globally performed. For 1), an MSHR entry is allocated for read requests as if they are read misses, and they are granted access to the data as soon as the store is acknowledged. At this point, the timestamp is determined and the *warp _ts* of the reading warp is updated accordingly. For 2), a hardware structure is needed to hold the old data while the store is pending. Moreover, we also need to ensure that the writing warp can only read the new data that it generates after the write is being globally performed.

Note that it is not an issue for Simultaneous Multithreading (SMT) [22] processors with conventional coherence protocol and write atomicity. Because before the write is globally performed, the new value is in the processor's write buffer and the old value is in the L1 cache. The other threads in the same processor can bypass write buffer and directly obtain the old data from L1 cache, this ensures write atomicity. If write atomicity is not supported, the threads in the same processor could read the new values from write buffer. However, conventional protocol never allows the read to observe new value before it is produced (as opposite to the example in Figure 10). Using a write buffer in GPUs increases the hardware complexity of the LDST unit and has a high area overhead. A single store instruction generates 2-4 memory write operations on average. With 48-64 concurrent warps executing the same code, those warps are expected to hit the same store instruction within a small time window meaning that the write buffer need to deal with ∼200 outstanding write requests per store instruction.

In this paper, we evaluated both approaches. Different from TC, we found that option 1 gives the better trade-off in GPUs. The performance overhead of delaying accesses to updated block is negligible, so we do not need to pay for the hardware cost for keeping multiple copies.

B. Request Combining in GPUs

The second challenge is the validity of the data serviced by L2 cache requested by multiple threads.

When multiple read requests from different warps with different *warp _ts* in the same SM try to access a cache block that is not present in L1 cache, these requests can be either all forwarded to L2 cache or just the first request is forwarded, with the hope that the other *warp_ts* will fall in the lease and be able to access the data. The two options indicate a trade-off between coherence traffic and performance. Forwarding all requests to L2 cache increases the traffic but assures that the requests are serviced as soon as the responses are returned from L2 cache. Forwarding only the first request and keeping the remaining requests in the MSHR preserves the bandwidth but may increase the latency of some requests if the allocated lease window cannot cover their *warp_ts* and incur additional renewals. This issue is significant in GPUs since the NoC bandwidth is one of the performance bottlenecks as shown in [13]. The choice between forwarding all requests and keeping them in MSHR has a significant impact on the performance since the latency of the NoC increases with the increase of the memory traffic generated by the SMs [23]. Forwarding all requests to L2 cache can increase the number of memory requests sent by SMs by 12% to 35% on average.

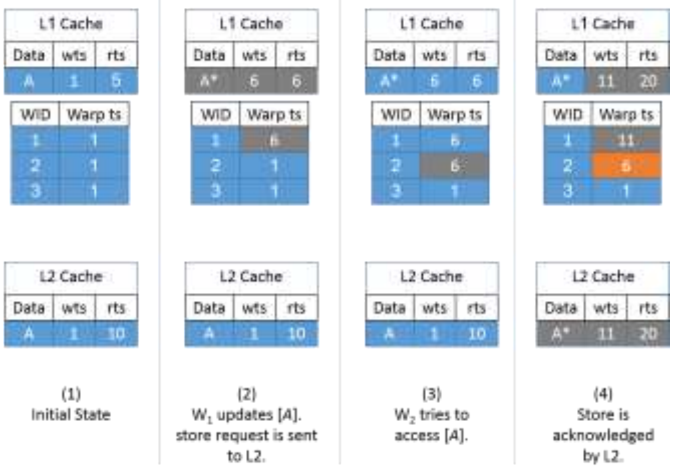


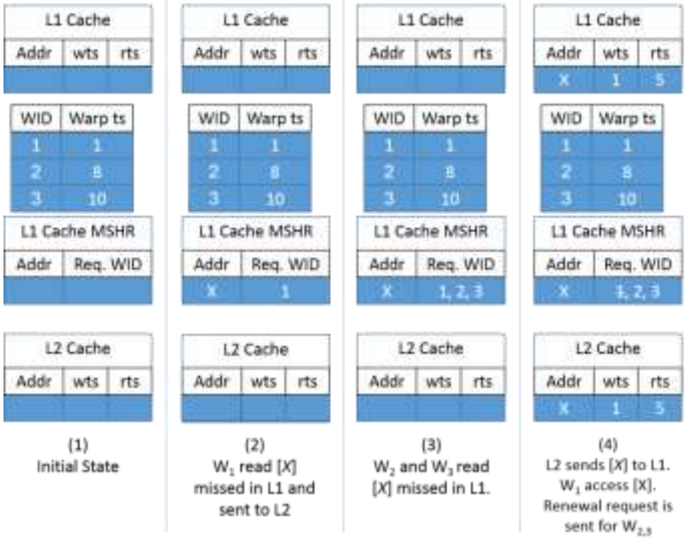
Fig. 10: Example of Update Visibility Challenge in GPUs

Consider the example in Figure 11. In step (2), a read request is sent to L2 with the *warp ts* of warp 1. In step (3), warp 2 and 3 try to read the same block. Assuming we only send one request, they do not generate extra messages from L1 to L2. Later in step (4), the response gives L1 the lease window [1,5], warp 1's request is removed from MSHR. Unfortunately, it is not sufficient for the other two requests so we need to send a renewal request for them and they still remain in L1's MSHR, see step (5).

In our approach we chose to keep the requests in MSHR and then send a renewal request in case the lease term expires before the waiting request can read the data. Where extra renewal requests are sent we still end up with saving some bandwidth because a renewal request generally has a smaller data response packet since the response from L2 contains the renewed lease information when no stor has been performed in the interim.

C. Non-Inclusive Caches in GPUs

As discussed in Section II-D2, TC has to force inclusion and incur delayed eviction. In timestamp ordering, it is possible to maintain only *one* timestamp in memory for the evicted blocks without introducing unnecessary stall, since timestamp ordering makes it possible to logically schedule an operation to happen in future by assigning a larger timestamp. Therefore, even if the timestamp in memory is increased by other evictions, a conflicting store can execute without stall by



(1)
Initial State

(2)
W_1 read [*X*] missed in L1 and sent to L2

(3)
W_2 and W_3 read [*X*] missed in L1.

(4)
L2 sends [*X*] to L1. W_1 access [X]. Renewal request is sent for $W_{2,3}$

Fig. 11: Example of Multiple Requests Challenge in GPUs

assigning a larger timestamp greater than the single coarse-grain timestamp stored in memory. Using timestamp ordering, we can support non-inclusive policy, which is compatible with current GPUs, and avoid the delayed eviction.

D. *Timestamp Overflows*

The experiments based on our benchmarks show that 16-bit timestamp is enough for all executions to make timestamp counter wrap-around sufficiently rare. Note that the L1 cache is flushed after each kernel and all timestamps are reset. In case of timestamp wrap around, the timestamp overflow mechanism can be handled at the L2 cache. The overflow can occur due to lease extension or assigning a timestamp to a new store operation (these are the only operations that increase the timestamp). The timestamps at the L1 caches are reflection of the timestamps assigned by the L2 cache (L1 cache does not increment the timestamps by itself). When a timestamp update causes an overflow, the L2 cache bank sends a reset signal to all L2 cache banks and then reset its timestamps. Upon resetting the timestamps in L2 cache bank, the write timestamp of all blocks is set to 1 and the read timestamp is set to (lease) and the memory timestamp is set to 1. Since the L2 cache has the up-to-date data of all blocks, there is no need to flush the cache. After resetting all timestamps, the L2 cache responds to every request with timestamp with a large value with a fill response along with the data even if the request is for a renewal. It also includes a timestamp reset signal with the response to inform the L1 cache that the timestamp is reset. When L1 cache receive a response with a reset message, it flushes its blocks and reset warp timestamp and then access the new data.

For comparison, TC uses a 32-bit local timestamp for each L1 cacheline, a 32-bit global timestamp for each L2 cacheline, a 32-bit entry per warp in the GWCT table and a 32-bit counter for each L1 and L2 cache.

VI. Evaluation and Discussion

A. *Evaluation Setup*

We implemented G-TSC in GPGPU-Sim version 3.2.2 [13]. We used GPUWattch [24] to estimate the power and energy consumption. The simulated GPU has 16 SMs, with 48KB shared memory, and 16KB L1 cache each. Each SM can run 48 warps at most with 32 threads/warp. L2 cache is partitioned into 8 partitions with 128KB each (1MB overall). In our evaluation, we used two sets of benchmarks: the first set requires cache coherence for correctness, and the other does not. The second set of benchmarks are used to show the impacts of coherence protocol on them due to the protocol overheads.

The performance of G-TSC is compared against TC. We implemented TC on GPGPU-Sim simulator and all the results presented in the paper are based on our implementation of TC on GPGPU-Sim. But to validate that our implementation of TC closely matches the original implementation we also ran TC on the same benchmarks with the same configuration setting using the original simulator used in the TC paper [9]. Table II shows the execution time of TC on our G-TSC simulation infrastructure (column four) and the execution time of TC on the original simulator (column five). As can be seen the two simulators provide very similar execution times. The few differences that are present may be attributed to the fact that the original TC used Ruby [25] to implement its cache and memory system, while we enhanced the default memory system implemented in GPGPU-sim for implementing the G-TSC memory system.

TABLE II: Absolute Execution Cycles of TC and Baseline (BL) in Millions

Benchmark	BL (G-TSC simulator)	BL [9]	TC (G-TSC simulator)	TC [9]
BH	0.55	1.26	0.84	1.03
CC	1.47	2.99	1.77	1.75
DLP	1.63	5.53	1.63	1.44
VPR	0.85	1.98	0.90	0.77
STN	2.00	4.66	1.74	1.62
BFS	0.79	1.95	2.32	1.87
CCP	13.50	13.59	13.50	13.47
GE	2.22	4.89	2.49	3.51
HS	0.22	0.22	0.23	0.23
KM	28.74	30.89	30.78	34.17
BP	0.84	1.61	0.69	0.58
SGM	6.08	5.74	6.14	5.91

We also simulated the baseline (BL) configuration, which essentially turns off the private cache to provide coherence, both on the original TC simulator and our G-TSC simulator. Table II shows the execution time of BL on our G-TSC simulation infrastructure (column two) and the execution time of BL on the original TC simulator (column three). The baseline execution times differ in the two models. We believe that the difference stems from how the two simulators implement *no L1 cache* design in the simulator. G-TSC implements BL by essentially sending all requests directly to the L2 cache over the NOC and assumes that there are no L1 cache tags to be checked or L1 cache MSHRs to be updated. Hence, any relative performance improvements over the baseline model reported in the original TC paper and our paper may be different. From here on we report all results relative to our baseline implementation on our simulation infrastructure. We implemented G-TSC and TC with SC and RC memory models.

B. *Performance Evaluation*

Figure 12 shows the performance (execution cycles) of G-TSC and TC with RC and SC normalized to the performance of coherent GPU with L1 cache disabled (therefore enforcing coherence through the shared L2 cache). There are two sets of benchmarks. The first set shown in the left cluster are benchmarks that require coherence and will not function correctly without it. The benchmarks in the right cluster do not require coherence. Hence, we show one new performance bar (the left most bar titled Baseline W/L1) using a baseline with L1 cache since they do not need coherence and can take advantage of L1 cache in the baseline.

The higher bars in Figure 12 indicate better performance. Our results show that the performance difference between RC and SC with G-TSC is smaller than the difference between RC and SC for TC. G-TSC does not incur much stall time due to unexpired leases, as opposed to TC. Hence, the difference between SC and RC with G-TSC is small, sometimes even negligible (e.g. BH, BFS and most of the applications that do not require coherence as shown in the right cluster). For G-TSC, benchmarks that requires coherence obtain 12% speedup with RC compared to SC. The overall average speedup is around 9% over all benchmarks.

Interestingly, for one benchmark (CC), SC is better than RC in G-TSC˙G-TSC-SC outperforms G-TSC-RC sometimes (e.g. CC) because the NoC traffic is limited by the fact that in SC only one outstanding memory request per warp is allowed. While RC could eliminate certain warp stalls, but it generates more coherence messages and allows more requests into NoC, which happens to have more negative impact on performance in CC. As a result, the average network latency goes down and the memory requests can be serviced faster in SC. In CC, we indeed confirm that the average network latency per request in G-TSC-SC is 29% lower than G-TSC-RC due to a 14% reduction in memory request rate generation. Previous work [13] showed the similar behavior.

G-TSC is able to achieve about 38% speedup over TC with RC; and about 84% speedup over TC with SC. G-TSC with SC outperforms TC with RC by 26% for benchmarks that require coherence for correctness. These significant performance improvements are mainly due to G-TSC's ability to avoid warp stalling caused by delayed writes and eviction. G-TSC also avoids the stalls caused by GWCT in TC before executing fence instructions. These stalls aggravate the performance penalty in SC since each warp is allowed to have at most one outstanding memory request.

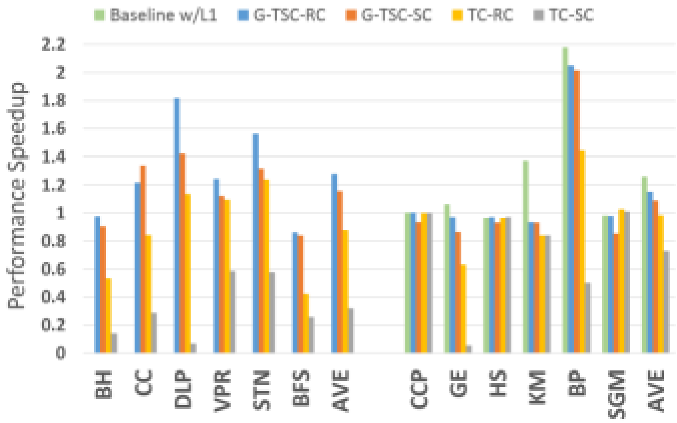


Fig. 12: Performance of GPU Coherence Protocols with Different Memory Models

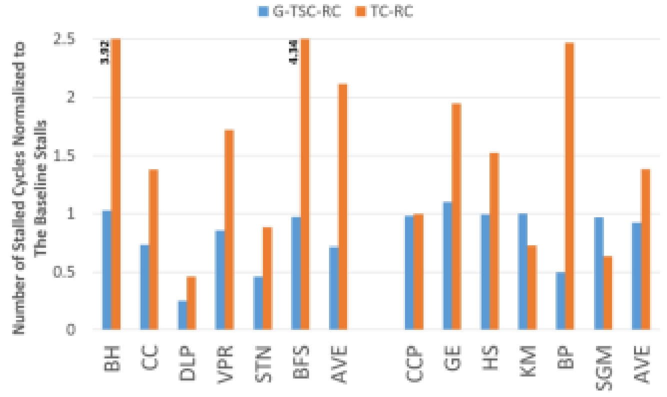


Fig. 13: Pipeline Stalls due to Memory Delay in G-TSC and TC Normalized to Stalls in No-L1-Cache Configuration

Benchmarks like CCP, HS, and KM (that do not require coherence) do not exhibit significant difference in performance between G-TSC and TC and between SC and RC. These benchmarks are compute-intensive benchmarks so the stalls imposed by the coherence protocols or consistency model requirement are overlapped with execution of other non-memory instructions.

Figure 13 plots the pipeline stalls due to memory delays normalized to baseline with no L1 cache configuration. The results shows that TC encounters around 45% more stalls than G-TSC for the first set of benchmarks and more than 1.4 x stalls for the second set of benchmarks.

The performance of GPU with L1 cache is also presented in figure 12 to show the performance overhead of G-TSCfor benchmarks that do not need coherence. We report the performance of the second group of benchmarks only since the presence of L1 cache with no coherence (which is the case here) affects the correctness of the first group of benchmarks. The figure shows that G-TSC overhead is around 11% with respect to the non-coherent GPU. It also shows that G-TSC can perform as good as the non-coherent GPU in most of the cases (e.g. CCP, GE, HS and SGM).

Figure 14 shows the performance of G-TSC with different lease periods with RC. G-TSC shows small sensitivity for lease values variation. This insensitivity is because lease period in G-TSC is not related to the physical time; it represents the logical time. Intuitively only very small and very large lease values may impact G-TSC. Small lease values can affect performance because of the excessive renewal requests. It also may aggravate the multiple reader issue discussed in V-B. Large leases cause the timestamp to roll faster and reduce the chance that multiple warps could access the cache block during its lease before renewal. But for a range of lease periods that we explored (8-20 cycles) G-TSC performance is unchanged.

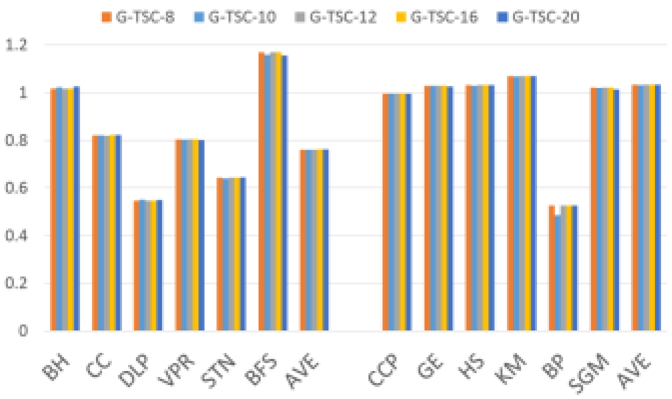


Fig. 14: Performance of G-TSC-RC with Different Lease Values

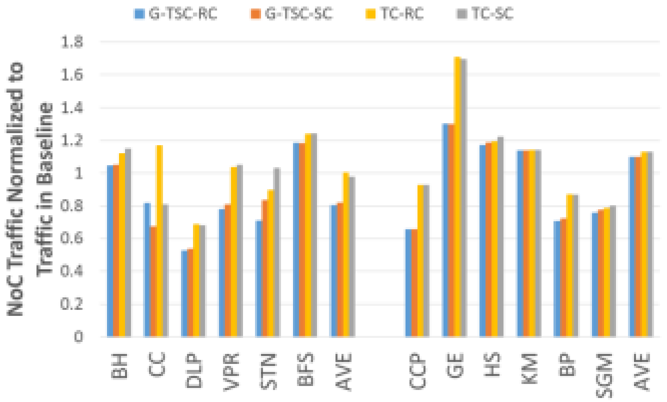


Fig. 15: NoC Traffic of GPU Coherence Protocols with Different Memory Models

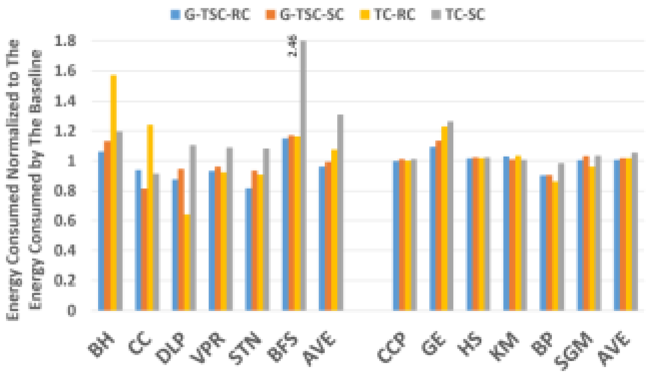


Fig. 16: Total Energy Consumption of GPU Coherence Protocols with Different Memory Models

C. Coherence Traffic

Coherence traffic in G-TSC and TC is mainly due to the lease renewal requests in L1 cache or fetching new data to replace old data. Since G-TSC is conducting its coherence transactions in logical time, it is able to reduce the coherence traffic compared to TC which operates coherence transactions in physical time. Since logical time in G-TSC rolls slower than the physical time, more load operation are able to access the cache block during its lease period in L1 cache. This reduces the number of renewal requests.

Another optimization for NoC bandwidth usage is that renewal response in G-TSC does not require sending the data again. Figure 15 shows the traffic in NoC for G-TSC and TC with RC and SC memory models normalized to the NoC traffic in a coherent GPU with no L1 cache. We see that G-TSC is able to reduce the traffic by 20% over TC with RC and 15.7% with SC for the first set of benchmarks. Note that the NoC traffic is almost the same for RC and SC in both G-TSC and TC for the second set of benchmarks; these benchmarks do not generate coherence traffic to begin with.

D. Energy

G-TSC is able to reduce the total energy of the GPU since it is able to enhance the performance and reduce the NoC traffic. Figure 16 shows the normalized overall energy consumption of evaluated benchmarks. G-TSC consumed 11% less energy than TC with RC for the first set of benchmarks. RC consumes more energy than SC for some benchmarks, like CC and BH, even though their performance is better. The reason for this behavior is that in SC implementations, the cores remain idle and do not consume much energy (except static energy).

We studied the energy saving of individual components of the GPU, mainly, energy consumed by L2 cache, main memory (DRAM and memory controller) and the interconnection network. G-TSC reduces the energy consumed by the L2 cache by 2%, the NoC by 4%, and the other GPU components by 5%. It also saves 1% more energy for the L2 cache, 3% for the NoC, and 5% for the other GPU components over TC. The

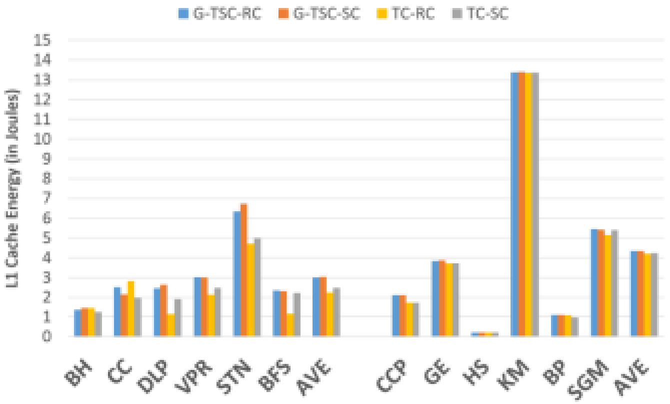


Fig. 17: L1 Cache Energy (in joules) of GPU Coherence

Protocols with Different Memory Models

total energy saving is 11% over the baseline, and 9% over the TC for the first set of benchmarks. The results in figure 16 includes the energy of L1 cache. We also presented the L1 cache energy consumption in figure 17. The figure shows that TC consumed slightly less energy than G-TSC.

We see that in GPUs, SC may not always be a bad choice, because it may offer better performance for certain benchmarks (as discussed before) and incur less energy due to the reduced NoC traffic. With TC, the majority of applications show a big gap between RC and SC. However, G-TSC reduces this gap and makes it much smaller. This motivates supporting SC feasible in GPUs, and some recent works came to the same conclusion [26].

E. Characteristics of G-TSC

Implementing cache coherence in logical time in G-TSC rather than physical time as in TC introduces some advantages. Kernels that have more load instructions than store instructions do not incur cache misses due to lease expiration since their timestamps roll slower. Our experiments show that the number of misses due to lease expiration has dropped by around 48%. This observation allows more accesses to hit in L1 cache which indeed translates into relatively longer lease in physical time. However, multiple results show that G-TSC is insensitive to small variations in lease values. It allows the implementation with relatively small lease values which limits the speed of timestamp rollover.

VII. Related Work

The use of timestamps in coherence protocols has been studied in multiple hardware and software protocols. Lamport [27] is one of the earliest efforts that tried to use logical times to order operations in distributed systems and avoid using synchronized physical clocks. They studied the use of logical timestamps to order operations in distributed systems. De Supinski *et. al.* in [28] evaluated the performance of the late delta cache coherence protocol, a highly concurrent directory-based coherence protocols which exploits the notion of logical time to provide support for sequential consistency and atomicity for CPUs. Min *et al.* [29] proposed a software-assisted cache coherence scheme which uses a combination of a compile-time marking of references and a hardware-based local incoherence detection scheme. Nandy *et al.* [30] is one of the first hardware coherence protocol that uses timestamps. TSO-CC [31] proposed a hardware coherence protocol based on timestamps. It supports total-store-ordering (TSO) memory consistency model and requires broadcasting and frequently self-invalidating cache lines in private caches. TC-Release++ [32] is a timestamp-based coherence protocol for RC that is inspired by TC and addresses the scalability issues of efficiently supporting cache coherence in large-scale systems. TC-Release++ eliminates the expensive memory stalls and provides an optimized lifetime prediction mechanism for CMP.

The previous protocols tightly couple timestamp with physical time. Tardis [12] is a timestamp coherence protocol that is based on logical time rather than physical time. Tardis is designed for CMP and implements SC. G-TSC builds on top of Tardis and focuses on GPU implementation. G-TSC optimizes the protocol requirements to fit the highly multi-threaded GPU cores. An imporved version of Tardis (called Tardis 2.0) [33] implements TSO consistency model and proposes optimized lease policies. Similar to Tardis, Martin *et. al* [34] proposed timestamp snooping scheme where processor and memory nodes perform coherence transactions in logical order. The network assigns a logical timestamp for each transaction and then broadcasts it to all processor and memory nodes without regard for order.

Self-invalidation in private caches has been explored in the context of cache coherence. Dynamic self-invalidation (DSI) [35] reduces cache coherence overhead and reduce invalidation messages by speculatively identifying which block to invalidate when they are brought into the cache but deferring the actual invalidation to future time. DSI still requires explicit messages to the directory to acknowledge self-invalidation. DSI can reduce the traffic by using *tear-off* blocks that are self-invalidated at synchronization instructions. A similar idea is proposed in [36] that extends the tear-off blocks to all cache blocks in order to entirely eliminate coherence directories. Last-Touch Predictors (LTP) [37] triggers speculative self-invalidation of memory blocks in distributed shared memory.

VIII. Conclusion

This paper proposes, G-TSC, a timestamp-based GPU cache coherence scheme that reduces the coherence traffic. Different than the previous work on time based coherence for GPUs, G-TSC conducts its coherence transactions in logical time. We implemented G-TSC in GPGPU-Sim and used 12 benchmarks in the evaluation. When using G-TSC to keep coherence between private L1 caches and the shared L2 cache, G-TSC outperforms TC by 38% with release consistency. Moreover, even G-TSC with sequential consistency outperforms TC with release consistency by 26% for benchmarks that require coherence for correctness. For the same benchmarks, the memory traffic is reduced by 20%.

Acknowledgment

This work is supported by the following grants: NSF-CCF-1719074, DARPAPERFECT-HR0011-12-2-0020, NSF-CCF-1657333, NSF-CCF-1717754 and NSF-CAREER-0954211

References

[1] X. Xie, W. Tan, L. L. Fong, and Y. Liang, “CuMF_SGD: Parallelized Stochastic Gradient Descent for Matrix Factorization on GPUs,” in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2017, pp. 79–92.

[2] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen, “Efficient GPU Spatial-Temporal Multitasking,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 3, pp. 748–760, 2015.

[3] “Khronos Group,” OpenCL, https://www.khronos.org/opencl/.

[4] “NVIDIA Corp.” CUDA C Programming Guide v4.2,, 2012.

[5] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O’Connor, and T. M. Aamodt, “Characterizing and Evaluating a Key-value Store Application on Heterogeneous CPU-GPU Systems,” in *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 88–98.

[6] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated Static and Dynamic Cache Bypassing for GPUs," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015, pp. 76–88.

[7] X. Xie, Y. Liang, G. Sun, and D. Chen, "An Efficient Compiler Framework for Cache Bypassing on GPUs," in *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*. IEEE, 2013, pp. 516–523.

[8] R. Behrends, L. K. Dillon, S. D. Fleming, and R. E. K. Stirewalt, "AMD Graphics Cores Next Architecture, Generation 3," Advanced Micro Devices Inc., Tech. Rep., August 2016.

[9] I. Singh, A. Shriraman, W. W. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 2013, pp. 578–590.

[10] K. S. Shim, M. H. Cho, M. Lis, O. Khan, and S. Devadas, "Library Cache Coherence," MIT, Tech. Rep., 2011.

[11] S. Zanella, A. Nardi, A. Neviani, M. Quarantelli, S. Saxena, and C. Guardiani, "Analysis of the Impact of Process Variations on Clock Skew," *IEEE Transactions on Semiconductor Manufacturing*, vol. 13, no. 4, pp. 401–407, 2000.

[12] X. Yu and S. Devadas, "Tardis: Time Traveling Coherence Algorithm for Distributed Shared Memory," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 227–240.

[13] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 163–174.

[14] Z. Cui, Y. Liang, K. Rupnow, and D. Chen, "An Accurate GPU Performance Model for Effective Control Flow Divergence Optimization," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 83–94.

[15] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang, and D. Fan, "Enabling Coordinated Register Allocation and Thread-level Parallelism Optimization for GPUs," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 395–406.

[16] D. J. Sorin, M. D. Hill, and D. A. Wood, "A Primer on Memory Consistency and Cache Coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011.

[17] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE transactions on computers*, vol. 100, no. 9, pp. 690–691, 1979.

[18] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *computer*, vol. 29, no. 12, pp. 66–76, 1996.

[19] J. Feehrer, P. Rotker, M. Shih, P. Gingras, P. Yakutis, S. Phillips, and J. Heath, "Coherency Hub Design for Multisocket SUN Servers with Coolthreads Technology," *IEEE Micro*, vol. 29, no. 4, pp. 36–47, 2009.

[20] N. Anssari, "Using Hybrid Shared and Distributed Caching for Mixed-Coherency GPU Workloads," Master's thesis, University of Illinois at Urbana-Champaign, 2013.

[21] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive Cache Management for Energy-Efficient GPU Computing," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 343–355.

[22] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism," in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ser. ISCA '95. New York, NY, USA: ACM, 1995, pp. 392–403. [Online]. Available: http://doi.acm.org/10.1145/223982.224449

[23] G. Koo, H. Jeon, and M. Annavaram, "Revealing Critical Loads and Hidden Data Locality in GPGPU Applications," in *Workload Characterization (IISWC), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 120–129.

[24] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *ACM SIGARCH Computer Architecture News*, vol. 41. ACM, 2013, pp. 487–498.

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

[26] B. A. Hechtman and D. J. Sorin, "Exploring Memory Consistency for Massively-threaded Throughput-oriented Processors," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 201–212. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485940

[27] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[28] B. R. de Supinski, C. Williams, and P. F. Reynolds, Jr., "Performance Evaluation of the Late Delta Cache Coherence Protocol," University of Virginia, Charlottesville, VA, USA, Tech. Rep., 1996.

[29] S. L. Min, J.-L. Baer, and M. Mn, "A Timestamp-Based Cache Coherence Scheme," 1989.

[30] S. Nandy and R. Narayan, "An Incessantly Coherent Cache Scheme for Shared Memory Multithreaded Systems," in *International Workshop on Parallel Processing*. Citeseer, 1994.

[31] M. Elver and V. Nagarajan, "TSO-CC: Consistency Directed Cache Coherence for TSO," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 165–176.

[32] Y. Yao, G. Wang, Z. Ge, T. Mitra, W. Chen, and N. Zhang, "Efficient Timestamp-Based Cache Coherence Protocol for Many-Core Architectures," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016, pp. 19:1–19:13. [Online]. Available: http://doi.acm.org/10.1145/2925426.2926270

[33] X. Yu, H. Liu, E. Zou, and S. Devadas, "Tardis 2.0: Optimized Time Traveling Coherence for Relaxed Consistency Models," in *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Sept 2016, pp. 261–274.

[34] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood, "Timestamp Snooping: An Approach for Extending SMPs," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IX. New York, NY, USA: ACM, 2000, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/378993.378998

[35] A. R. Lebeck and D. A. Wood, "Dynamic Self-invalidation: Reducing Coherence Overhead in Shared-memory Multiprocessors," in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ser. ISCA '95. New York, NY, USA: ACM, 1995, pp. 48–59. [Online]. Available: http://doi.acm.org/10.1145/223982.223995

[36] A. Ros and S. Kaxiras, "Complexity-Effective Multicore Coherence," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 241–252.

[37] A.-C. Lai and B. Falsafi, "Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*. IEEE, 2000, pp. 139–148.