Appears in the Proceedings of the 24th International Symposium on High Performance Computer Architecture (HPCA), 2018

High-performance GPU Transactional Memory via Eager Conflict Detection

Xiaowei Ren and Mieszko Lis
The University of British Columbia
{xiaowei, mieszko}@ece.ubc.ca

***Abstract*—GPUs transactional memory (TM) proposals to date have relied on lazy, value-based conflict detection, assuming that GPUs can amortize the latency by executing other warps. In practice, however, concurrency must be throttled to a few warps per core to avoid high abort rates, and TM performance has remained far below that of fine-grained locks.**

We trace this to the latency cost of validating transactions: two round trips across the crossbar required for most commits and aborts. With limited concurrency, the warp scheduler cannot amortize this, and leaves the core idle most of the time.

In this paper, we show that value-based validation does not scale to high thread counts, and eager conflict detection becomes more efficient as the number of threads grows. We leverage this insight to propose GETM, a GPU TM with eager conflict detection. GETM relies on a novel distributed logical clock scheme to implement eager conflict detection without the need for cache coherence or signature broadcasts.

GETM is up to 2.1 times faster than the state-of-the art prior work WarpTM (gmean 1.2 times), with 3.6 times lower silicon area overheads and 2.2 times lower power overheads.

I. Introduction

While GPUs have traditionally focused on streaming applications with regular parallelism, irregular GPU applications with fine-grained synchronization are becoming increasingly important. Graph transformation [1, 2], dynamic programming [3], parallel data structures [4], and distributed hashtables [5] have all been accelerated on GPUs using fine-grained locks. Fine-grained parallel algorithms have recently become a hardware optimization focus for commercial GPUs [6].

Unfortunately, high-performance parallel applications with fine-grained locks are challenging to program and debug. Indeed, reasoning about thread-based synchronized programs is difficult in general [7, 8], and even simple formal analyses that account for inter-thread synchronization are NP-hard [9] or undecidable [10]. In practice, the problem is exacerbated in accelerators like GPUs, because optimizing for performance is paramount — after all, if it weren't, the code would be running on a CPU. In GPUs, this problem is even worse, as the combination of lockstep warp execution and stack-based branch reconvergence can result in unexpected deadlocks in code that would be deadlock-free in CPUs [11].

Transactional memory (TM) [12, 13] offers an attractive solution. In contrast to the imperative style and global dependencies induced by locks, transactions enable a *declarative* programming style: the programmer specifies that a given code block constitutes an atomic transaction and leaves execution details to the runtime (see Fig. 1). Typically, the runtime (hardware or software) attempts to execute transactions optimistically, only aborting and retrying them when conflicts are detected; writes performed by aborted transactions are not visible to transactions that commit successfully. Because they maintain atomicity and isolation, transactions are *composable* [14], and substantially simplify code in complex codebases [15, 16], leading to many times lower error rates [17]. Recently, hardware-level transactional memory has appeared in production CPUs from major vendors [18–21], as well as in designs and proposals from other significant industry players [22, 23].

```
if (src > dst) { // acquire in-order to avoid deadlock
    outer = src; inner = dst;
} else {
    inner = src; outer = dst;
}
done = false;
while (!done) { // loop on flag to avoid SIMT deadlock
    if (atomicCAS(&locks[outer], 0, 1) == 0) {
        if (atomicCAS(&locks[inner], 0, 1) == 0) {
            accounts[src] -= amount;
            accounts[dst] += amount;
            locks[inner] = 0; // release
            locks[outer] = 0; // both locks
            done = true;
        } else { // acquired outer but not inner lock
            locks[outer] = 0; // release outer lock
        }
    }
}
```

```
txbegin
    accounts[src] -= amount;
    accounts[dst] += amount;
txcommit
```

Figure 1. CUDA ATM benchmark fragment using either locks or TM.

Early proposals for hardware-level transactional memory for GPUs solved key problems of interacting with the SIMT stack [24] and coalescing transactions at warp level [25]. Both rely on value-based validation, which requires one core ↔ LLC round trip to validate each transaction and another round-trip to finalize the commit. Combined with the massive concurrency present in GPU workloads, these long latencies create bottlenecks in the commit phase: even if transactional concurrency is restricted, 700 or more transactions may be queued in the commit phase on average [24].

Prior proposals have therefore limited transactional concurrency to very few warps per SIMT core [24, 25]. With few warps, however, the GPU can no longer effectively amortize commit latencies, so some performance is lost. Another proposal has been to proactively abort transactions by broadcasting conflict sets from the LLC back to the SIMT cores [26]; the bandwidth and latency of these broadcasts, however, limit this approach to extremely long transactions.

In this paper, we instead propose to directly reduce commit costs by detecting conflicts eagerly. If conflict detection is performed separately for each memory access — a latency well within a GPU's capacity to amortize even with concurrency throttling — a transaction that arrives at the commit point is guaranteed to commit successfully. Because there is no need for time-consuming value-based conflict detection at commit time, the commit itself can be taken off the critical path while the warp continues execution.

Specifically, we make the following contributions:

- we trace the inefficiency of prior GPU TM proposals to long, unamortized commit latencies;
- we show that the number of concurrent transactions in GPUs favours eager conflict detection;
- we propose a novel GPU TM system with eager conflict detection and lazy version management;
- we describe an efficient data structure that precisely tracks metadata for open transactions of unlimited size while approximately summarizing past commits.

To the best of our knowledge, this is the first full GPU hardware TM proposal with eager conflict detection, and the first to leave transaction commits out of the critical path.

II. Background

In this section we briefly sketch the design space of hardware transactional memory (HTM), describe the best-performing prior GPU proposal WarpTM [25], and identify the bottleneck mechanisms we replace in GETM.

A. The Transactional Memory design space

HTMs can be categorized along two axes: conflict detection and versioning. In eager conflict detection (e.g., LogTM [27, 28]), an inconsistent read or update attempt by a transaction is detected when the access is made, and one of the conflicting transactions is aborted. Lazy conflict detection (e.g., TCC [29]) defers this until later: often, the entire transaction log is validated during the commit process, and conflicts are discovered only then. In principle, the lazy technique can make better conflict resolution decisions because the entire transaction is known, but has longer commit/abort latencies because the entire transaction must be verified atomically. Typically, eager conflict detection leverages an existing CPU coherence protocol.

Version management can also be eager or lazy. Lazily-versioned TMs (e.g., TCC [29]) add transactional accesses to a *redo log*, which is only written to memory when the transaction has been validated and commits; if the transaction aborts, the redo log is discarded. In eager versioning (e.g., LogTM [27, 28]), the transaction writes the new value directly to the memory hierarchy, but keeps the old value in an *undo log*; if a transaction aborts, the undo log is written to memory.

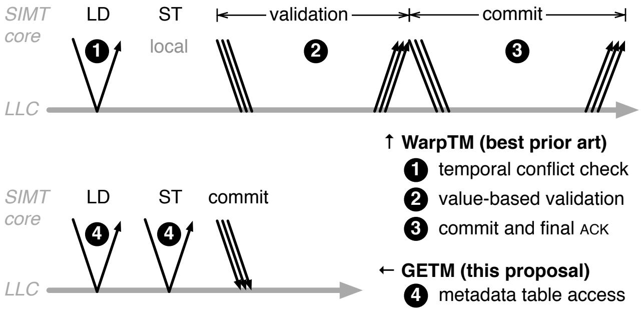


Figure 2. Messages required for transactional memory accesses and commits in WarpTM (top) and GETM (bottom).

B. GPU Transactional Memory

The state-of-the-art GPU TM, WarpTM [24, 25], combines lazy version management with lazy, value-based conflict detection.[1] Fig. 2 (top) shows the access and commit timing.

Firstly, WarpTM modifies the SIMT stacks to allow aborting and restarting transactions at thread granularity. GPUs execute many (32–64) threads in lockstep as a single warp; transactions are a thread-level abstraction, however, so it is possible that some of the threads in the warp commit while other threads abort. WarpTM adds special Transaction and Retry stack entry types that track which threads aborted and should run again when the transaction is restarted.

As transactions execute, their memory accesses are sent to a redo log, stored in the SIMT core's local memory.[2] For each address, loads record the value that was observed (for later validation), and stores record the newly written value. When the warp reaches `txcommit`, a tx log unit traverses the redo log to record all threads wishing to access each address; this allows the SIMT core to resolve all intra-warp conflicts and coalesce the warp's surviving transactions.

At commit time, the read and write logs of the coalesced transaction are sent to validation/commit units (VUs/CUs) colocated with each LLC bank. Each validation unit verifies that the value observed by each read in the log corresponds to the current value in the LLC, and sends a success/failure message to the SIMT core. The core collects these to check whether any addresses failed validation, and sends a commit/abort confirmation back to the CUs. Each CU then sends the write log values to the LLC, and ACKs to the core. Once the core has collected ACKs from all CUs, the warp continues execution. Transactional consistency requires each transaction to be validated and committed atomically, so while one transaction goes through the two-round-trip validation/commit sequence, other transactions must wait.

WarpTM also includes a temporal conflict check mechanism (TCD) that allows read-only transactions to commit silently. A TCD table at the LLC that records the physical

[1] We discuss other GPU proposals [26, 30, 31] in Sec. VII.

[2] In NVidia terminology, a GPU core's local memory is an address range of the global address space reserved for that core. As with the rest of the address space, local memory is cached in the GPU cache hierarchy.

clock cycle number of the last store to each address; the cycle numbers are updated as transactions commit. Each transactional load is immediately sent from the SIMT core to this TCD table; if a read-only transaction has only read locations modified in the past, it is allowed to bypass value-based validation and commit silently.

Because GETM uses eager conflict detection, transactions that have reached `txcommit` are guaranteed to be free of conflicts, and commit without additional validation or ACKS.

GETM retains the SIMT stack modifications and warp-level transaction coalescing of WarpTM. However, it replaces the value-based validation and TCD read-only silent commits with an eager conflict detection scheme (see Sec. IV), which greatly simplifies the validation/commit unit and substantially reduces the hardware overhead (see Sec. V and VI).

C. Eager conflict detection and GPUs

Although eager conflict detection is more suitable for high-thread-count architectures (see Sec. III), the lack of a natural conflict detection mechanism poses a challenge to implementing eager conflict detection in GPUs. Prior TMs with eager conflict detection (e.g., LogTM [27, 28]) have targeted CPUs, in which conflicts are naturally flagged when cache lines are invalidated by the coherence protocol. Unfortunately, extant GPUs lack hardware cache coherence, so another mechanism must be designed. Another challenge is scalability, since GPUs have large core counts and many concurrent warps in each core. This precludes, for example, mechanisms that collect and broadcast read/write signatures.

To provide a scalable eager conflict detection mechanism, we take inspiration from the software transactional memory system TL2 [32]. TL2 uses a global version-clock that is incremented by every transaction which writes to memory, and maintains last-written version-clock values for every memory location. As the transaction accesses memory, it collects version-clocks for all referenced locations. At commit time, these clocks are checked to ensure that the transaction observed a consistent state of memory; if there are no violations, TL2 acquires locks for all locations it intends to modify and finally writes the memory.

In TL2, logical clocks are used to ensure consistency, but conflict detection is still performed lazily at commit time. In addition, the global version clock must be shared among multiple cores, which relies on the underlying cache coherence protocol. We leverage the idea of providing consistency via logical clocks, but use them to implement early conflict detection, and design a distributed logical clock protocol that does not need cache coherence.

We propose GPU Eager Transactional Memory (GETM), a novel GPU hardware TM design. Unlike prior eager TMs, GETM does not rely on coherence or signature broadcast. Instead, GETM combines encounter-time write reservations with a logical timestamp mechanism to detect conflicts as soon as they occur, and to allow off-critical-path commits.

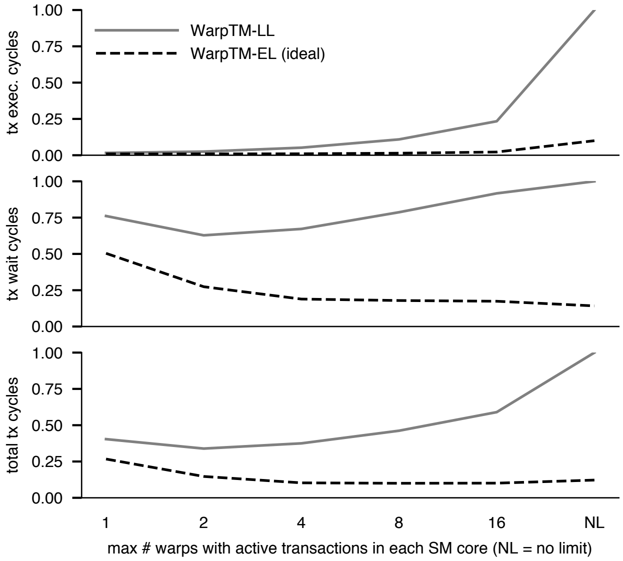


Figure 3. Time per transaction spent executing transactional code (top), waiting for aborting transactions in the same warp and concurrency limits (centre), and total time spent in transactions (bottom), as the number of warps allowed to concurrently run transactions grows. Measurements from the HT-H hashtable benchmark, normalized to the highest data point.

III. GPUs FAVOUR EAGER CONFLICT DETECTION

In this section, we argue that eager conflict detection is particularly suited to the large number of threads concurrently executing in a GPU, because the long commit latencies inherent in lazy detection form a key bottleneck as concurrency grows. This is not the case for CPUs, where TMs with eager conflict detection, such as LogTM [27], are outperformed by lazy [33] or partially lazy [34] variants.

To test this intuition, we modified the state-of-the-art GPU TM design WarpTM [25] to emulate eager conflict detection (cf. Fig. 2) and examined how it performs as the number of warps per SIMT core grows. WarpTM uses lazy conflict detection and lazy versioning (see Sec. II for details), and commits transactions via two core ↔ LLC round trips: (i) the transaction log is sent to be value-validated at the LLC banks; (ii) the LLC sends back validation success/failure status; (iii) the core collects the responses and (if all banks reported success) instructs the LLC to start commit; (iv) the LLC banks acknowledge commit completion; (v) the core can resume executing the relevant warp. Eager conflict detection needs to check only the currently accessed memory location, but must be repeated for every access; therefore, to emulate an eager-lazy design, we hacked WarpTM to run validation (i)–(ii) for every transactional access, with no latency.

Fig. 3 (top) shows how the original WarpTM (-LL) and idealized eager-lazy variant (-EL) perform as permitted concurrency grows on the hashtable insertion workload HT-H. With an increasing number of transactions, the number of cycles spent executing each transaction (including retries) grows much faster for the variant with lazy conflict detection than for the eager version. This is because increasing

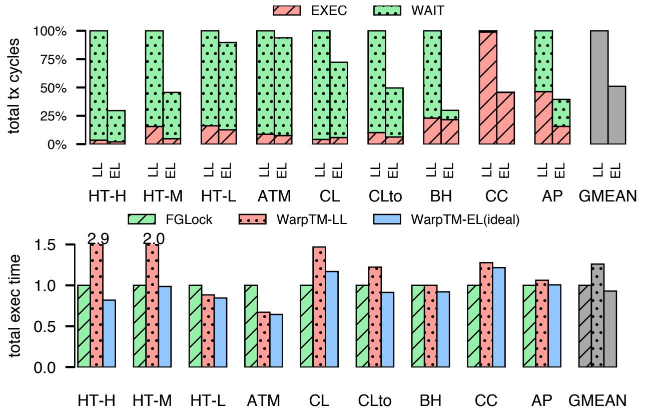


Figure 4. WarpTM with lazy and eager conflict detection compared with hand-optimized fine-grained lock implementations. Top: cycles for transactional segments only; bottom: tx and non-tx segments. Optimal concurrency is used for all configurations.

concurrency increases conflicts and causes transactions to be retried more times. For each retry, WarpTM-LL incurs the two round-trip latency of lazy value-based validation, making each attempt far more expensive than in WarpTM-EL.

Fig. 3 (centre) shows how long transactions wait to commit, either because of concurrency throttling or because of waiting for diverged threads in the same warp to abort the transaction. Because the value-based validations in WarpTM-LL are expensive, subsequent transactions wait longer than in WarpTM-EL. For WarpTM-EL, wait time decreases as more warps can execute and cover commit latency; for WarpTM-LL, however, increasing concurrency increases the commit queue backup and therefore the total wait cost.

The overall runtime spent in transactions is shown in Fig. 3 (bottom). This explains why the optimal concurrency for WarpTM-LL is 2 transactional warps per SIMT core [25], and demonstrates that eager conflict detection can support substantially more concurrency with much lower overheads.

Note that this effect is peculiar to architectures with high thread-level concurrency, such as GPUs. Most CPUs run 1–2 threads per core, and have few cores per die. This places them on the left of Fig. 3 (top), where the lazy and eager versions execute similar number of transactional cycles.

To quantify the overall performance potential of eager conflict detection, we simulated a range of TM benchmarks using the lazy and eager variants of WarpTM, as well as the equivalent non-TM versions using hand-optimized fine-grained locks. Fig. 4 (top) shows that execution and wait cycles spent in transactions are substantially reduced in the eager variant, and Fig. 4 (bottom) shows that this translates to faster overall execution time.

IV. GETM TRANSACTIONAL MEMORY

In this section, we sketch an overview of how GETM provides transactional atomicity, consistency, and isolation, and describe how it tracks the necessary metadata.

The description here focuses on the GETM protocol, how transactions execute, and how metadata evolves. The high-level architecture is shown in Fig. 5; implementation details, including the metadata and queueing data structures present at the LLC, are described in Sec. V.

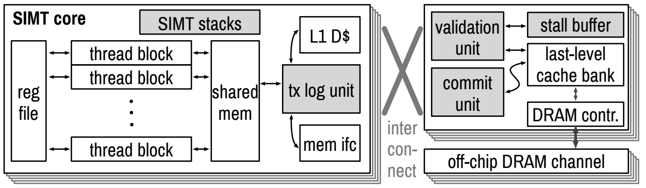


Figure 5. Overall architecture of a SIMT core with GETM. Shaded blocks are added for transactional memory support.

A. Atomicity, consistency, and isolation

We first describe the transaction logs that provide atomicity, and then the logical timestamp and access-time locking mechanisms used to ensure consistency and isolation.

Transaction logs. As in prior work [24, 25], transactions are managed at warp level, and each warp keeps a redo log in the SIMT core's existing local memory.

In contrast to GETM, prior work required sending the entire log (reads and writes) to the commit units for validation at commit time. Because GETM uses eager conflict detection, transactions that have reached `txcommit` are guaranteed to succeed, and commit-time validation is not necessary. Instead, a committing transaction transmits only the transactional writes from the redo log (typically a fraction of the entire log), so that the write data can be stored in the LLC.

In addition to being logged, all transactional accesses are sent to the LLC for eager conflict detection, using the timestamp and lock mechanisms described below.

Logical timestamps. GETM uses distributed logical timestamps to provide transactional consistency, and each transaction executes at a specific logical timestamp. To guarantee consistency, GETM must ensure that a running transaction (a) does not observe stale values of locations changed by logically earlier transactions, (b) does not observe values written by logically later transactions, and (c) does not alter values already seen by logically later transactions.

The logical timestamps tracked by GETM are shown in Table I. Firstly, each warp keeps a logical timestamp *warpts*, corresponding to the memory state observed by the last transaction. This timestamp starts at 0, and is advanced when transactions abort (as discussed below). All new transactions started by this warp execute at logical time *warpts*.

Each cache line in the shared LLC has a write timestamp *wts*, equal to one more than the logical time of the last write,

Tracked per warp	
warpts	the logical time at which transactions from this warp atomically execute
Tracked per LLC cache line	
wts	one higher than the logical time when this location was last written
rts	the logical time when this location was last read
#writes	# writes to this location (if non-zero, location is locked by a transaction)
owner	the owner of the write reservation (if *# writes* is non-zero)

Table I. Metadata tracked by GETM.

i.e., $1 + warpts$ of the logically latest transaction to attempt a write. If a transaction T attempts to access a cache line L where $L.wts > T.warpts$, it means that L was written by a transaction logically later than T, and T must abort.

Every cache line also contains a read timestamp *rts*, equal to the logical time of the last read, i.e., *warpts* of the last transaction to read it. A transaction T may read lines with any *rts*, but writing a cache line L where $L.rts > T.warpts$ would overwrite a value which has already been observed by a later transaction, and is not permitted.

The *rts* and *wts* timestamps are maintained eagerly: that is, transactional loads update *rts* and transactional writes update *wts* at the time of the request, regardless of whether the transaction will eventually commit. The updated timestamps are not reverted if a transaction aborts; while this might unnecessarily abort some future transactions, those will be restarted, and consistency is not compromised.

Encounter-time locks. Unlike timestamps, transactional write data is not stored in the LLC until the transaction reaches its commit point. This creates an isolation problem if a transaction T_1 modifies a location and a logically later but physically concurrent transaction T_2 accesses this location: the value that should be seen by T_2 depends on whether T_1 will commit successfully, but T_1 is still in progress.

To avoid this issue, GETM uses locks to prevent T_2 from reading the location until T_1 has committed. Each cache line has two additional fields to support this: *#writes* and *owner* (see Table I). When a transaction T first encounters a previously untouched cache line L, it reserves L by setting $L.\#writes$ to 1 and $L.owner$ to the transaction's global warp ID (because transactions are coalesced per warp, this uniquely identifies a running transaction; see Sec. II-B).

Now when T_2 accesses L (either for reading or writing), it must check whether L has been reserved. If $L.\#writes \neq 0$ and $L.owner \neq T_2$, transaction T_2 proceeds with the *rts/wts* checks described above; if the checks fail then T_2 is aborted, otherwise T_2 stalls until T_1 commits. (We discuss the stall buffer where stalled transactions are queued in Sec. V.)

The *owner/#writes* mechanism also allows a transaction to repeatedly write the same location. If T is already the *owner* of a cache line, it bypasses the *rts* and *wts* timestamp checks, and writes the line. This is safe because T must have previously satisfied the *rts* and *wts* timestamp constraints, and updated *wts*. As any other transaction attempting to update the line since that time would have been stalled, neither *rts* and *wts* could have been altered since T's reservation.

Aborts and advancing logical time. The logical time observed by each warp (*warpts*) advances when transactions are aborted. When a transaction aborts, it reports to the core the latest logical timestamp t it attempted to read or write (the abort cause). Since the transaction will fail again unless it restarts at a time later than t, *warpts* is set to $t + 1$.

For example, if a transaction T has aborted because of reading a cache line L, it must be because the cache line is logically newer than the transaction, i.e., $L.wts > warpts$. In this case, the SIMT core sets *warpts* to $L.wts + 1$, and T is restarted. Similarly, if T aborts because of a write, *warpts* is set to $\max(L.rts, L.wts) + 1$, and the transaction restarts.

Commit and cleanup. When all threads in a warp reach the end of the transaction (commit or abort), the SIMT core serializes the write logs of all threads and sends them to the LLC. For all threads that have successfully reached the commit point, the core sends the address, write data, and write count (since multiple writes may have been coalesced).

Once this commit/abort log is received, each entry is written to the LLC and the relevant *#writes* entry is decremented. Once *#writes* in a cache line has reached 0, the cache line fully reflects the atomic transaction update, and can now be accessed by other transactions.

Aborted transactions instead send the address and write count for each modified cache block to facilitate cleanup. The *#writes* in each cache line is updated as above; after *#writes* has reached 0, the cache line reflects its pre-transaction state, and may be accessed by other transactions.

The life of a transactional access. Fig. 6 shows how a transactional read or write is processed in GETM.

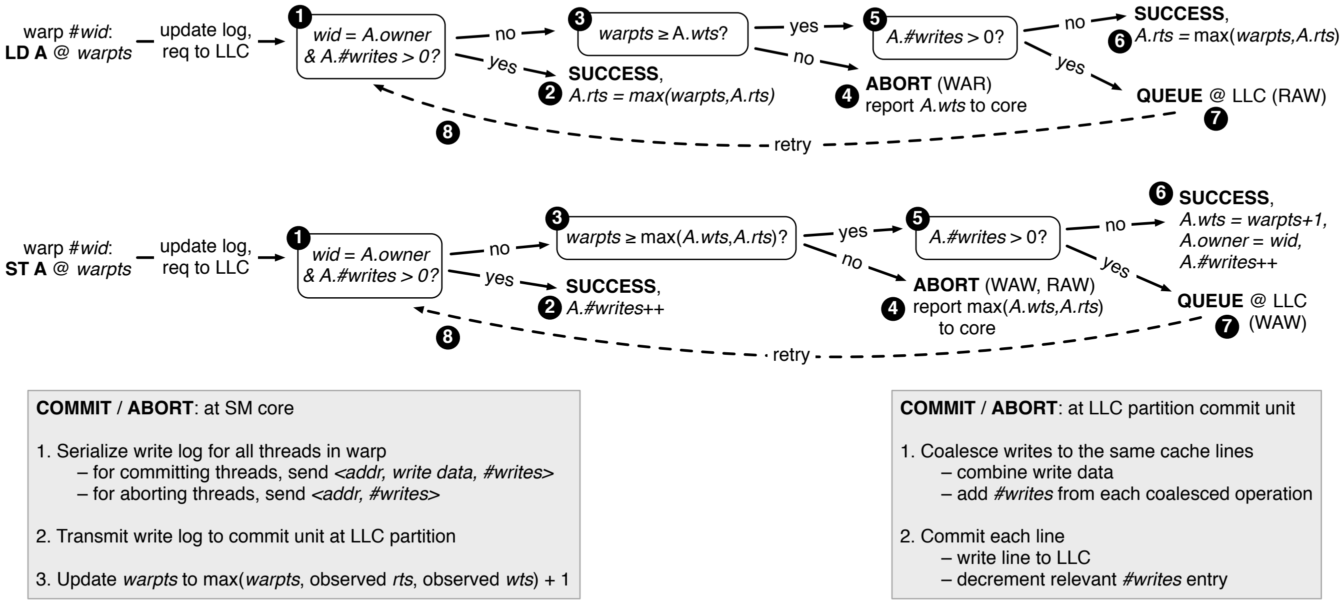
Owner check ❶. If *#writes* is non-zero but the owner field matches the current transaction, the line must be locked and the access succeeds ❷. Stores only increment *#writes* (since *wts* was already set by the previous write), while loads potentially update *rts* if it is less than *warpts*.

Timestamp check ❸. A transaction that attempts to load an address and finds its *wts* younger than the transaction's own *warpts* has detected a WAR conflict – i.e., another transaction with a younger *warpts* has already written to the location – and must abort ❹. Similarly, a transaction that writes a location but finds either *wts* or *rts* to be younger than *warpts* must also abort, since a logically younger transaction has either written the location or observed its value ❹.

Abort notification ❹. If the version check discovers a conflict, the transaction must be aborted. To minimize the chances of the transaction aborting again, the SIMT core is sent the highest timestamp seen so far at the LLC; this will be used to update *warpts* and restart the transaction. Meanwhile, the core notes that the thread has aborted, and will clean up any reservations made when the entire warp reaches *txcommit* or when all threads have aborted.

Write lock check ❺. Next, the transactional memory operation checks whether the accessed location has been reserved by another warp (i.e., whether *#writes* is non-zero). If not, the operation succeeds without conflict: a load will update *rts* (if < *warpts*) while a store will set *#writes* to 1 and update the location's *wts* with the transaction's *warpts* ❻.

Queue ❼ *and retry* ❽. Accesses that passed the timestamp check but do not own the active lock must be logically younger than the lock owner. To avoid unnecessary aborts, these requests are queued until the owner transaction commits. After the lock is released, the queued transactions will retry.



COMMIT / **ABORT**: at SM core

1. Serialize write log for all threads in warp
   – for committing threads, send *<addr, write data, #writes>*
   – for aborting threads, send *<addr, #writes>*

2. Transmit write log to commit unit at LLC partition

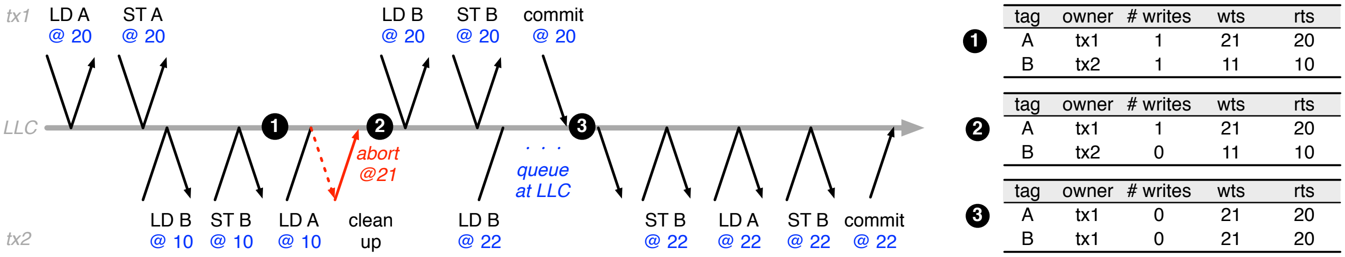
3. Update *warpts* to max(*warpts*, observed *rts*, observed *wts*) + 1

COMMIT / **ABORT**: at LLC partition commit unit

1. Coalesce writes to the same cache lines
   – combine write data
   – add *#writes* from each coalesced operation

2. Commit each line
   – write line to LLC
   – decrement relevant *#writes* entry

Figure 6. The flowchart for load, store, and commit/abort logic in GETM.



❶

tag	owner	# writes	wts	rts
A	tx1	1	21	20
B	tx2	1	11	10

❷

tag	owner	# writes	wts	rts
A	tx1	1	21	20
B	tx2	0	11	10

❸

tag	owner	# writes	wts	rts
A	tx1	0	21	20
B	tx1	0	21	20

Figure 7. Eager conflict resolution in GETM.

B. Walkthrough example

Fig. 7 illustrates how the GETM protocol operates on two conflicting transactions from the bank transfer example (Fig. 1); in this benchmark, accounts are modelled as unique memory locations. The first transaction (*tx1*) transfers some amount from account A to account B, while the second (*tx2*) transfers another amount from B to A. Transaction *tx1* starts at *warpts* = 20, and transaction *tx2* starts at *warpts* = 10. The central grey line represents the LLC, and the thinner black arrows represent messages between the cores and the LLC. The interleaving of the accesses from each transaction has been chosen to illustrate how the eager conflict detection and queueing mechanisms work; in reality, any interleaving of the two transactions could occur.

First, *tx1* loads and stores location A: the load updates A's *rts* to match the transaction's *warpts* (i.e., to 20), and the store updates the *wts* of A to exceed that of *tx1* (i.e., to 21). Then *tx2* does the same with B, updating its *wts* to 11 and *rts* to 10. At this point, *tx1* and *tx2* have accessed disjoint locations and so far do not conflict. The transaction metadata for addresses A and B at this point are shown in table ❶.

Next, *tx2* attempts to read location A, previously altered by *tx1*. Because *tx2.warpts* < *A.wts*, the load fails the version check and will abort *tx2* (cf. Fig. 6). The LLC will notify the requesting core that the transaction been aborted, and that the next *warpts* should be later than 21. The core will then send the write/abort log for *tx2* to the LLC, which will release the reservation for B by setting the *# writes* field to 0. When *tx1* now sends load and store requests for B, both requests succeed since *tx2* had an older version and its write lock was cleared as *tx2* aborted. At this point, the metadata for A and B correspond to table ❷.

Transaction *tx2* now restarts at the core, with a higher *warpts* of 22. When its first load request (for B) arrives at the validation unit, it passes the version check but finds B reserved; the load is therefore queued in the VU's stall buffer and will be retried as the conflicting transaction commits.

Meanwhile, *tx1* gets to its commit instruction. Because all of its memory accesses have passed eager conflict detection, the transaction is guaranteed to succeed. The core therefore sends the write log to the LLC and moves on. As the write log is processed, write reservations (*#writes*) for both A and B are reset. Table ❸ shows the metadata at this point.

Once the commit of *tx1* has finished and released the reservations on A and B, any stalled transaction accesses are retried; in this case, this is the load of B from *tx2*, which now succeeds. Transaction *tx2* can then continue with its remaining memory accesses, and will succeed.

V. Implementation details

Adding transactional memory support requires modifications to both the SIMT core and the memory partition that houses the LLC slice and a memory controller: we need to modify the core to retry aborted transactions and record redo logs, and to add validation and commit hardware to each memory partition. Fig. 5 shows the overall architecture components of a GPU core extended with GETM.

A. SIMT core extensions

SIMT Stack. Adding transactional memory support to a GPU's cores requires changing the SIMT stack to track which threads in the warp are executing transactions and which must be retried. To implement this, we leverage the modified SIMT stack proposed by Fung et al [24]. This mechanism is similar to branch divergence hardware [35]: for each warp, the top of the SIMT stack tracks the threads that are currently executing, while the stack entry immediately below tracks threads that have aborted and must be retried.

Transaction management. While individual threads can run separate transactions, commits occur at warp granularity when all threads in the warp have arrived at the commit point [25]. Nevertheless, transactions remain logically at thread granularity: when some of the warp's threads abort, they are automatically retried via the extended SIMT stack after the entire warp reaches the commit point [24].

Transaction logs. The GETM versioning mechanism is the same as in GPU transactional memory [24]. Logs are stored in each SIMT core's local address space, and cached by the L1/LLC caches. Although GETM only requires a write log, we also record a read log to permit intra-warp conflict detection [25]; in this technique, each transactional access is first checked against the local per-warp read and write logs and aborted if it conflicts with other threads in the same warp. At commit time, however, the read log is discarded and only the write log is sent to the commit units.

Forward progress. Aborted transactions ensure progress by restarting with a probabilistically increasing backoff [36].

B. Validation unit

GETM protocol actions on the LLC side are carried out by validation units (VUs), one of which is colocated with each LLC bank. Each VU consists of (a) metadata storage structures to track the last-written and last-read versions for each address, and (b) a structure to buffer requests that found a location locked but were younger than the current owner.

1) Transaction metadata storage: Because GETM explicitly tracks versions to enable eager conflict detection, it must keep all metadata (*wts*, *rts*, *# writes*, and *owner*; see Table I) for all locations that are part of any in-flight transaction, and some metadata (*wts* and *rts*) for all locations that have been (or could be) accessed transactionally.

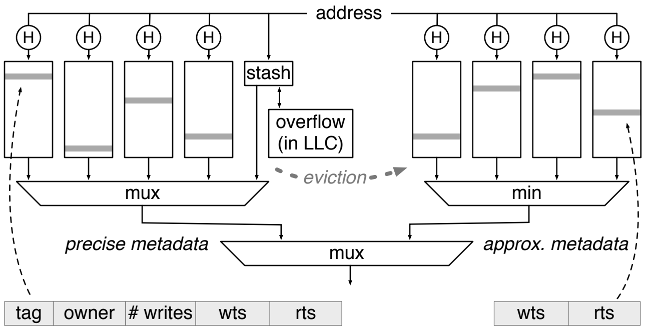


Figure 8. Transaction metadata table microarchitecture

These requirements pose some challenges: firstly, transactions could be very long (and, in general, unbounded), so fast access to a potentially large lookup structure is necessary; secondly, potentially all addresses could be accessed transactionally, and tracking metadata for them all is impractical.

Our solution relies on two observations. The first is that very long transactions are likely to be rare in well-tuned code; therefore the metadata table can be sized for the common case and provide a spillover mechanism (like in Unbounded TM [37]). The second is that metadata for addresses that are not being written by in-flight transactions can be maintained *approximately* provided that the only errors are overestimates: if the lookup mechanism reports a higher *rts* or *wts*, additional transactions may abort, but correctness will be preserved.

Fig. 8 shows the microarchitecture of the metadata storage structure. Our implementation has one such structure at every LLC partition, responsible for the same address range. It consists of two tables, accessed simultaneously during lookups: the first tracks precise metadata for addresses accessed by in-flight transactions, while the second tracks approximate *rts* and *wts* for all other addresses.

Precise metadata for in-flight accesses. The precise metadata table is similar to a cuckoo hash table [38], extended with a small stash [39] (conceptually similar to a victim cache); even a small stash allows the cuckoo table to maintain higher occupancy with limited resources [39]. When inserting a ⟨key, value⟩ pair causes too many swaps in the cuckoo table, the last ⟨key, value⟩ pair swapped out during the insertion process is placed in the stash, and during lookups the stash is searched in parallel with the cuckoo table itself. We use a four-way cuckoo table with four randomly generated H_3 hashes [40] and a 4-entry fully associative stash. To permit very long transactions, the precise table and stash can spill to an unbounded overflow space located in main memory and cached in the LLC. In our experiments the overflow space was never used, so we organized the overflow as a linked list; a commercial implementation would likely use an asymptotically faster design such as a balanced tree or another hashtable layer in main memory.

Unlike the original cuckoo table, our design allows the

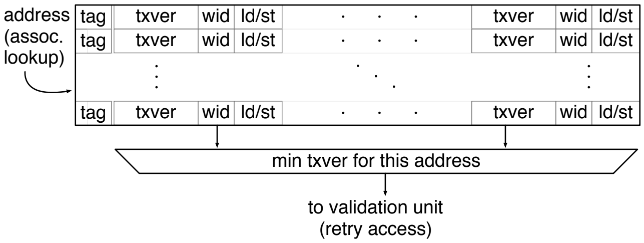


Figure 9. Stall buffer microarchitecture

insertion process to terminate by evicting an entry that has not been reserved by any transaction (i.e., *# writes* is zero). Since the remaining metadata — *wts* and *rts* — can be safely approximated, the evicted entry is inserted into the approximate metadata structure described below.

Approximate metadata for inactive locations. The simplest design for approximate version tracking is a pair of registers tracking the maximum *wts* and *rts* that have been evicted from the precise table. When a lookup misses in the precise table, it is reinserted using the approximate *wts* and *rts* values from the two registers. When we conducted experiments with this configuration, however, we found that the version numbers increased very quickly and caused many aborts.

To combine efficient storage of large numbers of evicted addresses with the ability to discriminate among many of them, we use a recency Bloom filter [24]. This structure consists of several (in our case, four) ways indexed by different hashes of the lookup address (we again use H_3 hashes). Each address maps to one entry in each way, and each entry stores the maximum *wts* and *rts* of all inserted addresses that map to it. On insertion, the *wts* and *rts* in each way are only updated if they exceed the stored values (which may have come from a hash collision), and on lookup the minimum *wts* and *rts* among the four ways are returned.

Timestamp rollover. Unlike physical timestamps [25], logical timestamps advance very slowly. In our experiments, the increment rates ranged from one increment in 1,265 cycles to one in 15,836 cycles, depending on the benchmark. At this rate and with a 1 GHz clock, 32-bit timestamps will roll over less than once every 1.5 hours, and 48-bit timestamps will roll over less than once every 11 years.

When a validation unit detects a rollover, it must ensure that (a) all validation units roll over atomically, and (b) all SIMT cores have rolled over. The first task can be accomplished via two messages (containing the VU ID to break ties) sent via a single-wire ring connecting all validation units. The first message indicates that the recipient should stall and forward the message to its neighbour; all VUs will be known to have stalled when the message reaches back to the originating VU. The second message indicates that the recipient should roll over and continue execution. (Alternately, the existing interconnect can be used for this purpose with an ACK–REPLY protocol). Cores roll over on a request from the VUs sent over the interconnect. Once the cores have ACKed the request, the VU knows that no requests are in flight; it flushes the stall buffer and metadata tables and resumes.

Baseline GPU	
SIMT core config	15 cores, 48 × 32-wide warps / core, 2 × 16-wide SIMD
warp scheduler	greedy then oldest (GTO)
in-core storage	32,768 registers / core, 16KB shared memory / core
L1 data cache	48KB per core, 128-byte lines, 6-way assoc.
L2 cache (LLC)	128KB / partition, 128-byte lines, 8-way assoc.,
interconnect	2 xbars (1 up, 1 down), 288GB/s each, 5-cycle latency
operating frequency	SIMT core: 1400 MHz, interconnect: 1400 MHz, memory: 924 × 4 (quad-pumped)
GDDR5	6 partitions, 32 queued requests each, FR-FCFS, Hynix H5GQ1H24AFR timing, total BW 177GB/s
memory scheduling latency	L1: 1 cycle; LLC: 330 cycles; DRAM: 200 cycles
Transactional memory support	
concurrency (tx warps/core)	1, 2, 4, 8, 16, unlimited (optimal for each benchmark)
operating frequency	validation unit: 1400 MHz, commit unit: 700 MHz
metadata storage	precise: 4K entries (total) in 4-bank cuckoo HTs, 4-entry stashes approx.: 1K entries (total) in 4-bank recency Bloom filters
stall buffer	4 lines with 4 entries each, per partition
validation BW	1 request/cycle per partition
commit BW	32B/cycle per partition
intra-warp conflict detection	two-phase parallel, 4KB ownership table / tx warp

Table II. Simulated GPU and memory hierarchy.

2) Stall buffer: Requests that passed the version check but found the address locked are queued in a *stall buffer* until the relevant transaction commits or aborts (see Sec. IV).

The organization of this structure, shown in Fig. 9, is similar to a store buffer or an MSHR, but tracks several requests for each address (from different warps contending for the same location). When a committing transaction decrements the *#writes* count to 0, it checks whether any stall buffer entries are waiting on the relevant address; if so, the oldest request (i.e., with the minimum *warpts*) re-enters the validation unit. If the buffer is full, the transaction aborts.

C. Commit-time coalescing

The commit unit receives write logs from SIMT cores, coalesces multiple accesses to the same 32-byte regions, writes the data to the LLC, and decrements the relevant *#writes* entries. While coalescing is not needed for correctness, it efficiently uses the GPU's wide LLC port.

To coalesce writes, we use a simplified variant of the ring buffer used in KiloTM [24] and WarpTM [25]. In contrast to these proposals, in GETM the commit unit receives only the write log, so the buffer can be substantially reduced; we conservatively size it to half of that in prior work.

VI. Results and discussion

A. Methods

Simulation setup. We follow the methodology established in previous GPU hardware transaction memory proposals [24–26]. GPGPUsim 3.x [41] is used to simulate the GPU and modified to implement GETM and prior proposals. We estimated area and power overheads of the structures required to implement TM by modelling them in CACTI 6.5 [42], conservatively assuming that all structures are accessed every cycle and accounting for the higher validation unit clock. We assumed a 32nm node (the smallest supported by CACTI 6.5).

name	abbreviation	description
Hash Table (CUDA)	HT-H	populate an 8000-entry hash table
	HT-M	populate an 80000-entry hash table
	HT-L	populate an 800000-entry hash table
Bank Account (CUDA)	ATM	parallel funds transfer (1M accounts)
Cloth Physics [45] (OpenCL)	CL	cloth physics (60K edges)
	CLto	tx-optimized version of CL
Barnes Hut [46] (CUDA)	BH	build an octree (30K bodies)
CudaCuts [47] (CUDA)	CC	image segmentation (200×150 pixels)
Data Mining [48] (CUDA)	AP	data mining (4000 records)

Table III. Summary of the benchmarks used for evaluation.

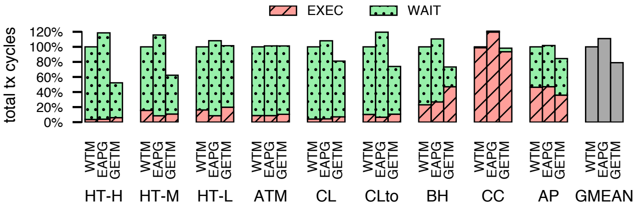


Figure 10. Transaction-only execution and wait time, normalized to WarpTM (lower is better). Note that EAPG is idealized.

Table II describes the simulation setup. For fair comparison of the eager conflict detection mechanism with the value-based detection from prior proposals, we keep the same baseline: a GPGPU similar to NVIDIA's GTX 480 (Fermi [43]) with 15 cores, 6 memory partitions, and latencies derived from microbenchmark studies [44]. To investigate scalability to higher core counts, we also simulated a configuration with 56 cores in 28 clusters, and a 4MB L2 cache in eight 8-way banks; for WarpTM, we doubled the recency filter size, and for GETM we doubled only the precise metadata table.

Baselines. We compare GETM against WarpTM [25], and an idealized implementation of the EarlyAbort/Pause-n-Go (EAPG) proposal [26].[3] We use TM benchmarks from prior work [24, 25]; they are summarized in Table III.

B. Results

Performance and crossbar traffic. Fig. 10 shows the total number of cycles spent executing transactions and waiting for other transactions to finish, normalized to the WarpTM baseline. For most workloads, GETM reduces both transaction execution time and wait time. CC and AP have contention over few memory locations, and GETM sees many aborts; because commits and aborts are cheap in GETM, however, this is still faster than WarpTM and EAPG. In CC and AP, transactions spend little time waiting because they account for a small portion of the total runtime. We find that, for these benchmarks, even idealized EAPG is ineffective, as only 5.2% aborts come from the early-abort mechanism and 1.3% transactions are ever paused. Essentially, by the time a broadcast update reaches the cores, most conflicting transactions have already been sent for validation/commit. In fact, EAPG underperforms WarpTM because the additional early-abort broadcasts congest the core ↔ LLC interconnect (even though these are idealized as single header-only flits). We expect that EAPG can be effective only with extremely long transactions.

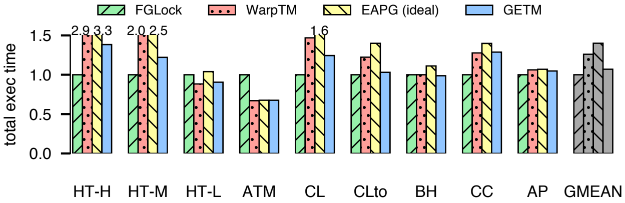


Figure 11. Execution time normalized to the fine-grained lock baseline, including transactional and non-transactional parts (lower is better).

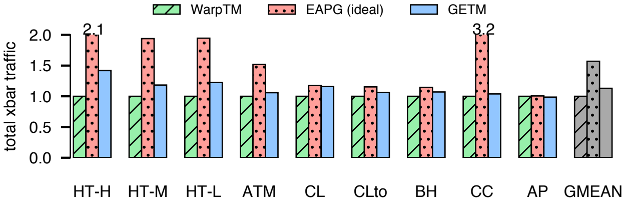


Figure 12. Crossbar traffic load normalized to WarpTM (lower is better).

Overall performance is shown in Fig. 11: on average, GETM outperforms WarpTM by 1.2× (gmean) and is within 7% of the fine-grained lock baseline. The trend mirrors that of the transactional execution and wait time above. Benchmarks with high contention benefit more, because GETM aborts doomed transactions without the need to queue at the LLC for value-based validation, and show substantial improvements (up to 2.1× for HT-H). Low-contention workloads perform comparably to WarpTM.

The improved performance comes at a minor cost in interconnect traffic compared to WarpTM (Fig. 12). Although GETM does not need to transmit the transaction read log at commit time, it needs to acquire locks for every write at encounter time, whereas WarpTM only contacts the TCD for loads. In addition, despite better performance, GETM has a higher abort rate, which adds to the interconnect traffic load.

Sensitivity to validation unit parameters. Because the validation unit contains a cuckoo-like structure where worst-case insertions can take many cycles, we measured the average number of validation unit cycles spent on accessing the metadata tables for each request (Fig. 13). Even under very high load factors (> 99%), long insert chains where all entries have #*writes* > 0 are very unlikely; when they do occur, the stash is effective as predicted theoretically [39].

We also investigated the effect of changing metadata table sizes and granularity (Fig. 14); we tested 2K, 4K, and 8K entries GPU-wide, and 16, 32, 64, and 128-byte granularity assuming 4K table entries GPU-wide. A 2K metadata footprint is too small (and, indeed, requires a larger stash), especially when parallelism is abundant (e.g., HT-H); because 8K entries do not significantly outperform 4K entries, we settled on 4K entries for other parts of the evaluation. Decreasing granularity generally improves performance because false sharing is reduced; however, it also reduces effective table size when parallelism is high and

[3] Specifically, write signatures broadcast to cores were idealized as 64-bit messages, refcount table updates on the LLC side were idealized to one cycle for the entire tx log, and the early conflict check was made instant.

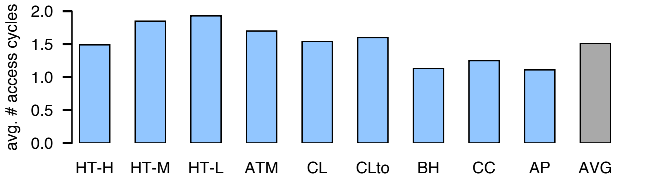


Figure 13. Mean latency of the cuckoo table in the metadata storage structure (≥ 1.0, lower is better). The combination of allowing evictions to the approximate table and the small stash makes insertions very efficient.

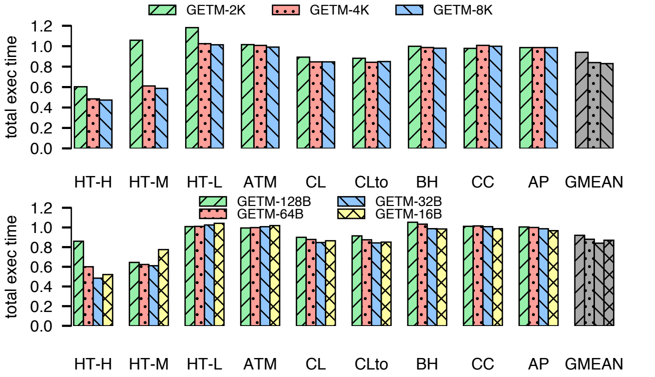


Figure 14. Sensitivity to metadata table size (top) and granularity (bottom). Execution time normalized to a WarpTM baseline (lower is better).

the total number of addresses accessed is higher. We chose 32-byte granularity for all other tests.

Since requests that pass the timestamp check but find their target location reserved are queued in the stall buffer, we measured stall buffer performance. Fig. 15 shows the maximum total occupancy of all stall buffers; this never rises above 12 requests across the entire GPU. Fig. 16 shows that very few requests are queued up on average for any given address. In the rest of the evaluation, we conservatively sized the stall buffers to 4 addresses with space for 4 requests each.

Abort rates under contention. Both WarpTM and GETM limit transactional concurrency to optimize performance. Table IV lists the best concurrency settings for each benchmarks — i.e., the number of warps in each core allowed to run transactions concurrently — and the resulting number of aborted transactions. With abundant parallelism (e.g., HT-H), GETM is efficient at higher concurrency than WarpTM. The eager conflict detection in GETM also translates to dramatically faster commits and aborts than the value-based conflict detection in WarpTM, so GETM can handle higher abort rates and still perform substantially better.

Scalability. To investigate scalability at higher core counts, we also simulated WarpTM and GETM in a configuration with 56 SIMT cores and a 4MB LLC; Fig. 17 shows the results. While performance differences vary slightly per benchmark, the overall trends match the 15-core setup.

Silicon area and power. Table V shows the area and power overheads introduced by adding TM support. Because GETM removes most of the structures needed by WarpTM, it has 3.6× lower area overheads and 2.2× lower power overheads (4.9× and 3.6× lower than EAPG). Overall, GETM adds ~0.2% area to a GTX 480 die scaled down to 32nm.

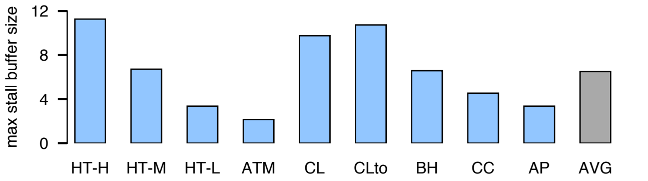


Figure 15. The maximum number of addresses queued at any given time (total of all stall buffers in the GPU).

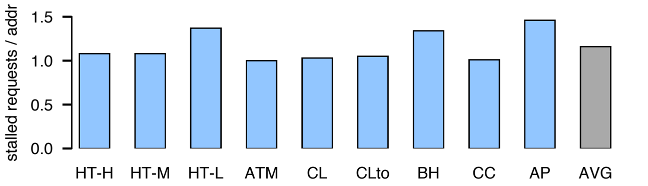


Figure 16. The average number of requests per address that concurrently reside in the stall buffer.

VII. Related work

GPU TM. To date, all hardware-level transactional memory proposals for GPUs have been based on KiloTM [24]; this system combines lazy version management with lazy, value-based conflict detection. Follow-up work [25] extended KiloTM with an intra-warp conflict detection mechanism and a silent-commit filter for read-only transactions based on physical timestamps. A later proposal [26] added global broadcast updates about currently committing transactions, and leveraged this to pause or abort doomed transactions; we use an idealized version of this as one of our baselines. GPU-LocalTM [30] is a limited form of transactional memory that guarantees atomicity only within a core's scratchpad; Bloom filters [49] are used for conflict detection. Software transactional memory proposals for GPUs have used either per-object write locks [50] or combined value-based detection with TL2-like timestamp approach [51]. Given special DRAM subarrays [52], and at the cost of substantial memory overheads and extensive OS/software changes, GPU snapshot isolation [31] can reduce abort rates in long transactions by buffering many concurrent memory states; it retains two-round-trip lazy validation and must update snapshot versions in DRAM, resulting in even longer commit latencies.

CPU HTM. Since hardware-level transactional memory was first proposed [12, 13], many CPU implementations have been proposed. Many leverage the existing inter-core coherence mechanism to identify conflicts, either my modifying the coherence protocol [23, 33, 34, 53], adding extra bits to the coherence state [27, 54, 55], or leveraging coherence to update read/write signatures [28, 56, 57]. Existing GPU coherence proposals, however, cannot support eager TM: they either rely on special language-level properties [58], eschew write atomicity [59], or cannot support detecting conflict times [60]. Other TM proposals [29, 33, 61–64] rely on signature or update broadcasts, or on software-assisted detection [65–67].

Timestamp-based TM. Transactional memory schemes

	best concurrency				aborts / 1K commits			
	WTM	EAPG	WTM-EL	GETM	WTM	EAPG	WTM-EL	GETM
HT-H	2	2	8	8	119	113	122	460
HT-M	8	4	8	8	98	84	83	172
HT-L	8	4	8	8	80	78	78	207
ATM	4	4	4	4	27	26	25	114
CL	2	2	4	4	93	91	119	205
CLto	4	2	4	4	110	61	72	176
BH	2	2	8	∞	93	86	145	865
CC	∞	∞	∞	∞	6	5	1	38
AP	1	1	1	1	231	237	204	9188

Table IV. Optimal concurrency (# warp transactions per core) settings and abort rates for different workloads. (WTM = WarpTM.)

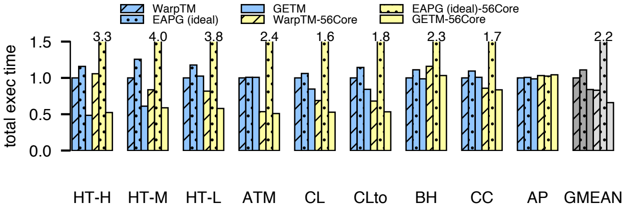


Figure 17. Execution time in GPUs in 15-core and 56-core configurations, normalized to 15-core WarpTM performance (lower is better).

element	area [mm²]	power [mW]
WarpTM		
CU: LWHR tables (3KB×6)	0.108	21.84
CU: LWHR filters (2KB×6)	0.03	12.00
CU: entry arrays (19KB×6)	0.402	100.62
CU: read-write buffers (32KB×6)	1.734	132.48
TCD: first-read tables (12KB×15)	0.375	113.25
TCD: last-write buffer (16KB total)	0.031	9.86
total WarpTM	2.68	390.05
EAPG (in addition to WarpTM)		
CAT: Conflict Address Table (12KB×15)	0.6	153.3
RCT: Reference Count Table (15KB×6)	0.294	75.6
total EAPG	3.574	618.95
GETM (independent of WarpTM)		
CU: write buffers (16KB×6)	0.522	85.56
VU: precise tables (64KB total)	0.181	69.59
VU: approximate tables (8KB total)	0.018	8.51
warpts tables (192B×15)	0.015	10.65
stall buffer (30B×4×6)	0.0004	2.67
total GETM	0.736	176.98

Table V. CACTI area and power (dynamic + static) estimates for WarpTM [25], EAPG [26], and GETM overheads (32nm node). CU: commit unit; TCD: temporal conflict detection; VU: validation unit.

based on logical clocks share commonalities with timestamp-based approaches. These have been used mainly in software TMs to maintain consistency [68]; hardware TMs have leveraged them to maintain fairness and forward progress [27, 37, 69], snapshot isolation [70], and in prior GPU work to avoid validation of read-only transactions [25].

Logical-time coherence and consistency. Lamport first observed that consistency guarantees can be maintained using logical clocks [71]; the read and write versions tracked by GETM use this insight. Logical clocks have also been used to implement coherence in a logically ordered bus (e.g., [72, 73]), to extend snooping [74, 75], and by directly tracking access timestamps [60, 76, 77]; two of these proposals [60, 76] also use logical timestamps to enforce sequential consistency. Logical clocks were also used to dynamically verify consistency models [78]. While the concepts of coherence and consistency are related to TM, they do not offer atomicity and isolation for access sequences as transactional memory does.

VIII. Conclusions

We have presented GETM, the first full GPU transactional memory mechanism with eager conflict resolution. By combining explicit version tracking with encounter-time write reservations, GETM enables efficient conflict detection and off-the-critical-path commits. GETM is up to 2.1× faster than the state-of-the-art GPU TM (1.2× gmean), while incurring 3.6× lower area overheads and 2.2× lower power overheads.

Acknowledgements

This research was supported by the Natural Sciences and Engineering Research Council of Canada. The authors are grateful to Tor Aamodt and Daniel Lustig, as well as the anonymous reviewers, for helpful discussion and suggestions.

References

[1] M. Méndez-Lojo *et al.*, “A GPU Implementation of Inclusion-based Points-to Analysis,” in *PPoPP*, 2012.

[2] Y. Xu *et al.*, “Lock-based Synchronization for GPU Architectures,” in *CF*, 2016.

[3] A. Li *et al.*, “Fine-Grained Synchronizations and Dataflow Programming on GPUs,” in *ICS*, 2015.

[4] N. Moscovici *et al.*, “POSTER: A GPU-Friendly Skiplist Algorithm,” in *PPoPP*, 2017.

[5] T. H. Hetherington *et al.*, “MemcachedGPU: Scaling-up Scale-out Key-value Stores,” in *SoCC*, 2015.

[6] NVIDIA. Inside Volta: The World’s Most Advanced Data Center GPU. Parallel Forall blog entry. [Online]. Available: https://devblogs.nvidia.com/parallelforall/inside-volta/

[7] D. C. Arnold *et al.*, “Stack Trace Analysis for Large Scale Debugging,” in *IPDPS*, 2007.

[8] E. A. Lee, “The Problem with Threads,” *IEEE Computer*, vol. 39, p. 33, 2006.

[9] R. N. Taylor, “Complexity of analyzing the synchronization structure of concurrent programs,” *Acta Informatica*, vol. 19, pp. 57–84, 1983.

[10] G. Ramalingam, “Context-sensitive Synchronization-sensitive Analysis is Undecidable,” *ACM Trans. Program. Lang. Syst.*, vol. 22, pp. 416–430, 2000.

[11] A. ElTantawy and T. M. Aamodt, “MIMD synchronization on SIMT architectures,” in *MICRO*, 2016.

[12] M. Herlihy and J. E. B. Moss, “Transactional Memory: Architectural Support for Lock-free Data Structures,” in *ISCA*, 1993.

[13] J. M. Stone *et al.*, “Multiple reservations and the Oklahoma update,” *IEEE Parallel Distributed Technology: Systems Applications*, vol. 1, pp. 58–71, 1993.

[14] T. Harris *et al.*, “Composable Memory Transactions,” in *PPoPP*, 2005.

[15] F. Zyulkyarov *et al.*, “Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server,” in *PPoPP*, 2009.

[16] C. J. Rossbach *et al.*, “TxLinux: Using and Managing Hardware Transactional Memory in an Operating System,” in *SOSP*, 2007.

[17] ——, “Is Transactional Programming Actually Easier?” in *PPoPP*, 2010.
[18] Intel, “Intel architecture instruction set extensions programming reference: Chapter 8: Intel transactional synchronization extensions,” Tech. Rep., 2012.
[19] R. Haring *et al.*, “The IBM Blue Gene/Q Compute Chip,” *IEEE Micro*, vol. 32, pp. 48–60, 2012.
[20] C. Jacobi *et al.*, “Transactional Memory Architecture and Implementation for IBM System Z,” in *MICRO*, 2012.
[21] H. W. Cain *et al.*, “Robust Architectural Support for Transactional Memory in the Power Architecture,” in *ISCA*, 2013.
[22] S. Chaudhry *et al.*, “Rock: A High-Performance Sparc CMT Processor,” *IEEE Micro*, vol. 29, pp. 6–16, 2009.
[23] J. Chung *et al.*, “ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory,” in *MICRO*, 2010.
[24] W. W. L. Fung *et al.*, “Hardware Transactional Memory for GPU Architectures,” in *MICRO*, 2011.
[25] W. W. L. Fung and T. M. Aamodt, “Energy Efficient GPU Transactional Memory via Space-time Optimizations,” in *MICRO*, 2013.
[26] S. Chen and L. Peng, “Efficient GPU hardware transactional memory through early conflict resolution,” in *HPCA*, 2016.
[27] K. E. Moore *et al.*, “LogTM: log-based transactional memory,” in *HPCA*, 2006.
[28] L. Yen *et al.*, “LogTM-SE: Decoupling Hardware Transactional Memory from Caches,” in *HPCA*, 2007.
[29] L. Hammond *et al.*, “Transactional Memory Coherence and Consistency,” in *ISCA*, 2004.
[30] A. Villegas *et al.*, “Hardware support for Local Memory Transactions on GPU Architectures,” in *TRANSACT*, 2015.
[31] S. Chen *et al.*, “Accelerating GPU Hardware Transactional Memory with Snapshot Isolation,” in *ISCA*, 2017.
[32] D. Dice *et al.*, “Transactional Locking II,” in *DISC*, 2006.
[33] H. Chafi *et al.*, “A Scalable, Non-blocking Approach to Transactional Memory,” in *HPCA*, 2007.
[34] S. Tomić *et al.*, “EazyHTM: EAger-LaZY hardware Transactional Memory,” in *MICRO*, 2009.
[35] A. Levinthal and T. Porter, “Chap – a SIMD Graphics Processor,” in *SIGGRAPH*, 1984.
[36] S. Lam and L. Kleinrock, “Packet Switching in a Multiaccess Broadcast Channel: Dynamic Control Procedures,” *IEEE Trans. Commun.*, vol. 23, p. 891, 1975.
[37] C. S. Ananian *et al.*, “Unbounded transactional memory,” in *HPCA*, 2005.
[38] R. Pagh and F. F. Rodler, “Cuckoo hashing,” in *ESA*, 2001.
[39] A. Kirsch *et al.*, “More Robust Hashing: Cuckoo Hashing with a Stash,” *SIAM J. Comput.*, vol. 39, pp. 1543–1561, 2009.
[40] D. Sanchez *et al.*, “Implementing Signatures for Transactional Memory,” in *MICRO*, Dec 2007.
[41] A. Bakhoda *et al.*, “Analyzing CUDA workloads using a detailed GPU simulator,” in *ISPASS*, 2009.
[42] N. Muralimanohar *et al.*, “CACTI 6.0: A tool to model large caches,” HP Laboratories, Tech. Rep., 2009.
[43] NVIDIA, “NVIDIA’s Next Generation CUDA Compute Architecture: Fermi,” 2009.
[44] H. Wong *et al.*, “Demystifying GPU microarchitecture through microbenchmarking,” in *ISPASS*, 2010.
[45] A. Brownsword, “Cloth in OpenCL,” in *GDC*, 2009.
[46] M. Burtscher and K. Pingali, “An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm,” in *GPU Computing Gems Emerald Edition*. Elsevier, 2011.
[47] V. Vineet and P. J. Narayanan, “CUDA cuts: Fast graph cuts on the GPU,” in *CVPRW*, 2008.
[48] G. Kestor *et al.*, “RMS-TM: A Comprehensive Benchmark Suite for Transactional Memory Systems,” in *ICPE*, 2011.
[49] B. H. Bloom, “Space/Time Trade-offs in Hash Coding with Allowable Errors,” *Commun. ACM*, vol. 13, pp. 422–426, 1970.
[50] D. Cederman *et al.*, “Towards a Software Transactional Memory for Graphics Processors,” in *EGPGV*, 2010.
[51] Y. Xu *et al.*, “Software Transactional Memory for GPU Architectures,” in *CGO*, 2014.
[52] V. Seshadri *et al.*, “RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization,” in *MICRO*, 2013.
[53] D. Dice *et al.*, “Early Experience with a Commercial Hardware Transactional Memory Implementation,” in *ASPLOS*, 2009.
[54] J. Bobba *et al.*, “TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory,” in *ISCA*, 2008.
[55] B. Saha *et al.*, “Architectural support for software transactional memory,” in *MICRO*, 2006.
[56] C. C. Minh *et al.*, “An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees,” in *ISCA*, 2007.
[57] J. Casper *et al.*, “Hardware Acceleration of Transactional Memory on Commodity Systems,” in *ASPLOS*, 2011.
[58] M. D. Sinclair *et al.*, “Efficient GPU Synchronization Without Scopes: Saying No to Complex Consistency Models,” in *MICRO*, 2015.
[59] I. Singh *et al.*, “Cache coherence for GPU architectures,” in *HPCA*, 2013.
[60] X. Ren and M. Lis, “Efficient Sequential Consistency in GPUs via Relativistic Cache Coherence,” in *HPCA*, 2017.
[61] T. Knight, “An architecture for mostly functional languages,” in *LFP*, 1986.
[62] L. Ceze *et al.*, “Bulk Disambiguation of Speculative Threads in Multiprocessors,” in *ISCA*, 2006.
[63] S. H. Pugsley *et al.*, “Scalable and reliable communication for hardware transactional memory,” in *PACT*, 2008.
[64] M. M. Waliullah and P. Stenstrom, “Starvation-free transactional memory-system protocols,” in *ECPP*, 2007.
[65] A. Shriraman and S. Dwarkadas, “Refereeing conflicts in hardware transactional memory,” in *ICS*, 2009.
[66] A. Shriraman *et al.*, “Flexible decoupled transactional memory support,” in *ISCA*, 2008.
[67] ——, “An integrated hardware-software approach to flexible transactional memory,” in *ISCA*, 2007.
[68] P. Felber *et al.*, “Time-Based Software Transactional Memory,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, p. 1793, 2010.
[69] R. Rajwar and J. R. Goodman, “Transactional Lock-free Execution of Lock-based Programs,” in *ASPLOS*, 2002.
[70] H. Litz *et al.*, “SI-TM: Reducing Transactional Memory Abort Rates Through Snapshot Isolation,” in *ASPLOS*, 2014.
[71] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” *Commun. ACM*, vol. 21, p. 558, 1978.
[72] B. Sinharoy *et al.*, “POWER5 system microarchitecture,” *IBM J. Res. Dev.*, vol. 49, p. 505, 2005.
[73] H. Q. Le *et al.*, “IBM POWER6 microarchitecture,” *IBM J. Res. Dev.*, vol. 51, p. 639, 2007.
[74] M. M. K. Martin *et al.*, “Timestamp Snooping: An Approach for Extending SMPs,” in *ASPLOS*, 2000.
[75] N. Agarwal *et al.*, “In-Network Snoop Ordering: Snoopy coherence on unordered interconnects,” in *HPCA*, 2009.
[76] X. Yu and S. Devadas, “TARDIS: Timestamp-based Coherence Algorithm for Distributed Shared Memory,” in *PACT*, 2015.
[77] X. Yu *et al.*, “Tardis 2.0: Optimized Time Traveling Coherence for Relaxed Consistency Models,” in *PACT*, 2016.
[78] A. Meixner and D. J. Sorin, “Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures,” in *DSN*, 2006.