Accelerating GPU Hardware Transactional Memory with Snapshot Isolation

Sui Chen Lu Peng Samuel Irving
Division of Electrical & Computer Engineering
Louisiana State University

ABSTRACT

Snapshot Isolation (SI) is an established model in the database community, which permits write-read conflicts to pass and aborts transactions only on write-write conflicts. With the Write Skew anomaly correctly eliminated, SI can reduce the occurrence of aborts, save the work done by transactions, and greatly benefit long transactions involving complex data structures.

GPUs are evolving towards a general-purpose computing device with growing support for irregular workloads, including transactional memory. The usage of snapshot isolation on transactional memory has proven to be greatly beneficial for performance. In this paper, we propose a multi-versioned memory subsystem for hardware-based transactional memory on the GPU, with a method for eliminating the Write Skew anomaly on the fly, and finally incorporate Snapshot Isolation with this system.

The results show that snapshot isolation can effectively boost the performance of dynamically sized data structures such as linked lists, binary trees and red-black trees, sometimes by as much as 4.5x, which results in improved overall performance of benchmarks utilizing these data structures.

CCS CONCEPTS

• **Computer systems organization** → **Single instruction, multiple data**; • **Computing methodologies** → *Concurrent computing methodologies*;

KEYWORDS

GPU, Transactional Memory, Snapshot Isolation

ACM Reference format:
Sui Chen Lu Peng Samuel Irving Division of Electrical & Computer Engineering Louisiana State University . 2017. Accelerating GPU Hardware Transactional Memory with Snapshot Isolation. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 13 pages.
https://doi.org/10.1145/3079856.3080204

1 INTRODUCTION

Transactional Memory (TM) is a programming model proposed by Herlihy and Moss [10] that enables a series of read and write operations to complete atomically similar to an atomic compare-and-swap command. The operations are encapsulated in transactions that will either succeed and commit to the data store, or abort and restart. A transaction must be aborted if it can result in inconsistent state resulting from concurrent reads/writes by other transactions into the system. Many TM system proposals followed, ranging from hardware to software and hardware-software co-designs [11, 12, 27, 28, 30]. When multi-core and many-core processors emerged, people began to focus on scalability of TM systems and inter-operation of different TM systems as well [23, 28, 31]. Transactional Memory has been implemented in consumer products such as the Intel Haswell and its successors[14].

The GPU is a throughput-oriented computing device characterized by large arithmetic density, high memory bandwidth and a high degree of parallelism, and is continually evolving towards a general-purpose computing device, with growing support for irregular workloads [3] and data structures [22] that are traditionally non-GPU oriented. Recently, hardware-based transactional memory systems for GPUs have been proposed [8, 9], offering performance comparable to fine-grained locking that are as easy to use as coarse-grained locking, making it a very competitive tool for exploiting the full potential of GPUs.

Most existing TM systems implement the 2-Phase-Locking (2PL) concurrency control mechanism, which aborts transactions on write-read conflicts and write-write conflicts. In comparison, Snapshot Isolation (SI) is another mechanism that only aborts on write-write conflicts and can greatly improve performance. However, at the same time SI permits the Write Skew anomaly, which must be addressed to obtain correct outputs.

In this paper, we propose applying SI to a GPU-based hardware transactional memory system in order to achieve better performance for complex transactions involving dynamically sized data structures such as the linked list and search trees, and a solution for eliminating the Write Skew anomaly on the fly. Specifically, we make the following contributions:

- We devise a versioned memory system for quickly creating versions and enabling Snapshot Isolation on the GPU;
- We propose a method for detecting dependency loops on-the-fly and eliminating the Write Skew anomaly that can scale to hundreds of concurrent transactions on the GPU;
- We demonstrate that the proposed SI-based hardware TM on the GPU brings speedup over the baseline system that aborts on all read-write conflicts in applications that utilize dynamically-sized data structures.

The rest of the paper is organized as follows. Section 2 introduces related background. Section 3 describes the organization and operation of the multi-versioned memory system. Section 4 describes

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080204

the mechanism for detecting and eliminating the Write Skew anomaly. Section 5 discusses how the proposed changes interact with the existing GPU architecture. Sections 6 and 7 discuss experimental results and we list related works in Section 8 and conclude the paper in Section 9.

2 BACKGROUND

2.1 Transactional Memory and Snapshot Isolation

A TM system usually provides a certain isolation level by implementing one concurrency control mechanism. The purpose is to make sure concurrent read-writes always result in valid system states. This is done by detecting and resolving conflicts, which is a condition when when two or more transactions access one data item simultaneously and at least one of them is a write. A frequently-used conflict resolution method is to abort all but one transactions in a read-write or write-write conflict. With this mechanism, concurrent accesses will be split into disjoint sets with no intersections in between. This is usually used with the 2-Phase-Locking concurrency mechanism, with the first phase called the "expanding phase", when transactions compete for and acquire locks, and the second phase called the "shrinking phase", when locks are released.

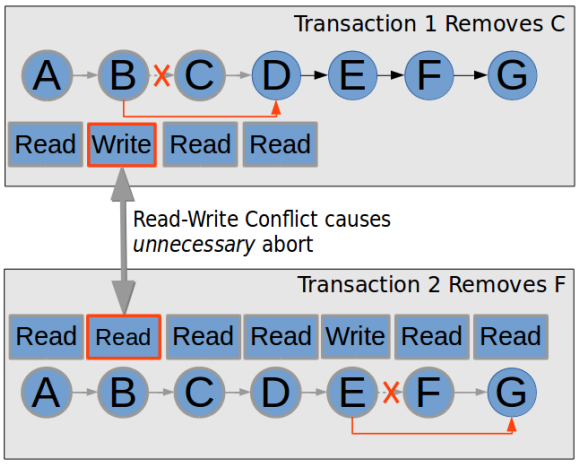

Figure 1: Example of unnecessary aborts in a linked list.

This conflict resolution mechanism is pessimistic in that it may abort more transactions than what is necessary to maintain a consistent program state. An example of why it can result in unnecessary aborts can be seen from Figure 1: Transactions T1 and T2 attempt to remove nodes **C** and **F** from a linked list. Both transactions need first iterate through the list to find the element to be removed and its neighbors and then change the list. In this example, T1 modifies **B** in order to remove **C** from the list, but T2 has read node **B** when it was looking for node **F**, so this pair is considered to be in conflict and one of them has to be aborted. But this abort is not necessary since the outcome of T1 and T2 both committing is still a valid linked list with nodes **C** and **F** removed.

The pessimistic approach is one reason why transactional memory systems sometimes run slower than fine-grained locking. In fine-grained locking, the critical section can be made small enough to lock only the most relevant data and block the smallest set of conflicting transactions; the lock itself can also be acquired using the best strategy. Using the same example in Figure 1, when T1 locks nodes **B**,**C** and **D** and T2 locks **E**,**F**,**G** using fine-grained locking, the abort can be avoided. However, fine-grained locking requires extra efforts, for example, a lock-sorting algorithm to avoid live-locks and deadlocks, so it is usually a much more difficult task than simply using transactional memory.

In contrast, Snapshot Isolation (SI) [1] significantly differs from 2-Phase Locking, as it allows write-read conflicts to pass. SI makes a guarantee that every transaction will see a consistent snapshot of the entire system and it will successfully commit only if it doesn't conflict with any concurrent updates made during its life time. This allows a transaction to commit its tentative changes in isolation, without being affected by other transactions. In the basic SI protocol, read operations will always complete and only write-write conflicts will be aborted.

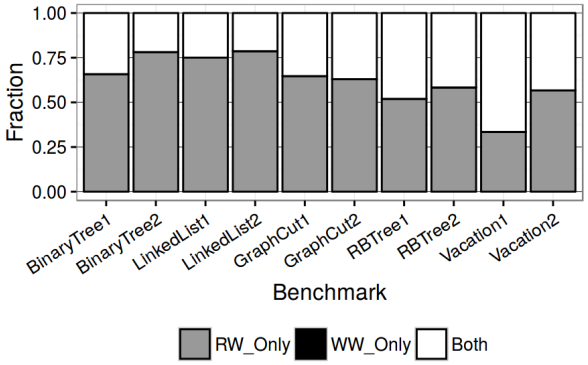

Figure 2: Types of conflicts between transaction pairs in some GPU TM workloads (*Write-write only* conflict pairs do exist, but their numbers are negligible.)

SI has the potential of greatly accelerating applications which involve many transactions aborted by read-write conflicts. For example, the breakdown of conflict types between temporally-overlapping transaction pairs in some GPU programs is shown in in Figure 2. From the figure it can be seen that transactional applications utilizing data structures such as binary search trees, linked lists and red-black trees contain a large portion of read-write only conflicts. These data structures are dynamically sized and are much easier to implement using TM than fine-grained locks.

However, Snapshot Isolation suffers from a well known problem called the Write Skew anomaly. The problem is that transactions produce correct results when running alone from their own snapshots can produce incorrect results when running together. This is a result of SI not providing full serializability, and must be addressed for an SI-based system to execute correctly and be useful. One method to ensure correctness is to identify the cause of the anomalies, either manually or by static/dynamic analysis and modify the application accordingly by introducing artificial locking or introducing write-write conflicts [2, 19]. There exist educational campaigns for database users to make them aware of and be able to prevent the

Write Skew anomaly. One notable theoretical foundation to these methods is types of dependency graphs, the Read Dependency Graph [5], which represents the relative serialization order of transactions that can create a system state equivalent to when transactions are concurrently executed.

A recent application of Snapshot Isolation to transactional memory is found in the SI-TM system [18] running on the CPU. It resolves the Write Skew anomaly through a trace-driven approach [19]: transaction execution traces are generated during runtime to form a Dynamic Dependency Graph (DDG) as defined by Fekete [5]. The DDG is collapsed into a Read Dependency Graph (RDG), with edges in the RDG representing the dependency loops in DDG, and vertices in the RDG representing the source code locations that generate them. The problem of eliminating dependency loops is solved by choosing a set of the read operations ("dangerous reads"), converting them into writes to introduce run-time write-write conflicts, thus eliminating the Write Skew anomaly. As with many test-driven approaches, the coverage of skew detection depends on the size of the sampled runs. To catch all possible "dangerous reads", a large number of experiments may be required, which may limit the usefulness of this approach under certain circumstances.

Recent developments in database [2] and coherence [29] have demonstrated ways to eliminate cyclic read-write dependencies in database systems and in memory dependency analysis. These works have outlined the building block for preventing the Write Skew Anomaly on-the-fly in a transactional memory system, thus saving the user from having to obtain execution traces and manually fix application source code.

3 VERSIONED MEMORY SUBSYSTEM

To meet the guarantee of Snapshot Isolation that transactions be able to see their own snapshots in isolation, an address should be allowed to map to multiple machine words, each of which being a version (we use the terms "version" and "snapshot" interchangeably in this paper.) This is similar to a version control system or a checkpointing infrastructure, where new versions may be appended to reflect the changes of the tracked word, and old versions are kept for access to the word's history. In terms of its semantics, *a version is the previous version combined with the changes between the two versions.* In the example shown in Figure 3, a linked list initially containing 7 nodes could have 2 nodes removed, then one more node removed, with 4 nodes left in the end. After each batch of removals, the linked list becomes a new version of itself, and any version itself is a valid linked list. Versions are also commutative: In the example, Node D is removed from Version 1, but may as well be removed from Version 0; in either case, when all 3 nodes are removed, the result will be Version 2. In this sense, a versioned memory system may be implemented as a collection of full snapshots, or the combination of snapshots and change sets. In our study, we choose the former, as the quick row copy primitive makes it easier to create full snapshots than to keep track of change sets.

Transactional memory systems serve as a mechanism to handle conflicts, a result of accesses from different threads that overlap both in space and time. This means there exist spatial and temporal locality in such access patterns. The locality is more noticeable in GPUs, because of the large number of concurrent threads, as well as the lock-step execution pattern of threads in a warp. On the other hand, the high degree of parallelism also requires a high speed in creating versions.

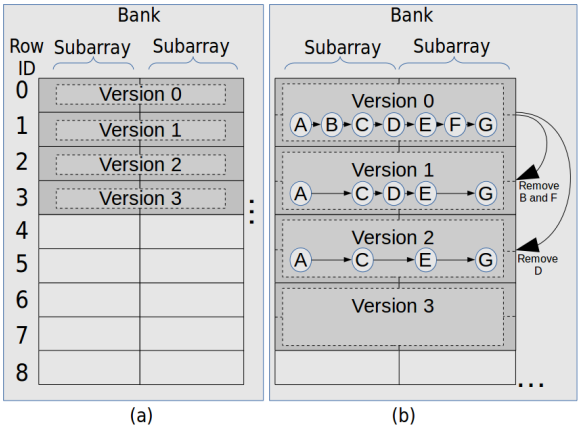

Figure 3: Versions and rows in a subarray of a bank of DRAM. A version may span (a) multiple subarrays and (b) multiple rows. Versions track the evolution of data structures (shown in (b)).

Because of this access pattern and the requirement, we identify RowClone [26] as a promising building block for our versioned memory system. RowClone exploits the internal organization of DRAMs to copy multiple kilobytes of data referred to as "rows" completely within memory, enabling fast creation of snapshots.

A DRAM can be thought of as a collection of rows, each many kilobytes in size. The entire memory is divided into multiple banks that can operate independently and each bank is made up of multiple subarrays that have limited physical size to keep signal transmission time short. A number of DRAM rows that belong to the same *subarray* are connected to a common sense amplifier which are able to read and write the rows. By connecting multiple rows to the amplifier, data can be copied from one row to another, eliminating the need to copy data through the processor, all at the granularity of a row buffer, which is generally larger than cache lines. As such, the row copy operation can reach large bandwidth with very small energy consumption.

Although a "DRAM row" shares the same name as a "database row", a "DRAM row" is not a smallest atomic unit as a "database row" is; the atomic unit in a transactional memory is usually a machine word, and a DRAM row is treated as and managed as a collection of many machine words. This suggests the granularity of the RowClone operation (multiples of DRAM rows) is a bulk operation rather than a fine-grained operation, and is more suitable for copying whole snapshots than keeping fine-grained track of change sets.

Given the RowClone mechanism we intend to use, the versioned memory subsystem should do the following: 1) Layout the versions in a subarray-aware fashion such that the versions may be created using row copy, 2) Fetch a specific version to service accesses quickly, 3) Manage the lifetime of versions by creating and

recycling versions, and 4) Remap the memory such that accesses to versioned/non-versioned regions works as expected.

Layout of versions Knowing that RowClone operates by copying between rows belonging to a same subarray and that version management requires multiple copy operations, we allocate rows that belong to the same subarray to the storage of versions, so that the copy operations can be made fast.

A version may be larger than the row buffer size. In this case, mulitple rows will have to be allocated for a single version, as indicated in case (b) of Figure 3.

Version lifetime management Logically, the lifetime of a version consists of 4 states (except Version 0, the initial version created at program start and will enter state S3 by definition,) some of which allow reads/writes:

State	Read	Write
S1. Space of the version is allocated and contents of the previous version is copied to it.	No	No
S2. The changes made by the committing transactions from a GPU thread warp (executed in lockstep) are being written to this version. Multiple non-conflict changes can be written to it; conflicts are resolved by aborting some transactions.	No	Yes
S3. When warp commit is finished, the version will be assigned a creation time and will not be written to; future transactions may start reading from it.	Yes	No
S4. The version is recycled, marked as unreadable when no transactions are reading from it anymore, and its space is recycled.	No	No

In addition to copying and recycling, transactional reads and writes interact with the versions as well. An example of how copying and recycling are involved in the different states is shown in Figure 4:

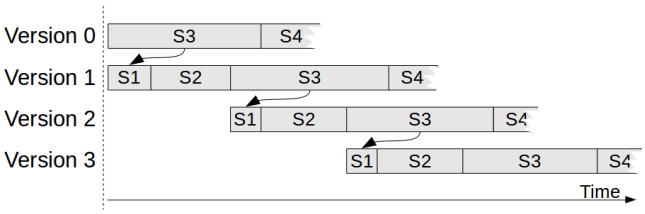

Figure 4: Life cycles of versions and the changes in the states (S1-S4). Arrows denote copy operations.

In this example, four versions are shown. When there exists a version at state S3, its next version will enter state S1 and start copying. When a next version enters state S2, it may be used as a write target. At any given time, there exists at least one version that can be read from, and when there is not an ongoing copy, there exists one version to be written to. During the copying, transactions that are executing will read from previous versions which are in state S3. Therefore, the copy operations are overlapped with the transaction execution.

The total number of versions allocated is a parameter determined at kernel launch and due to the limit, versions are recycled and enter state S4 when there are no references to them.

Accessing a version A version is accessed with an address and a timestamp. The timestamp is compared with the timestamps of the existing versions that are in lifetime state S3, and the latest version that is created before the given timestamp will be accessed.

Memory mapping In the multi-version memory scheme, a version may cause multiple regions in the device memory space to become unavailable, similar to the way in which system memory is shared between the CPU and the GPU in certain heterogeneous architectures. Depending on the memory mapping scheme, versions can take up multiple disjoint regions in the memory space. For example, suppose we have the following address mapping scheme:

```
Bit ID    31  27          16         7      0
          |   |           |          |      |
Bit Usage 0000RRRR RRRRRRRR BBBBCCCC CCCCCCCC
```

In this mapping scheme, R, B and C denote row, burst and column addresses, respectively. Suppose addresses 0x800000-0x8003FF is allocated 8 versions, each with a size of 0x400 bytes, then addresses 0x801000-0x8013FF, 0x802000-0x8023FF, ... 0x807000-0x807FFF will be occupied. In this example, accesses to those addresses will be redirected to a reserved space starting from 0xA00000, as illustrated in Figure 5.

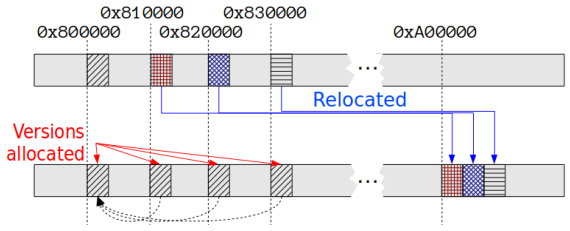

Figure 5: Memory mapping and reallocating scheme

The purpose of relocating the space occupied by versioned regions is to recover a continuous memory space.

Connecting all components with the Version Index Table To put the proposed multi-versioned memory in action, the list of versioned memory regions and their details are stored in the Version Index Table. The details include the length of the regions, the relocated location for the space occupied by the versions, the creation times of the respective versions, and the reference counts to the versions. The table is located on the level as the L2 cache, which all DRAM requests pass through. A program is allowed to allocate multiple versioned memory regions. In such cases, there will exist multiple entries in the Version Index Table.

Version Index Table

Start Addr	Length	Relocated	Creation Time	RefCount
0x800000	0x400	0xA00000	{17,123,193,255}	{1,23,50,5}
0x880000	0x10000	0xA10000	{12,144,156,300}	{0,9,35,47}

Figure 6: Version Index Table

An access will be handled based on whether it is versioned or not. In the case of a versioned access, the version will be retrieved by the accompanying timestamp. For the example in Figure 6, there exists two versioned memory regions, each having 4 versions. Access to 0x800000 in the first versioned region, when given timestamp

(1. Predecessor receives successor's PDTS and computes the *logical* timestamp to be sent to the successor)

```
1  OnPredecessorReceivesTS(pred, succ->PDTS) {
2    if (pred is committed) return PrevTS
3    if (succ->PDTS == -1) {
4        if (pred->PDTS == -1)
5            pred->PDTS = pred->CTS + 10
6    } else if (pred->CTS >= succ->PDTS) {
7        if (pred->PDTS == -1)
8            pred->PDTS = pred->CTS + 10
9    } else {
10       midpoint = (pred->CTS + succ->PDTS) / 2
11       if (pred->PDTS > midpoint)
12           pred->PDTS = midpoint
13   }
14   return pred->PDTS
15 }
```

(2. Successor receives the updated *logical* timestamp from the predecessor)

```
16 OnSuccessorReceivesTS(pred, succ, timestamp) {
17   if (succ->PDTS != -1 && timestamp >= succ->PDTS) {
18       // Declare a dependency loop
19   } else {
20       succ->CTS = ts + 1
21   }
22 }
```

(3. At commit or abort)

```
23 AtCommitOrAbort(txn, ts) {
24    txn->PrevTS = txn->CTS
25    txn->CTS = ts + 1
26    txn->PDTS = -1
27 }
```

Figure 7: Transaction-Level dependency loop detection *logical* timestamp updating operations

between 123 and 192 inclusive, will be directed to 0x810000. A non-versioned access to 0x810000 will be relocated and be reading from 0xA00000. Other non-relocated accesses will proceed as normal.

Reference Counts are incremented by transactions that read from the region for the first time and decremented by the aborting/committing transactions that have read from the region.

4 RESOLVING WRITE SKEW ANOMALY

According to prior works [19], a necessary condition for the Write Skew Anomaly to occur is the presence of inter-transaction write-read dependency loops in the Dynamic Dependence Graph of a program. Thus, eliminating dependency loops prevents the Write Skew Anomaly from happening. For this purpose, we devise a mechanism for removing Write Skew Anomalies by detecting and breaking write-dependency loops that arise during our Snapshot Isolation concurrency mechanism.

4.1 Detecting Dependency Loops Using Post-Dating

One issue inherent to GPU is scalability, which is a result of its enormous concurrency. Because of the presence of up to thousands of concurrent threads scattered throughout the system, the cost of communication can be high and the designer of a system has to think carefully when deciding where to use broadcasts. This also makes a precise graph tracking mechanism impractical on the GPU, because the size of the graph grows as the program runs.

In order to avoid building graphs with unbounded size, we choose to detect dependency loops (We use the term *dependency loop* to

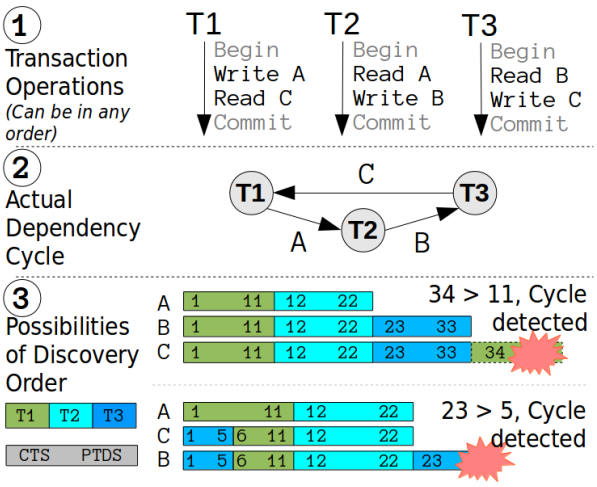

Figure 8: Detecting dependency loops using post-dated *logical* timestamps and an increment of 10. Note that discovery starting from B and C can be made equivalent to the illustrated example through rotating the edge numbers.

denote *cyclic dependencies*, to avoid confusion with *clock cycles*) on the fly using a *post-dating* mechanism similar to TimeTraveler [29], which uses this method to detect memory race loops. The algorithm is shown in Figure 7. With a little modification, this method fits for our purpose, allowing us to track dependency loops between transactions pairs, when both transactions are executing, as well as between executing and committed transactions.

In the post-dating method, each transaction is assigned 3 integer numbers that are updated throughout the post-dating process, which are called the current *logical* timestamp (CTS), the post-dated *logical* timestamp (PDTS), and the previous *logical* timestamp (PrevTS,) as shown in Figure 7. (It should be pointed out that the timestamps reflect the ordering between the transactions, *not* the time at which the transactions start, or when transactional accesses take place, thus they are called logical timestamps, to avoid confusion with the time-based timestamps associated with the snapshots.)

When two transactions write and read the same address, a dependency edge between these two transactions is formed, pointing from the writer being to the successor. The purpose of post-dating is to detect loops by updating the logical timestamps of predecessors and successors using a simple rule and checking the invariant that *a transaction's CTS must be smaller than its PDTS*, when the invariant is violated, a dependency loop is declared.

An example of the operation of the post-dating mechanism involving 3 transactions and 3 dependency edges is illustrated in Figure 8. In the beginning all transactions are initialized with (CTS=1, PDTS=-1). Between transaction begin and commit, all the write and read operations (①) go through the post-dating process. When a pair of read and write operations on the same data is seen by the post-dating mechanism, a dependency edge is discovered, which we denote as edges A, B and C (②). Depending on the order in which the three edges are discovered by post-dating, the three transactions may obtain different CTS and PTDS logical timestamps (③). When only

two out of the three edges are discovered, the invariant is still maintained, but when the third edge is constructed, one of the transactions will discover a violation of the invariant.

In the first case in ③, the three edges are discovered in the order of A, B, C. When C is being discovered, T1 must set its CTS to 34 since it's a successor to T3 in write-read dependency edge C, and T3 has a PTDS of 33, but 34 is greater than T1's PTDS of 11, thus violating the invariant. In the second case where the order is A, C then B, and when B is being discovered, T3 must set its CTS to 23 because it's the successor to T2 in edge B, but 23 is greater than T3's PDTS of 5, thus violating the invariant.

For the other cases, B,C,A and C,A,B are equivalent to A,B,C; B,A,C and C,B,A are equivalent to A,C,B. This can be proven by rotating the edge names. In all the cases, the loop will be discovered.

When the edges involve committed transactions, the committed transactions' CTS and PDTS will be set as usual; the only difference is when the committed transaction serves as the predecessor, Line 2 in Figure 7 will be executed and the PrevTS will be returned.

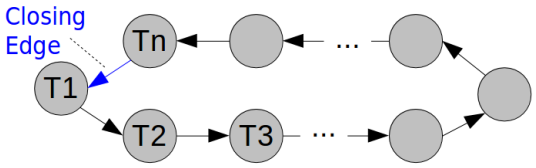

Figure 9: Post-dating for loops involving more than 3 transactions.

A dependency loop involving more than 3 transactions such as the one in Figure 9 can also be discovered by the post-dating algorithm because:

- Due to transitivity, given dependency $T_i \rightarrow T_j$ in a chain of dependency edges, the invariants will always hold before a dependency loop is discovered, regardless of the order in which the dependency edges are post-dated: 1) PDTS of T_i must be greater than CTS of T_i; 2) PDTS of T_j must be greater than PDTS of T_j and 3) CTS of T_j must be greater than PDTS of T_i.
- When the closing edge of a dependency of a loop is discovered, it will cause the aforementioned invariant to be violated, thus declaring a dependency loop.

The violation may also be caused by the logical timestamps running out of precision, resulting in false positives. Because false positives can only lead to aborts, it does not affect the correctness of the transactional memory system.

4.2 Making Post-Dating Scalable

In order to make run-time costs manageable, we have made the design choice of a *Single-writer* paradigm, allowing only one writer to own one machine word during the transaction execution stage. The choice of single-writer paradigm is reasonable in that the baseline Snapshot Isolation aborts on write-write conflicts, so the additional writers are likely to get aborted.

With the single-writer constraint, the difficulty of tracking dependency is reduced, because it now becomes feasible to track the sole writers of the currently active words in this system. We decide to put the Writer Table that stores the writer of words on the same level as the L2 cache, so it is accessible to all the SIMT cores, and is on the path of all transactional reads and writes. Transactions consult this table to perform post-dating.

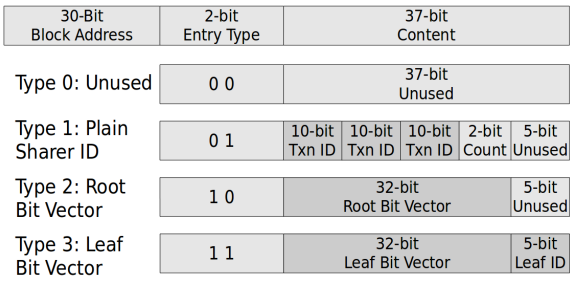

Figure 10: Structure of the Scalable Readers List.

To enable the writers to acknowledge all readers of the request to update logical timestamps, we choose to implement a Scalable Reader List, an efficient, scalable and exact scheme for representing readers similar to the SCD directory [25]. This scheme relies on efficient highly-associative caches proposed in the ZCache[24] and the Cuckoo Directory [7].

The Scalable Reader List is a directory with three types of entries illustrated in Figure 10. The table is indexed by the 30-bit block address. Given a block address, the relevant table entries are fetched, which can be one of the four following types:

- Type 0, Unused: the entry does not contain any useful information and can be deallocated.
- Type 1, Plain Sharer ID: The entry contains 1 to 3 10-bit Transaction IDs. The number of entries is stored in the 2-bit field.
- Type 2, Root-Level Bit Vector: Each bit in the 32-bit vector indicates whether one of the 32 leaf-level bit vectors are present.
- Type 3, Leaf-Level Bit Vector: Each bit in the 32-bit vector indicates whether each transaction in the group of 32 transactions is in the reader's list. The 5-bit field indicates the position of the Leaf in the Root and is used as an offset in the 1024-bit space. For example: the 11-th bit in the 21-th Leaf means the transaction with an ID of $32 \cdot 21+11=683$.

When a transaction reads an address, the read address and the reader will be appended to the Scalable Reader List. A writer uses this list to find the writers and update the logical timestamps.

The organization of two-level 32-bit vectors allows at most 1024 concurrent transactions, which is greater than the number of concurrent transactions used in our study. More concurrent transactions can be achieved by increasing the number of bit vector layers.

4.3 Overhead

The overhead of the components required for detecting dependency loops is listed in Table 1, which is estimated based on CACTI [17] using 40nm technology node. The total size of the four tables is designed to be 768kB, which may be increased when necessary. Currently the storage is split into the following parts:

Table 1: Area and Power Overhead of the Dependency Loop Detection Components

	Size	Area(mm^2)	Power(mW)
Writer Table	15kB	0.12	31.5
Logical Timestamp Table	12kB	0.01	37.8
Version Index Table	1kB	0.005	3.2
Scalable Reader List	740kB	5.0	2330

- The **Writer Table** contains 3000 entries. The number is chosen to match the capacity of the Scalable Reader List. Each entry in this table takes 40 bits (30 bits block address plus 10 bits writer ID), with a total size of 3000·40/8=15kB.
- The **Logical Timestamp Table** contains 1024 entries in total to accommodate 1024 concurrent transactions. (In our experiments, the number is limited to 960 to match the baseline system.) The entries are directly mapped to each transaction, each containing three 24-bit logical timestamps, with a total size of 1024·24·3/8=12kB.
- The **Version Index Table** takes 1kB. Each entry in this table takes 193 bits (2×30-bit original and relocated block addresses, 5 bits versioned area size and 8×32-bit creation times such that 60+8·32+5=193 bits). The space can accommodate 1024/(193/8)=42 versioned regions.
- The **Scalable Reader List** takes the rest of the space, which is 768-15-12-1=740kB. With each entry taking 69 bits (30 bits block address, 2 type bits, 37 bits content), the space accommodates 740·1024/(69/8)≈87856 entries. This translates to 87856/33=2662 addresses in the worst-case sharer scenario (each address taking 33 entries). Since worst-case does not always happen, the space is large enough to cover the same number of unique addresses as the Writer Table (3000).

Post-dating based loop detection are overlapped with loads and writes and therefore do not block. It's only required that all logical timestamp updates originating from a transaction must be completed before the transaction starts committing.

When any loop-detection tables (Writer Table or Scalable Readers List) overflow, all future transactions, as well as transactions trying to obtain entry in any of those tables, will abort and wait for current transactions to either commit or abort and free up space in the tables.

In the case timestamp in the Version Index Table overflows, all current versions will be invalidated, all concurrent transactions will abort, and the timestamps will be reset to zero.

In the case the post-dating timestamps (CTS and PDTS) overflow, the same strategy may be applied to abort all currently executing transactions and restart the CTS and PDTS to zero.

5 INTERACTION WITH EXISTING GPU HTM

Figure 11 shows the hardware components added to the baseline GPU HTM hardware, WarpTM [8]. These components enable SI and address the GPU-specific challenges for Write Skew Anomaly elimination: First, GPUs usually lack the ability to broadcast cache line invalidation messages across cores as on the CPU; Second, the single-writer paradigm and the corresponding components help keep dependency tracking cost low.

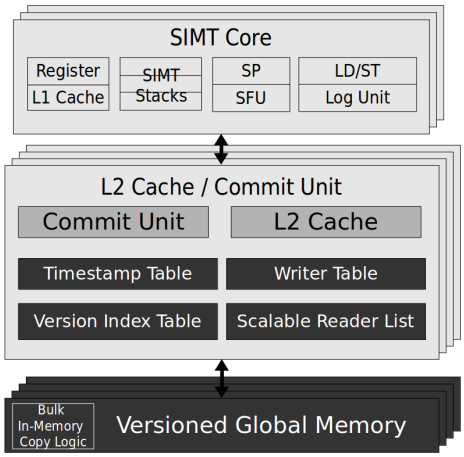

Figure 11: The SI-enabled GPU HTM architecture with GPU-specific components.

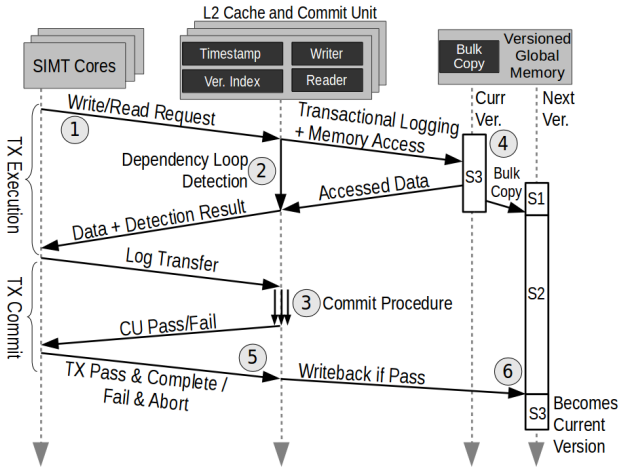

Figure 12: Transaction execution flow.

Figure 12 shows how the hardware components are involved in the execution of transactions. In the beginning, transactions perform access to snapshots in the **versioned memory** (step ①) by providing an address and its beginning timestamp, and the corresponding version is accessed through the Version Index Table, using the process described in Section 3 mean time. Transactional logging is also performed in the meantime.

In the meantime, **dependency loop detection** (step ②) is performed. For a write request, the writer's ID will be inserted into the Writer Table if no other writers to the address exists, and all the readers of the same address are extracted from the Reader List and their timestamps are updated using the post-dating mechanism; if there already exists a writer, the new writer is aborted, otherwise, the writer with a larger ID will abort. For read requests, the reader's ID will be inserted into the Scalable Reader List, the writer of the read address will be found from the Writer Table, and the same post-dating process will be run. The operations overlap with the

memory accesses and transactional logging in step ① which involve global memory writes. These processes are usually long enough to hide the post-dating latency. The loop detection results will be sent back to the transaction that made the access to abort transactions that may form dependency loops.

We do not need to peform intra-warp conflict detection as WarpTM does before a warp commits. The reason is intra-warp conflict detection aborts on all intra-warp read-write conflicts, which is not desirable in Snapshot Isolation. The write sets of transactions are sent to the multiple Commit Units based on the memory partition the write destinations are in.

The function of the Commit Units in our system is simpler than that in WarpTM and KiloTM. With SI, the Commit Unit only need to check transactional writes (the single-writer limit apples to running transactions only, so it is still possible for multiple writes to the same address to exist in the commit stage) and aborts transactions when the writes are in conflict with other committing transactions. For this reason, the Commit Unit only does not need to validate read requests and the read log is not transferred to the Commit Unit as in WarpTM. The commit procedure starts in each of the Commit Units when the logs have been transfered to it (step ③). The transaction IDs will be removed from the Writer Table or Scalable Reader Lists once a transaction passes the commit procedure.

During transaction execution and commit stages, the **Bulk In-Memory Copy Unit** will copy the content of the current snapshot, currently in state S3 to the next snapshot, currently in state S1 (described in Figure 4), in an asynchronous fashion(step ④). It scans the addresses in the regions indicated in the Version Index Table and issues copy commands to the corresponding rows, which will then copy the rows in parallel in the respective subarrays.

When the commit procedure is completed, the SIMT core will collect the results from the related commit units. When there is no write-write conflict in all its related Commit Units, the transaction is considered to have passed validation (step ⑤), and instructs the related Commit Units to write its changes to the new snapshot in state S2. Because GPU warps execute in lock step, a warp of transactions will be committing and writing back simultaneously, which allows the changes to be merged, reducing the number of versions that need to be created. When transactions from this warp finish writeback, the state of the snapshot will be changed from S2 to S3 (step ⑥), becoming the "current transaction" from the future transactions' perspective, and will be given a stamp and be read from, as described in Section 3. The execution flow repeats from step ① for the next transaction.

Number of concurrent snapshots We set a finite number of concurrent snapshots in the system due to capacity and practicality considerations. As described in Section 3, there may exist multiple snapshots in state S3 which are available for read. The more concurrent snapshots, the more recent the versions read from by transactions will be.

A transaction chooses the snapshot to read from by its transaction start time and the timestamp of the snapshots, namely the latest version available at its start time. With a limited number of concurrent transaction, a transaction will be forced to read an older snapshot. This still results in correct execution; however, it is similar to having the transaction start much earlier but get stalled until its actual start time, which may increase the chance of an abort.

When a transaction starts, the reference count of the snapshot it's using is incremented by 1. When it commits or aborts, the count is decremented by 1. When it restarts from an abort, it will choose the latest version in the pool of available snapshots, so older snapshots will eventually be referenced by no transactions. Then, it will be garbage-collected to make room for new versions.

Note that some applications may not have dynamic data structures as those shown in Figure 2, and SI won't have performance benefits for those applications. We will turn off the SI mechanism and use normal baseline configuration to avoid runtime overhead in this case.

Table 2: GPGPU-Sim Configuration

GPU	
SIMT Cores / SIMD Width	15 / 16 x 2
Warps/Threads per Core	48 warps × 32 = 1536 Threads
Memory Partitions	6
Core/Interconnect/Memory clock	1400/1400/924 MHz
Warp Scheduler Count	2 per Core
Warp Scheduler Policy	Greedy-then-oldest
L1 Data Cache per Core	60KB / 48KB, 128B Line, 6-Way (Not caching global accesses)
Shared Memory per Core	16KB
Interconnect Topology	1 Crossbar per direction
Interconnect Bandwidth	32B/cycle = 288GB/s per direction
Interconnect Latency	5 Cycles to traverse
DRAM Scheduler	Out-of-order, FR-FCFS
DRAM Scheduler Queue Size	16
DRAM Return Queue Size	116
DRAM Timing	Hynix H5GQ1H24AFR
Min DRAM Request Latency	330 Compute cycles
Transactional Memory	
Concurrency Control	2 Warps per core (960 Concurrent Txns)
Commit Unit Clock	700 MHz
Validation/Commit BW	1 Word per cycle per CU
Warp TM-Specific	
L2 Cache for all Cores	256KB × 6 partition = 1536KB
Intra-Warp CD Resources	4KB Shared memory per warp
Intra-Warp CD Mechanism	2-Phase Parallel Conflict Resolution
TCD Last Written Time Table	16KB (2048 Entries in 4 sub arrays)
TCD Detection Granularity	128 Byte
Snapshot Isolation-Specific	
L2 Cache for all Cores	128KB × 6 Partitions = 768KB
Writer Table	15KB
Scalable Reader List	740KB
Timestamp Table	12KB
Version Index Table	1kB
Concurrently Active Versions	8
Row Copy Size	2KB Per Subarray
Row Copy Latency	68 ns
Number of Subarrays	64 per bank

6 EXPERIMENT SETUP

We have extended the WarpTM system using GPGPU-Sim 3.2.1 [6], which simulates a device resembling the NVidia GTX480. WarpTM also includes the baseline KiloTM, with the difference being that it does not have intra-warp conflict detection.

We use applications utilizing dynamically sized data structures that involve many write-read conflicts. These applications are listed in Table 3, which we describe below:

Linked List is a linked list, which allows both concurrent insertions and deletions, where the *early-release* technique is not available. In Linked List 1 and 2 we spawn 100 and 200 threads respectively, each of which inserts one element in the linked list by

Table 3: Benchmark Properties

(Transaction length and read/write set size measured under baseline (WarpTM) settings)

Name	Threads	Read/Write Set Size	Average Tx Length (Cycles)
Linked List 1	100	114 / 4	17646
Linked List 2	200	240 / 4	59876
Binary Tree 1	1000	33 / 8	327025
Binary Tree 2	100	36 / 6	26910
GraphCut1	32	37 / 3	80859
GraphCut2	32	52 / 5	84249
Red Black Tree 1	200	47 / 14	86501
Red Black Tree 2	400	47 / 14	131218
Vacation 1	150	92 / 4	241258
Vacation 2	140	144 / 12	338872

first iterating to the insertion point then perform the insert. The lists are initially empty.

Binary Tree is a binary search tree with pre-initialized elements, which allows insertion, deletion and query operations. In Binary Tree 1, we initialize a tree with 1000 elements and spawn 1000 threads, each removing one element from the tree. In Binary Tree 2, we initialize a tree with 1000 elements and spawn 100 threads to perform 50 insertion and 50 remove operations.

GraphCut is an implementation of Karger's randomized algorithm [16] for computing the minimal cut of a graph. In Graph Cut 1 we solve a graph with 60 vertexes and 180 edges with 32 threads. In Graph Cut 2 we solve a graph with 90 vertexes and 270 edges with 32 threads.

Red Black Tree is the red-black tree implementation from the RSTM suite [20], which we adapted to the GPU. In Red Black Tree 1 and Red Black Tree 2 we initialize a tree with 1000 elements and perform 50/50 and 100/100 insertion/remove operations respectively.

Vacation is the Vacation benchmark from the STAMP benchmark suite [21], which we adapted to CUDA while keeping the semantics unchanged. It includes its own linked list and red-black tree implementation. Vacation Benchmark simulates a travel reservation system, which includes a database with four tables: Flight, Room, Car and Customers. Each Customer has a linked list storing the reservations made. A reservation points to flight, room or car. Thus, Vacation is similar to having 4 red-black trees and 1 linked list running simultaneously. In Vacation 1 we perform all "table-modifying" operations with no user queries, i.e. we add or delete entries in the 4 red-black trees all initialized with a size of 500 entries. In Vacation 2 we simulate 5 customers and all threads make reservations for the 5 customers. The table sizes are initialized to be 1000.

We run the benchmarks using 4 configurations as listed in Table 2:

- **WarpTM** is the baseline GPU hardware TM, with intra-warp conflict resolution.
- **KiloTM** is the other pre-existing system, which is WarpTM without intra-warp conflict resolution.
- **SITM_TS** is the Snapshot Isolation-enabled TM with loop detection using post-dated timestamps, with the single-writer restriction.
- **SITM_Oracle** is the Snapshot Isolation-enabled TM with perfect loop detection (we construct a dynamic dependence graph in the simulator) with no cost. There is no single-writer restriction for SITM_Oracle. SITM_Oracle has no cost in post-dating or row copy. (This is ***not*** achievable in reality.)

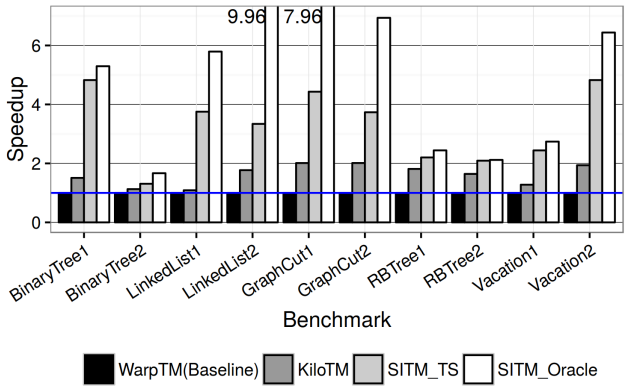

Figure 13: Overall Speedup.

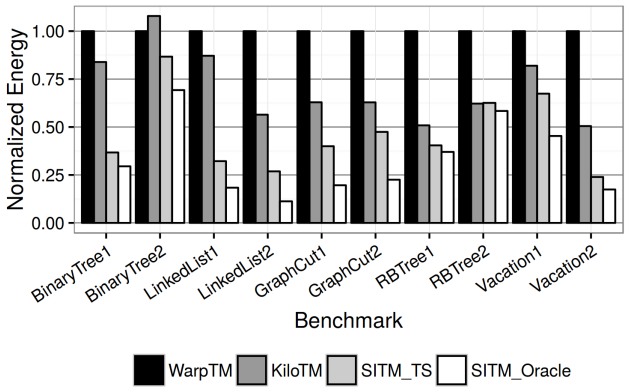

Figure 14: Normalized Energy Consumption.

For a fair comparison, WarpTM and KiloTM are given a larger L2 cache as indicated in Table 2 (1536kB), to account for the extra space taken by cyclic conflict detection in Snapshot Isolation.

7 RESULTS

Figures 13 and Figure 14 show the overall speedup and energy consumption of the benchmarks. Overall, the baseline WarpTM is the slowest among all of the benchmarks. In fact, it is slower than KiloTM, which indicates that intra-warp conflict detection based only on address does not help with the benchmarks used in our experiments. Since intra-warp conflict resolution aborts on *all* write-read conflicts, we believe this is the reason why it is slower than KiloTM. For the same reason, KiloTM is slower than both Snapshot Isolation-based systems because it aborts transactions with write-read conflicts in the Commit Unit. This is most obvious in the Linked List benchmarks. By comparing Linked List benchmarks, it can be

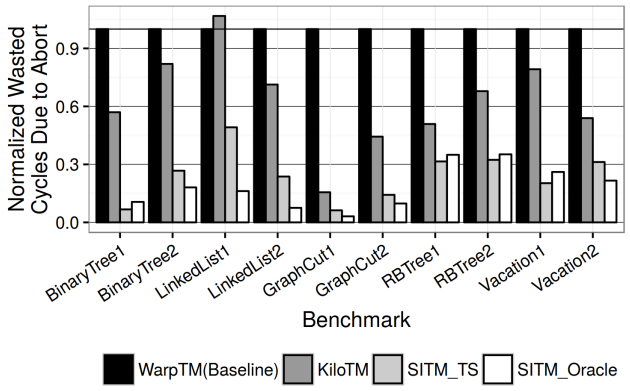

Figure 15: Normalized Wasted Work due to Aborts.

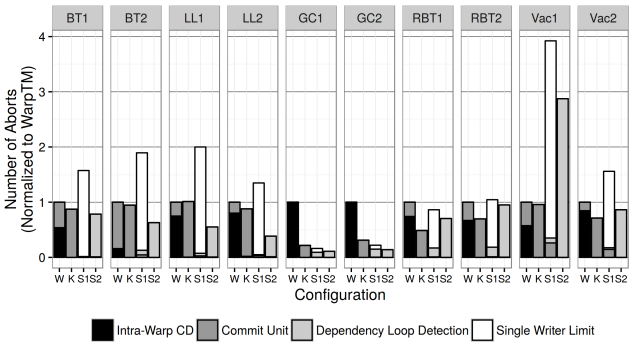

Figure 16: Breakdown for Abort Reasons.

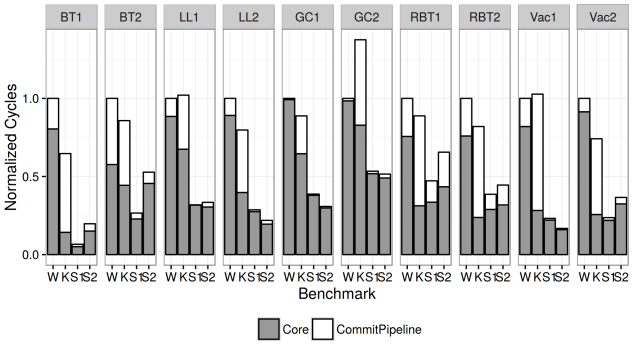

Figure 17: Normalized Duration of Transactions.

seen that KiloTM and WarpTM do not scale well; actually, the execution gets serialized in WarpTM and KiloTM.

Speedup for red-black tree and binary tree may be explained in a way resembling linked lists: a branch of a tree is similar to a linked list, but since there exists many branches in a tree, the serialization in a tree is not nearly as serious as in a linked list. However, there is still enough access patterns like those found in a linked list to allow Snapshot Isolation to perform much faster.

Similarly, in the Graph Cut application, a series of edge contraction operations are performed, each modifying in the structure of the graph. This can be compared to an insertion and a deletion in the linked list.

In fact, one may also think of graphs as generalized trees which can contain loops. Because the same access pattern is observed in all those data structures, dynamically sized lists, trees and graphs can all benefit from Snapshot Isolation.

The overall energy consumption is largely affected by the execution time. Although SITM_TS introduces extra power consumption on the Writer Table, Timestamp Table, Scalable Reader List and Version Index Table, the overall energy consumption is still less than WarpTM due to decreased running time. It can be even significantly lower than KiloTM when the running time difference is large enough, such as in Linked List.

We take a look into the finer details in the following analysis.

7.1 Speedup Analysis

The design of Snapshot Isolation is to avoid unnecessary write-read aborts. To give a quantitative measure of this, we compute the sum of the duration of all aborted transactions in clock cycles for each of the configurations and normalize them that of the WarpTM baseline, as shown in Figure 15. The higher the bars are, the more work is wasted on aborted transactions. Overall, the resultant speedup is generally inversely correlated to the amount of work wasted. KiloTM generally resulted in less wasted work than WarpTM, and both SI-based configurations resulted in even less wasted work than KiloTM. To compare SITM_TS and SITM_Oracle: the performance gap between SITM_TS and SITM_Oracle is correlated to the gap between the amount of wasted work of them. When SITM_TS has less or similar amount of wasted work than SITM_Oracle, its performance may be close to that of SITM_Oracle (in benchmarks BinaryTree 1 and 2, RBTree 1 and 2, Vacation 1). When SITM_TS wastes more work than SITM_Oracle, its performance will fall farther behind SITM_Oracle due to the overhead it has compared to SITM_Oracle.

Figure 16 shows the number of aborts caused in each of the benchmarks, normalized to the number of aborts in WarpTM. From the figure, we can see WarpTM often aborts more than KiloTM does; most of them are caused by intra-warp conflict resolution. For SITM_TS, many aborts come from the single-writer restriction. This type of aborts usually happens in early stages of transactions where the transactions have not performed much work. As a result, the length of the aborted transactions tend to be smaller, resulting in a smaller aggregated time even the number of aborted transactions may be larger.

Figure 17 shows the amount of time transactions spend in the Commit Unit and the SIMT cores. It can be seen from the figure that the amount of time spent in the Commit Unit is shorter in SITM_TS and SITM_Oracle because the Commit Unit does not need to perform value-based validation for the read sets like in WarpTM and KiloTM (Only the write set needs to be validated in Snapshot Isolation.) As a result the average durations of both SI-based configurations are shorter than that of both WarpTM and KiloTM, and this can match the observation of aggregate time for aborted transactions in Figure 15.

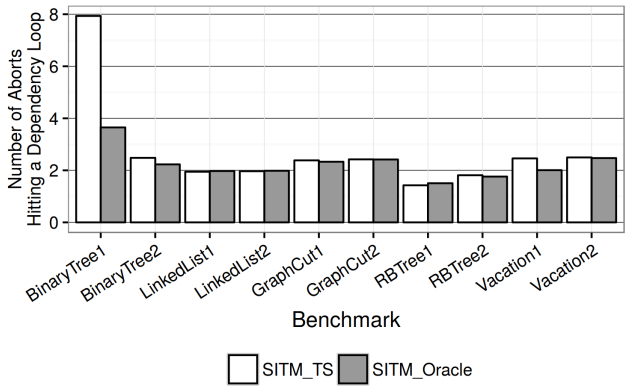

Figure 18: Number of Aborts Seen by Each Dependency Loop.

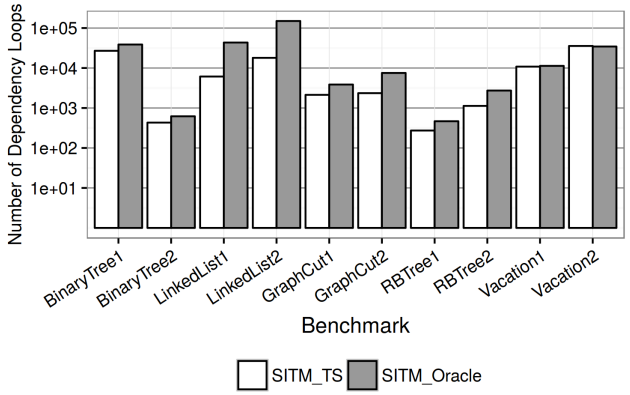

Figure 19: Number of Total Possible Dependency Loops.

It can be seen from Figures 16 and 17 that SITM_Oracle reduces the number of aborts for LinkedList 1 and 2, GraphCut 1 and 2, which is as expected. For the other benchmarks, SITM_Oracle resulted in a shorter transaction execution time, which turns into greater speedup. For SITM_TS, the absolute number of aborts may increase due to single-writer limits. Nevertheless, Figure 17 suggests SITM_TS's aborted transactions tend to be much shorter. Since SITM_TS still allows transactions with read-write conflicts to pass, this will overall results in less wasted work, leading to advantage over non-SI configurations.

7.2 Dependency Loop Detection

Figure 18 shows the number of loop-breaking aborts received per dependency loop. The numbers are the outcome of two factors: 1) how much do the loops themselves overlap and 2) the false-alarm rate. When the loops overlap more, it is more likely that one aborted transaction breaks more than one loop. False alarms may also cause a dependency loop to receive more than one abort. Overall, the number of aborts per dependency loop does not show a great difference between SITM_TS and SITM_Oracle except in BinaryTree1: In the early stages of this benchmark, many transactions attempt to replace

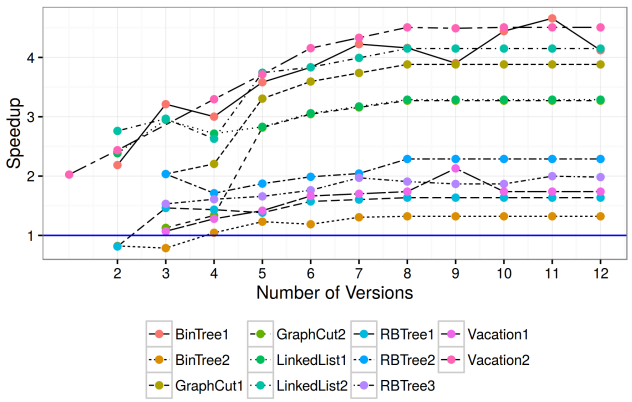

Figure 20: Sensitivity of speedup to number of versions available.

the parent of a tree node with the next largest element in the tree, resulting in many writes to the same addresses. This causes the gap in the number of aborts between to SITM_TS and SITM_Oracle to widen, which contributed to the difference in the overall numbers of aborts per dependency loop. In later stages of the benchmark as well as in other benchmarks, the difference between the number of aborts of SITM_TS and SITM_Oracle is more stable, resulting in similar numbers of aborts per loop.

Figure 19 shows the total number of possible dependency loops that can ever be formed by all transactions throughout the benchmark lifetime. It can be seen that the number for SITM_TS is much smaller than that of SITM_Oracle. The reason is many transactions are aborted due to single-writer abort, such that the transactions could not have the chance to overlap with other transactions to form dependency loops. This also suggests although the purpose of the single-writer restriction is to keep post-dating scalable, it also eliminates much potential dependency loops by coincidence by aborting transactions.

7.3 Sensitivity to Number of Versions and Postdating Delta

We performed a study on how the the number of concurrently available versions affects overall performance, shown in Figure 20. The results suggest that a larger number of concurrently available versions will generally imply better performance. This is because with more versions available, transactions can see more recent snapshots. On the contrary, With fewer versions available, transactions are more likely to see stale snapshots, which can turn into more frequent write-write conflicts. For example, when inserting into a linked list, a more recent snapshot will contain a longer linked list. A longer list is more likely to cause the insertion operations to modify addresses farther away from each other, lowering the chance of conflict.

As is shown in Figure 20, a concurrent version number of 8 results in near-optimal performance, we consider it to be the balance between performance and cost and decide to use this number in our experiments.

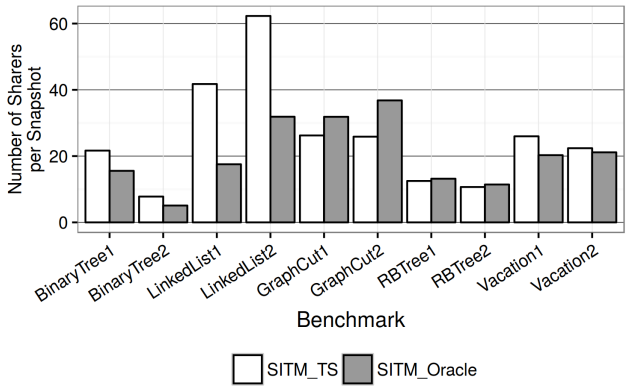

Figure 21: Number of Sharers per Snapshot.

Figure 21 shows there are many transactions that read from the same snapshot, which is a result of the high level parallelism on the GPU. In addition to the high level parallelism, transactions on a GPU are run in warps which execute in lock-step, so that all transactions in a warp start at the same time step, thus sharing the same snapshot. The large number of transactions overlapping in time means many commits of the transactions may be merged together, reducing the number of version creation and row-level copy needed, lowering the amortized cost of such operations. This is an example where a high degree of parallelism doesn't necessarily mean a high cost in versioning.

We have observed the choice of the post-dating delta (which is 10 in Figure 7) does not have a significant influence on the overall speedup.

8 RELATED WORKS

For GPU-based hardware transactional memory, existing works include KiloTM [9] and WarpTM [8] which is a hardware implementation of modified RingSTM [27]. This implementation is shown to benefit from two early conflict resolution schemes proposed by Chen and Peng [4] for certain workloads. For software TMs that can be implemented in CUDA, Holey and Zhai proposed Lightweight Transactions [12] and Xu et al. proposed GPU-STM [30]. Their designs eager and lazy approaches for both conflict detection and commit. One limitation of Software-based TM systems is memory copies are compiled into word level copy loops, so copies must be coalesced to be efficient. Furthermore, copy loops are compiled into load and store instructions, forcing data to make detour to the SIMT cores and back to the memory, consuming interconnect bandwidth and compute cycles. In contrast, hardware-based TM can achieve bulk copy from within the memory system, overcoming those limits and reducing run-time overhead considerably.

Snapshot Isolation was first proposed by Berenson et al. in 1995 [1], and has been implemented in popular databases such as PostgreSQL, Oracle and SAP Hana, implemented using methods such as version chains. The Write Skew Anomaly has been known since the introduction of SI to the database community and Fekete et al. [2] proposed the concept of Dynamic Dependency Graph which served as a theoretical foundation for removing the Anomaly.

The first application of Snapshot Isolation to transactional memory is described in SI-TM [18], with a system for the user to identify application source code lines that cause the Write Skew Anomaly and remove them [19]. This method is trace-driven and runs offline.

On-line detection of dependency loops using the Post-Dating method is used in Wait-n-Go TM [15], which does not integrate Snapshot Isolation and uses it to eliminate cyclic aborts. The method itself comes from TimeTraveler [29] which uses the method for the purpose of saving data race log file size by detecting cyclic memory races since acyclic memory races need not be recorded. Applying the Post-Dating method to GPU depends on the realization of a directory protocol for many-cores, such as the SCD [25] and other in-memory processing techniques such as RowClone[26] and other near-memory processing techniques[13].

9 CONCLUSION

We have applied the Snapshot Isolation mechanism to a GPU-based hardware TM system. With a feasible mechanism for creating versions quickly and detecting dependency loop, this system can work out well for applications involving linked lists, binary search trees, red-black trees and graphs, achieving great speedup over systems using conventional conflict detection mechanism. We believe this can expand the use cases for TM systems on the GPU: It will no longer only handle small and fixed-size transactions, but large ones with dynamically-sized, complex data structures as well.

10 ACKNOWLEDGEMENTS

We appreciate the invaluable comments from the anonymous reviewers. This work is supported in part by US National Science Foundation (NSF) grants CCF-1017961, CCF-1422408 and CNS-1527318. We also acknowledge the computing resources provided by the Louisiana Optical Network Initiative (LONI) HPC team.

REFERENCES

[1] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM International Conference on Management of Data (SIGMOD)*.

[2] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable Isolation for Snapshot Databases. In *ACM Transactions on Database Systems (TODS)*, Vol. 34. 1–42. https://doi.org/10.1145/1620585.1620587

[3] George. C. Caragea, Fuat Keceli, Alexadros Tzannes, and Uzi Vishkin. 2010. General-Purpose vs. GPU: Comparison of Many-Cores on Irregular Workloads. In *Proceedings of the Second Usenix Workshop on Hot Topics in Parallelism*. http://www.usenix.org/event/hotpar10/final

[4] Sui Chen and Lu Peng. 2016. Efficient GPU Hardware Transactional Memory through Early Conflict Resolution. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 274–284. https://doi.org/10.1109/HPCA.2016.7446071

[5] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making Snapshot Isolation Serializable. In *ACM Transactions on Database Systems (TODS)*, Vol. 30. 492–528. https://doi.org/10.1145/1071610.1071615

[6] Wilson W. L. Feng. 2013. GPGPU-Sim 3.2.1. http://www.ece.ubc.ca/~wwlfung/code/kilotm-gpgpu_sim.tgz. (2013). Retrieved on 2015-05-30.

[7] Michael Ferdman, Pejman Lotfi-Kamran, Ken Balet, and Babak Falsafi. 2011. Cuckoo Directory: A Scalable Directory for Many-Core Systems. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2011.5749726

[8] Wilson W. L.. Fung and Tor M. Aamodt. 2013. Energy efficient GPU transactional memory via space-time optimizations. In *Proceedings of the 46th International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/2540708.2540743

[9] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. 2011. Hardware transactional memory for GPU architectures. In *Proceedings of the 44th International Symposium on Microarchitecture(MICRO)*. https://doi.org/10.1145/2155620.2155655

[10] Maurice Herlihy and J. Eliot B. Moss. 1993. *Transactional Memory: Architectural Support For Lock-free Data Structures*. IEEE Computers Society Press. 289–300 pages. https://doi.org/10.1109/ISCA.1993.698569

[11] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2006. A Flexible Framework for Implementing Software Transactional Memory. In *Proceedings of the 21th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*.

[12] Anup Holey and Antonia Zhai. 2014. Lightweight Software Transactions on GPUs. *Proceedings of the 43rd International Conference on Parallel Processing (ICPP)* (Sep 2014). https://doi.org/10.1109/ICPP.2014.55

[13] Kevin Hsieh, Eiman Ebrahim, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W. Keckler. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 204–216. https://doi.org/10.1109/ISCA.2016.27

[14] Intel Corporation. 2016. Chapter 8, Intel Transactional Synchronization Extensions. (2016).

[15] Syed Ali Raza Jafri, Gwendolyn Voskuilen, and T. N. Vijaykumar. 2013. Wait-n-GoTM: Improving HTM Performance by Serializing Cyclic Dependencies. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[16] David R. Karger. 1993. Global Min-cuts in RNC, and Other Ramifications of a Simple Min-out Algorithm. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 21–30. http://dl.acm.org/citation.cfm?id=313559.313605

[17] HP Labs. 2009. CACTI 5.3. http://quid.hpl.hp.com:9081/cacti/. (2009). Retrieved on 2016-07-01.

[18] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P. Stevenson. 2014. SI-TM: Reducing Transactional Memory Abort Rates Through Snapshot Isolation. In *Proceedings of the 19th international conference on Architectural Support for programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/2541940.2541952

[19] Heiner Litz, Richardo J. Dias, and David R. Cheriton. 2014. Efficient Correction of Anomalies in Snapshot Isolation Transactions. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 4 (2014), 1–24. https://doi.org/10.1145/2693260

[20] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. 2006. Lowering the Overhead of Nonblocking Software Transactional Memory. In *Tech Report, Dept. of Computer Science, Univ. of Rochester*.

[21] Chí Cao. Minh, JaeWoong. Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. https://doi.org/10.1109/IISWC.2008.4636089

[22] Prabhakar Misra and Mainak Chaudhuri. 2012. Performance Evaluation of Concurrent Lock-free Data Structures on GPUs. *18th International Conference on Parallel and Distributed Systems (ICPADS)* (Dec 2012). https://doi.org/10.1109/ICPADS.2012.18

[23] Anurag Negi, Per Stenstrom, Manuel E. Acacio, Rubén Titos-Gil, and José M. Garcia. 2011. π-TM: Pessimistic invalidation for scalable lazy hardware transactional memory. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. https://doi.org/10.1109/PACT.2011.41

[24] Daniel Sanchez and Christos Kozyrakis. 2010. The ZCache: Decoupling Ways and Associativity. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2010.20

[25] Daniel Sanchez and Christos Kozyrakis. 2012. SCD: A scalable coherence directory with flexible sharer set encoding. In *Proceedings of the 18th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2012.6168950

[26] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2013. RowClone: Fast and Energy-Efficient in-DRAM Bulk Data Copy and Initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 186–197. https://doi.org/10.1145/2540708.2540725

[27] Michael F. Spear, Maged M. Michael, and Christoph von Praun. 2008. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, New York, NY, USA, 275–284. https://doi.org/10.1145/1378533.1378583

[28] Saša Tomić, Cristian Perfumo, Chinmay Kulkarni, Adrià Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. 2009. EazyHTM: EAger-LaZY hardware Transactional Memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 145–155. https://doi.org/10.1145/1669112.1669132

[29] Gwendolyn Voskuilen, Faraz Ahmad, and T. N. Vijaykumar. 2010. Timetraveler: Exploiting Acyclic Races for Optimizing Memory Race Recording. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/1815961.1815986

[30] Yunlong Xu, Rui Wang, Nilanjan Goswami, Tao Li, Lan Gao, and Depei Qian. 2014. Software Transactional Memory for GPU Architectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 49 – 52. https://doi.org/10.1145/2544137.2544139

[31] Lihang Zhao and Jeffrey Draper. 2014. Consolidated Conflict Detection for Hardware Transactional Memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)*. 201–212. https://doi.org/10.1145/2628071.2628076