Efficient GPU Hardware Transactional Memory through Early Conflict Resolution

Sui Chen and Lu Peng
Division of Electrical & Computer Engineering, Louisiana State University
{csui1,lpeng}@lsu.edu

ABSTRACT

It has been proposed that Transactional Memory be added to Graphics Processing Units (GPUs) in recent years. One proposed hardware design, Warp TM, can scale to 1000s of concurrent transactions. As a programming method that can atomicize an arbitrary number of memory access locations and greatly reduce the efforts to program parallel applications, transactional memory handles the complexity of inter-thread synchronization. However, when thousands of transactions run concurrently on a GPU, conflicts and resource contentions arise, causing performance loss.

In this paper, we identify and analyze the cause of conflicts and contentions and propose two enhancements that try to resolve conflicts early: (1) Early-Abort global conflict resolution that allows conflicts to be detected before they reach the Commit Units so that contention in the Commit Units is reduced and (2) Pause-and-Go execution scheme that reduces the chance of conflict and the performance penalty of re-executing long transactions. These two enhancements are enabled by a single hardware modification. Our evaluation shows the combination of the two enhancements greatly improves overall execution speed while reducing energy consumption.

1. INTRODUCTION

GPUs are designed for throughput-oriented computing using large numbers of light-weight parallel threads [12]. For this purpose, they are constructed with compute units that can house large numbers of resident threads and a deeply pipelined memory subsystem that can handle a large number of parallel memory accesses. The GPU is becoming more versatile with its feature set enriching ever since compute acceleration programming models such as CUDA [1] and OpenCL [3] were introduced: features like atomic operations and support for recursion have enabled the construction of programs involving more complicated inter-thread cooperation. The collection of lock-free data structures [16] and programming libraries keeps expanding as well.

Correct and efficient implementation of synchronization methods in parallel programs that can scale up well is not a trivial task. While accessing a few hash tables may require only a single fine-grained word-based lock, writing a concurrent red-black tree [13] is much more difficult. As such, transactional memory (TM) [8] has gained attention as a way to confront the challenges in the development of parallel programs, because it allows all read/write operations in a transaction to complete atomically as a whole, relieving the programmer of having to tackle fine-grained locks for performance and correctness. While most research efforts have focused on TM support on multi-core processors [17] and hardware TM support has begun entering commodity CPUs [4], both software and hardware transactional memory systems have been proposed for GPUs as well. One of such proposal is named “Kilo TM” [6], followed by its successor “Warp TM” [7].

Like ordinary GPU programs, GPU hardware transactional memory also faces the challenge of resource contention. As the high amount of concurrency on GPUs puts great pressure on the memory subsystem, a programmer needs to spend great amounts of effort in optimizing the memory access pattern of the program in question, otherwise the program would not scale well and even increasing the number of concurrent threads would harm performance [20]. The same phenomenon is also observed in Kilo TM and Warp TM, where having too many concurrent transactions may increase conflicts and resource contention, resulting in decreased performance.

In this paper, we analyzed the performance penalty resulting from conflict and contention with benchmarks running on Warp TM. Our analysis leads to two enhancements that can lead to performance improvement on top of Warp TM:

- **Early-Abort global conflict resolution:** Conflicting addresses are made accessible from the SIMT cores so that resolution of certain types of conflict can be done on the cores, reducing contention at the Commit Units and the interconnection network, and
- **Pause-and-Go execution scheme:** Running transactions are stalled when a conflict is likely to happen, which protects the work the transaction has done so far from being wasted.

According to our experimental results, the two approaches result in an overall speedup of up to 1.41x compared to Warp TM at an average power consumption of 0.8x. Further, the enhancements used in this paper may be applied on top of various TM implementations because its correctness is guaranteed by the underlying TM implementation.

We make the following contributions in this paper:

1. We analyze the performance overhead of conflicts in transactional memory on GPUs.

978–1–4673–9211–2/16/$31.00 © 2016 IEEE

2. We study GPU TM programs with short-running and long-running transactions.

3. We propose a simple hardware modification for reducing conflict and contention.

4. We propose two enhancements which may be used on GPU TM systems with various underlying implementations.

2. BACKGROUND

2.1 Transactional Memory

Transactional memory (TM) [8] is a technology that allows the programmer to mark code regions as "transactions" that satisfy serializability and atomicity. It may be viewed as a generalized version of the atomic compare-and-swap instruction, which can operate on an arbitrary set of data instead of just one machine word.

The lifetime of a transaction consists of four major states: 1) executing, where it performs speculative read/write operations that constitute a transaction, 2) committing, where it is being checked against other committing transactions to see if data hazards exist, 3) aborted, where a transaction fails conflict detection and is aborted, with its speculative execution squashed, and 4) committed, where a transaction passes conflict detection and its speculative results written to the memory.

Transactional memory systems fall into distinct subregions of the design space [5]. Design choice can be made on when conflict detection is performed. An eager TM system performs conflict detection during execution while a lazy one does it in the commit stage. Another choice can be made on where to store versioning meta-data. The meta-data may reside on the memory side in the form of ownership records, which are mapped to parts of the memory, such as locks associated with machine words [24] or objects [9] [10]. The meta-data may also reside on the thread side in the form of read/write logs, a buffer that keeps the speculative read/write values in a transaction.

Both hardware-based and software-based transactional memory systems have been proposed for GPU systems. Both have to take into account the characteristics of the GPU in order to be efficient. We will be discussing and improving on a hardware-based TM system, Warp TM[7] (which is built on Kilo TM [6]) in this paper.

2.2 Kilo TM

Kilo TM [6] is a hardware-based GPU transactional memory system, which features value-based validation and lazy version management, thus allowing the programmer to write weakly-isolated transactions in GPU kernel code. It is based on an algorithm similar to RingSTM [18].

In the base algorithm, the life time of transactions can be divided into execution and commit. During execution, the transactional loads and stores performed are buffered in read- and write-logs. During commit, the logs are transferred to the Commit Units in the memory partitions for conflict detection and resolution. Conflicts between committing transactions are detected using an algorithm similar to RingSTM [18], with transactions claiming entries in a ring buffer and committing in a global order. To speed up committing, the validation step is performed in two stages. The first stage involves probabilistically detecting overlap between the signatures of read- and write-sets encoded with bloom filters, which is also called *hazard detection*. The second stage is *value-based validation* and it is needed only when hazard exists and is responsible for resolving false hazards.

The Kilo TM design introduces new hardware units and new hardware features to enable the operation of transactional memory. The execution stage of transactions takes place in the SIMT cores. The load/store units are responsible for maintaining the read- and write-logs for transactions that are being executed. The SIMT stack is extended to handle transactional code blocks. The commit stage takes place in the Commit Units, which implements the ring buffers, bloom filter-based signatures and valid-based validation. The Commit Units are located in the memory partitions. The entire address space is divided into disjoint sets, each of which is managed by one Commit Unit. As such, the implementation is a distributed algorithm which involves multiple Commit Units, and a protocol that aggregates and broadcasts validation results between the Commit Units and SIMT cores. At the beginning of a commit, the transactional log walkers transfer the logs to the Commit Units. When validation results are computed, they are sent back to the originating SIMT cores.

2.3 Warp TM

Derived from Kilo TM, Warp TM [7] introduces the important concept of *warp-level transaction management*, based on the fact that threads in GPU are executed in lock-step, referred by NVidia as "warps". This is reflected in various aspects of the design of the GPU hardware. As a result it is advantageous to handle the threads in a warp as a single entity.

Based on this, Warp TM implemented two optimizations. First, scalar protocol messages in Kilo TM are superseded by coalesced protocol messages in Warp TM. Second, conflicts within a warp are resolved prior to the transfer of read/write sets for global commit and validation. This can reduce contention and delay at the Commit Units, improve performance and reduce energy consumption.

3. CONFLICT AND CONTENTION REDUCTION

In this section, we list the conflict types in a way that is relevant to the GPU architecture in question. Because of the GPU architecture, certain types of conflict can be resolved at the SIMT cores. The others have to travel through the Commit Units, as illustrated in Figure 1. So, more conflicts resolved on the core side means fewer transactions need to reach the Commit Unit for conflict resolution. Here we give types of conflicts and indicate the opportunities.

3.1 Spatial and Temporal Types of Conflict

Spatial: A conflict between two simultaneously running transactions in the GPU transactional memory may span across different levels of thread hierarchies, as illustrated in Figure 1. Depending on the location of the transactions in a conflict

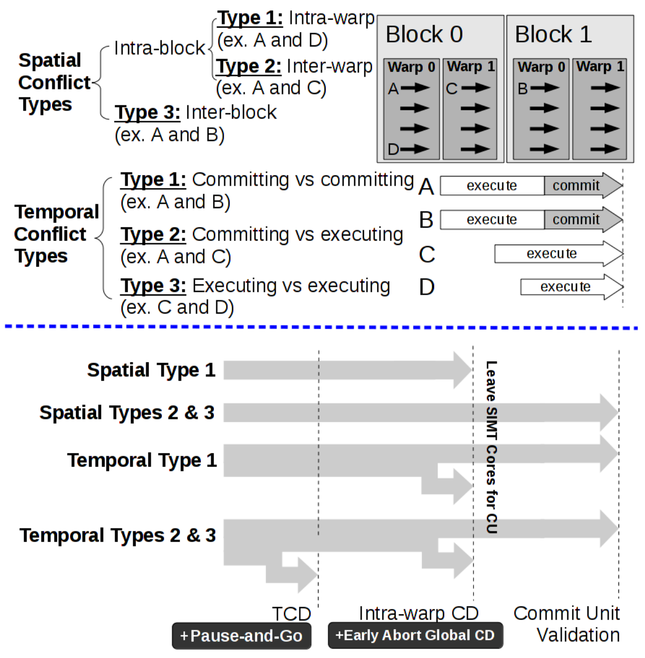


Figure 1: Spatial and Temporal Types of Conflict in a GPU TM and the resolution flow of conflicts (end of arrow = resolved). Pause-and-Go and Early-Abort Global Conflict Detection are added in this paper to resolve more conflicts before they reach for the Commit Unit.

pair there exist three non-overlapping spatial types: Type 1 (intra-warp), Type 2 (inter-warp, but in the same block), and Type 3 (inter-block).

Since the shared memory is the "most recent common ancestor" in the memory hierarchy accessible to threads in the same block, Type 1 conflicts can be resolved within the SIMT core using the shared memory, as is described as *Intra-warp Conflict Resolution* in Warp TM. Type 2 conflicts, however, are not handled by warp-level conflict detection due to the overhead of increased complexity and the rarity of Type 2 conflicts. Type 3 conflicts involve global read/write sets so they cannot be resolved in the core.

The detection of both Type 2 and 3 conflicts needs to be done in one level higher in the memory hierarchy which is the off-chip DRAM partitions, the "most recent common ancestor" in the memory hierarchy accessible to threads in different blocks.

One transaction may be involved in more than one type of conflict with other transactions, but all transactions cannot commit if one conflicts with any committing transaction at all. Performance improvement can come from resolving Types 2 and 3 conflicts at the SIMT core in addition to Type 1.

Temporal: For transactions not executing in lock-step, they must overlap both temporally and spatially to conflict with each other. Recall that the two steps in the life time of a transaction are execution and commit. Depending on the step the transactions in a pair are in, there exist three non-overlapping temporal types: Type 1 (committing and committing), Type 2 (committing and executing) and Type 3 (executing and executing). All 3 types of conflicts must be resolved at the Commit Unit with one exception that read-only

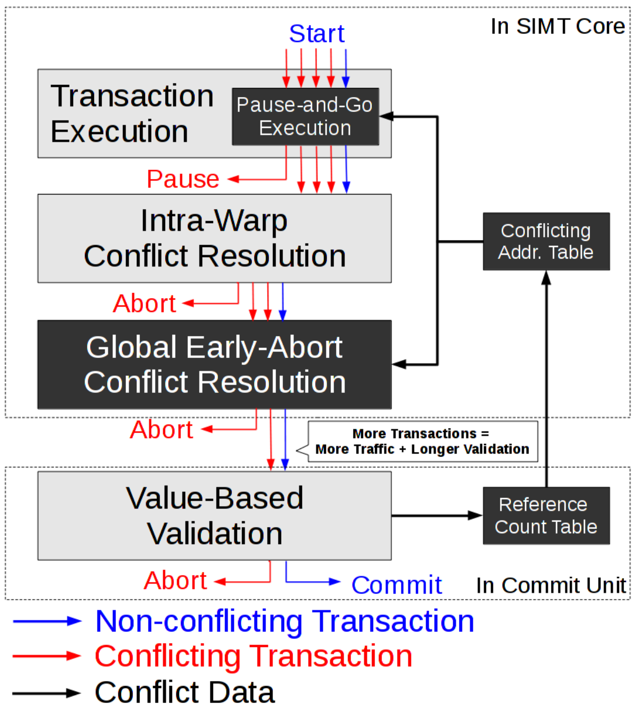


Figure 2: Transaction Flowchart

transactions may self-abort when it fails *Temporal Conflict Detection* [6] because only the Commit Unit has the information needed for conflict resolution.

To summarize, the more conflicts resolved before a running transaction reaches the Commit Unit, the less contention there will be in the entire system. In this paper we propose two new approaches that resolve conflicts of Spatial Type 3 (inter-block) and of Temporal Type 2. The approaches are enabled by modifying existing hardware that makes conflict data available in the SIMT cores illustrated in Figure 2. The two approaches are:

- **Early-Abort global conflict resolution** resolves conflicts between transactions about to commit and the ones already committing. Similar to intra-warp conflict detection, it reduces the number of transaction reaching the Commit Unit and saves validation cost.
- **Pause-and-Go execution scheme** resolves conflicts between executing transactions and committing transactions by temporarily stalling transactions that are about to execute load/store instructions that may result in a conflict. It reduces the incidence of conflicts.

The result is fewer conflicts, less resource pressure on the Commit Unit, and better overall performance. Both approaches are enabled by the hardware modification described in Section 4.

3.2 Concerns over Correctness

It is of utmost importance for a transactional memory system to guarantee the correct execution of transactions. In this paper, correctness is guaranteed by the last step in the transaction execution flow, value-based validation. Early-Abort can only abort transactions selectively and never commits

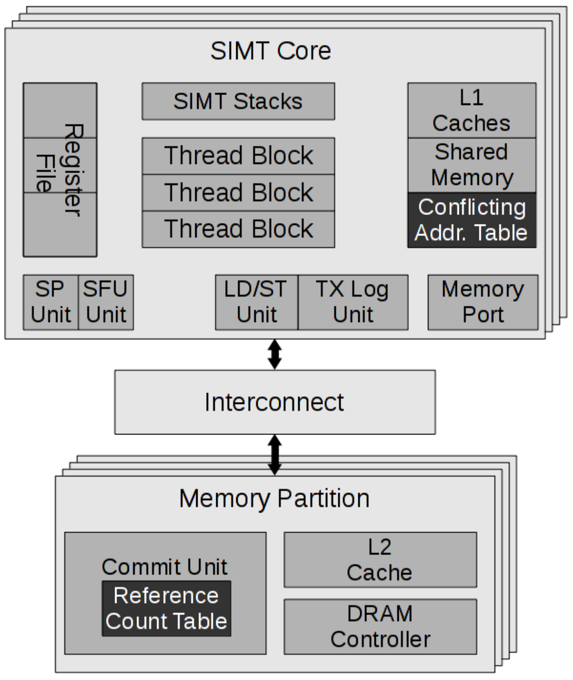


Figure 3: The TM-enabled GPU architecture. The Conflict Address Table and the Reference Count Table are added in this paper.

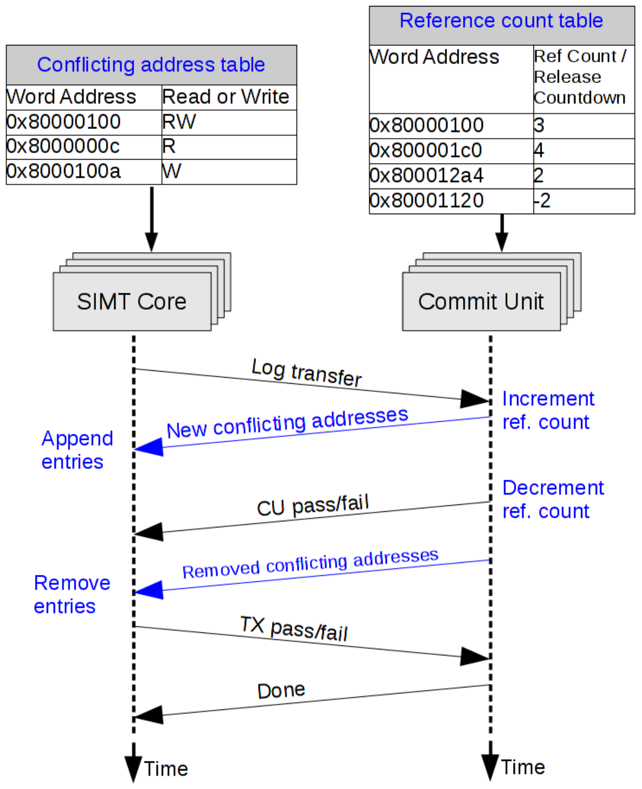
the write sets to the memory; Pause-and-Go does not abort any transaction so it does not cause inconsistency. More specifically, what may happen when a false positive or a false negative occurs is:

- If a non-conflicting transaction is aborted, its work up to the commit point is wasted and it will restart from the beginning of the transaction.
- If a conflicting transaction is not identified in Pause-and-Go or Early-Abort, it will eventually be aborted at the Commit Unit. This is the same expected behavior as in Warp TM.

Another potential problem that TM systems may encounter is livelocks, which are caused by transactions repeatedly aborting another. Similar symptoms are named by Bobba [5] as *Friendly Fire* and *Dueling Upgrades*. Pause-and-Go and Early-Abort do not introduce such "pathologies" because only committing transactions are allowed to abort executing transactions but not the other way around, resulting in an implicit conflict management policy that gives priority to committing transactions. Therefore, a pair of transactions could not repeatedly abort each other.

4. HARDWARE MODIFICATIONS TO WARP TM

The modified hardware is depicted in Figure 3. It consists of two address-indexed lookup tables. The per-SIMT-core Conflict Address Table (CAT) maps addresses to two bits indicating whether a word is written or read by committing transactions. The per-Commit-Unit Reference Count Table



Word Address	Read or Write
0x80000100	RW
0x8000000c	R
0x8000100a	W

Word Address	Ref Count / Release Countdown
0x80000100	3
0x800001c0	4
0x800012a4	2
0x80001120	-2

Figure 4: The Conflicting Address Table (CAT) in SIMT cores and the Reference Count Table (RCT) in Commit Units, and the operations related to the two tables on the Warp TM protocol message time line.

(RCT) maps addresses to the number of readers and writers. The Commit Unit maintains the Reference Count Table as it processes committing transactions. The Conflicting Address Tables on the cores are updated through the interconnection network.

When a transaction is executing, the SIMT core tries to resolve conflicts early utilizing the Conflicting Address Table.

4.1 Maintaining the Reference Count Table and the Conflict Address Table

When a transaction starts committing, it will potentially be involved in conflict with all other committing and running transactions over the addresses in its read and write sets. To detect conflict between a committing transaction and a running transaction, the read/write logs of the committing transaction have to be made visible to the SIMT cores. The two tables work in conjunction to achieve this goal.

The Reference Count Table in a Commit Unit keeps the number of readers and writers. It is updated when committing transactions are processed.

When log transfer is completed, read/write addresses reach the Commit Units. The read addresses are being appended to the ring buffer as well as the validation queue and are prepared for value-based validation. The write addresses are inserted into the Last Writer History Table, which is also an address-indexed lookup table. At these two points, the reader/writer count of an address is incremented for each address appearing in the read/write set.

As the Commit Unit receives validation reply from the L2 cache, it checks the value of each of the words in the read set. If all values in the read set match the values in the memory, the transaction passes value-based validation. If any address does not match, the transaction fails validation. Whether the outcome of the transaction is, the addresses will not be used by the transaction anymore. So, when the outcome is known, the Commit Unit traverses the read/write logs and decrements the reference counts.

The core-side Conflict Address Tables reflect which addresses are being referenced in the Commit Unit. Every time addresses are inserted into or removed from the per-Commit-Unit Reference Count Table, a message is sent from the Commit Unit to notify all the SIMT cores of the change in the address set. The message consists of a series of entries, each of which contain the word addresses, a read/write bit and an add/remove bit. Since the words are aligned to a multiple of 4 bytes, the two least significant bits in the addresses are always zero. Thus, each entry can be made exactly 4 bytes long. When the SIMT cores receive the packets, they update the Conflict Address Tables accordingly. Figure 4 gives a visual description of the two tables and how they are updated.

4.2 Table Size Limit

When the Reference Count Table is full, new entries will be simply ignored. The Conflict Address Tables will not be updated in this case, either.

Opportunity for reducing contention may be missed when the tables are full, but the correctness of transactions will not be affected since the correctness is guaranteed by the TM implementation in the value-based validation stage.

4.3 Early-Abort Utilizing the Conflicting Address Table

Early-Abort (EA) tries to detect Spatial Type 3 (inter-block conflict) conflicts at the intra-warp conflict resolution stage and abort the conflicting transactions. By avoiding sending the read/write sets of the conflicting transactions to the Commit Unit, Early-Abort reduces the resource contention in the Commit Unit.

This step requires matching the read/write sets of the threads in a warp with the addresses in the CAT. The match is implemented by performing a table look-up during the log scanning process. Assuming a 4-port L1 cache, the lookup takes up to 8 cycles for a warp with 32 active threads. The match overlaps with the mark-and-check *Intra-Warp Conflict Resolution* step. When a transaction reads/writes an address in the CAT, it is considered likely to be in conflict with another committing transaction and is aborted (except for read-read transactions.) After this step, the transactions in a warp are free of Spatial Type 1 (intra-warp) conflicts with reduced Spatial Type 3 (inter-block) conflicts.

4.4 Pause-and-Go Execution

Certain transactional applications contain very long transactions or large read/write sets and may encounter conflict over only a small fraction of the read/write sets, resulting

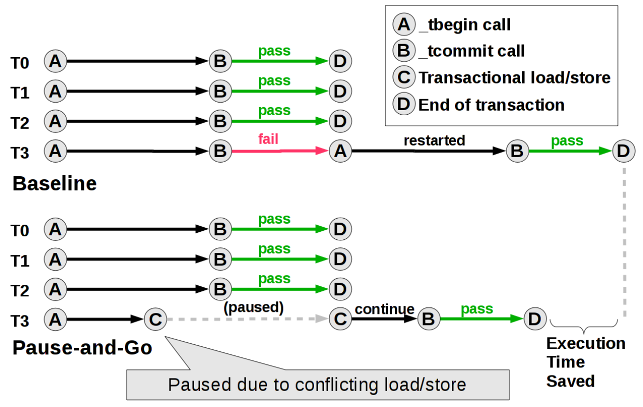


Figure 5: Pause-and-Go execution scheme showing two warps of 4 transactions committing, one of which fails validation.

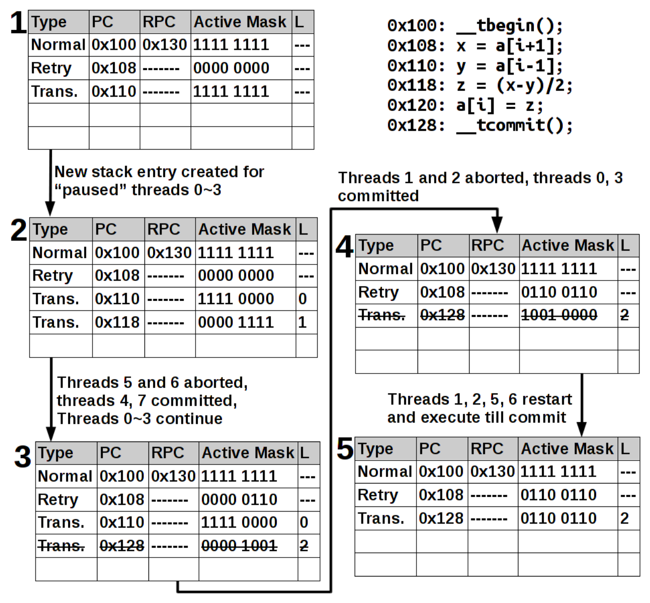
in most of the work being wasted. Pause-and-Go execution takes a more "gentle" approach to such conflicting transactions, trying to avoid aborting the entire transaction while resolving the conflicts. This approach is similar to Staggered Transactions [23] proposed for CPU-based transactional memory systems, where critical regions are serialized to avoid completely aborting transactions.

At each load/store instruction, the read/write addresses for a warp are checked against the Conflict Address Table, which may take up to 8 cycles with a 4-port L1 cache. A thread that is likely to conflict is temporarily "paused". The remaining threads will continue executing and attempt to commit. When the attempt is completed, the control flow will return to the paused threads.

By pausing a potentially conflicting transaction before the commit stage, the "paused" thread can simply avoid the conflict without having to restart execution. Restarting a transaction is costly especially for long transactions. Figure 5 illustrates one case where Pause-and-Go avoids wasting work done by an otherwise aborted transaction.

In order to correctly recover the transactional logs upon the resume of paused transactions, the Log Index (L column) is added to the SIMT stack. The Pause-and-Go execution scheme revolves around the SIMT stack and the transactional logs.

SIMT Stack: Pause-and-Go execution introduces new SIMT stack states. Figure 6 describes how they are maintained. In **1**, transactions start, with one `Retry` entry and one `Trans` entry pushed onto the SIMT stack. The threads execute the instruction at 0x110, which performs a transactional load. The addresses loaded by threads 0-3 exist in the Conflicting Address Table, so they are "paused" and will not continue executing with threads 4-7 for this commit attempt. This results in the SIMT stack configuration in **2**, where two `Trans` entries exist on the top of the stack. The top of stack which represents threads 4-7 attempt to commit, but only threads 4 and 7 passed validation. Bits 5 and 6 are set on the `Retry` entry. The top of stack `Trans` entry is popped from the stack, and the `Trans` entry representing the previously paused threads 0-3 becomes the top of stack. Threads 0-3 execute and attempt to commit, but only threads


1
New stack entry created for "paused" threads 0~3
2
Threads 5 and 6 aborted, threads 4, 7 committed, Threads 0~3 continue
3
Threads 1 and 2 aborted, threads 0, 3 committed
4
Threads 1, 2, 5, 6 restart and execute till commit
5

```
0x100: __tbegin();
0x108: x = a[i+1];
0x110: y = a[i-1];
0x118: z = (x-y)/2;
0x120: a[i] = z;
0x128: __tcommit();
```

Type	PC	RPC	Active Mask	L
Normal	0x100	0x130	1111 1111	---
Retry	0x108	-------	0000 0000	---
Trans.	0x110	-------	1111 1111	---

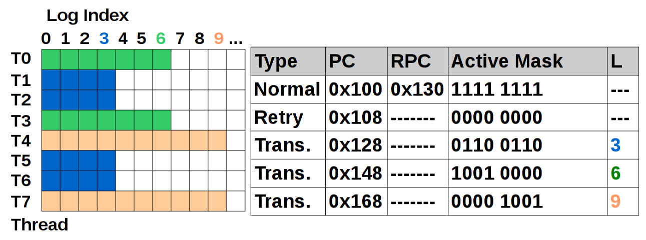
Type	PC	RPC	Active Mask	L
Normal	0x100	0x130	1111 1111	---
Retry	0x108	-------	0000 0000	---
Trans.	0x110	-------	1111 0000	0
Trans.	0x118	-------	0000 1111	1

Type	PC	RPC	Active Mask	L
Normal	0x100	0x130	1111 1111	---
Retry	0x108	-------	0000 0110	---
Trans.	0x110	-------	1111 0000	0
~~Trans.~~	~~0x128~~	-------	~~0000 1001~~	~~2~~

Type	PC	RPC	Active Mask	L
Normal	0x100	0x130	1111 1111	---
Retry	0x108	-------	0110 0110	---
~~Trans.~~	~~0x128~~	-------	~~1001 0000~~	~~2~~

Type	PC	RPC	Active Mask	L
Normal	0x100	0x130	1111 1111	---
Retry	0x108	-------	0110 0110	---
Trans.	0x128	-------	0110 0110	2

Figure 6: SIMT Stack handling in "Pause-and-Go" execution scheme.



Type	PC	RPC	Active Mask	L
Normal	0x100	0x130	1111 1111	---
Retry	0x108	-------	0000 0000	---
Trans.	0x128	-------	0110 0110	3
Trans.	0x148	-------	1001 0000	6
Trans.	0x168	-------	0000 1001	9

Figure 7: Transactional logs (left) and SIMT stack (right) after three rounds of "pause."

0 and 3 passed validation. Bits 1 and 2 are set on the `Retry` entry and results in the stack configuration in **4**. After the `Trans` entry is popped from the stack, the `Retry` entry becomes the top. After that, execution flow is identical to what is expected in the original Kilo TM and Warp TM, where the `Retry` entry is copied to create a new `Trans` entry.

Branch divergence inside paused threads is also handled in exactly the same way as in Kilo TM and Warp TM.

Transactional logs: When a warp is being executed, the Log Index in its corresponding SIMT stack entry is kept in sync with the log pointer of its transactional log walker, as illustrated in Figure 7. To back up the transactional logs for the paused transactions, the log pointer is stored in the Log Index field for the newly created `Trans` SIMT stack entry. To resume the transactional logs, the log pointer is simply reset to the value in the Log Index field in the SIMT stack when the paused transactions resume.

The reason backup and restore can be done with just modifying the log pointer is because a log entry represents all the transactions in a warp, including inactive ones, for which the entries contain zeroes. The logs are then interleaved into a linear buffer. As a result, the log entries of paused transactions will not get overwritten by other active transactions, and restoring the log state only includes changing the log pointer.

Table 1: GPGPU-Sim Configuration

GPU	
SIMT Cores	15
SIMD Width	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
L2 Cache for all Cores	128KB × 6 partition = 768KB
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. L2 Latency	330 Compute cycles
Warp TM	
Commit Unit Clock	700 MHz
Validation/Commit BW	1 Word per cycle per CU
Concurrency Control	2 Warps per core (960 concurrent Txns)
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
Contention Reduction	
Conflict Address Table per Core	12KB (3072 entries)
Reference Count Table per CU	15KB (3072 entries)

5. INTEGRATING WITH WARP TM

As Figure 2 shows, Early-Abort global conflict resolution (EA) and Pause-and-Go (PG) execution scheme are added to the transactional execution flow. Due to the fact that the Reference Count Tables are only updated when a transaction starts committing and the existence of delay in passing message from the Commit Units to the SIMT cores, EA and PG may cause false positives in aborting or pausing transactions. As is discussed in Section 3.2, the correctness of transactions will be guaranteed by the TM and therefore not affected. Together they constitute a hierarchical validation scheme similar to Warp TM and the GPU STM[24].

The two approaches proposed in this paper can achieve performance improvement regardless of the number of concurrent transactions per SIMT core.

The Reference Count Tables in the Commit Units may be constructed via minor modifications on the Last Write History Table. The Conflicting Address Tables in the SIMT cores may be constructed with the same hardware as the L1 cache and the Shared Memory. For a fair comparison, we give the baseline GPU of our comparison extra L1 cache with the size of the Conflict Address Table (extra 12 KB), as described in Table 1.

6. EXPERIMENTAL SETUP

Table 2: Benchmark Properties
(Transaction length and read/write set size measured under baseline settings)

Name	Threads	Read/Write Set Size	Average Tx Length (Cycles)
HashTable 1K entries (HT1K)	23040	2 / 4	8835
HashTable 512 entries (HT512)	23040	2 / 4	10135
ATM 25K accounts (ATM50K)	23040	3 / 2	1423
ATM 10K accounts (ATM25K)	23040	3 / 2	1803
Sparse Mat-Vec Mult. (SpMV)	13000	5 / 1	2221
Linked List (List)	23040	1 / 4	460
Binary Tree (BinTree)	1000	78 / 2	13320
Red-Black Tree small (RBT180)	180	33 / 17	16604
Red-Black Tree large (RBT450)	450	35 / 17	29455

We extend the Warp TM hardware platform using GPGPU-Sim 3.2.1 [2]. It simulates a device similar to NVidia GTX 480 (Fermi). Table 1 summarizes the key architectural parameters. The following benchmarks are used in our evaluation:

Hash Table is a benchmark used in Kilo TM, where each thread inserts into a hash table, each being a linked list. In this paper we use table sizes 1024 and 512 to create higher contention workloads.

Bank Account (ATM) is a benchmark used in Kilo TM, where each thread performs bank transactions between two out of a fixed number of accounts. In this paper we consider 25K and 10K accounts.

SpMV is a program that multiplies a vector and a sparse matrix represented in the Yale format. Transactions are used to update the destination vector.

List is based on the `ListRel` benchmark from DSTM2 [10]. The task of each thread is to insert a node into a linked list. Each thread first finds the insertion point in non-transactional code, then performs the insertion with a transaction.

BinTree is a generic binary search tree. The tree is programmed in a way similar to List, where each transaction first finds the insertion point in non-transactional mode and then performs the insertion with a transaction.

RBTree is a port of the red-black tree implementation in the RSTM [15] test suite. Each thread performs an insertion into the red-black tree with a transaction.

Table 2 summarizes the benchmarks. The benchmarks differ in transaction length, read/write set size, contention rate and working set size. This allows us to evaluate the early conflict resolution approaches in a wide range of situations.

We used GPUWattch [14] to estimate the average dynamic power consumed by each benchmark with the two proposed approaches added to Warp TM. This includes the lookup and maintenance of the Reference Count Tables and the Conflicting Address Tables, as well as the extra interconnection traffic required to update the Conflicting Address Table. We then multiply the average power by the execution time to obtain the total energy needed to execute each benchmark.

7. EXPERIMENTAL RESULTS

In this section, we analyze the performance improvements resulting from the proposed enhancements, Early-Abort (EA) and Pause-and-Go (PG) and compare the results with the baseline.

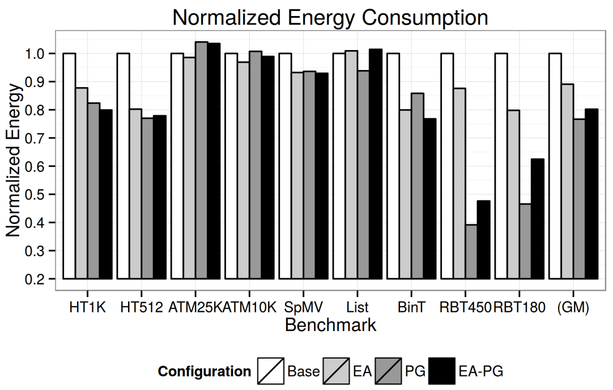


Figure 9: Normalized Energy Consumption. The lower the better.

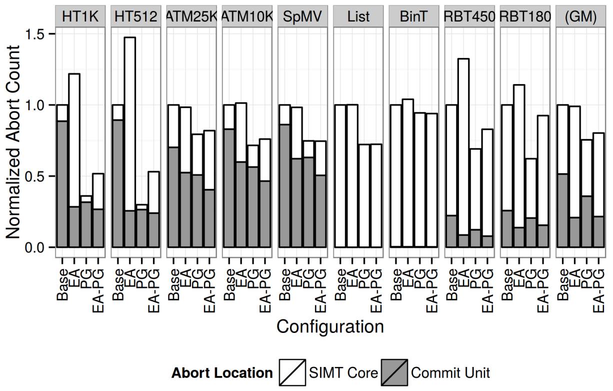


Figure 10: Overall Number of Aborted Transactions.

7.1 Performance and Power Consumption

Figure 8 and 9 shows the overall running time breakdown of the benchmarks and the overall energy consumption. Overall, the proposed approaches yielded a speedup of 1.41x (an average of 0.71x running time) compared to baseline Warp TM. On average, enabling both approaches yielded higher performance improvement than using either approach alone. The average energy consumption has also decreased due to the decrease in execution time. On average the energy consumption is 0.8x compared to the baseline Warp TM.

A closer look at the benchmarks suggest that the Early-Abort (EA) yields greater performance improvement than Pause-and-Go (PG) for Hashtable (HT1K and HT512), Bank Account (ATM25K and ATM10K) and SpMV; Pause-and-Go performs better on Binary Tree (BinT) and Red-Black Tree (RBT450 and RBT180).

The running time result shows that Hashtable (HT1K and HT512), Bank Account (ATM25K and ATM10K) and SpMV spend a significant amount of time in the Commit Unit and they perform better with Early-Abort. Since Early-Abort

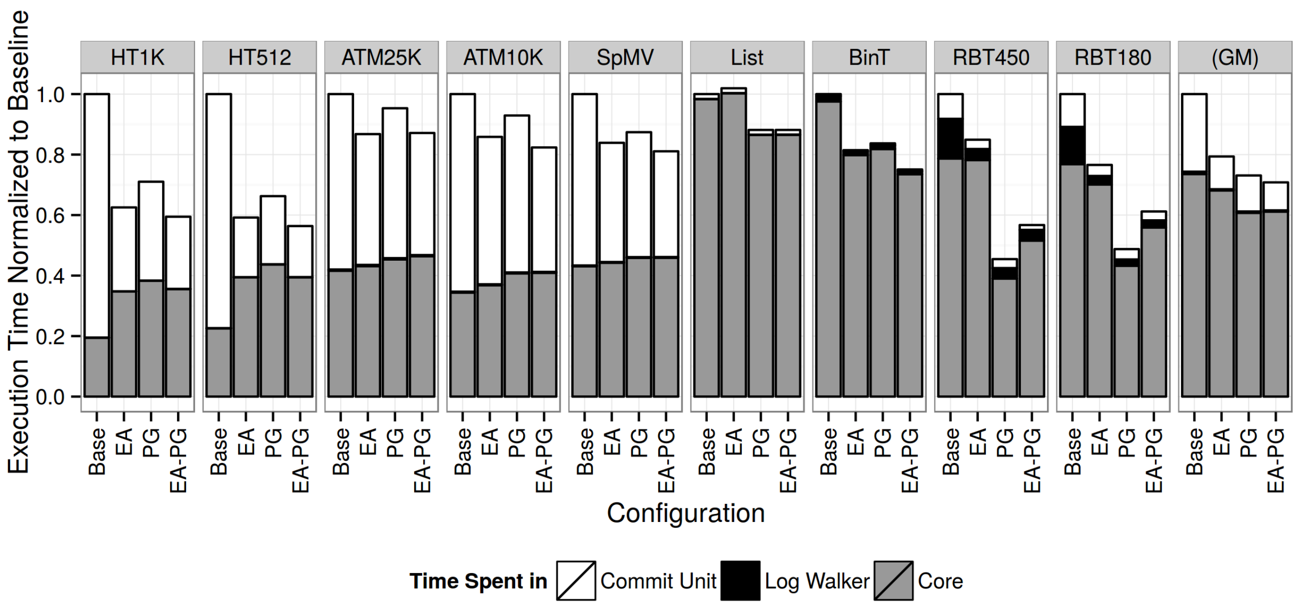


Figure 8: Overall Running Time and Per-Transaction Breakdown, where GM stands for Geometric Mean of all benchmarks. The lower the bar, the faster.

reduces the number of conflicting transactions entering the Commit Unit, transactions in these benchmarks can commit faster.

On the contrary, some of the benchmarks benefit more from Pause-and-Go and their running times are mainly spent in the SIMT core.

The aforementioned results are caused by multiple reasons including the average transaction length, types of conflict between the transactions, running time breakdown and the degree of branch divergence in each of the benchmarks.

7.2 Performance Improvement from Reduced Commit Unit Contention

For Hashtable (HT1K and HT512), Bank Transfer (ATM25K and ATM10K) and SpMV, we can see a correlation between the reduced number of transactions aborted at the Commit Unit (Figure 10) and the improved overall performance (Figure 8).

The reason behind the correlation is two-fold:

- First, aborting conflict transactions at the SIMT core side prevents resource contention at the Commit Unit, which makes committing faster;
- Second, due to the shortened delay, the SIMT core can complete committing the current warp and switch to other warps more quickly.

The correlation between the load on Commit Unit and overall performance is most obvious in Hashtable (HT1K and HT512): the ratio between aborts at the SIMT core and Commit Unit has shifted dramatically. With Early-Abort enabled, more than half of the transactional aborts occur in the SIMT cores. This means there were many conflict pairs that fall into the Type 2 (inter-warp) and 3 (inter-core) categories that were not resolved by intra-warp conflict resolution alone, but are resolved by Early-Abort global conflict resolution. As a result, despite the higher number of cumulative aborts, the transactions are able to complete faster because the Commit Units are less congested.

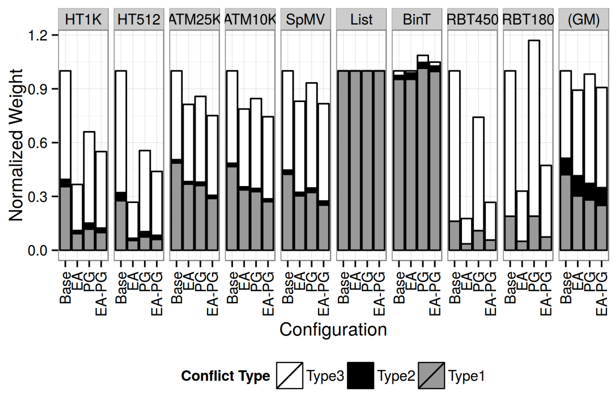


Figure 11: Overall Breakdown of types of conflict between all running transactions, counted in pairs of transactions, by whether read/write addresses overlap.

Figure 11 shows that in Hashtable (HT1K and HT512), Bank Account (ATM25K and ATM10K) and SpMV, more than half of the conflicts between transactions are of Type 3. Most of the conflicts can be detected by Early-Abort in the SIMT core, but would have to go through the Commit Unit in Warp TM with only intra-warp conflict resolution enabled. On the other hand, enabling Pause-and-Go execution scheme reduces the overall number of aborts and amount of

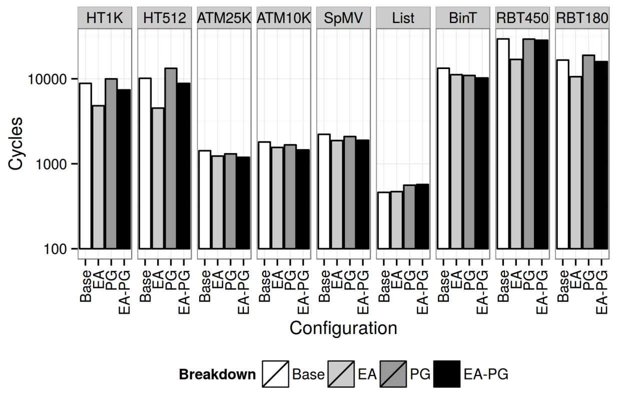


Figure 12: Average transaction length in cycles.

```
Hashtable insertion kernel:
   __tbegin();
   int hash = key, pool_slot = tid+1
   BaseEntry *base = &g_hashtable->mValues[hash];
   tTableEntry *ent = &g_entrypool[pool_slot];
   ent->mKey = key;
   ent->mValue = value;
   ent->mNext = base->mIndex;
   g_hashtable->mValues[hash].mIndex = pool_slot;
   __tcommit();

Binary tree insertion kernel:
   __tbegin();
   while (true) {
       if (val < curr->val) {
          if (curr->left == NULL)
              curr->left = my; break;
          else curr = curr->left;
       } else {
          if (curr->right == NULL)
              curr->right = my; break;
          else curr = curr->right;
   } }
   __tcommit();
```

Figure 13: Transactional code regions in Hashtable and Binary Tree.

conflicts most of the time.

7.3 Performance Improvement from Reduced Aborts

For List, Binary Tree (BinT) and Red-Black Tree (RBT450 and RBT180), Pause-and-Go execution scheme achieves similar or greater performance improvement than Early-Abort does. This is because of the following two reasons:

- The three benchmarks spend a significant portion of time in transaction execution, rather than in the Commit Unit. Thus, the speedup from reduced transaction re-execution becomes more significant. Figure 10 suggests that the absolute number of aborts are always reduced when Pause-and-Go is activated.

- The three benchmarks are inherently more divergent than Hashtable, ATM and SpMV, so branch divergence resulted from Pause-and-Go gets amortized with divergence, and do not affect the overall speedup much.

With fewer transactions aborted (Figure 10) and no significant increase in average transaction execution length (Figure 12), the time spent on re-execution is decreased, resulting in overall speedup for List, Binary Tree and Red-Black Tree.

Figure 13 shows the transactional part of Hashtable and Binary Tree. The Hashtable kernel does not have any `if` statements, so each thread executes the same code path. For Binary Tree, the data affects the code path executed by the threads in a warp and can cause branch divergence. This means that the length of transactions in Binary Tree is variable and can be much longer than that of a Hashtable even if the average transaction length is similar, making re-execution more costly. Also, the performance penalty caused by branch divergence resulting from pausing threads is less significant in Binary Tree. This can be justified by Figure 12, as the number of cycle per transaction for Binary Tree actually decreased.

In contrast, branch divergence induced by Pause-and-Go could cause transactions to run longer for HashTable. Applying either Early-Abort or Pause-and-Go reduces aborts at the Commit Unit to the same level for HashTable, but due to the longer average transaction length, Pause-and-Go does not deliver as much performance improvement as Early-Abort.

7.4 Combining Early-Abort and Pause-and-Go Execution Scheme

According to the experimental results we can observe the following:

- Enabling both approaches yields a greater performance improvement in Hashtable (HT1K and HT512), Bank Account (ATM25K and ATM10K) and SpMV than enabling either approach individually.
  In these applications, the number of aborts at the Commit Unit is fewer than when enabling either approach individually.

- Enabling both approaches in List gives the same improvement as enabling Pause-and-Go alone does. In fact, Early-Abort is never triggered in this benchmark.

- Enabling both approaches in Red-Black Tree (RBT450 and RBT180) is not as good as enabling only Pause-and-Go.
  The reason is, when Early-Abort and Pause-and-Go are enabled simultaneously, performance penalty resulting from false positives would arise (non-conflicting transactions are wrongly aborted at the SIMT core). The penalty is re-execution which could be expensive, so the result is less optimal than when using Pause-and-Go alone.

Therefore, we can devise the following rules for applying either or both early conflict resolution approaches:

- If the benchmark consists of large read/write sets (Red-black tree, Binary Tree), apply Pause-and-Go execution scheme;

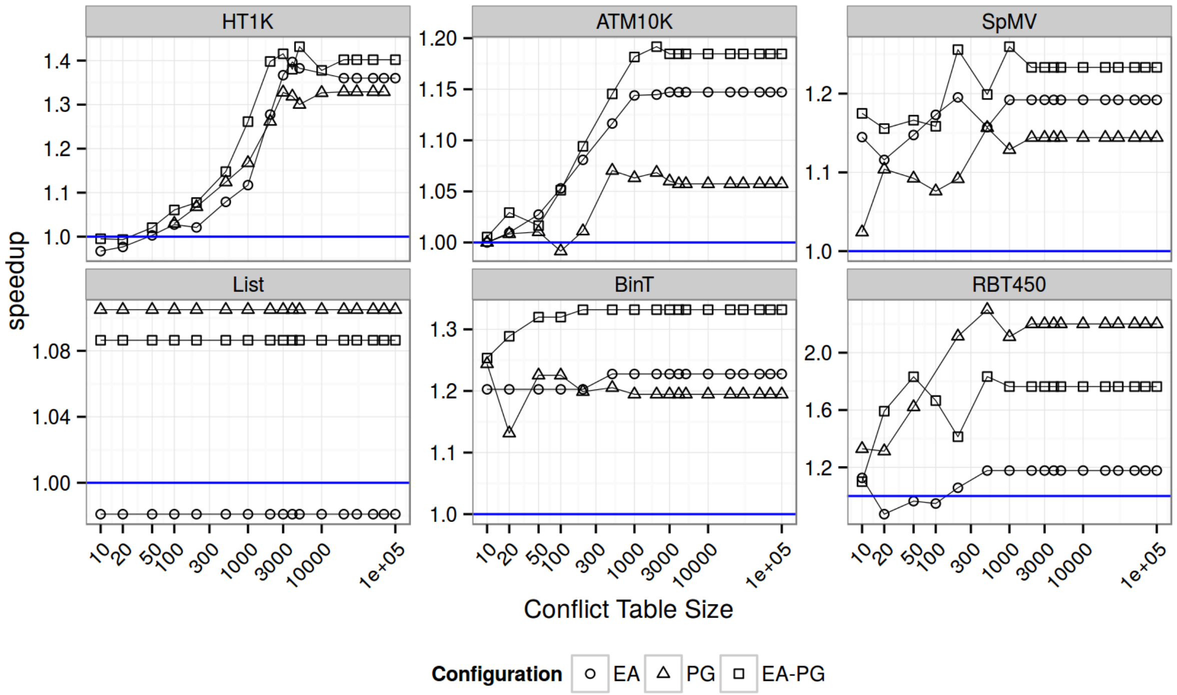


Figure 14: Sensitivity of Relative Speedup to Table Sizes.

- If the benchmark consists of mostly Type 1 conflicts and almost no conflicts of other 2 Types (example: List, Binary Tree), apply Pause-and-Go execution scheme;
- If the benchmark consists of considerable amounts of Type 3 and 2 conflicts, apply Early-Abort; further, if it consists of short transactions with small read/write sets, apply both Pause-and-Go and Early-Abort.

7.5 Sensitivity to Table Size

We vary the size of the Reference Count Table and the Conflicting Address Table and see how the performance improvement from applying either or both early conflict resolution methods changes. The sizes are varied from 10 through 100000. The results for HT1K, ATM10K, SpMV, List, BinT and RBT450 are shown in Figure 14.

Overall, for all benchmarks except List, a larger table size gives greater performance improvement. Increase of speedup slows down after the table size exceeds around 1000 for SpMV and RBT450 or around 3000 for HT1K and ATM10K. This means the table size of 3000 is enough for tracking all the addresses touched by the concurrent transactions. For List, a very small table size (say 10) performs as well as a large table.

7.6 Interconnection Network Delay and Traffic

Early-Abort and Pause-and-Go relies on passing of conflict address information from the Commit Units to the SIMT cores. For the benchmarks we used in this paper, it takes 5 to 25 cycles for the conflict address messages to travel from the Commit Units to the SIMT cores, as is shown in Figure 15. The results indicate that the delays are not directly related to the speedup.

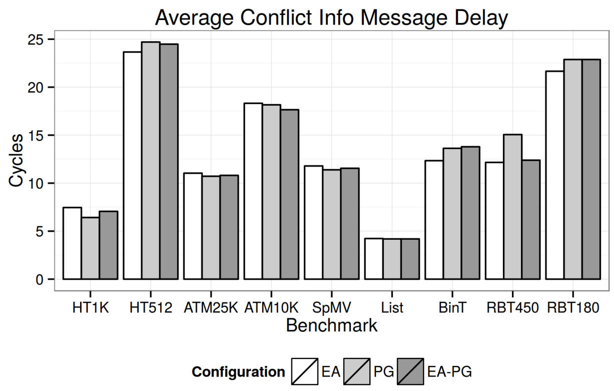


Figure 15: Average message delay between the Commit Unit and the SIMT cores.

Depending on the benchmark, the ratio between the extra traffic and the original traffic ranges from less than 1% (in Binary Tree) to around 20% (in HT 1K). The size of the extra traffic is on par with the transferred logs.

8. RELATED WORK

There are other proposals for GPU transactional memory other than Warp TM and Kilo TM. Various software-based implementations exist that can perform as well as CPU counterparts. Efforts have gone into various aspects concerning the design of GPU TMs: Xu et al. [24] proposed GPU-STM with encounter-time lock sorting to avoid deadlocks. Holey et al. [11] explored eager/pessimistic conflict detection on read/write operations.

Waliullah et al. pointed out that conflicts in hardware TM systems cause performance losses due to aborts and extra communication [21]. Their work introduced a new cache miss state that can help eliminate conflicts. With a similar goal, our two approaches aim at reducing aborts and communication between the core and the Commit Units.

There exist various proposals for accelerating transactional memory using hardware mechanisms. Wang et al. proposed TCache [22] which caches the shadow copy of transactional blocks, thus accelerating the re-execution for restarted transactions. Stipic et al. [19] proposed GTags, a hardware mechanism for fast access to transactional meta-data needed for conflict detection. Like both of these mechanisms, the Early-Abort approach also shortens the time needed to decide when to abort a transaction, thus accelerating transactions.

Xiang et al. proposed Staggered Transactions [23] that puts a thread into wait mode when a data conflict is likely to happen, rather than abort the thread eagerly. In their work, threads decide when to pause themselves by accessing locks using non-transactional loads/stores in a transaction. Pause-and-Go execution scheme shares the same philosophy but pauses threads from the warp scheduler's point of view, not from the scalar threads' point of view, due to the difference between GPU and CPU architectures.

9. CONCLUSION

We proposed two early conflict resolution methods, Early-Abort and Pause-and-Go execution scheme, for GPU hardware transactional memory systems. Our approaches are based on making conflict information available to the SIMT cores for early conflict resolution, shortening the time required to abort conflicting transactions and enabling pausing a transaction to avoid performing a conflicting load/store. Our evaluation showed the approaches reduced conflicts and Commit Unit contention, resulting in an average of 1.41x speedup at 0.8x energy consumption.

This paper demonstrates the effectiveness of utilizing information regarding conflicting transactions to resolve conflicts early. This insight may be gradually incorporated into future development of contention management and conflict resolution techniques on future transactional memory systems involving GPUs.

10. REFERENCES

[1] "About CUDA," https://developer.nvidia.com/about-cuda.

[2] "GPGPU-Sim 3.2.1 with Warp TM," http://www.ece.ubc.ca/~wwlfung/code/kilotm-gpgpu_sim.tgz.

[3] "OpenCL. The open standard for parallel programming of heterogeneous systems." https://www.khronos.org/opencl/.

[4] "Chapter 8, Intel Transactional Synchronization Extensions," in *Intel Architecture Instruction Set Extensions Programming Reference*, 2012.

[5] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance Pathologies in Hardware Transactional Memory," in *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, 2007.

[6] W. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware transactional memory for GPU architectures," in *Proceedings of the 44th International Symposium on Microarchitecture(MICRO)*, 2011.

[7] W. Fung and T. Aamodt, "Energy efficient GPU transactional memory via space-time optimizations," in *Proceedings of the 46th International Symposium on Microarchitecture (MICRO)*, 2013.

[8] M. Herlihy, J. Eliot, and B. Moss, "Transactional Memory: Architectural Support For Lock-free Data Structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, 1993.

[9] M. Herlihy and V. Luchangco, "Software Transactional Memory for Dynamic-Sized Data Structures," in *Proceedings of the 22th Symposium on Principles of Distributed Computing (PODC)*, 2003.

[10] M. Herlihy, V. Luchangco, and M. Moir, "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)*, 2006.

[11] A. Holey and A. Zhai, "Lightweight Software Transactions on GPUs," in *Proceedings of the 43rd International Conference on Parallel Processing (ICPP)*, 2014.

[12] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *Micro, IEEE*, vol. 31, no. 5, pp. 7–17, Sept 2011.

[13] J. H. Kim, H. Cameron, and P. Graham, "Lock-Free Red-Black Trees Using CAS," 2011.

[14] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013.

[15] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. S. III, and M. L. Scott, "Lowering the overhead of nonblocking software transactional memory," Technical Report 893, Dept. of Computer Science, University of Rochester, 2006.

[16] P. Misra and M. Chaudhuri, "Performance Evaluation of Concurrent Lock-Free Data Structures on GPUs," in *Proceedings of the 18th International Conference on Parallel and Distributed Systems (ICPADS)*, 2012.

[17] A. Shriraman, S. Dwarkadas, and M. L. Scott, "Flexible Decoupled Transactional Memory Support," in *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, 2007.

[18] M. F. Spear, M. M. Michael, and C. von Praun, "RingSTM: Scalable Transactions with a Single Atomic Instruction," in *Proceedings of the 20th International Symposium on Parallel Algorithms and Architectures (SPAA)*, 2008.

[19] S. Stipic, S. Tomic, F. Zyulkyarov, A. Cristal, O. Unsal, and M. Valero, "TagTM - accelerating STMs with hardware tags for fast meta-data access," in *Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2012.

[20] V. Volkov, "Better performance at lower occupancy," in *GPU Technology Conference (GTC)*, 2010.

[21] M. Waliullah and P. Stenstrom, "Classification and Elimination of Conflicts in Hardware Transactional Memory Systems," in *Proceedings of the 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2011.

[22] S. Wang, D. Wu, Z. Pang, W. Tang, and X. Yang, "Lowering the Overhead of Hybrid Transactional Memory with Transact Cache," in *Proceedings of the 9th International Conference for Young Computer Scientists (ICYCS)*, 2008.

[23] L. Xiang and M. L. Scott, "Conflict Reduction in Hardware Transactions Using Advisory Locks," in *Proceedings of the 27th International Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015.

[24] Y. Xu, R. Wang, N. Goswami, and T. Li, "Software Transactional Memory for GPU Architectures," *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2014.