Journal of Parallel and Distributed Computing 139 (2020) 1–17

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A general purpose contention manager for software transactions on the GPU

Qi Shen [a], Craig Sharp [b], Richard Davison [b], Gary Ushaw [b], Rajiv Ranjan [b], Albert Y. Zomaya [c], Graham Morgan [b,*]

[a] Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, China
[b] School of Computing, Newcastle University, UK
[c] School of Information Technologies, The University of Sydney, Australia

ARTICLE INFO

Article history:
Received 16 February 2019
Received in revised form 22 September 2019
Accepted 23 December 2019
Available online 28 January 2020

Keywords:
GPU
Parallel processing
High performance computing

ABSTRACT

The Graphics Processing Unit (GPU) is now used extensively for general purpose GPU programming (GPGPU), allowing for greater exploitation of the multi-core model across many application domains. This is particularly true in cloud/edge/fog computing, where multiple GPU enabled servers support many different end user services. This move away from the naturally parallel domain of graphics can incur significant performance issues. Unlike the CPU, code that is hindered from execution due to blocking/waiting on the GPU can affect thousands of threads, rendering the advantages of a GPU irrelevant and reducing a highly parallel environment down to a serial one in the worst case. In this paper we present a solution that minimises blocking/waiting in GPGPU computing using a contention manager that offsets memory conflicts across threads through thread re-ordering. We consider conflicts of memory not only to avoid corruption (standard for transactional memory) but also in the semantic layer of application logic (e.g., enforcing ordering to ensure money drawn from bank account occurs after all deposits). We demonstrate how our approach is successful across a number of industry benchmarks and compare our approach to the only other related solution. We also demonstrate that our approach is scalable in terms of thread numbers (a key requirement on the GPU). We believe this is the first work of its kind demonstrating a generalised conflict and semantic contention manager suitable for the scale of parallel execution found on a GPU.

1. Introduction

A Contention Management Policy (CMP) for parallel exploration of solutions to transactional conflict using the GPU is described. The approach utilises a priority rule based technique to explore multiple schedules of transaction permutations on the highly threaded GPU, in the context of resolving concurrent conflict in software transactional memory (the technique is labelled PR-STM).

Due to the execution nature of the GPU, transactions are enacted in batches. They are copied from main memory to the GPU and executed. Only when all transactions have completed on the GPU are results returned to main memory. Threads may not transfer from CPU during batch execution on the GPU (i.e., join and leave execution phases under pre-emptive CPU and operating system control). Therefore, the major research challenge is to determine a solution within which all GPU hosted transactions can be executed efficiently (in terms of timeliness) while avoiding livelock and deadlock.

We extend our CMP from [30] that addresses concurrent conflicts (ensuring ordered and correct memory access of a memory location) on GPU to also address semantic conflict. A semantic conflict occurs when there are application-specific factors affecting the order in which transactions must be completed (for example, funds must be placed into a bank account before they can be withdrawn). A CMP which only addresses concurrent conflict (interference of state resulting in possible inconsistency) will promote a lack of logical progress to the application level; there will be no state conflicts, but applications cannot progress logically in the context of their execution requirements. For example, transactions trying to withdraw money when there is no money available may commit successfully due to no concurrent conflicts resulting in inconsistency of memory, but will semantically hinder the application in a logical sense. The ordering of transactions, in our model, includes consideration of concurrent and semantic conflict.

* Corresponding author.
E-mail addresses: shenqi@act.buaa.edu.cn (Q. Shen), craig.sharp@newcastle.ac.uk (C. Sharp), richard-gordon.davison@newcastle.ac.uk (R. Davison), gary.ushaw@ncl.ac.uk (G. Ushaw), raj.ranjan@newcastle.ac.uk (R. Ranjan), albert.zomaya@sydney.edu.au (A.Y. Zomaya), graham.morgan@newcastle.ac.uk (G. Morgan).

https://doi.org/10.1016/j.jpdc.2019.12.018
0743-7315/

We add a semantic conflict management policy (CMP) to our existing concurrent conflict resolution policy, whereby a thread that is executing a transaction must check whether a semantic conflict occurs. If so, the transaction is delayed and the thread searches for the next uncommitted transaction. In order to achieve this on the parallel architecture of the GPU we introduce a global transaction table which tracks the commit status of multiple transactions. This itself utilises the parallel nature of the GPU, bringing scalability to our solution.

The effectiveness of our approach to PR-STM on the GPU is demonstrated by injecting semantic conflict into well-known benchmarks for contention management policies. We utilise the *Bank* benchmark (which is provided within *tinySTM* [11]), the *Vacation* and *Kmeans* benchmark (commonly associated with Stanford Transactional Applications for Multi-Processing (STAMP) [21]), the *Skiplist* benchmark [18] and a sample graph processing benchmark. The performance figures presented show that our approach handles semantic conflict appropriately. We argue that the overhead introduced by our approach is offset by the increase in timely throughput at the logical progression layer of the application. We compare our work against the comparator GPU-STM [34], the only other possible comparator at this time. We show that our approach performs better than GPU-STM when semantic conflict is introduced.

This is the first time that semantic conflict has been considered in a CMP that operates on the GPU. While there is an additional overhead in handling semantic conflict, our generalised approach to contention allows the CMP to be used for arbitrary permutations of transaction, without requiring any sorting by the user application.

2. Background and related work

This section introduces the GPU architecture, its execution model, and the notion of concurrent conflict together with concurrency control in the context of software transactional memory. We consider existing approaches to CMP, on both CPU and GPU to provide a broader context for our work. As a relatively new concept, we dedicate a portion of our text to a more detailed overview of semantic conflict resolution.

2.1. General purpose GPU execution model

To design an algorithm for GPU computing, it is necessary to take into account the differences in execution between the CPU and GPU. In a multi-core CPU, threads are delegated to cores with such execution continuing in parallel across cores with the only coordination occurring when memory is shared (conflict). This may require one or more threads to wait. In essence, a multi-core CPU may support unrelated execution traces of distinct types occurring across cores that share the same memory and such threads may experience a degree of blocking before their completion. Standard programming languages (such as C++) can support multi-threading using language primitives with the operating system handling the pre-emptive nature of execution. For example, one thread may be downloading information from a port while another renders an interface to a user and only when the interface requires the port information do they share memory through pre-defined locking strategies (e.g., transactional memory).

Due to the nature of the GPU's hardware, a dedicated programming environment is required. In our work this is provided by CUDA (Compute Unified Device Architecture) [26]. CUDA provides a General Purpose GPU (GPGPU) computing API. In essence, this allows non-graphically related programs to be constructed for execution on the GPU.

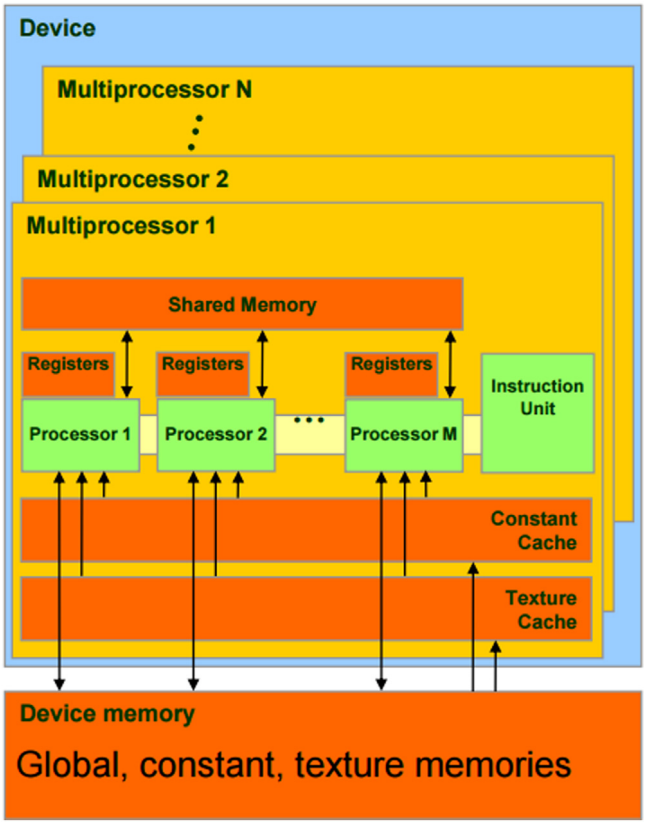


Fig. 1. CUDA device architecture.

Fig. 1 shows an abstract architecture of a CUDA device. CUDA applications can run on any card which supports this architecture (usually provided by Nvidia). However, as each GPU device has different specifications, they may have slightly different sets of supported features and different numbers of available computational resources. Any CUDA supported device should have some Streaming Multiprocessors (SMs), each of which consists of some processor cores (ALUs), one multi-thread instruction unit, a shared memory and a set of registers [25].

A CUDA application requires a compute kernel to be invoked that represents the desired execution to take place on the GPU. In a graphics environment these may be termed compute shaders and may refer to the graphical techniques run on a GPU in highly parallel ways. However, in CUDA these may represent arbitrary, more general, computations but are executed and managed in the same manner as their graphical counterparts. From a programmer's perspective, threads are organised into thread blocks that carry out the desired execution defined in the compute kernel within which such threads may share memory. Therefore, it is the programmer's task to allocate these executions appropriately from the CPU and handle their outputs back to the CPU once execution has finished.

Thread block scheduling is handled by a streaming multiprocessor that assumes responsibility for scheduling thread blocks in parallel and attempting to execute thread blocks when resources permit. From a hardware perspective, thread blocks are deployed across warps that expect hosted threads to execute the same code. Thread divergence may occur given the existence of branching in code (e.g., if statement). A warp incurs overheads as it attempts to execute divergent threads. This is because warps may only handle divergence of execution by deactivating divergent threads, according to a well known policy, and executing

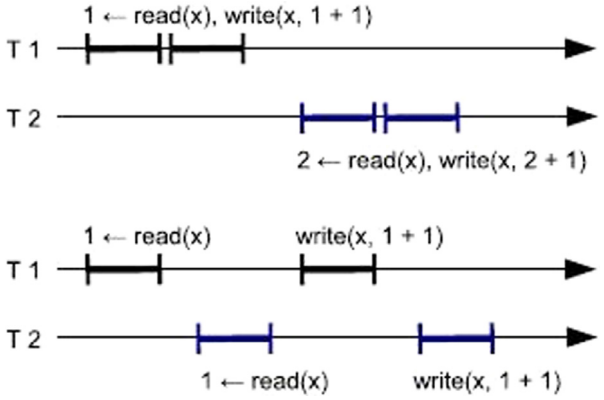


Fig. 2. A race condition.

them in series. This incurs substantial overhead by reducing the parallel efficiency of the execution and highly divergent environments can create serial environments on a GPU. This outlines the significant difference between CPU and GPU: the GPU performs at its best with limited divergence whereas the CPU is accustomed to highly divergent thread hosting (i.e., thread context in terms of programming logic are of little importance to the CPU in terms of scheduling).

A key point to highlight in the context of this work is that a substantial amount of wasted resources may be witnessed by the GPU unnecessarily if logical issues in thread execution are a problem. An "if" statement waiting on a memory change (e.g., if account is greater than zero), can cause substantial overheads on a GPU. The fact that we are concerned with a batch programming approach means that we simply cannot retry the logical execution again and again as on the CPU as it will lead to deactivation of a large number of threads which must wait for all previous threads to finish on the GPU. Therefore, we want to ascertain the appropriate ordering of threads in a manner that ensures the most optimum and successful execution once a compute kernel is enacted on warps. In graphical programming this is of little issue as compute shaders are naturally parallel in their exploitation (the reason for a GPU architecture). However, in GPGPU scenarios this can cause significant problems and overheads.

2.2. Concurrent conflict and concurrency control

We now explain conflict resolution in the context of well known concurrent issues in parallel programming. Race conditions are a well-known type of concurrency error that may result in inconsistent data across multi-threaded programs. Fig. 2 provides an example with two threads (T1 and T2) which read the same location (x) in shared memory and increase the value of that shared memory by 1. The possibility of a race condition means that there could be two different final values for memory location x. In the top scenario, the memory holds the value 2 when there is no interference between T1 and T2. However, in the bottom scenario, interleaved thread execution produces a final memory value of 1.

In this example the possibility of a race condition has removed the element of determinism from the program; it is no longer possible to say what value will be held at x. Concurrency control aims to implement mechanisms which prevent this loss of determinism. The two most prominent approaches to implementing concurrency control are pessimistic and optimistic concurrency control. Pessimistic concurrency control (PCC) protocols aim to prevent non-deterministic conflict occurring in advance, normally by using blocking synchronisation such as locks or semaphores. Optimistic concurrency control (OCC) detects conflicts after they have occurred and then implements steps to correct such conflicts [20], for example using abortable transactions.

Blocking synchronisation by mutual exclusion [9] is a common approach to pessimistic concurrency control and transactional memory [15] is a popular optimistic method for concurrency control. As there exist many kinds of applications in which concurrency control is needed, no single approach is likely to be the best solution for all scenarios. However, whether a method is pessimistic or optimistic, the main aims of a concurrency control mechanism are:

1. *Correctness*: Prevent the logical inaccuracy.
2. *Efficiency*: Resources should be used efficiently.

2.3. Transactional memory

Transactional memory is a well known example of optimistic concurrency control. In the transactional memory paradigm, threads can access shared data within the execution of a transaction. However, modifications to shared data are not made permanent until the transaction has completed successfully. These changes are made to shared data only if no concurrency conflict is detected, or otherwise the potential changes must be aborted and the transaction will restart. The main benefit of transactional memory compared to mutual exclusion is that deadlock can be avoided [15]. This is because all threads can operate without interference from other threads. Transactional memory implementations, however, typically require an overhead in excess of that required by a more simple blocking approach, which may lead to the degradation of performance when contention is high.

2.4. Contention management policies

To alleviate the degradation in performance of transactional memory in the presence of high contention a Contention Management Policy (CMP) may be employed. A variety of CMPs have been successfully implemented, which can be categorised as either wait-based or schedule-based. Wait-based CMPs, such as *Greedy*, *Karma* and *Polka* are well-known approaches which are relatively straightforward to implement, offering versatility and improved performance [14,27]. An inefficiency with wait-based approaches was identified in [16], as the dynamic nature of execution of STM leads to difficulty in finding an adequate back-off period.

Schedule-based CMPs typically reschedule or serialise aborted transactions. An example is described in [2] whereby transactions are distributed among the threads, with transactions that are likely to conflict being assigned to the same thread, thus assuring serialisation. In [1] a *Steal-on-Abort* approach is described in which various techniques are considered for rescheduling transactions across threads, with additional work queues created when the number of transactions passes a defined threshold. A collision avoidance and resolution approach to schedule-based CMP is described in [10] which also reassigns conflicting transactions to the same work thread.

Wait-based and schedule-based CMP take no consideration of the semantics of the application that they support. Several approaches to STM have been developed which employ a universal construction (UC). The concept was introduced in [17], enabling multiple threads to access a shared data structure in a wait-free manner. The UC technique was subsequently applied to transactions to handle failure conditions [7,33]. The approach was further extended in [8] to remove the abort semantics of STM. Thread level speculation is introduced to transactional memory in [3]. In this approach platform parallelism is exploited to explore different permutations of transactional elements. This is

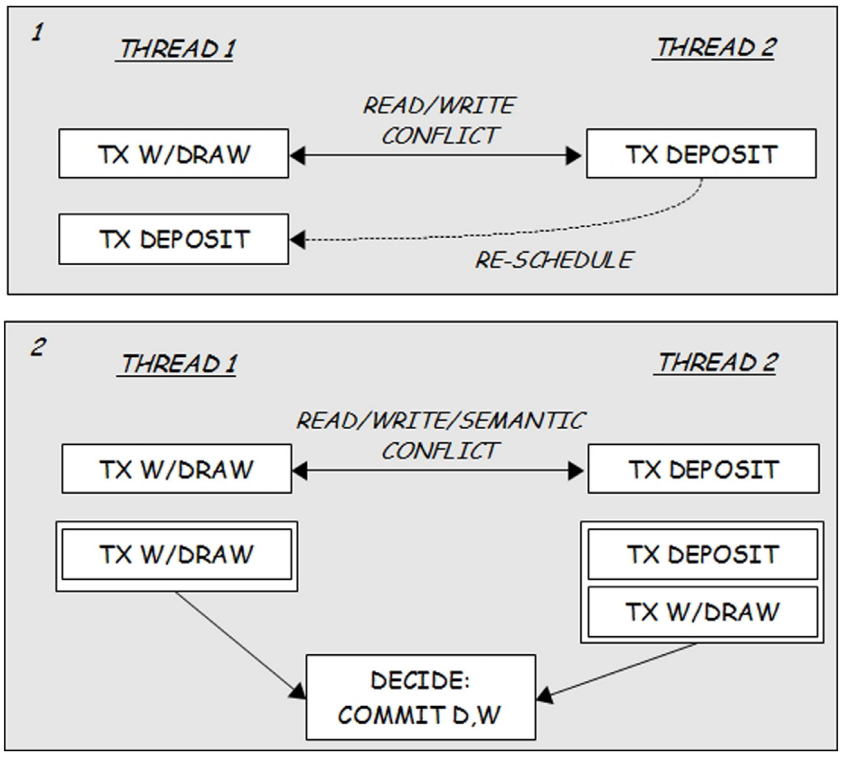


Fig. 3. In scenario 1 a read/write conflict has occurred between the w/draw and deposit transactions. The depositor is aborted and rescheduled to execute after the withdrawer has committed. In scenario 2, thread 1 aborts the depositor but then also aborts itself because of a semantic conflict caused by attempting to execute a withdrawal before a deposit. The conflict is resolved by the execution of an ordering which allows both to commit in the correct semantic order (deposit then withdraw).

achieved through reordering the internal execution of a transaction to better reflect concurrent schedules. This technique reorders executions at a lower level than that presented in our paper, as we seek to reschedule whole transactions to better accommodate the semantics of execution.

2.5. Parallel exploration of transaction scheduling on CPU

The CMP described in [29] and [28] uses a UC technique to resolve conflicts (labelled *Hugh2*). The threads which are considered are those which contain transactions that have been aborted due to semantic conflicts. The technique was first presented for object-based STM in [29] and extended to resolve semantic conflicts in word-based software transactional memory in [28].

Universal Construction allows any sequential data structure to be transformed into a linearisable representation that can be accessed and updated by a number of threads [17]. UC consists of three phases. Firstly a thread proposes an input to be added to the UC. Secondly each thread which has proposed an input performs consensus to decide which input will be added. Lastly the winning thread updates a log of inputs to reflect the decision. *Hugh2* accepts as input a permutation of one or more sequentially executed transactions and decides which permutation will be added to the log.

Two hypothetical scenarios are shown in Fig. 3, both involving a *depositor* and a *withdrawer* transaction which access a shared object. In scenario 1, the CMP reorders transactions to avoid concurrent conflicts. Although the withdrawer transaction can commit, it may need to re-execute in future (if deposits must precede withdrawals for example). In scenario 2, our approach is illustrated where a semantic abort occurs and each thread re-executes a different permutation of the aborted transactions.

The CMP is activated when thread *a* encounters a semantic conflict (causing *a* to explicitly abort its transaction). Before the aborted transaction is restarted, thread *a* enters a new *session* mode. During session mode, *a* re-executes its own transaction in addition to the transactions of any other session mode threads. Each session mode thread executes a single permutation of transactions, to discover a schedule of transaction execution which resolves the semantic conflict.

When there are no further transactions to execute, each thread performs consensus to determine the permutation to be committed. Consensus is managed using the UC, which is essentially a linked-list that may be concurrently appended to by threads engaged in session mode. Each new entry of the UC identifies the transactions that have been committed during a particular session. Once a session has terminated each participating thread can determine whether its own transaction was committed or aborted by reading the log of the UC. Those threads whose transactions remain uncommitted perform a new session, while the threads of the committed transactions return to non-session mode.

This approach to CMP for STM was the first to use additional threads to provide multiple serialised executions of aborted transactions in parallel. This was achieved without the overhead of a thread-pool. However, the number of additional threads required to calculate multiple serialised execution for aborted transactions is based on the number of aborted transactions. The probability of having large amounts of aborted transactions is high as there are many more threads on the GPU which makes the conflict rate higher than on the CPU. The expense of having more threads providing multiple serialised executions is not acceptable as it is running on the CPU.

2.6. Transactional memory on GPU

The ability of the GPU to handle considerably more threads than the CPU has recently led to increased interest in utilising transactional memory for GPU. Both hardware and software

transactional memories have been proposed for the GPU architectures.

On the hardware side, Kilo TM was proposed [13] in 2011. It implements a specific Commit Unit to carry out lazy conflict detection and version management. WarpTM [6,12] improved this by considering read-only transactions, reducing bus communication and early conflict detection. GPU-LocalTM [31,32] is an eager approach on the other hand, which supports synchronisation at local memory level. All these approaches must modify current GPU hardware architecture which means they cannot directly be applied on current commercial GPUs.

The first STM on the GPU that operates at the granularity of a thread-block (as opposed to the granularity of individual threads) is described in [5]. The thread-block approach avoids dependency between threads within a single block. This coarse granularity reduces contention due to the typically high thread numbers available on GPU, but does not accommodate workloads more appropriate for GPU execution. Generic modifications to improve the performance of algorithms with irregular access patterns have been proposed [24]. GPU techniques to speed up execution by reducing the usage of atomic operations have also been explored [23].

An approach, labelled GPU-STM, which does operate at the granularity of the individual thread, is described in [34]. The technique is based on a hierarchical validation system which is a combination of time-stamp and value based validation. Locks must be sorted whenever transactional reading or writing takes place, to avoid the possibility of livelock. The technique described in our paper avoids the necessity for sorting locks due to the introduction of a static priority rule. The lock sorting is a novel way to solve extra livelock possibilities introduced by lock-step execution mode on the GPU, but the overhead is comparatively high as every read or write operation in a transaction has to search and/or update the lock table. We utilise GPU-STM as the comparator during the evaluation of our system.

Three lightweight techniques for software transaction management on GPU are described in [19]. Firstly ESTM (*eager*) updates shared memory during transaction execution while maintaining an undo log which is used to remove those updates upon an abort. Secondly PSTM (*pessimistic*) is a simpler version of ESTM treating reads and writes in the same manner, which increases the effectiveness when transactions regularly read and write to the same shared data. Finally ISTM (*invisible*) which can represent invisible reads to reduce conflicts during a transaction. However none of these three techniques allow for lock-stealing based on thread priorities. While [19] compares the performance of each algorithm with the CPU, only basic fine-grain and coarse-grain locking benchmarks are employed.

2.7. Contribution

We consider the increased threading available within the GPU as an opportunity to explore transactional ordering that not only resolves concurrent conflicts at the data integrity layer, but at the application semantic layer also. In doing so we present a CMP implementation that improves overall application performance in terms of successful transactional commit ordering while promoting the logical progression of an application. We demonstrate our approach with well known benchmarking applications and compare our approach against the closest state-of-the-art. However, as this is the first time semantic consideration for thread scheduling on the GPU is employed, we also demonstrate the usefulness of our unique technique. A significant contribution in the context of developing threaded applications is that programmers do not have to consider the ordering of concurrent actions to promote application progression and avoid semantic conflict.

The reason why aborted transactions (divergence of execution) is of significant importance on the GPU is the batch nature of execution. In a warp an increase in divergence of execution amongst threads increases the serial nature of the execution, rendering the batch approach quite inappropriate.

To date the authors are not aware of any works addressing semantic conflict in transaction scheduling on the GPU.

3. Implementation

We have introduced our semantically aware universal construction for transactional memory systems [28,29] in previous works. We now describe how to modify and apply the technique to GPGPU scenarios via a priority rule-based software transactional approach, which we label PR-STM. The work provides a generalised contention manager which removes a requirement for the application programmer to order transactions appropriately to optimise performance by minimising concurrent and semantic conflicts resulting from thread divergence in GPGPU scenarios.

3.1. Priority rule based STM for GPU

In order to evolve our semantic approach to CMP for use on the GPU, we introduce a *priority rule* based STM (PR-STM). Two fundamental differences between the CPU and GPU must be taken into account when implementing a CMP for GPU. Firstly there is potentially a significantly higher number of threads available on a GPU, and secondly GPU threads are grouped together as a warp with similar execution traces. A common instruction counter is used by all threads in a warp, so they execute each instruction in a lock-step fashion. High levels of contention are more likely due to the increased number of threads available. Furthermore, deadlock and livelock are possible as threads in the same warp cannot coordinate their access to locks in the same manner as on the CPU and any divergence present could result in the deactivation of many threads (reducing the parallel nature of execution much more than a CPU counterpart).

We introduce a lock-stealing algorithm which prevents the possibility of livelock and deadlock. This is achieved by assigning a unique static priority to every thread. A thread may then steal a lock from any thread with a lower priority than its own. This addresses the issue of deadlock as a thread can always determine its next action when locked data is encountered. The possibility of livelock is also removed as threads can never attempt to perpetually steal each others locks (the thread with highest priority would take the lock while other threads rollback their actions). The way in which PR-STM addresses livelock and contention management is illustrated in Fig. 4.

A thread with high priority may cause a starvation problem during parallel computing on CPU. However, due to the execution model of the GPU, all threads will be assigned the same amount of transactions at start-up, with no more transactions added during their execution. Even if the thread with highest priority blocked all other threads (the worst case), other threads can still carry out their transactions after the highest priority one terminates. The possibility of starvation is therefore eliminated partly due to the nature of execution on the GPU.

In PR-STM, threads attempt to acquire locks at the end of their transactions (i.e., when ready to commit). Before committing, a thread must first attempt to validate its transactions by pre-locking the shared data. The priority rule still allows pre-locked data to be stolen by a thread with a higher priority. If validation is successful without locks stolen by other threads, the thread commits the transaction. Invisible reads are implemented to allow threads to maintain a version of the data they have accessed so

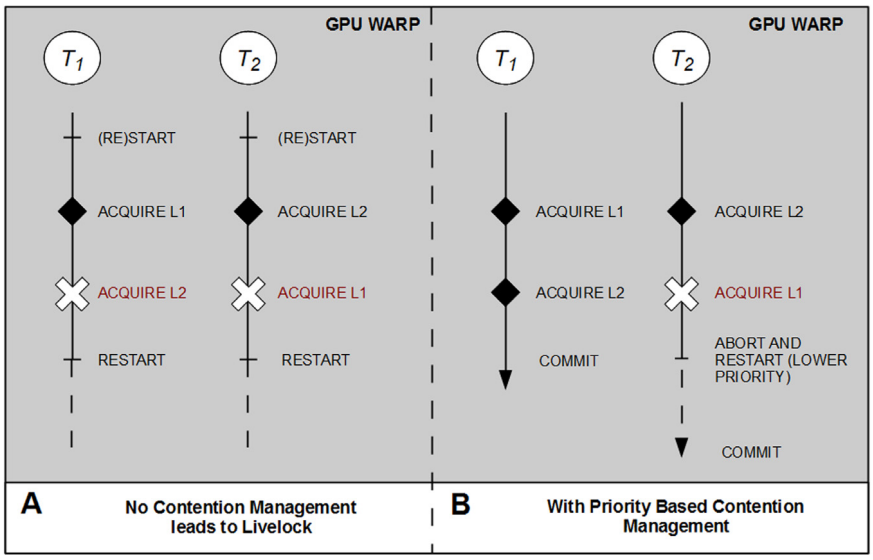


Fig. 4. Livelock avoidance on GPU.

an early abort can occur if a conflict is detected. The cost of false conflict is therefore reduced as a thread which encounters data that has been locked by a thread which will abort in the future does not itself need to abort.

Two types of metadata are required for PR-STM - *global* metadata which is shared among all threads, and *local* metadata which is private to a particular thread. The metadata used is as follows:

- Global: The *global lock table* contains a set of unique locks, each of which references shared data. Each lock can reference a variable number of words of data, enhancing the scalability of the system. However the greater the number of words of data referenced from a single lock, the greater the potential for false conflict based on shared locks. Each entry in the global lock table is an unsigned integer composed of the version (11 bits), owner (19 bits), locked flag (1 bit) and pre-locked flag (1 bit).
- Local: The *local read set* contains the current thread's set of reads. Each entry comprises a memory location, version and value. The *local write set* contains the current thread's set of writes.
- The *local lock set* contains the locks set by the current thread. Each entry consists of a lock index into the global lock table, and the lock version. Thread priorities along with lock versioning provide the required information for lock-stealing.

A number of functions are required for correct operation of the PR-STM. Algorithm 1 shows the pseudocode for each of these functions. The '⟨⟩' tuples means all data inside the tuple will be passed (read or write) at the same time (e.g. line 4 means checking whether that address and a value need to be set to that address is already in local write set).

3.1.1. txStart

Before a thread begins or restarts a transaction, the function *txStart* is called. The thread's local read, write and lock sets are initialised to be empty. The thread then sets its local abort flag to false.

3.1.2. txRead

When a thread attempts to read shared data from global memory, the *txRead* function is invoked. If the shared data is already locked by another thread, then the transaction is aborted and restarted. If the data has not already been locked, then the local write set is checked to see if the data has already been added to it (if so, the stored value is returned). If the data is not in the local write set then it is retrieved from the global memory, and added to the local read set, before being returned from the function.

3.1.3. txWrite

The *txWrite* function is invoked each time the thread attempts to write data. If the data is already locked then the *abort* flag is set, indicating that the transaction must abort and restart. If the data is not already locked, it is added to the local write set (or, if the data is already in the local write set, the value is updated).

3.1.4. txValidate

Before a transaction commits, the *txValidate* function is invoked. A lock is attempted on all shared data that the thread wants to write to, and validation is performed on all shared data that the thread has read. The thread first checks whether it has the highest priority on all shared data in its read set and write set by invoking *prelock*. The data in its read set is then validated by checking that the version of each piece of data has not been changed. The final validation step is to attempt to lock all the data. If any step is unsuccessful then the transaction is aborted.

3.1.5. txCommit

If a transaction has been successfully validated, the *txCommit* function is invoked. The global shared memory that is referenced in the thread's local write set is written to, and a *thread fence* is executed. The thread then updates the version number in the global lock table for each lock in its lock set, either incrementing it, or resetting it if the value has reached its maximum.

3.2. CMP for PR-STM

The contention management policy for PR-STM uses locks to protect shared data and to implement the priority rule policy. Each lock is a 32-bit word comprised of:

- Bits 1–11 contain the current version of the lock. This version is incremented whenever an update transaction is successfully committed. The version would be reset to 0 when it exceeds the value that these 11bits can represent (2^{11}, 2048), so there is no limitation of the number of modifications on a memory address.

ALGORITHM 1: Pseudocode for PR-STM functions *txRead*, *txWrite*, *txValidate* and *txCommit*

```
   function txStart()
1      readSet ← writeSet ← lockTable ← ∅;
2      abort ← false;
   function txRead(Address addr)
3      if getLockBit(g_lock[hash(addr)]) = 0 then
4          if < addr, valWritten >∈ writeSet then
5              return valWritten ;
           else
6              value ← atomicRead(addr);
7              version ←
               getVersion(atomicRead(g_lock[hash(addr)]));
8              readSet ← readSet ∪ {< addr, value, version >};
9              return value;
       else
10         abort ← true;
11         return 0;
   function txWrite(Address addr, Value val)
12     if getLockBit(g_lock[hash(addr)]) = 0 then
13         if < addr, valWritten >∈ writeSet then
14             < addr, valWritten > ← < addr, val >;
           else
15             idx ← hash(addr);
16             version ← getVersion(g_lock[idx]);
17             writeSet ← writeSet ∪ {< addr, val >};
18             lockSet ← lockSet ∪ {< idx, version >};
       else
19         abort ← true;
   function txValidate()
20     if tryPreLock() = true then
21         for all< addr, value, version >∈ readSet do
22             if getVersion(g_lock[hash(addr)]) ≠ version then
                   return false;
23         return tryLock();
       else
24         return false;
   function txCommit()
25     for all < addr, val >∈ writeSet do
26         *addr ← val;
27     _threadfence();
28     for all< idx, version >∈ lockSet do
29         if version < maxVersion then
30             setVersion(g_lock[idx], version + 1);
           else
31             setVersion(g_lock[idx], 0);
```

- Bits 12–30 contain the priority of the thread which has pre-locked the data. The lower the value, the higher the priority. These bits can present 2^{19} (524 288) threads which is enough for current GPU frameworks. If there are more than 524 288 threads, we could use an unsigned integer to present the pre-locked thread number, and the limitation would be expanded to 2^{32}, but the storage overhead would increase in the mean time.
- Bit 31 is a flag to indicate whether the lock is currently pre-locked. Threads with a higher priority may steal a pre-locked lock.
- Bit 32 is a flag to indicate whether the lock is locked. If set, no other thread can steal the lock.

The storage overhead of the system on global memory is the number of locks integer. Because the locks number can be manipulated on demand by modifying the lock coverage, the overhead could be from the number of items integer (which can achieve the best performance, there would be no fault conflict as each item is bound to a different lock) to 1 integer (which will achieve the worst performance, all items were bound to a single lock, a sequential computing likely fashion).

ALGORITHM 2: Pseudocode for locking mechanisms for PR-STM functions *tryPreLock*, *tryLock* and *releaseLocks*

```
   function tryPreLock()
1      for all< idx, version >∈ lockSet do
2          repeat
3              tmpLockVal ← g_lock[idx];
4              if getVersion(tmpLockVal) ≠ version
5              or getLockBit(tmpLockVal) = 1
6              or(getPreLockBit(tmpLockVal) = 1 and
               getOwner(tmpLockVal) < threadIdx) then
7                  releaseLocks();
8                  return false;
9              preLockVal ←
               calcPreLockedVal(version, threadIdx);
           until atomicCAS(g_lock+idx,tmpLockVal,preLockVal) =
           tmpLockVal;
10     return true;
   function tryLock()
11     for all< idx, version >∈ lockSet do
12         PreLockVal ← calcPreLockedVal(version, threadIdx);
13         FinalLockVal ← calcLockedVal(version);
14         if atomicCAS(g_lock+idx,PreLockVal,FinalLockVal) ≠
           PreLockVal then
15             releaseLocks();
16             return false;
17     return true;
   function releaseLocks()
       for all idx ∈ PreLocked do
18         preLockVal ← calcPreLockedVal(version, threadIdx);
19         atomicCAS(g_lock+idx,preLockVal,preLockVal-1);
       for all idx ∈ Locked do
20         unLockVal ← calcUnlockVal(version);
21         g_lock[idx] ← unLockVal;
```

The handlers required to manage the locking mechanisms for PR-STM are *tryPreLock*, *tryLock* and *releaseLocks*. The pseudocode for these handlers is shown in Algorithm 2.

3.2.1. tryPrelock

When a thread attempts to pre-lock shared data, the *tryPreLock* handler is called. Each lock in the local lock list is checked for three things: its availability, whether the lock versions are consistent, and whether there is an existing pre-lock from a thread with a higher priority. If any of these checks fail, the thread releases all locks that it has previously pre-locked and aborts. Otherwise the thread attempts to pre-lock the lock with an atomic *compare and swap* (CAS). If the CAS fails then another thread must have accessed the lock, so the checks are repeated until either the transaction is aborted, or the CAS succeeds (meaning this thread has the highest priority of the threads attempting to pre-lock the lock).

3.2.2. tryLock

When a thread has successfully pre-locked every lock in its local lock set, the *tryLock* handler is invoked. The thread attempts to lock each of its pre-locked locks with a CAS operation. If the CAS fails then the lock must have been stolen by a higher priority thread, so this thread releases all its locks and aborts.

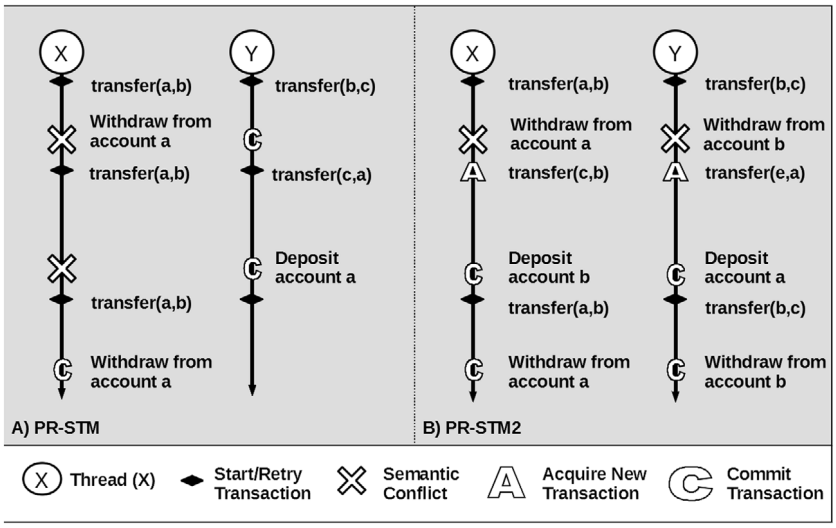


Fig. 5. Unnecessary retries are avoided by acquiring new transactions when a semantic conflict occurs.

3.2.3. releaseLocks

When a thread either commits or aborts, it invokes the *releaseLocks* handler. All locks on the local lock list are released (both pre-locked and locked). A CAS must be used to release pre-locked locks, in case that lock has been stolen by a higher priority thread.

3.3. Semantic conflict

Unlike threads on the CPU, groups of GPU threads execute in lock-step fashion, sharing a single instruction counter. In GPU terminology, this group of threads is called a warp and may consist of 1 to 32 threads. We have introduced a priority-rule based approach (PR-STM) for resolving concurrency conflicts on the GPU within software transactions. PR-STM avoids livelock, which is caused by a warp of GPU threads continuously generating the same concurrency conflicts due to lock-step memory access.

When semantic conflicts are introduced to the application they: (a) increase the possibility that transactions must abort; (b) reintroduce the possibility of livelock. For example, a transaction may now abort due to either a concurrency conflict or the semantics of the application preventing the transaction from completing (e.g. attempting to remove from a shared buffer which is empty). Furthermore, if the shared buffer remains empty, then the transaction may never commit and is in a state of livelock.

It is the objective of PR-STM to reduce the rate of aborts caused by semantic conflict and reduce the possibility of livelock. Eliminating livelock completely requires some transaction to always exist that will resolve the semantic conflict. For example, a transaction which appends items to an empty buffer or deposits funds in a bank account from which another transaction is attempting to dequeue/withdraw. Meeting this requirement is outside the scope of PR-STM. Furthermore, the application programmer must identify one simple priority permutation by tagging each type of transaction with a number. For example, in a bank scenario, the programmer must tag the deposit function with a number 1 and the withdraw function with a number 0 indicating that a deposit should precede a withdraw. This is the only consideration required by a programmer to ensure appropriate progression of the system.

Fig. 5(A) shows transaction execution without a semantic CMP. Thread *x* attempts to withdraw funds from a bank account that is empty and aborts/retries until the account has been deposited with funds by thread *y*. The number of extra retries depends on the time taken for another transaction to deposit funds into the account which can be arbitrarily large causing unnecessary resource consumption. Furthermore, as thread numbers grow so does the potential for a greater amount of unnecessary resource consumption.

In Fig. 5(B) all transactions are now stored in a *global transaction table*. As the scenario begins both threads encounter semantic conflicts. Unlike Fig. 5(A), semantic conflicts now cause each thread to abandon their transactions rather than retrying. Instead, the threads acquire new transactions from the global transaction table. At some future time the abandoned transactions are re-executed, possibly resolving the original semantic conflicts if conditions have changed sufficiently (if withdrawals can now be made because deposits have now taken place).

3.4. Semantic contention management policy

We use a postpone strategy to deal with semantic conflict and a global transaction table to store transactions (see Fig. 6). When semantic conflict occurs, the thread aborts the transaction and reads the next uncommitted transaction from the global transaction table. As the search wraps back to the start of the allocated transaction block, threads will retry previously aborted transactions. After all transactions belonging to a thread complete, the finish flag for this thread will be set, and after all threads are finished the result will be transferred back to CPU memory.

Pseudocode for the management of semantic conflict in this manner is shown in Algorithm 3.

mainKernel is invoked when the GPU launches a thread to carry out a block of transactions. Firstly a thread gets a transaction block allocated. Because all transactions one thread needs to execute are sequential in instruction table, a thread can use its thread index multiply transaction number each thread needs to execute to get the start instruction index in instruction table (line 1), and calculate the finish instruction index by calculating the start point of next thread (line 2). Then it tries to execute all transactions in that block sequentially. When a thread reads a transaction from the allocated block of the global transactions table, it checks whether this transaction has already been committed. If this transaction has not been committed, it will call the *doTransaction* function to execute it. If a transaction is successfully committed, this thread will set the finish flag to true in the global transaction table.

When a thread encounters a concurrent conflict and cannot complete because of lower priority, the thread will try to

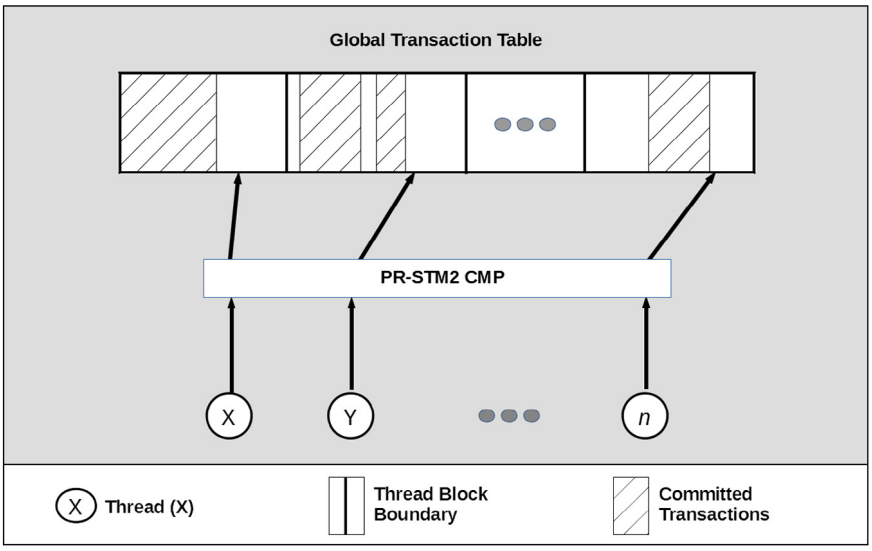


Fig. 6. PR-STM maintains a global transaction table to provide GPU threads with transactions to execute.

ALGORITHM 3: Pseudocode for mechanisms to deal with semantic conflict

```
   function mainKernel()
1      startIns ← threadIdx ∗ transEachThread;
2      finishIns ← (threadIdx + 1) ∗ transEachThread;
3      while startIns < finishIns do
4          anyNotFinish ← false;
5          for i ← startIns to finishIns do
6              if g_insTable[i].finish = false then
7                  if doTransaction() = succeed then
8                      g_insTable[i].finish ← true;
9                  if doTransaction() = concurrentConflict then
10                     doTransaction();
11                 if doTransaction() = semanticConflict then
12                     anyNotFinish = true;
13             if anyNotFinish = false then
14                 startIns ← i + 1;
   function doTransaction()
15     if txRead() = false then
16         return concurrentConflict;
17     if semanticConflictDetect() = true then
18         return semanticConflict;
19     if txWrite() = false then
20         return concurrentConflict;
21     if txValidate() = true then
22         txCommit();
23         return succeed;
       else
24         return concunrrentConflict;
```

execute the transaction again. If a semantic conflict is encountered the thread will skip the transaction and begin the next available transaction while setting a variable to indicate that a previous transaction has been skipped. The thread is finished when all transactions in the allocated transaction block have been committed.

doTransaction is called after a thread reads a transaction from the global transaction table. There can be different types of transaction (e.g. deposit, transfer or withdraw) defined by the transaction type read from the global transaction table. If the executing transaction is not read-only, a thread has to check whether it encounters a semantic conflict (e.g. a withdraw transaction occurring before a required deposit transaction). If so, a *semanticConflict* flag is returned. If a concurrent conflict is encountered the thread will abort with a *concurrentConflict* flag, otherwise, it will commit updates with a *succeed* flag. The semantic conflict result must be determined by the programmer by identifying which situation is a semantic and returning a certain value in the code (e.g. in the bank scenario, the programmer needs to return 2 when insufficient money is in one account), and the system will check whether there is a semantic conflict by detecting return values.

The overhead of this semantic conflict management mechanism is 15% in timely execution. We compared our approach with a previous version [30] which has no mechanism for dealing with semantic conflict.

4. Results and evaluation

To evaluate our system, we compare the performance of PR-STM with GPU-STM [34]. The GPU-STM is the only acceptable related work as it is also a software transactional memory deployment on the GPU which focuses on concurrent conflict resolution. The authors of GPU-STM compared their work to other STMs on the GPU, indicating that GPU-STM performs well (as they do not have the sophistication found in GPU-STM regarding contention management). Therefore, by comparing our work to GPU-STM we identify the appropriateness of our solution in the context of the wider comparisons GPU-STM has already achieved.

Systems on the CPU that can deal with semantic conflict can only launch a few threads and are not comparable to our system. This is primarily because the CPU is a completely different problem in the area of concurrency control as the overhead involved in its resolution is much lower in terms of thread blocking compared to its GPU counterpart: warps must deactivate divergent threads and serialise execution — described more in the earlier background section of this paper. Furthermore, threads may enact quite different execution traces on the CPU and provide a continuous execution rather than batch as witnessed on the GPU. Therefore, comparing CPU contention management with GPU contention management makes little sense.

Evaluation is made in the context of transaction throughput and scalability. We evaluate performance for generalised application usage (semantic transactions present). Only transactions

successfully committed are counted for performance and those retry transactions are not counted. In each case we compare results for instruction sets which have and have not been pre-ordered to avoid semantic conflict. We then present results which evaluate the performance of each CMP as the ratio of semantic transactions is increased from 0% to 100%.

Five scenarios are used to benchmark our technique against GPU-STM to demonstrate that PR-STM is a generalised solution. The benchmarks *Bank* (a simple benchmark provided as part of *tinySTM* [11]), *Vacation and Kmeans (from Stanford Transactional Applications for Multi-Processing (STAMP) [21]), SkipList* [18] and *Graph Processing* are used. *Bank* is a relatively straightforward scenario showing performance in a simple application. *Vacation* involves a much higher conflict rate due to the fixed number of room types accessed by a high number of threads. *SkipList* and *Graph Processing* are commonly used benchmarks for parallel computing system performance with complex data structures. *Kmeans* is a widely used algorithm in clustering.

For each benchmark scenario, results are presented as three parameters are varied:

- The number of threads. Increases in the number of threads utilised increases the chances of conflict as threads are more likely to compete for the shared resource. The default number of threads used in the experiments is 6720.
- The lock coverage. Each lock covers a greater amount of shared resource; the potential for conflict increases but there is a reduction in the memory requirement. The default hash value used in the experiments is 1. Both GPU-STM and PR-STM use a hash number approach to explicitly control how many memory addresses share one lock. With a higher hash number, a greater contention rate can be expected.
- The ratio of semantic transactions. The performance of each system is assessed as the amount of semantic conflict introduced by the application increases. The default semantic transaction ratio used in the experiments is 100% (representing generalised usage).

For each of these experiments four graphs are presented comparing the throughput of PR-STM and GPU-STM. The graphs show results from experiments where the transactions are either all read/write or 20% read-only. For each of these cases results are shown with and without a stage when the transactions are pre-sorted to avoid semantic conflict (replicating the task of the application programmer).

All experiments are carried out on a server with a CPU (Intel Xeon E5-2630 v3) running at 2.4 GHz. The GPU was a NVidia GeForce GTX 1080 with a clock speed of 1733 MHz, 8 Gb of GDDR5X memory and 20 streaming multiprocessors each of which has 128 CUDA cores. The operating system was Linux. The two CMPs (PR-STM and GPU-STM) were implemented with the CUDA 9.2 runtime library.

The shared data, including global lock table, are allocated in off-chip global memory. In the PR-STM implementation, local metadata is stored in thread-local memory, whereas for GPU-STM, metadata is stored in global memory but the pointers to the data are in local memory [34]. For both implementations, the local metadata is cached at the L1 and L2 levels, and the global data is cached only at the L2 level as the L1 cache is not coherent [30].

4.1. Implementation of bank and vacation benchmark

The *bank* scenario was the first to be used to benchmark the performance of the two systems. *Bank* consists of an array of bank account structures and allows the execution of a number of transaction types on these simulated bank accounts. Each transaction stands for one behaviour operated by a person, they can enact a deposit, withdraw, transfer or a combination of these options.

As many threads are available on a GPU, a sizeable number of accounts were created in shared data for the *bank* scenario. A memory block of 10MB was set aside for the creation of roughly 2.5 million accounts. This allows the observation of both low and high contention as the scenario parameters are varied. The *bank* scenario was adapted to our needs in a number of ways:

- The hashing function used by both CMPs was modified so that the amount of shared data covered by a single lock can be varied. This allows investigation into the amount of false sharing.
- Results are included where the number of threads is increased, to observe the contention caused by the high numbers of threads available on GPU.

The *Vacation* scenario implements a hotel room coordination system which handles the booking and cancelling of certain types of hotel room concurrently. It is commonly used as part of the STAMP benchmarking suite [21]. The *vacation* scenario involves transactions which tend to execute more statements of greater complexity than those in the *bank* scenario [28].

As the *vacation* benchmark is designed for CPU implementation, some adjustments were made to ensure compatibility with GPU operation. The variables available in the *Vacation* scenario for evaluation include the hash number and the number of transactions per thread (TPT). This provides the base environment for the performance testing. The TPT is used to examine the vitality of a thread. The more transactions a thread executes, the longer it is active, which allows us to evaluate the ability a thread has to proceed through different concurrent situations.

In the *Vacation* scenario, semantic conflict occurs when an attempt is made to book a type of room that is sold out. For sequential computing, it is easy to deal with this eventuality — the unsatisfied customer is added to a waiting list, and when a suitable room becomes available the first customer on the waiting list is allocated that room. However, in the context of parallel computing, there is no trivial solution to this issue because the instructions are executed in an arbitrary order and the threads are isolated from each other. One way to handle this issue is to have the threads that want to book an unavailable room listen to the account of the type of room. Once the type of room is available again, all the threads compete for it using some atomic operation, with one being selected to be successful.

PR-STM provides a solution to semantic conflict by running through the instruction table, if a semantic conflict occurs, the instruction is added to an array of instructions which have incurred semantic conflict. After the kernel finishes the whole instruction table, it will continue to execute instructions from the recorded array until all the semantic conflicts are resolved or abandoned [4].

GPU-STM provides no semantic conflict solution. To make benchmarking more scalable for this evaluation, a simple semantic conflict solution has been added to GPU-STM. If conflict occurs, the instruction will not be aborted instantly. Instead it is given a limited number of chances to retry. The number of chances is recommended to be adjusted according to the size of data [34]. As we use a fixed size of shared data, the retry limit for GPU-STM is set to 100 for all benchmarks.

Results are shown for the case when all threads perform update transactions (i.e., read and write operations), and also the case when 20% of threads execute read-only transactions. This approach allows the impact of invisible reads in the scenarios to be analysed. Each test lasted for 5 s, and was executed 10 times with the average results presented.

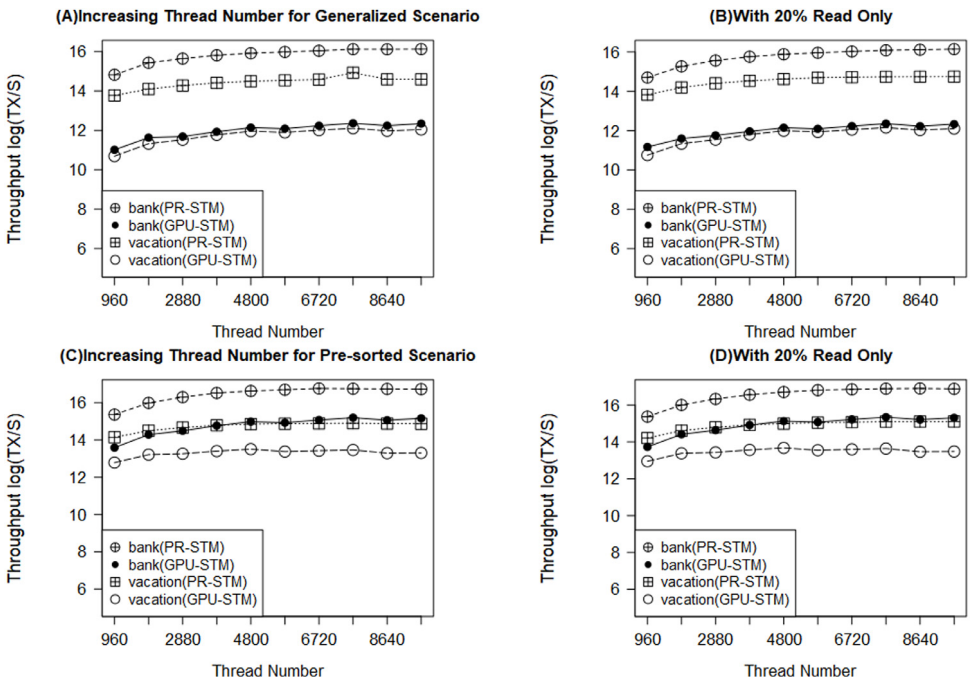


Fig. 7. Bank and vacation scenario average throughput with increasing number of threads. Graphs A and C have all read/write transactions, graphs B and D have 20% read-only transactions. Graphs A and B have no pre-sorting of transactions to avoid semantic conflict, whereas graphs C and D do.

4.2. Performance of bank and vacation benchmark

The graphs in Fig. 7 show the transaction throughput achieved when the number of threads available for resolution is increased. In each case the performance of PR-STM is compared to that of GPU-STM on the GPU. Graphs A and B show the results for the generalised application (i.e. 100% semantic transactions with no pre-ordering). As GPU-STM is not designed to handle semantic conflicts it performs poorly in these circumstances (the y axes are presented as logarithmic to better distinguish the performance of GPU-STM as the thread count increases). Graphs C and D show results where the application's instructions have been pre-ordered to avoid semantic conflict, so that the transactions are more suited to GPU-STM, allowing us to assess both CMPs in a less generalised context.

The number of threads available was varied between 960 and 9600. The throughputs achieved by PR-STM and GPU-STM for the cases when all threads perform read and write transactions, and when 20% of threads perform read-only transactions respectively are presented. In all cases PR-STM outperforms GPU-STM. The introduction of 20% read-only threads causes only a minimal change in throughput in both cases, so it appears that read-only transactions have little effect on GPU performance.

The *vacation* benchmark scenario utilised consists of 150,000 customers attempting to book rooms of 1000 different room types (with 150 rooms of each type available). For a booking transaction, a random customer is allocated a random room type to attempt a booking. For a cancellation transaction, an already-booked customer is selected at random to cancel a room of the selected room type. The choice of hash number in each algorithm gives a certain number of consecutive data addresses only one lock, so it is necessary to make the room type account non-consecutive. Otherwise, the processing will simply occur sequentially. Therefore, we set the interval between each room type account to 100.

Increasing the number of threads means there is an increase in the available computing resource. Our results show an increase in throughput which is a little less than linear. This is because as more threads work in parallel, the conflict rate also increases (as the total shared resource is steady). The scalability of PR-STM is demonstrated by the increase in throughput, compared to GPU-STM across all thread numbers. An important aspect of a GPU based solution is it can launch a much greater number of threads than a CPU solution, so scalability is of interest. Even in the Pre-sorted scenarios, PR-STM still perform better than GPU-STM because GPU-STM has to execute a search and insert operation in every read and write function, which is costly because each time this occurs all threads in the same must be deactivated and wait for the slowest one. On the other hand, PR-STM only performs the divergence pre-lock operation in the commit session once each transaction and consume much less time than GPU-STM.

Tests were also carried out with a modified hash function, which determines the number of accounts that can be covered by a single lock. The lower a hash value, the less chance of a thread trying to access the same lock when reading or writing to different shared data. The number of threads remained at 6720. The results are shown in Fig. 8 for the cases when all threads perform read and write transactions, and when 20% of threads perform read-only transactions respectively. The y-axes show the number of transactions per second as logarithmic, and the x-axes show the hash function value, which is the number of accounts covered by a single lock.

A decrease in throughput is seen in both GPU-STM and PR-STM as the number of accounts per lock is increased. The performance of both CMPs decreases because of increased false conflict, but reduced lock querying counters this somewhat (as both CMPs use lock-sets), as does reduced bus traffic when querying the status of those locks (due to coalescence of memory). When 20% of the threads are read-only, throughput is only slightly improved for both CMPs.

4.3. Bank and vacation semantic transactions

We now consider performance under varying levels of semantic conflict to assess the performance of our algorithm in comparison to that of GPU-STM in situations where transactional

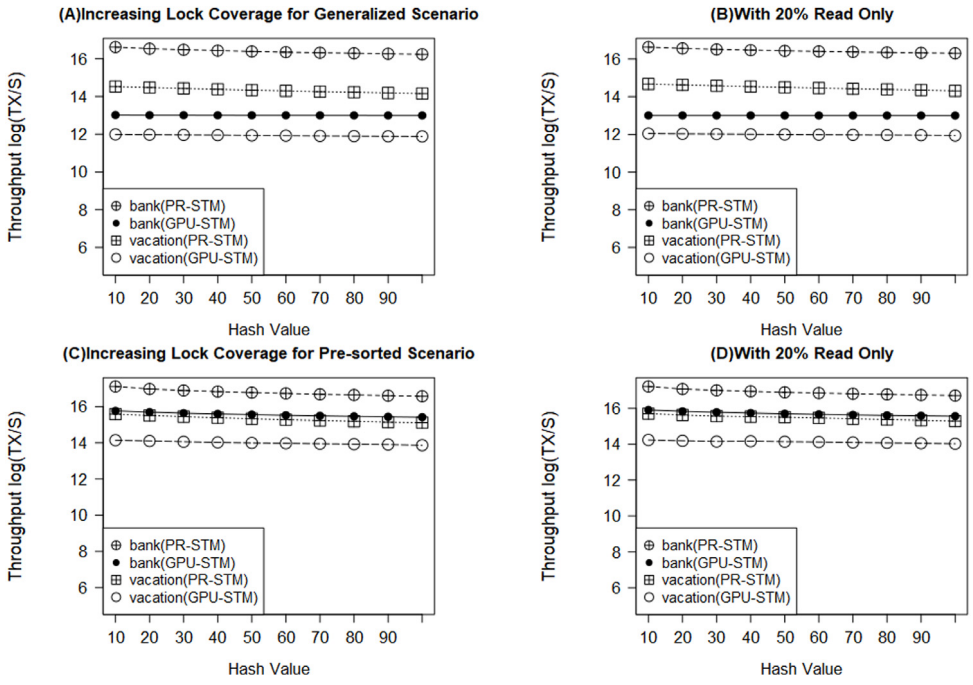


Fig. 8. Bank and Vacation scenario average throughput with increasing lock coverage. Graphs A and C have all read/write transactions, graphs B and D have 20% read-only transactions. Graphs A and B have no pre-sorting of transactions to avoid semantic conflict, whereas graphs C and D do.

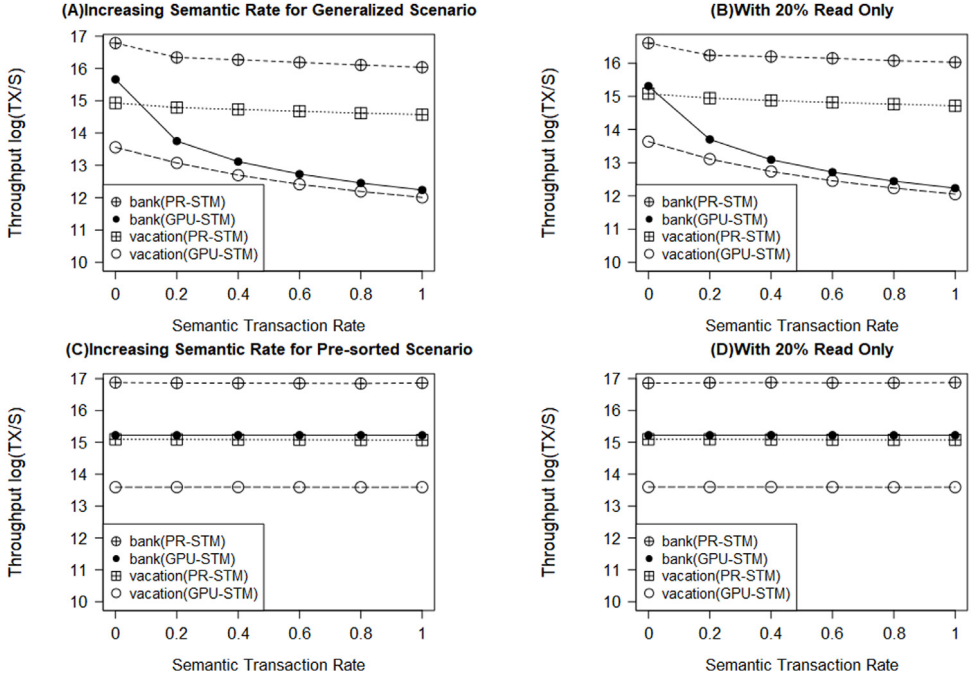


Fig. 9. Bank and vacation scenario average throughput with increasing ratio of semantic transactions. Graphs A and B show results when the application has pre-ordered transactions to avoid semantic conflict, graphs C and D show results when that application-level ordering has not occurred.

conflict is caused by the semantics of the application. Fig. 9 shows the rate of transaction throughput as the semantic transaction ratio is increased from 0% to 100% for two situations. In the first the application has not pre-ordered transactions to avoid semantic conflict, whereas in the second that ordering has occurred. The number of threads used was kept constant at 6720 and the hash number was 1 (i.e. every account gets a lock to avoid false conflict). In each graph, Y-axes show the number of transactions committed per second as logarithmic and X-axes show the percentage of semantic transactions in the transaction table.

Fig. 9 C and D show results after the transaction table has been sorted to avoid semantic conflict. This allows us to compare performance of GPU-STM and PR-STM in a situation which both are expected to handle — the onus on sorting the transactions is left to the application programmer. For example, a semantic conflict could be resolved by ordering a withdrawal transaction shortly after a deposit transaction, thus allowing the

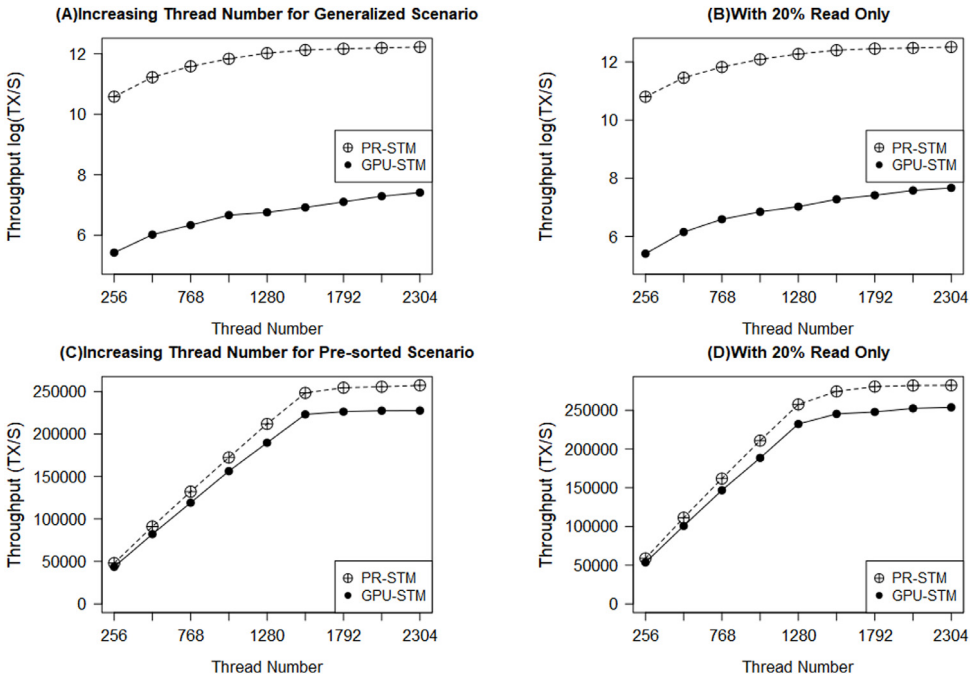


Fig. 10. Throughput of skiplist scenario with increasing number of threads. Graphs A and C have all read/write transactions, graphs B and D have 20% read-only transactions. Graphs A and B have no pre-sorting of transactions to avoid semantic conflict, whereas graphs C and D do.

withdrawal to succeed with minimum retries. PR-STM produced higher throughput than GPU-STM even when the transactions were sorted. As the semantic conflict rate was increased GPU-STM decreased in throughput very slightly while PR-STM began to improve. This is because PR-STM can benefit from temporarily abandoning transactions with semantic conflicts and searching for new transactions, causing fewer conflicts overall.

The graphs in Fig. 9 A and B show the results with no sorting of the transaction table (i.e., for a generalised situation). In this case, GPU-STM cannot deal with semantic transactions — the simulation aborts after 100 failed attempts. Introduction of semantic conflicts representing as little as 5% of the overall transactions shows a very rapid decline in performance. Note that the y-axes are logarithmic to better illustrate the relative performance of GPU-STM as the ratio of semantic conflict increases. PR-STM, however, copes well with the increased ratio of semantic conflict with little degradation after the initial drop-off.

While there is an increased cost in PR-STM's ability to handle semantic conflict, the fact that it can handle any form of conflict makes it an appealing solution. The requirement for the application programmer to correctly order transactions to provide no semantic conflict is removed, and a generalised solution is provided.

4.4. Implementation of skiplist benchmark

A skip-list is, effectively, a hierarchical linked-list. The *skiplist* benchmark is commonly used to assess contention resolution in transactional memory. In our implementation of the benchmark we use an initial array of 5000 entries, with sufficient memory allocated to expand this to 10 million entries (so that there are no allocation issues). The maximum number of hierarchical levels is set to 5. Three types of transactions are invoked by threads at random - a thread may insert a new element into the skip-list, delete an element, or simply search an element (i.e., a read-only transaction).

The use of *skiplist* as a benchmark allows us to further assess the scalability of the CMP. The hierarchical nature of the skip-list means that, in order to complete an insert or delete transaction, multiple nodes must be read from and written to. A higher rate of lock contention can therefore be reached than with the other benchmarks that have been considered [22].

4.5. Performance of skiplist benchmark

The concept of semantic transactions does not apply to the *skiplist* benchmark, as the application cannot attempt to insert a duplicate entry or remove a non-existent one. Evaluation was therefore limited to increasing the threads count and the hash number, allowing an assessment of PR-STM compared to GPU-STM for an application to which both are suited.

Fig. 10 shows the throughput achieved by both CMPs in the *skiplist* benchmark as the number of threads is increased. The hash number was set to the default value of 1 for these experiments. As there is no semantic conflict, the performance of GPU-STM is improved in comparison to that achieved in the *bank* and *vacation* scenarios. However PR-STM still achieves higher throughput in all cases. Again the introduction of 20% read-only transactions has a minimal effect on throughput for both CMPs.

The results from evaluating performance as the hash number is increased in the *skiplist* benchmark are shown in Fig. 11. In these experiments the thread number was set at 1536. Once again PR-STM is outperforming GPU-STM in all cases as PR-STM does not overly rely on global memory for evaluating the outcomes this execution domain.

4.6. Implementation of K-means benchmark

The *kmeans* benchmark groups objects into K clusters. In our implementation, we try to partition around 1 million points (which consist of two integers indicating a 2D position) into 16 clusters, and the maximum iteration time is 5. Two types of operations are assigned to threads in different stages: assignment stage and update stage. In the assignment stage, each thread is allotted a certain number of points and calculates which cluster each should belong to. In the update stage, threads are invoked

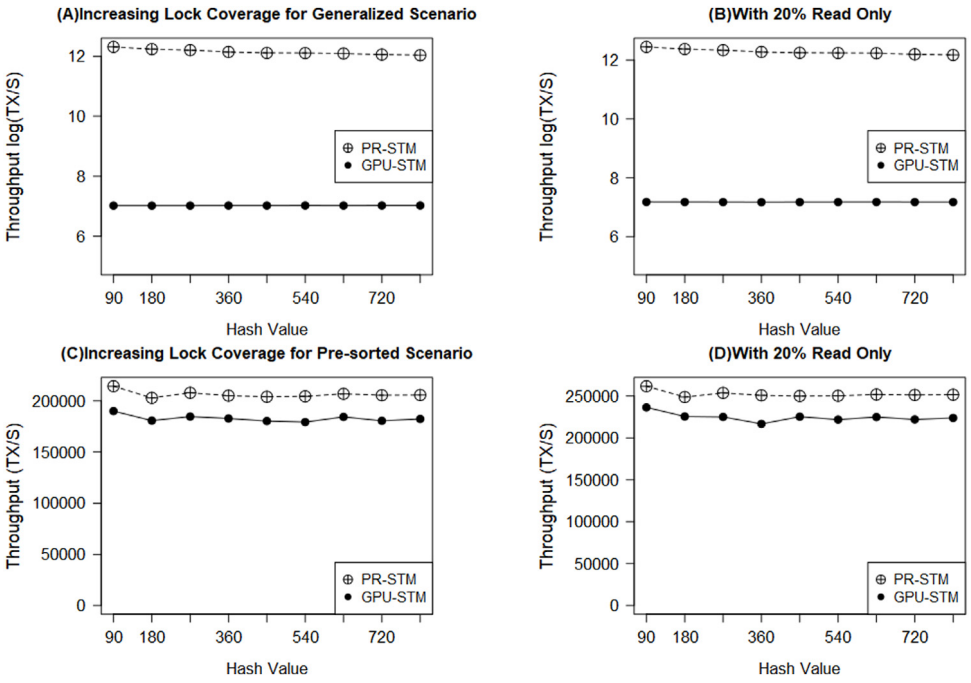


Fig. 11. Throughput of skiplist scenario with increasing hash number. Graphs A and C have all read/write transactions, graphs B and D have 20% read-only transactions. Graphs A and B have no pre-sorting of transactions to avoid semantic conflict, whereas graphs C and D do.

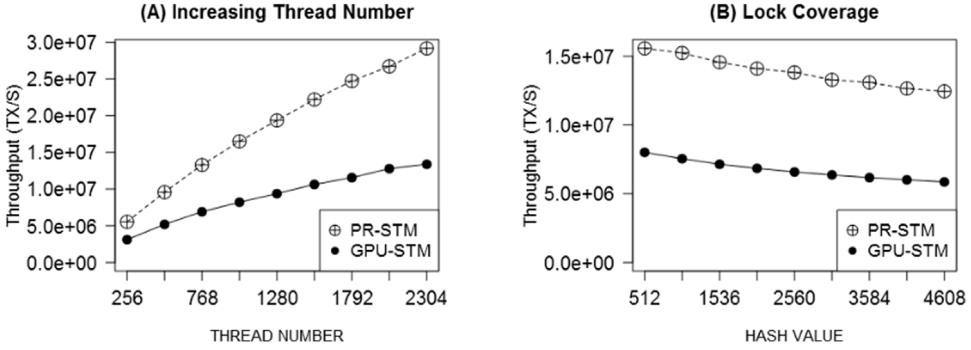


Fig. 12. Throughput of kmeans scenario with increasing threads number and hash number.

to calculate the new centroids of clusters. In practice, the first stage transactions are not required as there is no conflict, so transactions are only invoked in the second stage. In the update stage, different threads that try to update the same cluster leads leading to conflict.

The *kmeans* benchmark allows us to discover how the CMP works when the read conflict rate is much higher than the write conflict rate. Moreover, the update stage takes much more time than the assignment stage because conflicts only arise in this stage.

4.7. Performance of K-means benchmark

The concept of semantic transactions does not apply significantly to the *kmeans* benchmark, as the initial clusters are selected randomly and all integers can be assigned to a calculated cluster. However, as there is no read-only thread, the evaluation is limited to manipulating the threads number and hash number, in order to assess the efficiency and scalability of both systems.

Fig. 12 shows the throughput achieved by both CMPs in the *kmeans* benchmark when the number of threads and hash number is increased. The hash number was set to default value of 1 for Graph A. The performance of GPU-STM improved less than PR-STM because the algorithm it employs modifies the version even with read operations. In experiments of manipulating hash number, the thread number was set at 6720. Once again PR-STM is outperforming GPU-STM in all cases due to the same reason outlined earlier.

4.8. Implementation of graph processing benchmark

A compressed sparse row (CSR) graph is implemented on the GPU. Values in 3 arrays represent linked vertexes count, target vertexes' IDs and values of vertexes respectively. In this format the graph cannot add or remove vertexes nor edges, but the values can change. We initialise a graph with 1 million vertexes and 10 million edges. The value of vertexes is random from 1 to 10 000. Threads are launched allowing different vertexes to calculate the minimal value of all its predecessor vertexes. Each transaction sends the value of a source vertex to all its neighbours, and updates those vertexes if their value is greater than the source vertex. Only vertexes updated will send their values again, and the transaction will terminate when no further vertexes need to update. Conflict happens when a vertex value needs to be modified by multiple threads.

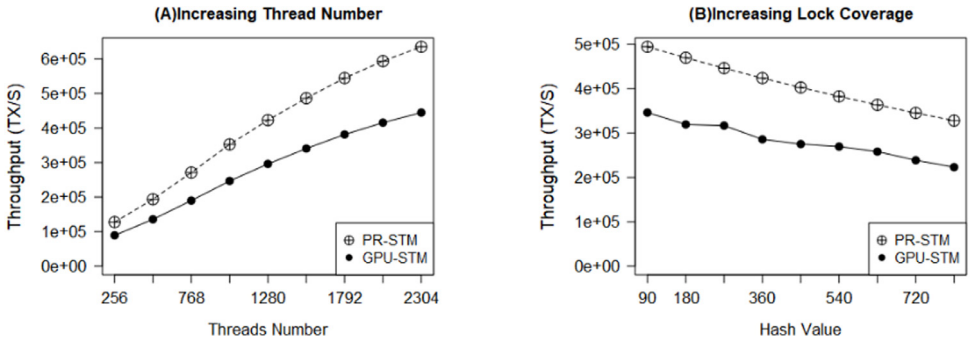


Fig. 13. Throughput of graph processing benchmark with increasing thread numbers and hash number.

4.9. Performance of graph processing benchmark

The concept of semantic transactions does not apply to the *graph processing* benchmark because if all neighbour vertexes have smaller values than a vertex, then that transaction can terminate. There is no possibility to generate read-only threads because read-only transaction only occur when all neighbour vertexes have smaller value than the start vertex, which requires pre-knowledge of the graph. So the evaluation is again limited to manipulating the thread number and hash number, in order to assess the efficiency and scalability of both systems.

Fig. 13 shows the throughput achieved by both CMPs in the *graph processing* benchmark when the number of threads and hash number is increased. The hash number was set to 1 for Graph A and B as before while the thread number was set to 1536 in experiments of manipulating hash number. The throughput drops more sharply than previous benchmarks when hash number increasing is because the false conflict rate is much higher in this scenario. If several threads are updating adjacent vertexes which are covered by the same lock, then every step may cause a false conflict.

5. Conclusions

A contention management policy for GPUs (PR-STM) has been introduced which utilises increased parallelism to explore transactional ordering solutions in a scalable manner to remove inter-thread contention. Our technique advocates a priority rule-based approach that increases thread throughput, improves application timeliness and removes the requirement for manual intervention of the programmer. We show that our approach is general purpose in the context of a variety of application domains using industry acknowledged benchmarks. We compare our solution to the only other similar comparator to demonstrate how the handling of semantic conflict can improve overall resource usage while improving timeliness of execution.

A traditional CMP (on the CPU or GPU) may reach a state where transactions are completed when considering concurrent conflict alone, but logical progression of the application does not occur (e.g., ordering withdrawal before deposit in banking scenario). This task is traditionally left to the programmer to enact coordination techniques, difficult to achieve in a pre-emptive transactional environment. However, our approach demonstrates that we can achieve this on the CPU and with this latest work achieve this on the GPU. The GPU, due to its hardware configuration and execution environment, is more reliant on avoiding semantic conflict as we show they would have a much more damaging impact on performance and resource utilisation. Our approach pioneers the automation of the resolution of semantic conflicts in co-ordination with transactional conflict, providing the first practical approach for handling software engineering issues in semantic conflict on the GPU (a complete solution for the programmer).

Due to the nature of how memory is utilised in our algorithms (local/global) we also perform better than our comparator in all benchmarks. Identifying that for those applications that do not consider semantic conflict an issue we still provide the most competitive solution.

Our priority rule approach has much scope for further development, in particular the introduction of dynamic priority settings that may configure themselves to react to the nature of execution may yield performance gains. One way to approach this problem is to consider AI influenced CMP for determining optimisation of CMP parametrisation during runtime.

In the future we expect session locking mechanisms within a distributed STM application to open some interesting possibilities in further scalability gains across cloud infrastructures (as we only consider single GPU deployments). In addition, combining the GPU and CPU within a heterogeneous transaction manager will be of greater interest as we expect such environments to be more pronounced in the future. We should be able to formulate a transaction allocation strategy which assigns thread exploration of transaction schedules to CPU or GPU when appropriate.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.jpdc.2019.12.018.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful feedback. This work is partly supported by the National Key R&D Program of China under Grant 2016YFB1000103.

References

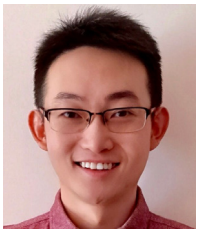
[1] Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris Kirkham, Ian Watson, Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering, in: High Performance Embedded Architectures and Compilers, Springer, 2009, pp. 4–18.

[2] Tongxin Bai, Xipeng Shen, Chengliang Zhang, William N. Scherer, Chen Ding, Michael L. Scott, A key-based adaptive transactional memory executor, in: Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, IEEE, 2007, pp. 1–8.

[3] Joao Barreto, Aleksandar Dragojevic, Paulo Ferreira, Ricardo Filipe, Rachid Guerraoui, Unifying thread-level speculation and transactional memory, in: Proceedings of the 13th International Middleware Conference, Springer-Verlag New York, Inc., 2012, pp. 187–207.

[4] Colin Blundell, E. Christopher Lewis, Milo M.K. Martin, Subtleties of transactional memory atomicity semantics, Comput. Arch. Lett. 5 (2) (2006).

[5] Daniel Cederman, Philippas Tsigas, Muhammad Tayyab Chaudhry, Towards a software transactional memory for graphics processors, in: EGPGV, 2010, pp. 121–129.
[6] S Chen, L Peng, Efficient gpu hardware transactional memory through early conflict resolution, in: Proceedings of 22nd International Symposium on High Performance Computing Architecture, ACM, 2016, pp. 274–284.
[7] Phong Chuong, Faith Ellen, Vijaya Ramachandran, A universal construction for wait-free transaction friendly data structures, in: Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures, ACM, 2010, pp. 335–344.
[8] Tyler Crain, Damien Imbs, Michel Raynal, Towards a universal construction for transaction-based multiprocess programs, in: Distributed Computing and Networking, Springer, 2012, pp. 61–75.
[9] E.W. Dijkstra, Solution of a problem in concurrent programming control, Commun. ACM 8 (9) (1965).
[10] Shlomi Dolev, Danny Hendler, Adi Suissa, Car-stm: scheduling-based collision avoidance and resolution for software transactional memory, in: Proceedings of the Twenty-Seventh ACM Symposium on Principles of Distributed Computing, ACM, 2008, pp. 125–134.
[11] Pascal Felber, Christof Fetzer, Torvald Riegel, Dynamic performance tuning of word-based software transactional memory, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2008, pp. 237–246.
[12] Wilson W.L. Fung, Tor M. Aamodt, Energy efficient gpu transactional memory via space-time optimizations, in: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2013, pp. 408–420.
[13] Wilson WL Fung, Inderpreet Singh, Andrew Brownsword, Tor M Aamodt, Hardware transactional memory for gpu architectures, in: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2011, pp. 296–307.
[14] Rachid Guerraoui, Maurice Herlihy, Bastian Pochon, Toward a theory of transactional contention managers, in: Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, ACM, 2005, pp. 258–264.
[15] T. Haerder, A. Reuter, Principles of transaction-oriented database recovery, in: ACM Computing Surveys, ACM, 1983.
[16] Tomer Heber, Danny Hendler, Adi Suissa, On the impact of serializing contention management on stm performance, J. Parallel Distrib. Comput. 72 (6) (2012) 739–750.
[17] Maurice Herlihy, Wait-free synchronization, ACM Trans. Program. Lang. Syst. (TOPLAS) 13 (1) (1991) 124–149.
[18] Maurice Herlihy, Victor Luchangco, Mark Moir, A flexible framework for implementing software transactional memory, in: ACM Sigplan Notices, Vol. 41, ACM, 2006, pp. 253–262.
[19] Anup Holey, Antonia Zhai, Lightweight software transactions on gpus, in: 2014 43rd International Conference on Parallel Processing (ICPP), IEEE, 2014, pp. 461–470.
[20] H.T. Kung, T. Robinson John, On optimistic methods for concurrency control, ACM Trans. Database Syst. (1981) 213–226.
[21] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, Kunle Olukotun, Stamp: Stanford transactional applications for multi-processing, in: IEEE International Symposium on Workload Characterization, 2008. IISWC 2008, IEEE, 2008, pp. 35–46.
[22] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, Kunle Olukotun, An effective hybrid transactional memory system with strong isolation guarantees, in: ACM SIGARCH Computer Architecture News, Vol. 35, ACM, 2007, pp. 69–80.
[23] Rupesh Nasre, Martin Burtscher, Keshav Pingali, Atomic-free irregular computations on gpus, in: Proceedings of the 6th Workshop on General Purpose Processor using Graphics Processing Units, ACM, 2013, pp. 96–107.
[24] Rupesh Nasre, Martin Burtscher, Keshav Pingali, Morph algorithms on gpus, in: ACM SIGPLAN Notices, Vol. 48, ACM, 2013, pp. 147–156.
[25] C. Nvida, Programming Guide, Nvida, 2008.
[26] Jason Sanders, Edward Kandrot, CUDA by Example: an Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010.
[27] William N. Scherer I.I.I., Michael L. Scott, Advanced contention management for dynamic software transactional memory, in: Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, ACM, 2005, pp. 240–248.
[28] Craig Sharp, William Blewitt, Graham Morgan, Resolving semantic conflicts in word based software transactional memory, in: Euro-Par 2014 Parallel Processing, Springer, 2014, pp. 463–474.
[29] Craig Sharp, Graham Morgan, Hugh: a semantically aware universal construction for transactional memory systems, in: Euro-Par 2013 Parallel Processing, Springer, 2013, pp. 470–481.
[30] Qi Shen, Craig Sharp, William Blewitt, Gary Ushaw, Graham Morgan, Prstm: Priority rule based software transactions for gpu, in: Euro-Par 2015 Parallel Processing, Springer, 2015.
[31] Alejandro Villegas, Rafael Asenjo, Angeles Navarro, Oscar Plata, David Kaeli, Lightweight hardware transactional memory for gpu scratchpad memory, IEEE Trans. Comput. 67 (6) (2018) 816–829.
[32] Alejandro Villegas, Rafael Asenjo, Angeles Navarro, Oscar Plata, Rafael Ubal, David Kaeli, Hardware support for scratchpad memory transactions on gpu architectures, in: European Conference on Parallel Processing, Springer, 2017, pp. 273–286.
[33] Jons-Tobias Wamhoff, Christof Fetzer, The Universal Transactional Memory Construction, Tech Report, University of Dresden, Germany, 2010, p. 12.
[34] Yunlong Xu, Rui Wang, Nilanjan Goswami, Tao Li, Lan Gao, Depei Qian, Software transactional memory for gpu architectures, in: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, ACM, 2014, p. 1.



Qi Shen is a postdoctoral researcher in Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, where he is investigating data processing systems for geographically distributed clusters. He received his Ph.D. degree in 2017 from The University of Newcastle Upon Tyne for work on semantically aware transactional concurrency control for GPGPU computing. His research interests include parallel programming, software transactional memory, and distributed architectures.

Craig Sharp is currently working for Newcastle University as a Data Scientist, specialising in the field of Convolutional Neural Networks. Craig previously worked in the Games Lab at Newcastle University (2013–2018) and was the lead developer in several projects implementing Gamification of Distributed Healthcare Diagnosis and Recovery, particularly in the area of 3D vision and stroke rehabilitation. Craig was awarded his P.h.D. in Computing Science in 2013 from Newcastle University, in the area of applied Contention Management in Transactional Memory applications.

Richard Davison is a lecturer in video game technology at Newcastle University, where he investigates uses of virtual reality in training scenarios, and in GPU accelerated data processing. He received his Ph.D. in 2016 from Newcastle University for work on utilising commodity video game hardware to detect rehabilitation markers in software designed for post-stroke physiotherapy. His interests include real time computer graphics, GPGPU technology, and medical gamification.

Gary Ushaw is Director of Business and Engagement for the School of Computing at Newcastle University. He is a senior lecturer in the Networked and Ubiquitous Systems Engineering group, with a particular interest in video game engineering and real-time simulation techniques. He attained his Ph.D. in 1996 with the Signal Processing Group at the University of Edinburgh. Gary was the engineering lead on multiple high profile projects for the computer games industry.

Rajiv Ranjan has been able to establish an international reputation as a leader in the field of Scalable Computing and in particular Cloud Computing and Big Data Analytics. For over 10 years, he has conducted seminal research around the development of generic resource management models and methods for efficient scheduling and resource allocation for all types of parallel and distributed computing systems such as grid computing and cloud computing. He also easily extended his methods to new computing trends we see in the industry such as Internet of Things (IoT) and Edge Computing. As a prolific researcher and as of November 2016, he has published 180 scientific publications that include 113 journal articles, 47 conference papers, 12 book chapters, and 8 edited research books. His research has received excellent recognition from the research community as evidenced by the citations, Google scholar h-index 33, i-10 index 66, 6700+ citations (November 2016).

Albert Y. Zomaya is currently the Chair Professor of High-Performance Computing and Networking and Australian Research Council Professorial Fellow in the School of Information Technologies, The University of Sydney. He is also the Director of the Centre for Distributed and High-Performance Computing which was established in late 2009. Professor Zomaya is the author/co-author of seven books, more than 450 publications in technical journals and conferences, and the editor of 14 books and 19 conference volumes. He is the Editor in Chief of the IEEE Transactions on Computers and Springer's Scalable Computing Journal serves as an associate editor for another 19 journals including some of the leading journals in the field.

Graham Morgan is the director of Networked and Ubiquitous Systems Engineering (NUSE) Group, jointly with Professor Rajiv Ranjan, in the School of Computing. Graham also created and leads the Game Technology Lab. The Game Technology Lab is a research and teaching laboratory that works with the video games industry on optimised resource management, streamed/networked gaming and real-time graphical simulations. Members of the lab regularly work on many of the top selling global video games. In addition to commercial activity, he has led research in a wide area of distributed systems topics, including the development of large scale real-time gaming technologies for cloud infrastructures and applied such work in the area of digital health for stroke rehabilitation and cognitive therapies. His work has won best paper awards in leading IEEE and ACM conferences and he has published in leading IEEE and ACM journals.