Eurographics Symposium on Parallel Graphics and Visualization (2010)
J. Ahrens, K. Debattista, and R. Pajarola (Editors)

Towards a Software Transactional Memory for Graphics Processors

Daniel Cederman, Philippas Tsigas and Muhammad Tayyab Chaudhry

Department of Computer Science and Engineering
Chalmers University of Technology, Göteborg, Sweden

Abstract

The introduction of general purpose computing on many-core graphics processor systems, and the general shift in the industry towards parallelism, has created a demand for ease of parallelization. Software transactional memory (STM) simplifies development of concurrent code by allowing the programmer to mark sections of code to be executed concurrently and atomically in an optimistic manner. In contrast to locks, STMs are easy to compose and do not suffer from deadlocks. We have designed and implemented two STMs for graphics processors, one blocking and one non-blocking. The design issues involved in the designing of these two STMs are described and explained in the paper together with experimental results comparing the performance of the two STMs.

Categories and Subject Descriptors (according to ACM CCS): D.1.3 [Software]: Programming Techniques—Concurrent Programming

1. Introduction

Computer processor research has previously been focused on increasing the clock speed, but as of late the trend has shifted towards increasing the number of processors instead. This has led to increased pressure for applications to become multi-threaded to take full advantage of the new computing power. But with increased parallelism comes the problem of efficient synchronization. Threads that concurrently access shared memory have to synchronize in order to maintain a non-corrupted view of the data.

The traditional way of synchronizing memory accesses has been to use mutual exclusion, using locks to only allow one process to access shared memory areas at any given time. However, this kind of lock-based synchronization makes it hard to compose function calls and leads to problems such as deadlocks, where two processes are both waiting for the other to give up a lock, and convoying, where a process that holds a lock gets swapped out causing other processes to wait unnecessarily long to acquire that lock.

Transactional memory (TM) provides an alternative concurrency control that can eliminate these problems or at least minimize them. A TM allows the programmer to mark a section of the code that is to run atomically, i.e., it should appear to take place instantly. The TM logs all read and write operations in the code block and only store the new data if there was no conflict with another process. If a transaction notices that another transaction has written to memory read in the transaction, the transaction will be restarted. The lack of commonly available hardware transactional memories has led to most implementations of transactional memory being completely software based, so-called Software Transactional Memories (STM).

An STM tries to automatically offer some degree of parallelism to the application without having the programmers concentrate on the mechanism of synchronization, as this is taken care of by the STM itself. There is, however, a trade-off when it comes to performance. Code written using STMs often has difficulties competing in performance compared to solutions that are highly optimized by hand. Caşcaval et al. argue that the overhead introduced by STMs might be hard to overcome [CBM*08].

The high bandwidth and many-core design of current graphics processors have caused a big interest in applying them for general purpose computing. With APIs such as CUDA and OpenCL it is only a matter of time before they become standard auxiliary processing units that most pro-

grammers would like to take advantage of in their applications.

In this paper we examine if an introduction of STMs to graphics processors could help simplify the relatively complex amount of synchronization needed when running programs on a many-core platform. More specifically, we evaluate two STM designs, with two different types of progress guarantees, in an effort to better understand the specific challenges involved providing synchronization for many-core graphics processor systems.

2. Related work

Transactional memory was originally intended for hardware and was first introduced in a paper by Knight and a paper by Herlihy and Moss [Kni86, HM93]. Shavit and Touitou then later introduced the concept of a pure software transactional memory [ST95]. Their STM required the programmer to specify beforehand which memory locations to access and could not adapt to values read in the transaction.

This was changed in Herlihy et al.'s dynamic STM (DSTM) where they also introduced the concept of a contention manager that should decide which transaction to abort [HLMS03]. Harris and Fraser have presented an STM that works on the word level as opposed to the object level, called WSTM [FH07].

The previously mentioned STMs perform their operations on local copies of the objects or words which are then either discarded or written back, but it is also possible to take a more optimistic approach and write directly to the objects or words [ATLM*06, HPST06, SATH*06]. This, however, requires the STM to store the original values so that the changes can be undone if there is a conflict and also introduces the problem of visibility – should other transactions be able see the values written?

There has also been work on creating hybrid transactional memories that use both hardware and software [DFL*06, KCJ*06]. If the hardware has support for transactional memory, the transaction is started in hardware and handled there until it gets too large for the hardware to support. In those cases the transaction is taken over by the STM.

Ennals argues that STMs should be blocking as the advantages of doing a non-blocking implementation are small and prevents several optimizations that can be done in a blocking system [Enn06].

There is a large amount of literature on designing STMs [DSS06, HF03, Moi97] and for a good overview we recommend the *Transactional Memory* paper by Larus and Kozyrakis [LK08].

3. System Model

CUDA was introduced by NVIDIA as a general purpose parallel computing architecture, making it possible to execute computationally complex problems on NVIDIA's graphics processors. The software part of CUDA provides a compiler for a language based on C, but with extensions that allow functions to be executed on the graphics processor instead of on the CPU.

```
atomic {
    ltail = read(tail);
    write(queue[ltail], value);
    ltail++;
}
```

Figure 1: *Example use of a software transactional memory.*

A CUDA compatible graphics processor consists of several so called multiprocessors, each of which can execute SIMD instructions on eight memory locations at a time. Threads are scheduled on the multiprocessors in groups called thread blocks. All threads in a thread block remain at the same multiprocessor until they have finished executing and can use the processor's extremely fast local memory, called the shared memory, to communicate with each other. It is up to the programmer to decide how many threads should be in each block and how many blocks to start in total. Depending on how many threads there are in a block, one or more blocks could run on the same multiprocessor.

Threads of the same as well as different blocks can perform read and write operations on the main graphics memory known as the global memory. There is no cache support when using the global memory, but if threads with consecutive thread id's are accessing consecutive memory locations, the memory accesses could be coalesced by the hardware to dramatically speed up the reading and writing speed to the memory. There are also texture and constant memory that have cache support, but they are read-only.

Newer versions of CUDA-compatible graphics processors also support atomic primitives, such as Compare-And-Swap, which can be used, for example, to implement locks or more advanced lock-free data-structures [Her88, PDC09]. Cederman and Tsigas took advantage of this to compare blocking and non-blocking dynamic load balancing schemes on graphics processors [CT08].

4. STM Design

On the user level, a basic software transactional memory needs to support, either directly or indirectly, four operations. A *begin* operation that marks the start of a transaction, a *read* operation that provides a snapshot of a memory location, a *write* operation that logs the updates to the memory that should be performed if the transaction is successful, and finally a *commit* operation that performs the writes if no other processes have touched the memory read by the transaction and restarts the transaction otherwise. The begin and

commit operations are often performed indirectly, as in Figure 1, where they are part of the *atomic* keyword.

However, when deciding how to implement the functionality behind these operations, there are several important design decisions that have to be made. There is a vast design space for STMs and as of yet there is no definitive way to design an STM. One major divisional line is that which progress guarantees to provide. More basic guarantees can achieve better performance under low contention, while more advanced guarantees can give more independence from the scheduler at a cost in complexity. We argue that this design choice is one of the most important, as most graphics processors perform their scheduling in hardware in non-standard ways. For this reason we have implemented two different STMs that differ mainly in the type of progress guarantees that they provide. The first STM is designed to be as simple as possible to lower resource requirements and improve performance. This will be known for the remainder of this paper as the *blocking STM*. The second is based on the STM by Harris and Fraser, which is more complex and designed for general multiprocessors, but offers better progress guarantees [HF03].

In the following subsections we will go through some of the different design parameters and elaborate on our design decisions.

4.1. Progress Guarantees

Progress guarantees are often divided into one of four categories. The strongest is *wait-freedom*, which guarantees that, in the context of STMs, a transaction always succeeds in a bounded number of its own steps. This guarantee is typically only provided in real-time systems, where predictability is critical, as it often hampers performance. A weaker and more practical guarantee is *lock-freedom*, which guarantees that at least one transaction will be successful in a bounded number of its own steps. A transaction in a lock-free STM is always able to make progress, even in the case of all other threads controlling transactions being suspended. Despite the name, lock-freedom does not preclude locks, as long as these can be revoked. An even weaker guarantee is *obstruction-freedom*. It guarantees that a transaction will always succeed if it is executed without conflicts with other transactions. A contention manager is often used to achieve this, by arranging for one of the conflicting transactions to back off. The final category includes the *blocking* algorithms, which uses irrevocable locks and provides no guarantees at all.

Despite the lack of progress guarantees, we decided to make the first STM we designed blocking. This allowed for a simpler design and, according to Ennals, a potentially more efficient implementation [Enn06]. The disadvantage of using a blocking implementation is that it makes the STM much more dependant on the scheduler. We had concerns of whether the hardware scheduler on the graphics processor would be able to handle locks or not, as if the scheduler is not fair, it could swap out the lock holder and repeatedly just schedule the processes waiting for the lock. However, we experienced no such problems during our experimentation.

The second STM that we designed is based on the STM by Harris and Fraser and is an obstruction-free one [HF03]. When a transaction is to be committed, it tries to acquire all the locks it requires to get exclusive access to its write locations. But, at the same time, it publicly announces the actual values that it is going to write. This gives conflicting transactions two options. If the transaction has managed to acquire all locks, but not yet written the new values, the conflicting transaction can steal locks from it, using the new value that is to be written. If the transaction has not yet acquired all its locks, the conflicting transaction may abort the original transaction before attempting to acquire its own locks. As a transaction never has to wait for another transaction to finish, the STM is non-blocking.

4.2. Conflict Detection Granularity

STMs are often designed with different levels of granularity for conflict detection depending on the language they are written for. For object oriented languages, such as Java, it is often more convenient to use objects as the basic unit. Two transactions accessing the same object will then conflict, even if they are accessing different fields in the object. This is known as a false conflict. For languages such as C, with no standard object type, it is more common to use individual words as the basic unit. Often, to lower the overhead of having a lock for each individual word, the memory is divided into several stripes, where every n:th word shares a lock. As multiple words might share the same lock there is a potential for false conflict, but this can be mitigated by increasing the number of strips.

For both the blocking and non-blocking STM we decided to put the granularity at the object level. The reason for this is that we wanted to take advantage of the graphics processor's ability to coalesce memory reads and writes into larger memory operations. For the blocking we shared locks between objects, whereas for the non-blocking we had one lock per object.

4.3. Log or Undo-Log

As there is normally no way of knowing if a transaction will succeed before it has tried to commit, there must be a way to undo transactions. The most common way is to keep a thread-local log where the changes to be performed are stored. The first time a word or object is read, a copy of it is stored in the log. All subsequent writes are then performed on the local copy. When all locks have been acquired at the end of the transaction, the items in the log are written to

Features	Blocking STM	Non-Blocking STM
Progress Guarantee	Blocking	Obstruction-free
Conflict Detection Time	Commit-time	Commit-time
Locks	Shared	Unique
Conflict Handling	Aborts the transaction	Steals locks or aborts other transaction
Conflict Detection Granularity	Object based	Object based
Visibility	Local updates	Local updates
Log or Undo-Log	Log	Log

the shared memory. An alternative, and more optimistic approach, is to acquire the write locks immediately at the first write and then store the data written inside the transaction directly to the shared memory. This is faster in cases where there are no conflicts, but to be able to abort, there needs to be an undo log that holds the old values of the words or objects written to.

Both STMs use the log method as we expect much contention and we want to avoid the problem with visibility, which occurs when other transactions read data that is yet to be committed. In a non-managed environment, this might lead to infinite loops or crashes.

4.4. Conflict Detection Time

Most STMs use some incarnation of a lock to provide mutual exclusion when writing the result from the transaction. These locks can either be acquired early, the first time that the word or object they protect is accessed, or as late as at commit time. Acquiring locks immediately have the advantage that conflicts will be detected early, but this might also, unfortunately, lead to more false conflicts. To assure that the transaction is not working on inconsistent data, it is possible to do the read validation whenever data is read or written.

The design decision here was that the blocking STM should use the same method as the non-blocking and lock at commit time.

4.5. Backoff

A backoff function is used whenever there is a conflict and a transaction needs to abort. It forces the process to wait before it tries to perform the transaction again. This lowers contention and increases the probability that at least one transaction is successful.

Backoff is often an important part of the contention management in STMs, together with the policy choice of which transaction to abort in case of a conflict. There are several ways of backing off, including linear, where the time to back off is increased linearly for every abort, and exponential, where the time to wait is, for example, doubled each time [GHP05, SS05]. We designed the STMs to be able to use both linear and exponential backoff.

5. Implementation

As mentioned in the STM Design section, an STM needs to support four basic operations. The following subsections will detail the implementation specifics for the blocking STM. For the non-blocking STM we refer to the paper by Harris and Fraser [HF03].

5.1. Begin

Each thread block is assigned a transaction descriptor to keep track of objects that have been read and objects that should be written back at commit time. Figure 2 shows an entry in the transaction descriptor. When a new transaction is initiated the transaction descriptor is cleared. The member variables in the descriptor will be motivated in the following subsections.

```
struct TransDescItem {
    int version;
    void* global;
    void* local;
    int size;
    bool readonly;
}
```

Figure 2: *Transaction descriptor item.*

```
struct VersionLock {
    int version(31bit);
    bool lock(1bit);
}
```

Figure 3: *Combined version number and lock.*

5.2. Read

The read operation transfers an object from the publicly available global memory into a private part that only the reading thread block has access to. A check is made to see if the lock that covers the object is taken. This is to make sure that no other thread block is currently writing to the object and to wait if anyone is. The lock consists of a version

number with a lock-bit, as can be seen in Figure 3. As the lock-bit is the least significant bit, one can interpret odd values of the lock as the lock being taken and even values as the lock being available.

A copy of the version number of the object is stored in the transaction descriptor and the object is copied to the local part of the memory. The version number is read again and compared to the one in the transaction descriptor. If they match, then the local copy of the object represents a consistent snapshot of the object. If they do not match, another thread block must have written to it and we have to read it again until the version numbers matches.

A pointer to the local copy and a pointer to the public object is stored in the transaction descriptor once the object have been successfully read, together with a marker that indicates whether the local copy has been updated or not. The pointers are needed so that the commit operation knows where to write back the local copy.

The read operation then returns a pointer to the local copy of the object. If there already exists a local copy of the object, due to it being read earlier in the transaction, it is just a matter of returning the pointer to that local copy.

5.3. Write

When a local copy of an object has been updated, it needs to be marked as such so that it is updated at commit time. This is done by going through the transaction descriptor looking for the pointer to the object and then marking it when found. Since all writes are being performed locally, there is no need for any locks in this phase.

5.4. Commit

At commit time the STM needs to make sure that no other thread block has changed any of the objects read or written to inside the transaction before it can write back the updated objects. The objects that are to be updated are therefore checked to see if their current version number matches the ones in the transaction descriptor. If they do, the version number is incremented by one atomically using Compare-And-Swap to lock the objects. Using Compare-and-Swap, this will only succeed as long as the version number has not changed in the mean time. The thread that locked the object now has exclusive access to it. Any failure in acquiring write locks, or version numbers that do not match, causes the transaction to abort. The updated objects are then written back to public memory once all locks have been acquired. The locks are released by increasing their version number by one and the transaction is successful. By combining the version number and the lock we make sure that any concurrent read invocation does not see any intermediate state during the writing back of the updated object to the global memory.

6. Experimental Evaluation

For evaluation we used four concurrent data-structures. All of them used software transactional memory in their design. We measured their respective performance when faced with different contention levels and using different backoff strategies. In addition to measuring the number of operations per second, we also measured the number of aborted transactions in order to better understand their respective behavior and how it affects the given performance.

6.1. Hardware

The experiments were performed on the high-end graphics processor GTX280 with 30 multiprocessors. Each multiprocessor has 8 cores, giving us a total of 240 cores. The processor clock rate is close to 1.3 GHz and the optimal memory bandwidth is 141 GB per second.

6.2. Test-Bed Applications

Binary Tree Each thread block inserts a fixed quantity of randomly picked values, uniformly distributed, into a binary tree. As the tree grows wider there should be fewer conflicts.

Queue Each thread block performs an even amount of enqueue and dequeue operations on a single queue. This benchmark should provide the highest level of contention as only one enqueue or dequeue operation can take place at any given time.

Hash-map Each thread block inserts a fixed amount of randomly picked values, uniformly distributed, into a hash-map with 128 buckets, each bucket being an individual list. This benchmark is similar to the queue benchmark, but lowers contention by dividing access to it over several buckets. The transactions are longer since they need to find the end of the list before they can insert their element.

Skip-list Each thread block performs an even amount of insert, find, and delete operations on a skip-list with a maximum of 7 levels. This is a more complex benchmark that is expected to scale similarly to the tree. To compare the performance of the respective STMs with a highly parallel design of an advanced data-structure, we also compared the respective STM skip-list implementations with the lock-free skip-list by Sundell and Tsigas [ST04].

6.3. Experiment Settings

To see how the STMs react to different contention levels we have tested them with two scenarios. One where the test application performs some local work before accessing the data-structure, a low contention scenario, and one high contention scenario where there is no pause between transactions. The time for the local work is picked randomly after

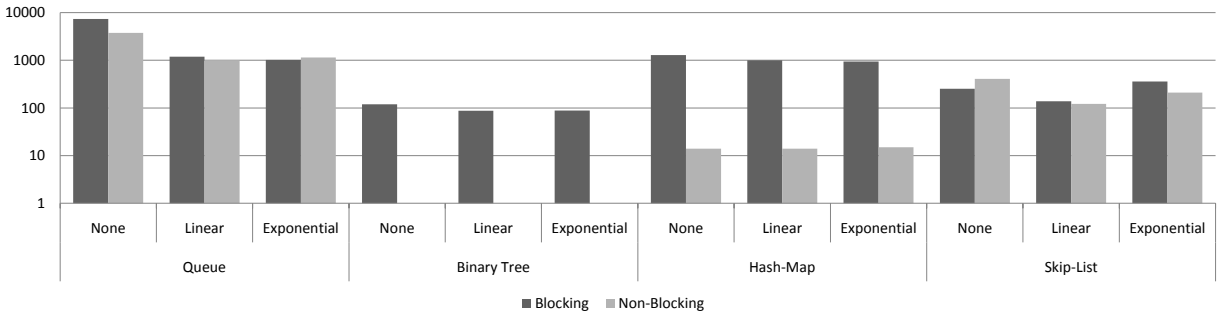

Figure 4: *Average number of aborts per transaction with **low** level of contention using 60 thread blocks (**logarithmic scale**).*

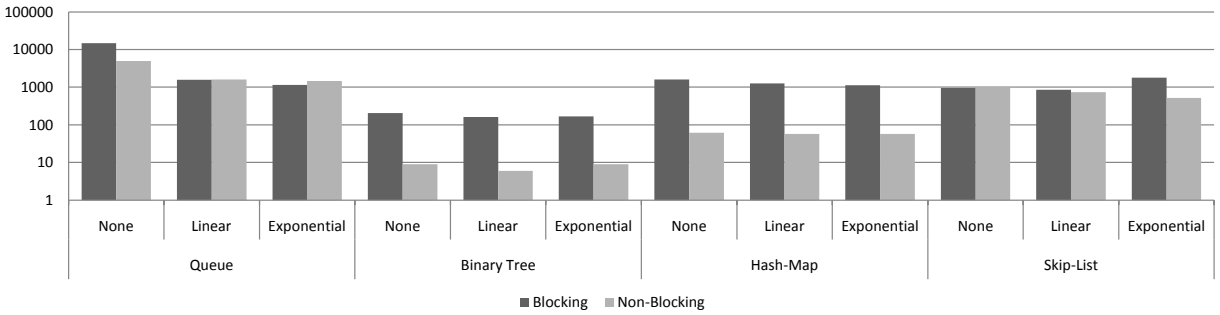

Figure 5: *Average number of aborts per transaction with **full** level of contention using 60 thread blocks (**logarithmic scale**).*

each transaction from a uniform distribution and takes a total of ∼450 ms for one thread block to complete and around ∼500 ms for 60 thread blocks to complete in parallel.

Since the choice of backoff-function is important, we did each experiment with two types of backoff, one linear and one exponential. We also performed the experiments using no backoff at all.

Each of the data-structures were evaluated with a varying number of thread blocks. We did not vary the number of threads in each thread block, as we are only synchronizing the accesses by the thread blocks and not the individual threads within a block. The measurements were repeated 50 times.

7. Discussion

At a low level of contention, the blocking and the non-blocking skip-list and binary tree both scale well, see Figure 8. Looking at the number of aborts for the skip-list, Figure 4, one can see that the average number of aborts is about the same, whereas for the binary tree there is a distinct difference. The blocking STM has an average of over one hundred aborted transactions for each thread block, while the non-blocking STM has none at all. With greater contention, Figure 5, the number of aborts remains the same for the blocking STM while it has increased to ten for the non-blocking STM. Despire this, the performance in number of operations per ms is much better for the non-blocking STM; see Figure 9.

In Figures 4 and 5 we see that the backoff has quite a large effect on the average number of aborts for the queue and that there does not seem to be any difference between the linear and the exponential backoff. However, the backoff does only slightly alter the number of operations per ms, which can be seen in Figures 6 and 7. For the other benchmarks there is hardly any difference between the results for the different backoff schemes, both when it comes to operations per second and when it comes to aborted transactions per number of thread blocks.

The queue does not scale well for either the blocking or non-blocking STM. This is not surprising since only one enqueue or dequeue operation can take place at any given time. The hash-map and binary tree both scale much better with the non-blocking STM and it can be clearly seen that the non-blocking has a lot fewer aborted transactions when it comes to these benchmarks. This can be attributed to the fact that the non-blocking version can steal locks to continue working without aborting.

In Figure 10 the result from the comparison between the respective STM skip-lists and the lock-free skip-list by Sundell and Tsigas is presented [ST04]. By the figure it is clear that the respective STM skip-lists are significantly slower

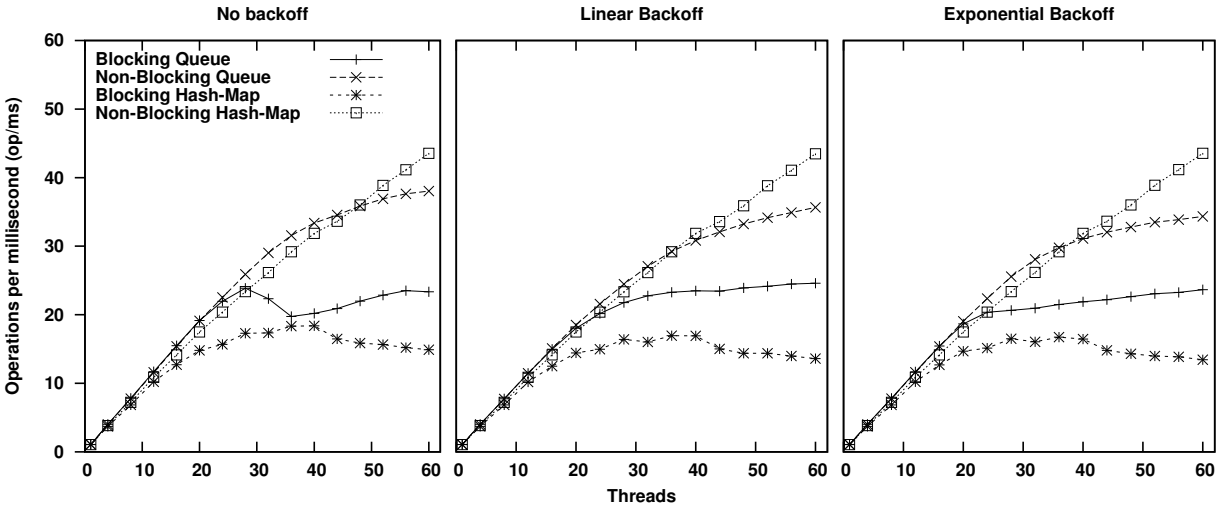

Figure 6: *Experimental result for the queue and hash-map with **low** level of contention.*

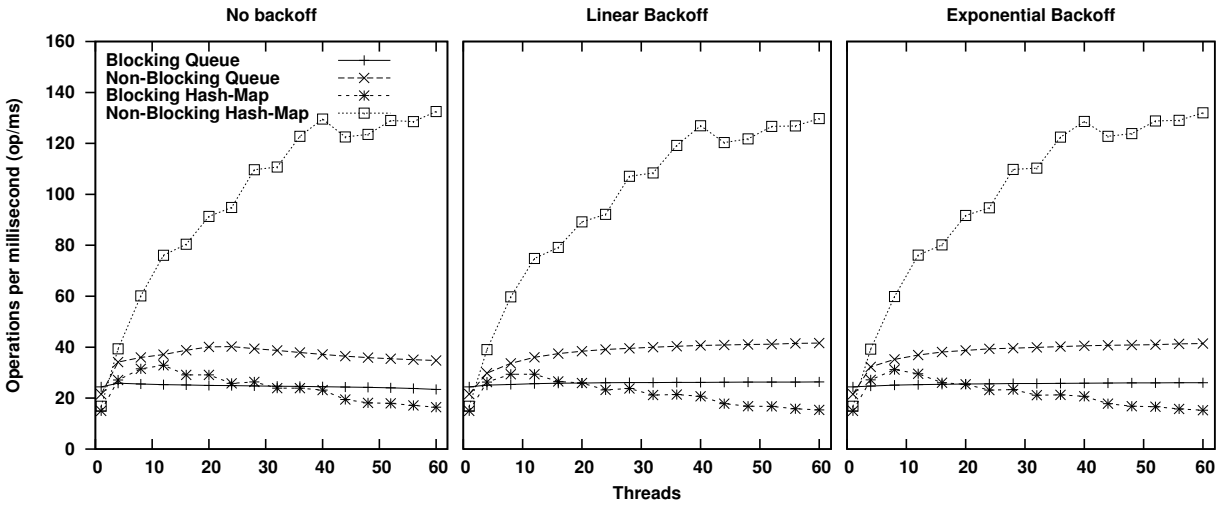

Figure 7: *Experimental result for the queue and hash-map with **full** level of contention.*

than the non-STM skip-list. This is to be expected, as one can gain a lot in performance by using more complex synchronization techniques during the design phase of the data-structure. However, there is a trade-off, as these techniques require much more time and expertise to get the design right, than to use an STM.

8. Conclusion

Software Transactional Memory has attracted the interest of many researchers over recent years. We have designed and implemented two STMs for graphics processors, one blocking and one non-blocking. The design issues involved in the designing of these two STMs are described and explained in the paper together with experimental results comparing the performance of the two STMs. We found that while a blocking STM is simpler to implement, providing additional progress guarantees, such as obstruction-freeness, improves performance and lowers the number of aborted transactions.

Acknowledgements

This work was partially supported by the EU as part of FP7 Project PEPPHER (www.peppher.eu) under grant 248481 and the Swedish Research Council under grant number 37252706. Daniel Cederman was supported by Microsoft Research through its European PhD Scholarship Programme.

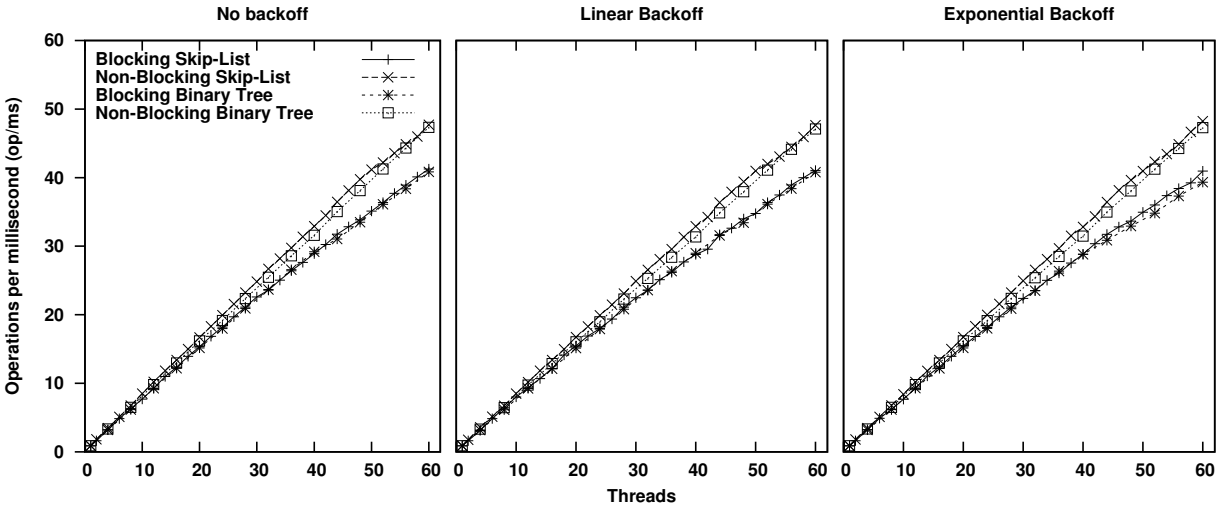

Figure 8: *Experimental result for the binary tree and skip-list with **low** level of contention.*

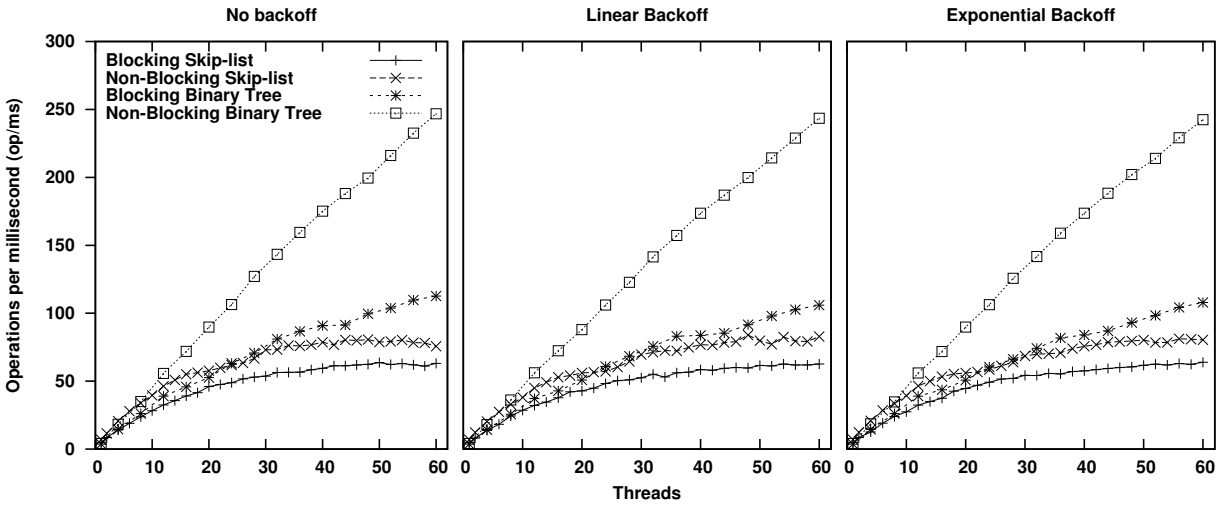

Figure 9: *Experimental result for the binary tree and the skip-list with **full** level of contention.*

References

[ATLM*06] ADL-TABATABAI A.-R., LEWIS B. T., MENON V., MURPHY B. R., SAHA B., SHPEISMAN T.: Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2006), ACM, pp. 26–37.

[CBM*08] CAŞCAVAL C., BLUNDELL C., MICHAEL M., CAIN H. W., WU P., CHIRAS S., CHATTERJEE S.: Software Transactional Memory: Why Is It Only a Research Toy? *Queue 6*, 5 (2008), 46–58.

[CT08] CEDERMAN D., TSIGAS P.: On Dynamic Load Balancing on Graphics Processors. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2008), Eurographics Association, pp. 57–64.

[DFL*06] DAMRON P., FEDOROVA A., LEV Y., LUCHANGCO V., MOIR M., NUSSBAUM D.: Hybrid Transactional Memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2006), pp. 336–346.

[DSS06] DICE D., SHALEV O., SHAVIT N.: Transactional Locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)* (2006), pp. 194–208.

[Enn06] ENNALS R.: *Software Transactional Memory Should Not Be Obstruction-Free*. Tech. Rep. IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[FH07] FRASER K., HARRIS T.: Concurrent programming with-

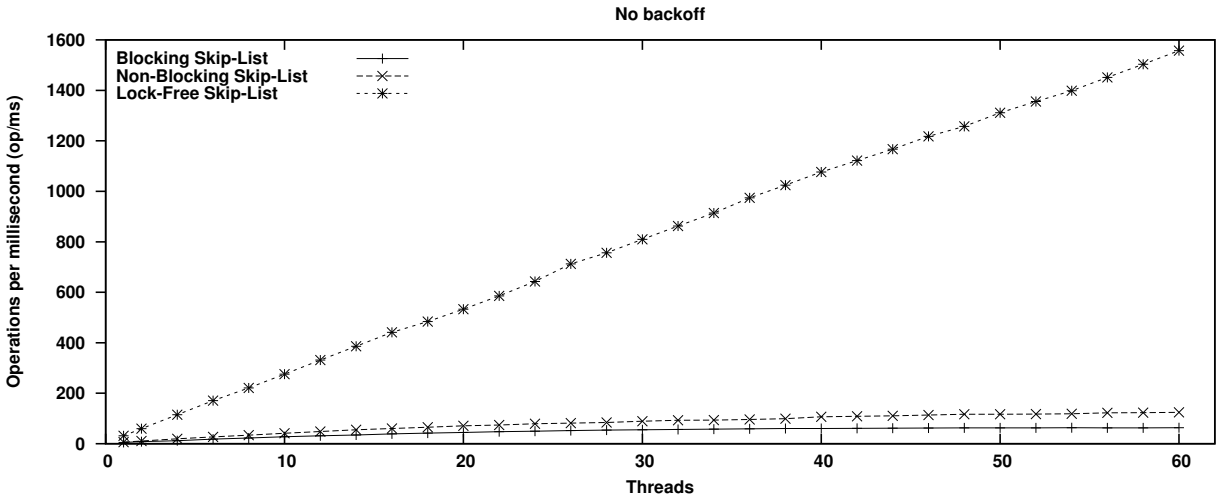

Figure 10: *Experimental result for the STM skip-lists compared with the skip-list by Sundell and Tsigas with **full** level of contention [ST04].*

out locks. *ACM Transactions on Computer Systems 25*, 2 (2007), 5.

[GHP05] GUERRAOUI R., HERLIHY M., POCHON B.: Polymorphic Contention Management. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)* (2005), vol. 3724 of *Lecture Notes in Computer Science*, pp. 303–323.

[Her88] HERLIHY M. P.: Impossibility and universality results for wait-free synchronization. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing* (New York, NY, USA, 1988), ACM, pp. 276–290.

[HF03] HARRIS T., FRASER K.: Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications* (New York, NY, USA, November 2003), vol. 38, ACM Press, pp. 388–402.

[HLMS03] HERLIHY M., LUCHANGCO V., MOIR M., SCHERER W.: Software Transactional Memory for Dynamic-sized Data Structures. In *Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (July 2003).

[HM93] HERLIHY M., MOSS J. E. B.: Transactional Memory: Architectural Support For Lock-Free Data Structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture* (1993).

[HPST06] HARRIS T., PLESKO M., SHINNAR A., TARDITI D.: Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2006), ACM Press, pp. 14–25.

[KCJ*06] KUMAR S., CHU M., J. HUGHES C., KUNDU P., NGUYEN A.: Hybrid Transactional Memory. In *Proceedings of Symposium on Principles and Practice of Parallel Programming* (Mar 2006).

[Kni86] KNIGHT T.: An architecture for mostly functional languages. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming* (New York, NY, USA, 1986), ACM Press, pp. 105–112.

[LK08] LARUS J., KOZYRAKIS C.: Transactional memory: Is TM the answer for improving parallel programming? vol. 51, ACM.

[Moi97] MOIR M.: Transparent Support for Wait-Free Transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms* (1997), Springer-Verlag, pp. 305–319.

[PDC09] P. DUBLA K. DEBATTISTA L. S., CHALMERS A.: Wait-Free Shared-Memory Irradiance Cache. In *Eurographics Symposium on Parallel Graphics and Visualization* (March 2009), Eurographics, pp. 57–64.

[SATH*06] SAHA B., ADL-TABATABAI A.-R., HUDSON R. L., MINH C. C., HERTZBERG B.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2006), ACM, pp. 187–197.

[SS05] SCHERER III W. N., SCOTT M. L.: Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2005), ACM, pp. 240–248.

[ST95] SHAVIT N., TOUITOU D.: Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1995), ACM, pp. 204–213.

[ST04] SUNDELL H., TSIGAS P.: Scalable and lock-free concurrent dictionaries. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing* (New York, NY, USA, 2004), ACM, pp. 1438–1445.