Software Transactional Memory Should Not Be Obstruction-Free

Robert Ennals

Intel Research Cambridge
15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK
robert.ennals@intel.com

Abstract. Much previous work on Software Transactional Memory has gone to great lengths to be "obstruction-free" — meaning that a transaction is guaranteed to make progress when all other transactions are suspended.
In this paper we argue that obstruction-freedom is not an important property for software transactional memory, and demonstrate that, if we are prepared to drop the goal of obstruction-freedom, software transactional memory can be made significantly faster.

1 Introduction

The benefits of transactions have been known in the database community for a long time [2]. Transactions offer a simple programming model that takes much of the pain out of concurrent programming. Programmers do not have to worry about deadlock, livelock, data consistency, atomicity, priority-inversion, or lock placement – in fact they barely have to think about concurrency at all [14].

The idea of using transactions for general purpose programming is almost as old as transactions themselves [6, 7]; however it has recently gained renewed popularity, and the name "Software Transactional Memory"(STM) [13, 9] has been widely adopted to refer to such lightweight transactions. This renewed interest has arisen partly because, with the introduction of multi-core processors, parallelism is moving into the mainstream, creating a demand for techniques that might make such devices easier to program; and partly because, as the gap between processor speed and memory speed has grown larger, the performance overhead of STM has become less significant.

Recent work on Software Transactional Memory has gone to great lengths to be "obstruction-free" [4] — meaning that a thread is guaranteed to make progress when all other threads are suspended. While this property is essential in distributed systems (which is the background of many STM researchers), we argue that it is not appropriate for non-distributed STM. Our argument for this proceeds as follows:

- We expect that the majority of people who use STM will use it to improve the performance of programs that would otherwise execute largely sequentially.
- In the programming paradigms to which programmers are accustomed, it is allowable for one atomic operation to block other atomic operations of the same or lower priority. We believe that programmers consider this to be acceptable behaviour.
- It is easy to adaptively vary the number of tasks in an application to match the number of cores available — making it unlikely that a transaction will be blocked by a transaction that has been switched out.

– If we abandon the principle of obstruction-freedom, we can produce an STM implementation that is considerably faster.

The remainder of this paper proceeds as follows. Section 2 examines the way in which programmers currently use concurrency. Section 3 argues that obstruction-freedom is not an important property for real programs. Section 4 explains why the requirement for obstruction-freedom reduces STM performance. Section 5 presents a design for a non-obstruction-free STM. Finally, Section 6 shows that our STM outperforms the best previous STM designs: by a factor of 5 under high contention and by a factor of 2 when memory bandwidth limited.

Threads and Tasks

It is important that we distinguish between *threads* and *tasks*. Throughout the rest of this paper we will use the following definitions:

– A **thread**, is a programmer-level construct used to specify that several blocks of code can be executed in parallel.
– A **task**, is a OS-level construct that runs on a core and executes code from threads. The runtime can dynamically adjust the number of tasks to match the number of cores available to the application, thus minimising the number of OS-level context switches. The runtime multiplexes several threads onto each task.

2 What do People Use Threads For?

Programmers use threads for two reasons: for convenience, and for performance. It is important to distinguish between the two.

Threading for Convenience

Before the advent of multi-core processors, the vast majority of computers had only a single processing core. Since there is no performance advantage to using multiple threads on such a machine (only one can execute at a time) programs designed to run on single-core machines do so purely for programming convenience — to allow one computation to proceed without blocking others. For example, a program might have separate compute and GUI threads.

Such programs tend to use very simple synchronisation mechanisms. Many programs ensure mutual exclusion using a single shared lock, which is held whenever a thread wishes to do something atomically. The single shared lock provides a simple programming interface that makes it easy to avoid deadlock and easy to avoid corrupting data. If there is only one processing core then there is no performance cost to using only a single lock rather than a more complex locking strategy, since threads can only execute one at a time.

Some programs may additionally need to ensure that high-priority threads are not blocked by long running low-priority operations. This is often done by having several locks, some of which should only be held for brief periods of time, and having the high-priority threads only use high-priority locks.

We expect that programs that use threading for convenience will be ported to STM by simply converting all lock-protected atomic blocks into transactions. Note that, when such an STM program is run, it is entirely acceptable for a transaction to block a transaction of the same or lower priority – since this is the behaviour that would be experienced in the original program.

Threading for Performance

More recently, as multi-core processors have moved parallel processing into the mainstream, more attention has been paid to the use of threads for performance. In such cases, a program that would most naturally be written using only a small number of threads is instead broken down into a greater number of threads, so that the program can take advantage of multiple processing cores.

Such programs will often be written by starting with sequential code, and modifying it so that operations that would have been done sequentially are instead performed in parallel. As with "threading for convenience" we expect that such programs will be ported to STM by converting all lock-protected atomic sections into transactions.

Note that, as before, it is entirely acceptable for a transaction to block the execution of transactions of the same or lower priority, since this is the behaviour that would be experienced in the original, non-parallelised, program, in which the parallel computations were performed sequentially.

3 Why Obstruction-Freedom is Unnecessary

There are three arguments used in favour of obstruction-freedom that we need to rebut:

- Obstruction-freedom prevents a long-running transaction blocking others
- Obstruction-freedom prevents the system locking up if a thread is switched part-way through a transaction
- Obstruction-freedom prevents the system locking up if a thread fails

Long-running transactions block others

Some have argued that, if an STM is not obstruction-free, a long-running or non-terminating transaction can cause other transactions to block.

We observe that obstruction-freedom does not make such problems go away. Consider a transaction that reads an object, computes for a year, and then writes to the same object. The only way in which *any* STM can allow such a transaction to complete is if it blocks all other transactions that manipulate that object for a year. Either the long-running transaction must be able to block other transactions, or the long-running transaction must be prevented from completing.

Neither obstruction-freedom, nor the stronger property of lock-freedom guarantee that a given transaction will make progress in a situation in which other transactions are executing. Obstruction-freedom only guarantees progress if there are no other conflicting transactions, while lock-freedom only guarantees that the system as a whole will make progress.

Moreover, as we argued in Section 2, while it can be useful to prevent low-priority transactions blocking high-priority transactions, we believe that it is perfectly acceptable for the execution of a transaction to block the execution of other transactions of the same or lower priority, within the same application.

Context Switching

Some have argued that, if an STM is not obstruction-free, the system could grind to a halt if the OS switches out a task that is holding onto a vital resource.

We observe that, if the OS switches a task out, it will always switch it back in again eventually; thus such interruptions are always temporary.

Moreover, if the runtime system behaves appropriately, such interruptions should be so rare as to be unimportant. It is easy for the runtime system to adaptively vary the number of active tasks to match the number of processor cores that the operating system is making available to it. If the runtime system finds that its tasks are being regularly switched out, then it reduces the number of tasks that it is running until context switches are sufficiently rare that the cost of switch related-blocking matches the cost of allowing unused processor cores to go idle.

Similarly, it is easy for the runtime system to arrange that when its tasks perform user-level context switches between threads they do so between, rather than during, transactions. On operating systems with suitable support one can also encourage the OS to perform context switches between, rather than during, transactions.

We thus argue that it is entirely acceptable for a switched out transaction to block the rest of the system, since it causes only a temporary interruption when it happens, and it can be easily made sufficiently rare to have a negligible impact on performance.

Independent Failure

Some argue that it is important that the system cope with transactions failing silently and independently, due to software or hardware failure.

In the case of a software failure, we note that software failure would break the original, non-STM program, so it is acceptable for it to break the STM program also.

In the case of a hardware failure, we respond two-fold: Firstly, while independent failure is a really important issue in distributed systems, on a multi-core or SMP system it is so rare that it is not worth worrying about. Secondly, as before, hardware failure would have broken the original non-STM program, so it is acceptable for it to break the STM program.

4 Why Obstruction-Freedom makes Efficient Implementation Difficult

If obstruction-freedom were merely unnecessary, then we would have no objection to it; however, as we argue in this section, making an STM obstruction-free prevents it from applying several important optimisations that would boost performance. In particular, obstruction-freedom prevents an object-based STM from applying the following important optimisations:

– Object metadata should be stored either adjacent to object data, or in a region that is private to the current processor, thus making it unlikely that extra cache misses will be incurred when loading it.

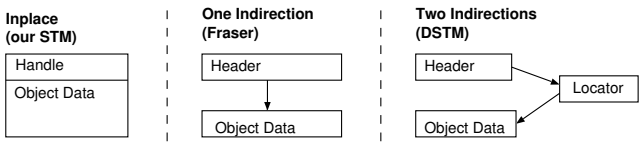

Fig. 1. How many indirections are needed to find object data?

– The number of active transactions should not exceed the number of available cores, thus avoiding unnecessary transaction conflicts.

In the following sections we explain these issues in more detail. We address these issues in the context of an object-based STM [4, 1, 10]; however we believe that similar issues apply to word-based STMs [3, 13].

Cache-Locality

All obstruction-free object-based STM designs presented so far [4, 1, 10] require a program to follow an indirection from the object metadata in order to find the current version of the object data (Figure 1). This is undesirable since it requires a program to load multiple cache lines for every read and write. In cases where the program is memory-bandwidth limited, this can halve performance relative to a design in which object data is stored in-place, adjacent to the object metadata (See Section 5 for results).

To understand why an obstruction-free STM cannot store object data in-place, consider the case in which a transaction has started writing to an object and then gets switched out. If another transaction needs to access that object, what should it do?

– It could wait for the first transaction to finish with the object, but that would not be obstruction-free, since the first transaction would be blocking the second transaction.
– It could start working on the object, but that would not be safe, as the first transaction might wake up and write over the version of the object that the second transaction is working with.
– It could forcibly abort the first transaction, but there is no safe, portable, way to do this without blocking until the first transaction acknowledges that it has been aborted – which obstruction-freedom does not allow. Moreover, if any transaction can abort any other transaction then livelock is likely to arise – since the first transaction might restart and abort the second transaction.

Excessive Active Transactions

Suppose we have N transactions running on N cores, one transaction per core. What should we do if we wish to start another transaction?

The most efficient approach is for a new transaction to wait for a core to become free before it starts executing. In this way we minimise the number of transactions executing, while making full use of all available cores.

If the STM is obstruction-free then it cannot wait for existing transactions to finish, and thus it must use context switching to share N cores between $N + 1$ transactions. Unfortunately, as the number of concurrently executing transactions increases, so does the frequency of conflicts between them, thus reducing overall performance.

5 Our Implementation

To demonstrate that obstruction-freedom restricts the performance on an STM implementation, we have implemented a new object-based STM implementation that is not obstruction-free and which uses the techniques, described in Section 4, that obstruction-freedom prevents. We show in Section 6 that this STM significantly outperforms the previous best performing STM implementations.

The Basic Idea

The fundamental concurrency control technique used by our implementation is similar to that used by DSTM [4]. We use *revocable two-phase locking* [2] to manage writes, and *optimistic concurrency control* [5, 2] to manage reads.

- **Revocable Two Phase Locking for Writes:** A transaction locks all objects that it writes and does not release these locks until the transaction terminates. If deadlock occurs then one transaction aborts, releasing its locks and reverting its writes.
- **Optimistic Concurrency Control for Reads:** Whenever a transaction reads from an object, it logs the version it read. When the transaction commits, it verifies that these are still the current versions of the objects.

Memory Layout

Figure 2 illustrates the memory layout used by our algorithm. Unlike previous STMs, we divide memory into public and non-public regions:

- **Public Memory:** can be accessed by any transaction. This region contains only objects.
- **Private Memory:** Each transaction has its own region of private memory that only that transaction can access. This region is used for storing book-keeping information, in the form of read and write descriptors.

Descriptors in private memory are not freed until a transaction commits, and cannot be seen by any other transaction. We can thus allocate them sequentially in a continuous block of memory, and immediately re-use this memory when another transaction starts on the same core.

Cachegrind [11] simulation reveals that the vast majority (typically ∼99.5%) of cache misses are in public memory. Private memory is relatively small and rapidly recycled, and so we expect it to stay in the local processor cache.

Relative to the original non-STM program, the only data we add to public memory is an additional *handle* field on each object. This field is adjacent to the object data, and so is likely to be in the same cache line. The meaning of the handle depends on its lowest order bit. If the lowest order bit is 1 then the handle is a version number, otherwise it is a pointer to a write descriptor:

- **Version number** v: no transaction currently has a lock on the object. The object data is version v of the object[1].

[1] Version number roll-over is fine, provided our scheduler ensures that no more than two billion transactions commit while another transaction is active.

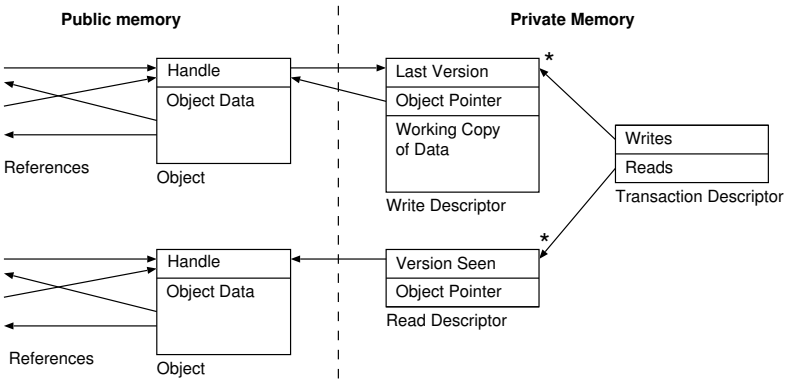

Fig. 2. Memory Layout

– **Pointer to a write descriptor** w: the object is locked by a transaction t and w is a pointer to a write descriptor in t's private memory. (Figure 2)

Reading and Writing Objects

To write to an object o, a transaction t must obtain an exclusive lock on o, and create a working copy of o's data that it can write to. If o's handle is a version number v, then t uses an atomic compare-and-swap operation to replace v with a pointer to a new write descriptor. This write descriptor has its *last-version* field set to v and its working copy initialised to the current version of the object data.

If o's handle is a pointer to a write descriptor, w, then t waits until the owning transaction, s, sets the handle to be a version number — indicating that it has finished with the object. If t has waited for more than a given number of cycles and s is of lower priority, then t requests that s abort itself. t can find a descriptor for s by zeroing the low-order bits of w. If deadlock is detected, then the system aborts one of the transactions in the cycle.

To read from an object, a transaction waits for the object handle to be a version (as for writing) and then logs the version number in a read descriptor.

Committing

Our commit algorithm is similar to that of DSTM [4]. To commit, a transaction checks that it is valid, and then makes its writes visible to other transactions.

A transaction is considered to be valid if none of the objects that it read from have subsequently been written to by other transactions. For each written object, the transaction makes its writes visible by copying across its working copy of the data and setting the handle to a new version number.

Like other STMs that use optimistic concurrency control for reads [4, 1], it is necessary for the runtime to periodically abort any transactions that are found to be invalid. If this was not done then a transaction might go into an infinite loop as a result of having seen inconsistent data. Similarly, a transaction that segfaults can retry if it is found to be invalid [1].

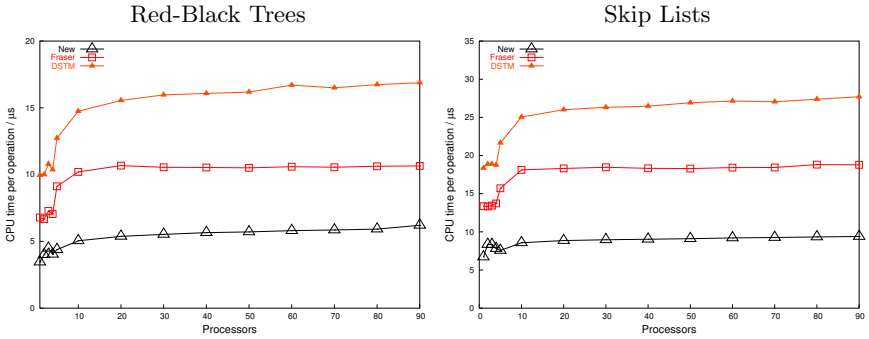

Fig. 3. Scalability under low contention (key space of 2^{19})

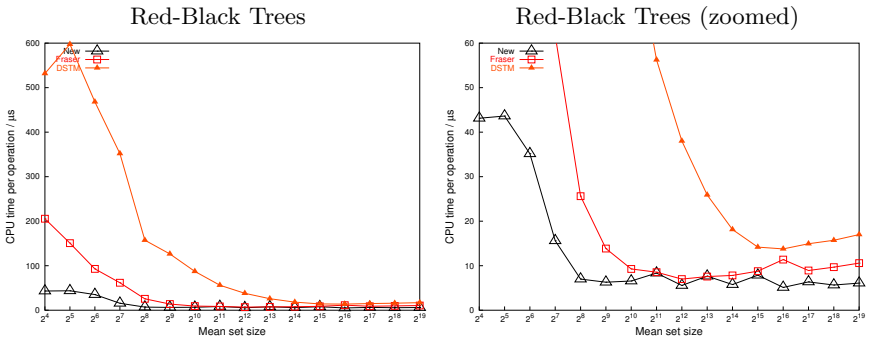

Fig. 4. Performance under varying contention (90 processors)

6 Performance Evaluation

To ensure the fairest comparison with other STMs, we asked Keir Fraser to benchmark our algorithm for us on the exact same setup as he used to benchmark his STM [1] . These tests were performed using the same machine, the same benchmarks, the same workload, and the same DSTM implementation as he used in his thesis [1].

The machine on which tests were run is a SunFire 15K server populated with 106 UltraSparc III processors, each running at 1.2Ghz. The benchmarks are Fraser's red-black tree and skip-list programs, both of which read and write random elements in a set. The benchmarks are run with a mix of 75% reads and 25% writes (which Fraser argues is representative of real programs). Performance is compared against Fraser's STM [1] and Fraser's C re-implementation[2] of DSTM [4][3] — which are currently established as the two best performing STM implementations [8, 1].

Figure 3 shows the performance under low contention, with the red-black tree benchmark on the left and the skip-lists benchmark on the right. Here, the benchmarks are run with a large data set (2^{19} objects) ensuring that the transactions rarely attempt to read or write the same object [1]. Our algorithm consistently takes around 50%-60% of the time taken by Fraser's STM and around 35% of the time taken by DSTM. In this case we believe that our algorithm wins by requiring a processor to load fewer cache lines than the

[2] The original implementation is in Java, and so could not have been fairly compared.
[3] Using the POLITE contention manager, which is considered to be one of the best [12].

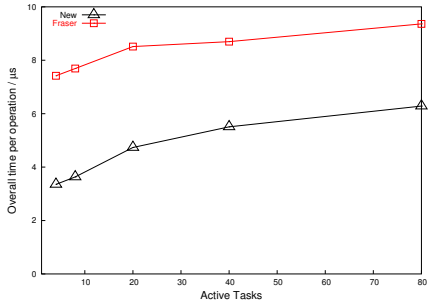

Fig. 5. Performance as task count increases (4 cores, key space of 2^{19}, red-black trees)

other algorithms. Indeed, the processor performance counters tell us that, per transaction, our STM incurs only 41% of the L2 misses, 58% of the L1 misses, and 22% of the TLB misses incurred by Fraser's STM.

Figure 4 shows performance under varying contention. Here, the number of processors is kept static at 90 and the data set size is varied from 16 to 2^{19} elements. Smaller data sets cause greater contention as transactions are more likely to attempt to manipulate the same element. Under high contention DSTM's contention manager copes poorly and comes close to livelock, while Fraser's STM is almost five times slower than ours. We believe that the poor performance of Fraser's STM is due to its use of helping[4]; if a transaction is blocked by another, then it will "help" the other transaction to complete. In practise it is better to simply wait for the other transaction to finish of its own accord. If transactions help each other then one can end up with 90 processors all trying to perform the same commit operation and all fighting over the same cache lines.

Figure 5 shows performance under varying numbers of tasks. These tests were done on a 4-way SPARC machine, rather than the 106-way machine used for the previous tests – in order to provoke the operating system into context switching between our tasks. As the number of tasks increases, context-switches during transactions become more common, transaction conflicts increase, and performance generally decreases. Our STM is affected more than Fraser's, since it allows a switched out transaction to block others; however our STM remains the fastest. Note that, when used normally, our STM does not allow there to be more tasks than available cores.

7 Conclusions

We have argued that obstruction-freedom is not an important property for STMs and have demonstrated that a non-obstruction-free STM can achieve significantly better performance than an obstruction-free one. We thus believe that future STM designs should not attempt to be obstruction-free.

[4] At one point we experimented with a version of our algorithm that had helping, and preliminary results suggested that its high-contention performance was similar to Fraser's STM.

Availability

Our implementation is available on SourceForge at `http://sourceforge.net/projects/libltx`. The source files used in our benchmarks can also be found at that URL.

Acknowledgements

We would like to thank Keir Fraser for providing us with his STM implementation and for benchmarking our algorithm on his testbed. We would also like to thank Michael Fetterman, Keir Fraser, Tim Harris, Maurice Herlihy, Gianlucca Iannaccone, Anil Madhavapeddy, Alan Mycroft, Matthew Parkinson, Rirchard Sharp, and Eben Upton for making useful suggestions.

References

1. Fraser, K. *Practical Lock Freedom*. PhD thesis, University of Cambridge, 2003.
2. Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
3. Harris, T., and Fraser, K. Language support for lightweight transactions. In *Proceedings of the 18th Annual ACM-SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '03)* (Oct. 2003).
4. Herlihy, M., Luchangco, V., Moir, M., and Scherer, W. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC '03)* (July 2003), pp. 92–101.
5. Kung, H. T., and Robinson, J. T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems 6*, 2 (June 1981), 213–226.
6. Liskov, B., and Scheifler, R. Guardians and actions: linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems 5*, 3 (July 1983), 381–404.
7. Lomet, D. B. Process structuring, synchronization and recovery using atomic actions. In *Proceedings of an ACM Conference on Language Design for Reliable Software* (Mar. 1977), D. B. Wortman, Ed., ACM, ACM, pp. 128–137.
8. Marathe, V. J., Scherer, W. N., and Scott, M. L. Design tradeoffs in modern software transactional memory systems. In *Proceedings of the Seventh ACM Workshop on Languages, Compilers and Run-time Support for Scalable Systems* (Oct. 2004).
9. Marathe, V. J., and Scott, M. L. A qualitative survey of modern software transactional memory systems. Tech. Rep. TR839, University of Rochester, June 2004.
10. Moir, M. Transparent support for wait-free transactions. In *Distributed Algorithms, 11th International Workshop* (Sept. 1997), vol. 1320 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 305–319.
11. Nethercote, N. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, Nov. 2004.
12. Scherer, W. N., and Scott, M. L. Contention management in dynamic software transactional memory. In *Proceedings of the Workshop on Concurrency and Synchronisation in Java Programs* (July 2004).
13. Shavit, N., and Touitou, D. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)* (Aug. 1995), pp. 204–213.
14. Weikum, G., and Vossen, G. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control*. Morgan Kaufmann, 2001.