Efficient Software Transactional Memory

Robert Ennals

IRC-TR-05-051

Research at Intel

Efficient Software Transactional Memory

Robert Ennals

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

Abstract. Transactions offer a simple programming model that takes much of the pain out of concurrent programming. Rather than having to battle with a sea of locks, programmers need merely mark blocks of code that they wish to execute atomically, and let their compiler and runtime do the work of ensuring that the behaviour really is atomic.
We present a new algorithm for implementing efficient lightweight transactions that we have found to significantly outperform the best previous algorithms. When run on our 106-processor test machine, our algorithm is almost five times as fast as the best previous algorithm under high contention, almost twice as fast when using large data sets, and we have yet to find a situation in which another implementation outperforms it. As we explain in our paper, we achieve these results by trading off theoretical elegance for practical efficiency, and by taking care to minimise cache contention.

1 Introduction

The benefits of transactions have been known in the database community for a long time [7]. 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 [25].

While transactions provide the programmer with a very convenient programming model, they also impose significant bookkeeping overheads on the processor. In the world of databases, these overheads are acceptable, since IO, not processor speed, is typically the main factor limiting performance. However, for most other applications the overhead of transactions has kept them out of general use.

It seems that things may soon change. A number of groups have presented designs for processors that can execute transactions in hardware [12, 20, 19, 21, 2]. Known as Transactional Memory, such designs avoid the bookkeeping overhead normally associated with transactions and thus make them practical for general purpose computing.

These hardware designs have been accompanied by a flood of programming languages that use transactions as their primary concurrency control mechanism, assuming that the underlying hardware and runtime will be able to implement

them efficiently. Several such languages have major industrial support, including Cray's Chapel [3, 5], IBM's X10 [4], and Sun's Fortress [24, 1].

This move towards transaction based programming has created a demand for efficient software implementations of transactional memory. Known as Software Transactional Memory (STM) [23, 15], such designs are needed for two reasons:

- To allow programs written with atomic transactions to run on existing hardware that does not have transaction support.
- To act as a fall-back mechanism for cases in which the hardware transaction implementation has exhausted it's physical resources.

It is not practical to simply re-use transaction implementations from databases, since Software Transactional Memory has very different requirements. Unlike databases, STMs store all data in non-persistent main memory, and thus they need not concern themselves with issues such as durability, distributed operation, replication, logging, checkpointing, disk scheduling, and fault tolerance.

1.1 Our Contribution

On the modern multi-processor machines on which STM implementations are designed to run, cache behaviour has a significant effect on performance [9]. Every time a processor attempts to access data that it does not have in its cache, it must wait for potentially hundreds or even thousands of cycles. Moreover, if two processors attempt to write to the same area of memory then the cache line will bounce between the caches of the two processors, causing significant slowdown.

In this paper we present a new object-based STM design that aims to maximise performance by minimising cache contention and memory bandwidth requirements. We do this by making four significant deviations from previous STM designs:

- **Object versioning information is stored inline**, removing the need for a transaction to access additional cache lines in order to find the current data for an object.
- **Book-keeping information is always private to the thread that created it**, allowing it to stay in that processor's cache.
- **Book-keeping information is allocated sequentially, and its memory is immediately re-cycled when a transaction completes** (no need for GC or ref counting), thus keeping it small and localised.
- **We do not guarantee that a transaction will make progress while another transaction is descheduled by the OS.**

While allowing a transaction to be obstructed by descheduled transactions is theoretically inelegant, we do not believe it is a problem in practice. If a transaction t is obstructed by a transaction s, then t can send s a signal, asking s to abort itself. If s is currently descheduled, then it will abort itself as soon as

the OS reschedules it. The only way that t can be permanently obstructed is if the OS is starving s, but if the OS can starve s then it could also starve t[1] and so the STM has not made things any worse then they were before.

Although depending on properties of the operating system's scheduling and ability to send messages between transactions complicates the theoretical understanding of the algorithm, from a practical standpoint these seem simple and common assumptions to make. We are after implementable efficiency and not theoretical elegance.

One might worry that performance will degrade when the number of transactions exceeds the number of processor cores, since there are more descheduled transactions that can obstruct executing transactions. In section 4 we show that, while performance does indeed degrade to some extent, it does not degrade by much, and our STM implementation continues to beat its rivals even when the number of transactions significantly exceeds the number of cores.

2 How It Works

The fundamental concurrency-control technique used by our algorithm is similar to that used by DSTM [11]. We use *revocable two phase locking* [7] to manage writes and *optimistic concurrency control* [13, 7] to manage reads:

- **Revocable Two Phase Locking for Writes:** Whenever a transaction t wishes to write to an object o, it must first obtain an exclusive (but revocable) lock on o. Transaction t holds onto these locks until it either commits or is forcibly aborted. If t wishes to write to an object that is locked by another transaction, s, then t will either wait for s to finish, or steal the object by forcing s to abort. Crucially: **Revocable two phase locking does not suffer from deadlock, and is thus able to use finer-grain locks than would be practical with non-revocable locks.**

- **Optimistic Concurrency Control for Reads:** Whenever a transaction reads from an object, it logs the version of the object it saw. When the transaction commits, it checks the current versions of all objects it read and verifies that the versions it read are still current. Crucially: **Unlike read-locks, optimistic concurrency control does not cause multiple readers to experience cache contention.**

In the rest of this paper, we will use the following terms:

- **Transaction:** A block of code that the programmer wishes to execute atomically.
- **Processor:** A CPU, hyper-thread [16], or core on a multi-core chip.
- **Object:** A block of data[2] that is manipulated by the user program — such objects would exist even in the absence of a transaction system.

[1] It may be necessary to use priority inheritance to ensure that if t can be scheduled then s can also be scheduled.

- **Valid:** A transaction is valid if none of the objects it read optimistically have been changed by another transaction since it read them.
- **Commit:** If a transaction completes its work, and is valid, then it can commit, at which point it will atomically make visible all of the changes it made to objects.
- **Obstructed:** A transaction t is obstructed by a transaction s if s prevents t from making progress.
- **Abort:** If a transaction t obstructs another transaction then t may be aborted. After a transaction has aborted, the program state will be as if it never executed.
- **Retry:** After abortion, a transaction will usually re-try its execution, repeating the entire operation from the start.

2.1 Memory Layout

Figure 1 illustrates the memory layout used by our algorithm. Unlike previous STMs, we divide memory into three categories:

- **Public Memory**: can be accessed by any transaction. This area contains only objects.
- **Private Memory**: Each transaction has its own region of private memory that only that transaction can access. This memory is used for storing bookkeeping information, in the form of read and write descriptors (Figure 1).
- **Semi-Private Memory**: Each transaction t has its own region of *semi-private* memory is which it stores its transaction descriptor. This descriptor is only used when there is a conflict between two transactions, as discussed in Section 2.6.
  Given a pointer into a transaction's private memory, one can obtain a pointer to the corresponding transaction descriptor by zeroing the low order bits[3] of the pointer (Figure 1).

All data in private memory is allocated cheaply on a thread-local stack that is re-used by the next transaction to start on the same thread. This is possible since this data cannot be seen by other transactions and is not freed until the transaction commits.

Tests show that our STM design causes very little cache-miss overhead relative to a non-STM program. Cachegrind [18] 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. Semi-Private memory is tiny and we expect it to only be accessed very rarely.

[2] This might be a C struct, C++ object, dynamically allocated block, or other data construct. An object can of course contain pointers to other objects.

[3] If private memory grows very large, then this can instead point to an indirection to the descriptor.

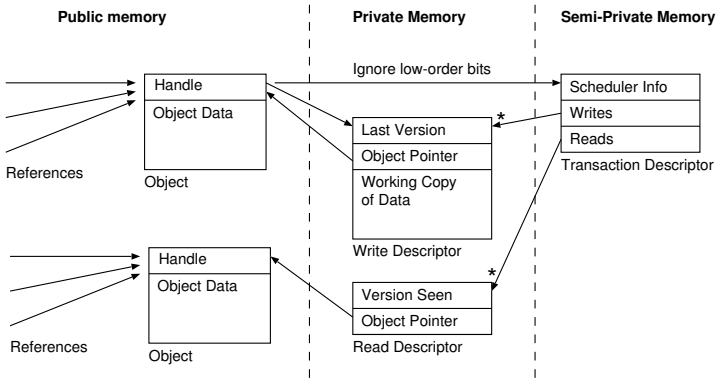

Fig. 1. Memory Layout

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[4].
- **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 1)

2.2 Writing Objects

To write to an object o, a transaction t follows the algorithm given in Figure 2. The function "get_write_pointer" is not visible to the programmer. It represents the code that must be executed every time the user's code writes to an object, in order to get a pointer to the object's writable data.

Object o's handle h tells t whether o is locked, and if so by who. If o is unlocked then the bottom bit of h will be 1 and h will be a version number. If o is locked, then the bottom bit of h will be 0, h will be a pointer to a write descriptor, and t can find the identity of the transaction that owns o by zeroing the low order bits of h.

To write to o, t must obtain an exclusive lock on o and produce a private working copy of o's data that it can write to. If o is already owned by t then t can use the working copy that it created earlier, otherwise t must lock o and create a new working copy.

If o is currently unlocked then t locks it, using an atomic compare-and-swap operation to replace o's handle with a new write descriptor. This write descriptor

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

```
get_write_pointer(t,o){
    h = o->handle;
    if(is_locked(h) && handle_owner(h) == t){
        return &(((writedesc*)(h->handle))->data);
    }else{
        writedesc = new write_descriptor;
        v = wait_for_object(o,writedesc);
        writedesc->last_version = v;
        writedesc->object_pointer = o;
        writedesc->data = o->data;
        return &(writedesc->data);
    }
}

wait_for_object(o,writedesc){
    while (true) {
        h = o->handle;
        if(is_unlocked(h)){
            if(writedesc != NULL)
                compare_and_swap(&(o->handle),h,writedesc);
            wakeup_any_in_wakeup_list();
            return h;
        }else{
            s = get_handle_owner(h);
            switch(ask_contention_manager_what_to_do(us,s)){
                case KILL_THEM:
                    send_abort_signal(s);
                    wait_for_ack_signal(s);
                    add_to_wakeup_list(s);
                    break;
                case WAIT_UNTIL_FINISHED:
                    wait_until_finished();
                    break;
                case WAIT_A_LITTLE:
                    sleep_briefly();
                    break;
            }
        }
    }
}

when_receive_abort_signal(sender,receiver){
    abort(receiver);
    send_ack_signal(sender);
    wait_for_wakeup_signal();
    restart_from_the_beginning(receiver);
}
```

Fig. 2. Pseudocode for obtaining a write pointer

has its *last-version* field set to the version that o had before t locked it and has its working copy initialised to the current version of o's data.

If o is currently locked by another transaction s then t asks the contention manager how the conflict should be resolved. The contention manager will tell t to either kill s, wait for s to complete, or wait for a short time and then try again.

If the contention manager says that t should kill s then t sends s a signal, asking it to abort. Once t has acquired o, t will send s another signal, telling it to restart. This second signal is necessary in order to prevent a race condition in which s restarts and re-acquires o before t has had a chance to do so.

It is the responsibility of the contention manager to ensure that deadlock and livelock cannot occur. Our contention manager avoids deadlock by placing transactions in a total order and decreeing that a transaction t waits for higher priority transactions and kills lower priority transactions. Our contention manager also avoids livelock; if t is the highest priority transaction then t must eventually succeed in acquiring o since eventually all transactions that conflict with t will have been aborted by t and will be waiting for t to acquire o. Other contention managers may use other techniques, including exponential back-off [11] and deadlock detectors.

There is no guarantee that the transaction s that t aborts is the same transaction that obstructed t; however this does not affect our progress guarantees. It may be that s committed and its descriptor was reused by a new transaction in the time between t observing that it was obstructed and t sending s an abortion signal. This is not a problem as the runtime system ensures that a transaction descriptor can only be reused by a transaction of the same or lower priority, so, while t can abort the wrong transaction, t can never abort a transaction of lower priority.

2.3 Reading Objects

```
get_read_pointer(t,o){
    h = o->handle;
    if(handle_owner(h) == t){
        return &(((writedesc*)(h->handle))->data);
    }else{
        readdesc = new read_descriptor;
        v = wait_for_object(o,NULL);
        readdesc->version_seen = v;
        readdesc->object_pointer = o;
        return &o->data;
    }
}
```

Fig. 3. Pseudocode for obtaining a read pointer

To read an object o, a transaction t follows the algorithm given in Figure 3. If t already owns o then t reads from the working copy that it created previously. If o is unlocked then t reads from o's current data. If o is locked by another transaction then t resolves the conflict using the same procedure as used for writing.

Every time a transaction reads an object, a note is made of the version that the transaction saw. These versions are checked at commit (Section 2.4) to ensure that another transaction did not write to an object while t was reading it.

2.4 Committing

```
commit(t){
    for each read descriptor r{
        if(r->object_pointer->handle != w->version_seen){
            abort(t);
            restart_from_the_beginning(t);
        }
    }
    block_abort_signal();
    for each write descriptor w{
        w->object_pointer->data = w->data;
        w->object_pointer->handle = next_version(w->last_version);
    }
    free_all_read_and_write_descriptors(t);
    unblock_abort_signal();
}
```

Fig. 4. Pseudocode for commiting a transaction

To commit, a transaction t follows the algorithm given in Figure 4. Transaction t first checks that no other transaction has written to any of the objects that it read. Then it makes its writes visible by copying across its working versions and setting the object handles to new versions.

There is a risk that t may be obstructed by a lower priority transaction s that writes to objects that t reads from. To avoid this problem, if t fails its read-check more than a fixed number of times, the runtime system tells t to retry using two-phase-locking for reads.

Unlike DSTM [11] the commit operation is performed by exposing each written object in sequence, rather than atomically flipping a status word that tells other transactions to look at t's private data. This allows us to avoid an extra indirection, keep bookkeeping data private, and immediately reuse descriptors in a new transaction without worrying that another transaction might be referencing them.

The downside of this policy is that, unlike DSTM, once t has started to commit, it must be allowed to finish committing, even if it is obstructing higher priority transactions. Any abort signals from other transactions will be ignored while a transaction is in its commit phase. In practice, this is not a problem as the commit phase lasts for a bounded time and is usually very quick. Even if t is descheduled while committing, the OS will usually reschedule it again quite soon. Standard priority-inheritance techniques can be used to ensure that a low-priority transaction is rescheduled promptly if high-priority transactions are waiting for it.

Like other STMs that use optimistic concurrency control for reads [11, 6], it is necessary for the runtime to periodically check that all transactions are valid and abort any that are not. 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 [6].

2.5 Aborting

```
abort(t){
    for each write descriptor w{
        w->object_pointer->handle = w->last_version;
    }
    free_all_read_and_write_descriptors(t);
}
```

Fig. 5. Pseudocode for aborting a transaction

To abort, a transaction t follows the algorithm given is Figure 5. The transaction iterates through all the objects that it has exclusive locks on and sets their handles back to what they were before it locked them.

2.6 The Contention Manager

Like DSTM [11], our STM separates the mechanism of implementing transactions from the policy of what should be done in the case of a conflict, placing the latter decision in a separate contention manager.

Our standard contention manager works by placing all transactions in a total order based on the lexical ordering of their user-assigned priority and the time at which they started. If two transactions have the same user-assigned priority then the transaction that started earlier will be considered higher priority.

If a transaction t is obstructed by a transaction s of higher priority then the contention manager will tell t to wait for s to finish (WAIT_UNTIL_FINISHED). If t is obstructed by a transaction r of lower priority then t will be obstructed

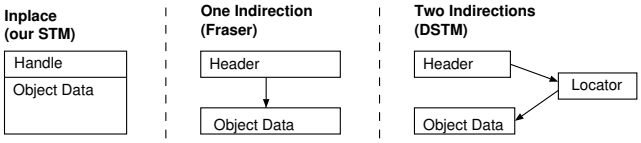

Fig. 6. How many indirections are needed to find object data?

to wait briefly (WAIT_A_LITTLE), and then forcibly abort r if r has not yet completed (KILL).

This contention manager guarantees that a transaction can never be permanently obstructed by a lower priority transaction, and guarantees that the system cannot deadlock or livelock. It does not however guarantee that the system is lock-free [12], since a non-terminating high-priority transaction can block all other transactions. If we did not allow a non-terminating hi-priority transaction to block lower priority transactions then we would have to allow low priority transactions to abort terminating high priority transactions – since one cannot generally tell whether a transaction will terminate.

In our design we have made the deliberate decision to chose to honor transaction priorities, rather than to provide lock-freedom. This may not be the right choice for all situations, and only experience will tell whether this is the right trade-off to make. If a programmer wishes to ensure that non-terminating transactions cannot block others, then they can impose a resource bound on their transactions.

3 Related Work

Although a number of people have proposed algorithms for software transactional memory, no STM that we are aware of has applied the key principles that motivate our design (Section 1.1). None have attempted to place the object handle in the same cache line as the object data, none have kept book-keeping information in private memory, and none have managed book-keeping information on a stack.

This does not mean that our design is necessarily *better* than previous work. Most of the previous STM designs have been significantly more theoretically elegant than our work, often having properties such as lock-freedom [12] and wait-freedom [10] that our design does not have. With our design we made a deliberate decision to trade off theoretical elegance for performance.

3.1 Object-based STMs

DSTM [11] represents an object using a pointer to a *locator*, which in turn points to the current object data. This triple-indirection requires a transaction to load three cache lines in order to access an object - even if it is only reading it (Figure 6).

Fraser's STM [6] uses optimistic concurrency control for both reads and writes. Fraser represents an object using a locator which points to the objects current data – requiring the loading of two cache lines for every read and write (Figure 6). Fraser estimates that this lower level of indirection is responsible for the bulk of his performance advantage over DSTM [6]. A key feature of Fraser's algorithm is its use of *helping*. If a transaction is obstructed by a another transaction then it will *help* the other transaction to commit. While this prevents transactions being obstructed by descheduled transactions, we have found that it degrades performance in practice (Section 4).

3.2 Word-based STMs

Word-based STMs [8, 23] perform concurrency control at the granularity of machine words, rather than objects. It is thus necessary to place book-keeping information in a separate location — requiring an additional cache line. Moreover, handles are typically placed closely together and so *false contention* may occur.

Moir's STM [17] is similar to both DSTM and word-based STMs. Moir divides his memory into fixed-size blocks, each of which is controlled by a handle word. Unfortunately for cache-performance, not only is the control information stored away from the block, but it is necessary to follow an indirection in order to find the current version of a block.

4 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 [6]. 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 [6].

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 [6] and Fraser's C re-implementation[5] of DSTM [11][6] — which are currently established as the two best performing STM implementations [14, 6]. More details of the setup are provided in Fraser's thesis [6].

Figure 7 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

[5] The original implementation is in Java, and so could not have been fairly compared.
[6] Using the POLITE contention manager.

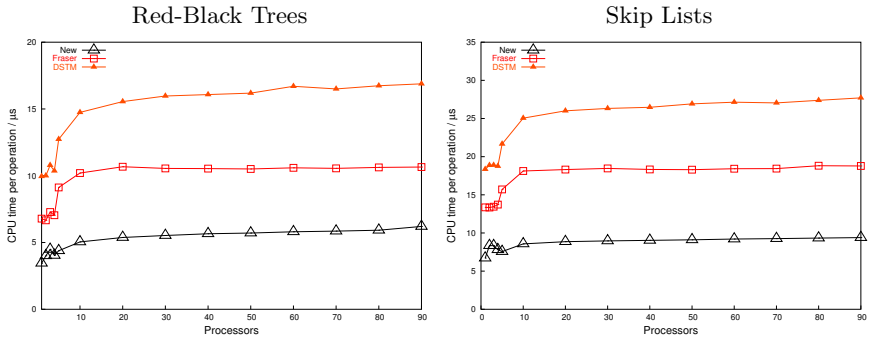

Fig. 7. Scalability under low contention (key space of 2^{19})

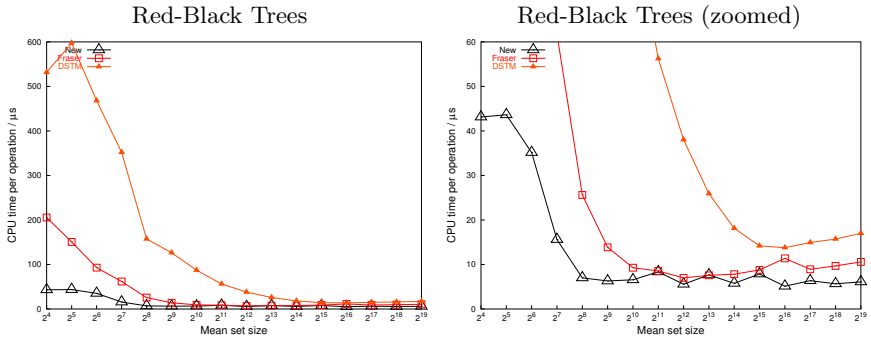

Fig. 8. Performance under varying contention (90 processors)

rarely attempt to read or write the same object [6]. 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 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 8 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 copes poorly and comes close to livelock, while Fraser's STM is almost five times slower than ours. It is possible that DSTM would perform better if a different contention manager was used [22].

We believe that the poor performance of Fraser's STM is due to its use of helping[7]. 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

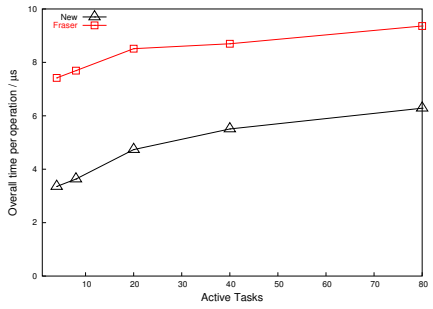

Fig. 9. Performance as task count increases (4 cores, key space of 2^{19}, red-black trees)

can end up with 90 processors all trying to perform the same commit operation and all fighting over the same cache lines.

Figure 9 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, even when there are 20 transactions per processor.

5 Conclusions

We have presented an implementation of Software Transactional Memory that significantly improves on the performance of previous algorithms while guaranteeing that no transaction can be permanently obstructed by a lower priority transaction. We believe that people interested in implementing transactional memory in software should seriously consider using our algorithm.

As further work, we plan to explore how our algorithm could be integrated with existing proposals for hardware transactional memory, such that software and hardware transactions can execute simultaneously and transactions can smoothly transition between hardware and software modes.

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.

[7] 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.

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 Richard Bornat, Michael Fetterman, Keir Fraser, Tim Harris, Maurice Herlihy, Gianlucca Iannaccone, Anil Madhavapeddy, Alan Mycroft, Matthew Parkinson, Ravi Rajwar, Bratin Saha, Ripduman Sohan, Richard Sharp, and Eben Upton for making useful suggestions.

References

1. Allen, E., Chase, D., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr, G. L., and Tobin-Hockstadt, S. *The Fortress Language Specification.* Sun Microsystems, Inc., July 2005.
2. Ananian, C. S., Asanavi'c, K., Kuszmaul, B. C., Leiserson, C. E., and Lie, S. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA'05)* (Feb. 2005).
3. Callahan, D., Chamberlain, B. L., and P.Zima, H. The cascade high productivity language. In *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)* (Apr. 2004).
4. Charles, P., Donawa, C., Ebcioglu, K., Grothoff, C., Kielstra, A., von Praun, C., Saraswat, V., and Sarkar, V. X10: An object oriented approach to non-uniform cluster computing. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Systems Languages and Applications (OOPSLA'05)* (Apr. 2005).
5. Cray Inc. *Chapel Specification 0.4.* Cray Inc, Feb. 2005.
6. Fraser, K. *Practical Lock Freedom.* PhD thesis, University of Cambridge, 2003.
7. Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.
8. 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).
9. Hennessy, J. L., Patterson, D. A., and Goldberg, D. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2002.
10. Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems 13*, 1 (Jan. 1991), 124–149.
11. 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.
12. Herlihy, M., and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)* (May 1993), ACM Press, pp. 289–301.
13. Kung, H. T., and Robinson, J. T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems 6*, 2 (June 1981), 213–226.

14. 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).
15. Marathe, V. J., and Scott, M. L. A qualitative survey of modern software transactional memory systems. Tech. Rep. TR839, University of Rochester, June 2004.
16. Marr, D., Binns, F., Hill, D., Hinton, G., Koufaty, D., Miller, J., and Upton, M. Hyper-Threading technology architecture and microarchitecture. *Intel Technology Journal 6* (2002).
17. 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.
18. Nethercote, N. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, Nov. 2004.
19. Rajwar, R., and Bernstein, P. A. Atomic transactional execution in hardware: A new high performance abstraction for databases. In *Proceedings of the 10th Workshop on High Performance Transaction Systems* (Oct. 2003).
20. Rajwar, R., and Goodman, J. R. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th ACM SIGMICRO International Symposium on Microarchitecture* (Dec. 2001).
21. Rajwar, R., Herlihy, M., and Lai, K. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)* (June 2005).
22. 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).
23. 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.
24. Steele, G. Parallel programming and parallel abstractions in fortress. In *Proceedings of the fourteenth conference of Parallel Architectures and Compilation (PACT'05)* (Sept. 2005).
25. Weikum, G., and Vossen, G. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control*. Morgan Kaufmann, 2001.