Summary:

The paper introduces *Transitional Locking II (TL2)*, a Software Transactional Memory (STM) algorithm, which tries to overcomes most of the safety and performance issues of former STM implementations.

As mentioned above, former STM implementations suffer mainly from two drawbacks: the need for a Closed-Memory Allocation System and for a Specialized Managed Runtime Environment.

Using a Closed-Memory Allocation System means, that memory used transactionally cannot be used non-transactionally and vis versa. Hence, the memory is partitioned into two disjunct regions: an non-transactional and a transactional region. These regions are fixed, meaning that memory committed to one region, cannot moved to another, thus virtually limiting the available memory for use.

A Specialized Manged Runtime Environment is needed to limit the execution of code, which could possibly cause unsafe behaviour due to the execution on an inconsistent program state. Such unsafe behaviour might include infinite loops or illegal memory accesses. To detect illegal memory accesses, the environment needs to catch traps and signals by the operating systems. For infinite loops even compiler support is required, since detection code has to be included in every loop, which could be speculatively executed.

The TL2 algorithm tries to overcome the mentioned drawbacks of former approaches by using a global version-clock and a two-phase locking scheme, that employs commit-time lock acquisition. The global version-clock for each transaction system is incremented by each writing transaction in the system. To avoid potential race conditions, the clock is only incremented by atomic compare-and-swap (CAS) instructions. Using CAS it is possible to detect contention among different threads while writing to the global clock. Additionally, each transactional memory location is annotated with a special versioned lock entry. The version number in this lock entry corresponds to the value of the global version-clock, on which the entry was last written to. Since the current transaction knows the version of the clock at the begin of the transaction, this value can be saved and used at commit time to easily detect interferences with other transactions and abort the current one. Especially, this even allows the algorithm to abort a transaction during its execution immediately when a changed value is read. Hence, transactions can be aborted *before* they could cause any unsafe behaviour, rendering the need for a managed runtime environment unnecessary. The lock itself is used to avoid a conflict between different transactions when finally updating the "real" memory location, since *all* locks have to be acquired by the target thread, before the commit can be executed (two-phase locking scheme). Remarkable for the algorithm design is the fact, that the lock doesn't have to be stored in the memory location, thus not restricting the algorithm on a particular memory allocation strategy. Furthermore, the algorithm maintains the typical STM data structures for a transaction, such as a write-set with pairs (address, value) for written and a read-set for read memory locations. As mentioned earlier, these values are used together with the version of the clock at transaction begin and the version of a memory location for detection of a conflict among running transactions.

Issues / Open Questions:

1. Apparently the clock is incremented as well for failing transactions, which could lead to an overflow, if there is a vast number of such transactions. This could finally lead to some kind of ABA problem, in which the safety of the algorithm isn't guaranteed.

2. The authors don't state explicitly how an object / a memory area is moved between the transactional and the non-transactional heap space.

   Additionally to prevent an memory location to be manually freed, before the transaction completes, the memory management routines (such as `malloc` and `free`) apparently need to be modified in some way. The authors, however, state explicitly, that the use of the standard memory management routines should be possible and is a huge benefit of their implementation. (Maybe this applies only to the underlying management algorithm, not the interface, which could be somehow overlayed?)

Summary S2

In the paper we discuss this week the authors present "Transactional Locking 2" as an approach for software transactional memory. It is the standard STM system which the technique we saw in the last week is based on. The problem with programming concurrent data structures is the loss of performance because of locks. TL2 tries to avoid this by locking only at commit time which is more efficient. As the authors claim TL2 is the first approach that overcomes the need for a closed memory system and for a special managed runtime environment.
TL2 uses a global version-clock that is incremented by each writing transaction. All memory locations have their own lock containing a version number. When a transaction starts the global version-clock is read and stored in a thread local variable. Then the speculative execution is run. A read-set containing addresses that are read and a write-set containing address / value pairs are maintained by adding instructions to every load and store in the original code. Then the transaction checks that the according lock is free and that its version number is less or equal to the recently stored value of the global version-clock. If this is the case the lock is acquired and the global version-clock is incremented. Simultaneously its current value is stored because it is needed for commit. After the validation of the read-set this step includes writing the values from the write-set to the correct memory locations and release their locks. Because performing all these steps for a writing transaction is costly TL2 provides low-cost read-only transactions in order to improve the performance. In this case only the first two steps have to be performed and no read-set has to be constructed and validated.

Questions / Opinion:

1. What does "bounded spinning" mean? (2.1.3)

2. Is this approach efficient in reality with such a high amount of overhead.

1 Summary

In this paper, the authors present Transactional Locking II (TL2), a software transactional memory algorithm, that allows both unbounded and dynamic transactions. TL2 features recycling of transactional memory to non-transactional memory and back, using *malloc* and *free* style operations or a garbage collected language. Furthermore, the algorithm efficiently avoids vulnerabilities related to reading inconsistent memory states, that for example might cause zombie transactions. The algorithm is designed to offer high performance for read-only transactions.
As part of their system, they implemented a shared global version-clock variable, that is incremented by each writing transaction, and used to detect recent changes to the data fields. As in the LiteSTM paper, every transaction holds a read-set of addresses and a write-set of address/value pairs; conflict detection on these sets is performed by a Bloom filter. TL2 offers two techniques on how locks are being applied. The first technique is called *per object* (PO), and assigns a lock to every shared object. The other technique is *per stripe* (PS). Here, based on a hash function, the memory is partitioned into stripes, and every stripe is assigned one lock. Although the PO technique outperformed PS in the benchmark, PS can be applied to various data structures without in-depth compiler analyses or intervention of the programmer.

2 Questions

In section 2.1 the authors claim that acquiring the locks in fixed sorted order to avoid deadlock is not worth the effort. Does this mean they allow deadlocks? If so, how do they resolve them?
Why did they not add another page to give the performance graphs more space?

Transactional Locking II
Summary

This paper introduces a new STM algorithm that makes use of a global clock for ordering of the transactions, and a fine-grained locking mechanism to maintain consistent memory states. Like STMlite algorithm, this also uses read-sets and write-sets. However, creation of zombie transactions are avoided in TL2, since algorithm operates only on consistent memory states. Hence TL2 doesn't need any special runtime environment support to detect and kill zombies. Algorithm also takes care of recovering freed memory space between transactional and non-transactional environments. Each memory location(granularity may vary) is augmented with a lock, and the time at which that lock was released last.

Initially, each transaction copies global clock value into a local variable. Read-set and write-sets are generated. Deviation from STMlite is that before each read operation a validation is performed to check if that location has been modified since the start of this transaction. If so, transaction is aborted. After this step, transaction tries to acquire locks on each address in its write-set. Global clock value is advanced after acquisition of all locks. After incrementing the clock, read-set validation is performed again to ensure that some other transaction has not modified these locations while this transaction was waiting to acquire lock for locations in the write-set. Values in the write-set are committed to memory. Corresponding clock value for each such modified location is also updated.

To advance the global clock, transaction might have to wait to acquire lock on that. To avoid this, algorithm proposes an optimization in which global clock is split into lock version number and thread ID pair. So a transaction has to update this clock only if version number differs.

For read-only transactions, algorithm performs better because there is no need of constructing read-set and write-sets.

Open Questions:

1. In section 2.2, Low Contention Global Version-Clock Implementation, algorithm allows reading the global clock while other transactions may be writing. Doesnt this lead to reading of the unstable memory content?
2. Transactions can not commit to memory until they acquire all the locks. Isn't there a possibility of starvation?

Transactional Locking II

Summary

This paper describes TL2, a software transactional memory (STM) system, as an advancement of its predecessor TL.

The system consists of the following components: a global version-clock which is a simple counter shared among all threads, a set of versioned locks associated with memory areas (either per object or per memory stripe), and – per thread / transaction – a read and a write set which are linked lists as well as two simple fields called the read and the write version.

The system replaces sequences of memory access by transactions that are then processed as follows (here, each validation fail causes a transaction abort):

The simple case is the read-only transaction. At the beginning of the transaction the value of the global version clock is stored in the read-version field. Given that, for each load it is validated that the clock version associated with the memory area read is $\leq$ the read version to assure that the memory has not been modified during the transaction execution.

The execution of the lock-saved writing (and reading) transactions is more difficult. After storing the global version-clock into the read-version field, we again start the execution of the transaction, but now fetching each write attempt in the write set list and each read action that is not covered by a former write of the same transaction (keyword bloom filter) in a read set, including the post-validation of lock versions from above. Then, all locks related to the write area of the transaction are taken, all versions are – again – validated against intermediate memory modifications, the global version clock is incremented to indicate our own memory modifications and, if all validation and re-validation succeeded, we commit all writes and release the locks again.

After a hint on how one might ease the contention at the global clock, the paper also in short discusses a mechanism to mix transactional with non-transactional memory which basically replaces any free() method (in the C/C++ case) with an equivalent call to a so-called quiesce function. This is controlled by the STM and ensures that the memory block is freed after the last transaction that attempts to modify it has released it.

They conclude with a short evaluation on an obviously well chosen example of a tree map which – given its recursive and by design partitioned structure – is a good candidate to be parallelized by TL2.

Open Questions

- During the speculative execution of a write transaction, post-validation performs version checks that are repeated afterwards, apparently in order to abort destroyed transactions sooner. Does that additional overhead (which is obviously not needed for soundness, see step 5) really pay off?
- As pure STMs seem to be beneficial in a rather small subset of applications or even only parts of those, are there attempts to heuristically detect those candidate code parts and apply the STM locally?

Summary Transactional Locking II

The authors introduce TL2, a transactional memory system. Its main contributions compared to previous systems are that it:

- allows to recycle memory between transactional and non-transactional parts of the program when using general free/malloc style allocation
- doesn't require a special environment to handle side effects

while still being competively performant. To achieve this, their approach features a global clock, and one lock per memory region. Regions also have an associated version number, used to track changes to regions. Like in the previous paper, each transaction maintains its own read and write set. Each transaction starts by creating a local copy of the current clock time. It then performs the actual execution; each time a read is encountered, the read adress is written into the read set before the data is read; if there's an entry in the write set for that address, the value stored there is used instead. Writes store a address/value pair in the write set. After each read, the version number of the associated address is checked; if it is larger than the clock value read during the start of the transaction, execution is aborted. As an optimization for read only-transactions (which must be explicitly annotated as such), the write locks are not aquired, but only checked. The version numbers are also compared, just as with read/write transactions.

Open Questions

- Does the post-validation after loads prevent zombie transactions?
- What exactly is the convinient order in which the spinlocks are aquired? Can it be optimized to prevent unnecesary aborts?
- The authors evaluate their approach on a concurrent red-black tree. A red-black tree most likely leads to rather small transactions, touching not that many memory regions. What happens when the transactions become larger?
- In the other STM paper, the datastructures used for read/write sets were more fancy than the linked lists used in this paper. Do the lists really not affect the performance, especially with large transactions?
- Are read--only transactions really that more common than read/write ones? Has this been empirically evaluated?

Summary S2

This week's paper is about Transactional Locking 2 (TL2), a software-based transactional memory system. STMs before TL2 were either based on locks, or lock-free. Both caused a lot of overhead and had certain requirements: lock-free systems require a closed memory system, lock-based systems either required closed memory systems too, or a specialized version of malloc() and free(). Furthermore, they needed a specialized managed runtime environment, as they operate on inconsistent states which could cause unsafe behavior.

TL2 claims to be the first STM which eliminates all of those requirements. TL2 makes use of a so called "version-clock", which is essentially a counter. At the beginning of a transaction the counter is read and stored locally. At commit it is checked if the global version-clock is still equal or less than the one stored locally. If this is the case, the commit is performed, if not the transaction is aborted. This global version-clock is incremented as soon as a transaction committed successfully.

Each transaction uses a read-set and a write-set which store information about memory accesses. Read-sets store the memory location for loads, write-sets store pairs of the address and the value written to this location. Each time a load or store is performed, the sets are updated accordingly.

It is also worth noting, that TL2 is a lock-based system. But in contrast to other STMs, the locks are only acquired while committing, making it more efficient than its competitors. Every memory location has its own lock, so only the memory locations which are updated are locked while committing.

Another specialty of TL2 is that there is an additional type of transaction just for performing read-only operations. Instead of performing 6 steps like a normal transaction, a read-only transaction only needs to perform 2. Particularly, it does not need to maintain a read-set which makes a read-only transaction highly efficient.

Transactional Locking 2

1 Summary

The paper introduces a STM system which utilizes global version management and global memory locking for dependence violation checking called TL2. The technique supports unbounded memory usage and dynamic memory allocation.
The work necessary to introduce TL2 driven speculative parallelism into a sequential or manually parallelized program using locks can be performed 'mechanically', meaning the translation is not dependent on the data flow and may be performed statically.
Further TL2 supports transactional memory to be recycled as non transactional memory by allowing dynamic malloc/free operations in transactions and it ensures that each transaction only observes consistent memory states (strong atomicity).
To lock memory regions accessed in a transaction TL2 may use two different methods, it may employ a lock for each object (PO) or it may use locks for partitions/stripes of memory (PS).
When executing a transaction TL2 will check for each load instruction if the memory was changed by another thread since the beginning of the transaction. All read locations are maintained in a read-setand all written memory locations are maintained in a write-set. Before a transactions commits both sets are validated again and the global version is incremented.
If a transaction is read-only there is no need to maintain read/write sets and the simple validation per load instruction is sufficient to find data dependence violation.
They evaluate TL2 by comparing it against other STM systems on the parallelization of a red-black tree implemented in java. In their examples TL2 performs better than the other STM systems most of the time but quite similar or even worse than the original TL system. They explain, that TL2 scales better with bigger structures since it has less overhead relative to the amount of data read transactionally because of the special handling of read only transactions.

2 Questions

- Is TL2 completely hardened against zombie transactions? They claim to avoid zombie transactions caused by reading inconsistent states but can zombie transaction be caused in other situations? (i.e. System Environment)

- In the second step of a write transaction described in Section 2.1 they state: "A load instruction sampling the associated lock is inserted before each original load, which is then followed by post-validation code checking that the location's versioned write-lock is free and has not changed." and that additionally "we make sure that the lock's version field is $\leq rv$ and the lock bit is clear."
  Where is the difference between these two tasks? Is there a difference between the "associated lock" and the "write-lock" of a memory region? Isn't "sampling the lock" the same thing as checking if the lock bit is set?