Summary:

The paper introduces STMLite a Software Transactional Memory (STM) system, which has been designed to specifically integrate with compiler parallelization techniques.

In order to support techniques such as thread-level speculation, the run-time system needs to detect potential dependency violations across different parallel executions. Read and write operations on the memory, which are issued by different executions, are regarded as part of a transaction of the specific execution, thus creating a so-called Transactional Memory (TM). Each transaction remains without any visible effect on the memory, until it has been committed. This makes it easy to allow different executions without the drawback of race-conditions. Each execution simply creates their own transaction, which can be committed separately. The dependency check will be finally performed during the commit-phase, which leads, in case of a conflict, to the abortion of the transaction or the write-back to the “real” memory. This guarantees that the memory is consistent at any point of the overall execution.

TM can be implemented in both, hardware and software, although hardware support for TM is less common and very complex to implement. STMs don’t require hardware support, but can cause a large runtime overhead, since the transactional state is maintained in software too. The STM needs to replace every load and store operation with a call to the specific STM operation. This may cause a large performance bottleneck, especially for the maintenance and validation of read sets in read-write transactions. Additionally, the use of global locks might be necessary to write back the “correct” data of a transaction. For example, taking the lock of an memory range might be necessary to avoid concurrent writes for different transactions, which leads to the use of atomic locking instruction during the commit-phase, thus making the commit very expensive.

STMLite tries to address these issues by avoiding explicit locking during memory write-backs and transactional data structures update. The locking problem is avoided by having a dedicated thread, the Transaction Commit Manager (TCM), which is responsible for performing the transaction. This makes the transaction data structures exclusively private to the TCM without the need for locking. The TCM ensures additionally that for a given time, only one write is allowed to a specific memory location, which makes locking here irrelevant too.

Another important point for the efficiency of STMLite is the efficiency of maintaining the transactional data structures by using hash functions, computing so-called signatures. These signatures are hash-based representations of all reads and write operations performed during the execution. When the dependency checking of different transactions is performed, it’s sufficient to compare the signatures inthe read set of one transaction with the signatures write set of another.

Notably for the efficiency of STMLite is the lazy conflict detecting scheme, which is employed during read and write operations. STMLite doesn’t check for a conflict each time a read or write access is performed, which would cause a significantly affect on the performance of the program. However the lazy detection scheme makes STMLite vulnerable to zombie transactions. Transactions are called a zombie,if they are always in conflict with another transaction, thus never being committed to the memory. To avoid such issues, the TCM considers transactions with a run time above a specific threshold as potential zombies and performs explicit conflict detection. If a potential zombie transaction shows conflicts, it is aborted.

Open Questions:

1. At the end of section 3.2 the authors state “[..] the signature operations can be inserted in a decomposed fashion [..], enabeling more aggressive compiler optimizations [..] with the aid of pointer alias analysis”. Does this hint, that (some) signatures could be generated by the compiler, since e.g. a pointer’s address could be determined statically?

2. STMLite tries to commit all transactions in the order in which they arrived at the TCM. Is this always deseriable? (except loop iterations) A single transaction could cause multiple following non-dependent transactions to fail, which could make reordering of transactions in some cases desirable.

Summary S1

The paper we are going to discuss this week deals with a light-weight software transactional memory system called STMLite which brings less overhead than other STMs. In this context an approach for automatic loop parallelization based on STMLite is shown.
Usual STMs involve a high amount of overhead because of the maintenance and validation of read sets when a read-write transaction is performed. This is because for every load the system has to check the timestamps for conflicts when committing. Furthermore when a transactional write is performed global locks are needed to commit data from the write sets to the original memory location. Besides these new advantages of STMLite it brings further efficiency improvements by loosening some limitations that are not important for the domain of loop parallelization like for example strong atomicity, handling of local variables and zombie transaction.
For managing the transactions STMLite provides a Transaction Commit Manager (TCM) running as a thread on a separate core of the CPU and ensures that at most one transactional thread is writing to a certain memory location. In order to realize this every transaction has to maintain a so-called transaction header including a read and a write signature which are computed during execution. When the commit point is reached these are copied to the precommit log. After that the TCM checks for conflicts between them and the entries in the commit log. If there is no such conflict it notifies the corresponding transaction to write back its write set to the commit log or otherwise to stop and restart the execution.
Using STMLite the authors developed a loop parallelization framework supporting DOALL-counted and DOALL-uncounted loops. Furthermore a tracking technique ensures that the right live-out registers are captured after a parallel loop execution.

Questions / Opinion:

1. Why has loop parallelization no need for strong atomicity?

2. What is the difference between lazy and eager version management?

3. Is it realistic to get such speedups with an STM when there is this high amount of additional instructions for bookkeeping?

1 Summary

In this paper, the authors present a low-cost software transactional memory model, called STMLite. Traditional STM systems have to use global locks to guarantee coherence between transactions, adding a considerable amount of overhead.
STIMLite eliminates the need for read log maintenance during transaction execution and explicit locking during memory writebacks. Furthermore, STMLite avoids having a single point of serialization. STMLite runs a dedicated thread for managing transactions, called the Transaction Commit Manager (TCM). The TCM is responsible for committing or aborting transactions. In addition, the TCM needs to make sure no concurrent writes are happening to the same address during writeback of a committed transaction.
Every transaction gets assigned a start version and a commit version. Therefore, the system uses a global clock, that is incremented by the TCM whenever a writing transaction commits. Additionally, every transaction holds a read signature and a write signature. These signatures essentially are hash-based representations of all read and write operations performed during execution of the transaction.
All transactions that have not yet been successfully committed reside in a pre-commit log, all successfully committed transactions are kept in the commit log. When a transaction finishes its execution, it sets its *Ready* flag. The TCM, which is constantly polling *Ready* flags of pre-commit log entries, will at some point find a transaction ready for commit. The TCM then performs a conflict check for this transaction. First, the transaction's start version is compared to the commit version of the commit log entries. If the transaction ready for commit does not overlap with any entry in the commit log, there is no conflict. Otherwise, the TCM has to perform an in-depth collision check, using the read signature of the transaction to commit, and the write signatures of the commit log entries. If the transaction to commit does conflict with any commit log entry, the transaction's *Abort* flag is set. Otherwise, the TCM finally has to ensure the transaction's writeback does not interfere with any other transaction that is currently writing back to memory, before setting the *Commit* flag.
The transaction to commit will at some point recognize that the *Commit* flag is set, and write back it's changes to memory, or it aborts.

In the remainder of the paper, the authors introduce their loop parallelization framework and adapt STMLite for parallelizing speculative DOALL loops.

SUMMARY: Parallelizing Sequential Applications on Commodity Hardware using a Low-cost Software Transactional Memory

Paper discusses about a lightweight memory transaction model called STMlite in which each parallel thread is treated as a transaction. STMlite obviates the need for strict atomicity, need for acquiring locks for a variable, and delays commit phase till the end, thereby increasing the performance.

Transactions are started together optimistically. Zombie transactions which read wrong data or taken wrong data paths might lead to infinite loops. Paper discusses a way to detect such transactions , and abort such transactions to ensure correct execution of the program. Because STMlite forces transactions to write to local logs before committing to memory, zombie transactions do not have to be rolled back.

STMlite uses a central Transaction Commit Manager to synchronize the commits of different transactions. In STMlite, each address read/written by a transaction is maintened using read-sets and write-sets respectively. Instead of raw addresses, their hash values are stored for performance reasons. Whenever a transaction tries to read a value, its write-set is consulted first to check if that variable has been written by the same transaction before. Once a transaction is finished, TCM checks if there are any coflicts between its read-set and variables in the write-set of an already committed overlapping transaction. Variables in the write-set are written to memory only if there are no such conflicts. And such writes may have to wait if another transaction's write-set is being written into the same locations in memory. If there are conflicts in the read-set, TCM uses abort signal to kill the zombie transaction. To avoid zombies that run into infinite loops, a threshold value is compared against the time for which a transaction is running. If that threshold value is exceeded by a transaction, abort signal is sent.

STMlite uses few additional data structures for loop parallelization. It maintains modification to a live-out variable by each iteration. If any of such parallel iteration encounters a break statement, it sets a variable which prevents the creation of any further iteration threads.

Parallelizing Sequential Applications on Commodity Hardware using a Low-cost Software Transactional Memory

Summary

The paper describes an approach to automatically parallelize loops in sequential programs using static analysis at compile-time combined with a transactional memory model to allow speculation for parallel loop execution at run-time. The latter one which is the key part of the paper is called *STMlite*. It is software based and therefore faces the problem of computational overhead introduced by the maintenance of its data structures.

The strategy to minimize those impacts on computation time is to build STMlite in a way such that it does not need a read log in order to serve valid memory data on read access resulting in non-blocking read accesses. That is achieved by the following design: The transaction commit manager (TCM) – that is core part of the system – runs on its own dedicated core of the system and receives memory read and write attempts (or a combination of both) from the program logic. These attempts are queued in a "'PreCommit Log". From there, the TCM pulls them and first of all serves the read part for each memory cell by either fetching the value directly from memory, or, if a write commit is already filed, the value that will be written. This is efficiently detected by some hash based "Commit Log" that contains write attempts that need to be executed after their read part has been served. A system of additional flags and occasional aborts/rescheduling of memory tasks ensures the soundness of the system but also seems to introduce some busy waits. A final commit back to the calling thread assures that the desired memory modification has taken place and that the execution can proceed.

In order to support speculation for the parallelization of not necessarily statically bounded loops – which the final aim – the system also utilizes a version system for its write commits. Due to that, it is able to cope with loop iterations that retrospectively break the loop and therefore need to abort "later" iterations that are speculatively, concurrently executed in parallel and to rewind their filed memory commits.

The authors use these mechanisms combined with a diversity of static analyses (e.g. some kind of not further described loop bound analysis) to obtain binaries of sequential programs that are able to benefit from customary (in 2009) multicore architectures. The effectiveness of that approach for certain settings is shown using a variety of benchmarks, showing that speculation can speedup sequential applications on parallel systems without specific hardware support.

Open Questions

- Is it the busy wait in the TCM commit polling mechanism which made them put the TCM on a dedicated core?
- Because of that, is the given approach suitable for mobile/energy efficient/reactive systems? Would hardware interrupts be a suitable alternative?

Summary S1

The authors present a novel approach to software transactional memory with focus on being usable for concurrency obtained by thread level speculation. They argue that existing systems incurred a high overhead for transactional stores and loads. Their new approach can obtain better results, partly because it gives weaker guarantees. They mention that e.g. for speculative loop parallelization, they don't have to care about strong atomicity guarantees; claiming that transactional and non-transactional execution can never happen at the same time. One might wonder, however, if this doesn't limits the amount of speculation which a compiler can use.

Key to the authors implementation is a central thread called TCM which takes care of managing the transactions. Each transactions maintains a set of signatures, which are hashes representing reads and writes. The TCM then uses this signatures to check if a transaction is valid, by checking for read/write conflicts with other transactions. To avoid locking, the TCM ensures that at the time of actual memory writeback, only one thread writes to a specific location. For this, an additional list containing all write signatures is used. Once the checks occured, the actual writeback is done by the transaction thread.

Open questions:

- Is it really necessary/benefitical to allocate one core to the TCM thread instead of leaving the scheduling to the OS? If so, why?
- Who manages the TCMs in case of TCM virtualization? Or how are conflicts between multiple transactions prevented when there is more than one TCM?
- How are the signatures actually computed?
- Can the busy polling to check ready flags in the TCM function be avoided? Also, isn't there a race between a transaction setting the ready flag and the TCM polling it?
- Is every zombie transaction required to have a conflict with another transaction at some point? If not, wouldn't the proposed zombie cleanup mechanism fail?
- The authors mention that they use a heuristic to select which loops they parallelize. Is that done offline or online?

Summary S1

This week’s paper introduces a lightweight software transactional memory system (STM) called “STMlite”, which aims to eliminate common shortcomings of standard STMs.

STMs are used for speculative parallelization: threads are running speculatively in parallel, storing their work in transactions. Transactions commit if they do not interfere with data from other transactions or abort and restart if they do. Those systems naturally introduce a lot of overhead which makes them inefficient to use for common applications, according to the authors.

STMlite was developed to deal with this problem. It is designed to throw a lot of the normally necessary checks and locks away by only managing a few speculative threads compared to standard STMs.

The paper focuses on loop parallelization and provides several improvements they could perform there. First of all in loop parallelization one can relinquish the need for strong atomicity - which is not trivially achieved - as at most only one parallel loop will run at a time.

Furthermore local variables in loops are not shared between the iterations as otherwise the loop couldn’t have been parallelized to begin with. Therefore, no special treatment is requirement.

Also, so called “zombie transactions” are left out of the picture as the only places they could occur won’t be parallelized anyway. Consequently, extra mechanisms to deal with those kind of transactions are not needed.

Once a transaction has finished execution, it needs to commit. For committing, STMlite provides a Transaction Commit Manager (TCM) which ensures that at most one transaction is writing to memory. Before a commit can be performed, one has to check whether or not it interferes with other transactions. This is done by writing the locations where read and/or write operations were performed to a so called “pre-commit log”. The TCM checks for conflicts in this log and tells the transaction to commit or to abort depending on the outcome of the check.

Parallelizing Sequential Applications on Commodity Hardware using a Low-cost Software Transactional Memory

1 Summary

The paper describes the implementation and usage of the novel software-transactional-memory system STMlite. By limiting the scalability and by defining the scope of their approach very strictly the authors achieve to reduce the overhead of their STM system. They set their goal to automatically speculatively parallelize loops for systems that use 2 - 8 parallel threads.
Further they avoid handling data dependences rooting from local variable usages in loop bodies by assuming they cannot parallelize those loops and they avoid they high memory locking overhead produced by other more generic STMs by moving the memory bookkeeping into another dedicated management thread. They claim that they do not need atomic memory access since all concurrent memory actions are performed through transactional memory.
While other STMs utilize a readlog which tracks what memory addresses are read by a transaction, STMlite uses a system where each transaction has a header composed of signatures which define what memory locations are accessed by using a hashing function.
All transactions are handled by a single transaction commit manager which performs the bookkeeping for the transaction headers, the conflict checking and keeps a clocking mechanism to evaluate the order in which transactions started and committed. Additionally the TCM will watch the runtime of each transaction and will forcibly abort transactions that are presumed to be zombie transactions that might never commit. Later they show how to implement an automatic parallelization algorithm using this STMlite, they split loops that can be parallelized in chunks and execute each chunk in parallel. They introduce some new bookkeeping measures for example to handle premature loop exit (break). They benchmark this implementation in a quite extensive evaluation showing that for most problems tested they reach speedup compared to other STMs and to the sequential execution.

2 Questions

- They state, that they can recognize overlapping memory access by using the signatures of each transaction. In paragraph 3.1 they talk about checking the signatures for 'hash collisions'. How is this hashing implemented to support these claims?
- They claim that they don't need atomic memory access but speak of polling flags which reside in memory accessed by multiple threads. How do they avoid race conditions in their own implementation?
- In paragraph 3.3 they describe that when checking if a transaction can commit, the TCM will wait for all conflicting transactions to finish writing before this transaction can write back. Doesn't this mean they ignore write after write dependences?