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

Mojtaba Mehrara Jeff Hao Po-Chun Hsu Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan Ann Arbor, MI 48109
{mehrara,jeffhao,pchsu,mahlke}@umich.edu

Abstract

Multicore designs have emerged as the mainstream design paradigm for the microprocessor industry. Unfortunately, providing multiple cores does not directly translate into performance for most applications. The industry has already fallen short of the decades-old performance trend of doubling performance every 18 months. An attractive approach for exploiting multiple cores is to rely on tools, both compilers and runtime optimizers, to automatically extract threads from sequential applications. However, despite decades of research on automatic parallelization, most techniques are only effective in the scientific and data parallel domains where array dominated codes can be precisely analyzed by the compiler. Thread-level speculation offers the opportunity to expand parallelization to general-purpose programs, but at the cost of expensive hardware support. In this paper, we focus on providing low-overhead software support for exploiting speculative parallelism. We propose STMlite, a light-weight software transactional memory model that is customized to facilitate profile-guided automatic loop parallelization. STMlite eliminates a considerable amount of checking and locking overhead in conventional software transactional memory models by decoupling the commit phase from main transaction execution. Further, strong atomicity requirements for generic transactional memories are unnecessary within a stylized automatic parallelization framework. STMlite enables sequential applications to extract meaningful performance gains on commodity multicore hardware.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features–Concurrent programming structures; D.3.4 [*Programming Languages*]: Processors–Code generation, Compilers

General Terms Languages, Algorithms, Design, Performance

Keywords Software transactional memory, Automatic parallelization, Thread-level speculation, Loop level parallelism, Profile-guided optimization

PLDI'09, June 15–20, 2009, Dublin, Ireland.

1. Introduction

As the scaling of clock frequency and complexity of uniprocessors has reached physical limitations, the industry has turned to multicore designs. Example systems include special purpose processors like the Sony/Toshiba/IBM Cell processor that consists of 9 cores [20], the NVIDIA GeForce 8800 GTX that contains 16 streaming multiprocessors each with eight processing units [30], and the Cisco CRS-1 Metro router that utilizes 192 Tensilica processors [12] and more general purpose processors including the Sun UltraSparc T1 that has 8 cores [22]. Furthermore, Intel and AMD are producing quad-core x86 systems today and larger systems are on their near term roadmaps. One of the most difficult challenges going forward is software: if the number of devices per chip continues to grow with Moore's law, can the available hardware resources be converted into meaningful application performance gains? Multiple cores readily help where threads are plentiful, such as web servers. However, they provide little or no gains for sequential applications. In fact, performance of sequential applications may suffer due to the use of simpler cores and smaller caches per core.

Many new languages have been proposed to ease the burden of writing parallel programs, including Atomos [5], Cilk [14], and StreamIt [41]. Despite these and other languages, the effort involved in creating correct and efficient parallel programs is still far more substantial than writing the equivalent single-threaded version. Developers must be trained to program and debug their applications with the additional concerns of deadlock, livelock, and race conditions. Converting an existing single-threaded application is often more challenging, as it may not have been developed to be easily parallelized in the first place. The lack of necessary compiler technology is increasingly apparent as the push to run general-purpose software on multicore platforms is required.

Techniques for parallelizing Fortran programs [3, 8, 15] usually target counted loops that manipulate array accesses with affine indices, where memory dependence analysis can be precisely performed. Unfortunately, these techniques do not often translate well to C and C++ applications. These applications, including those in the scientific and media processing domains, are much more difficult for compilers to analyze due to the extensive use of pointers, recursive data structures, and dynamic memory allocation. More sophisticated memory dependence analysis, such as points-to analysis [31], can help, but parallelization often fails due to unresolvable memory accesses.

Thread-level speculation (TLS) offers an opportunity for parallelizing C and C++ applications. With TLS, the architecture allows optimistic execution of code regions before all values are known [16, 21, 40, 45]. Hardware and/or software structures track register and memory accesses to determine if any dependence violations occur. In such cases, register and memory state are rolled

back to a previous correct state and sequential re-execution is initiated. With TLS, the programmer or compiler can delineate regions of code believed (but not provably) to be independent and amenable to parallelization [7, 11, 25, 27]. Profile data is often utilized during this process. The POSH compiler is an excellent example where TLS yielded approximately 1.3x speedup for a 4-way CMP on SPECint2000 benchmarks [25]. More recent work has shown that additional loop-level parallelism is covered up by a small number of register and control dependences, but can be unlocked with several dependence breaking transformations [44]. Outer-loop pipeline parallelism has also been identified as a key parallelization opportunity. Bridges *et al.* report a geometric mean of 5.5x gain on SPECint2000 (with variable number of threads up to 32) using decoupled software pipelining [4].

Proponents of TLS advocate hardware support for speculation generally in the form of transactional memory or similar techniques [16, 40]. Bulk tracking of memory dependences using signatures along with dedicated structures for managing speculative state provide an efficient environment for TLS [6]. However, the cost and complexity of implementing hardware or hybrid hardware/software TMs are high. With the notable exception of the Sun Rock processor, hardware support for TLS has not made it into mainstream multicore systems yet.

Alternatively, software TMs, or STMs, offer the opportunity for TLS support without any dedicated hardware. The first STM by Shavit *et al.* maintained read and write access locations in order to roll back in case of a transaction abort [35]. Many other works [19, 17, 26, 32, 10] proposed different forms of STM to tackle various performance and correctness issues involved in the STM paradigm. However, these STM implementations are far too expensive in terms of run-time overhead. For parallel applications, STMs typically result in visible slowdowns of 2x or more. The problem is even worse for compiler parallelized sequential applications where all the gains and more are typically wiped out by the STM.

STMs generally focus on flexibility to support a wide variety of transactions and scalability to enable many concurrent threads. STM control is fully distributed to the running threads. In this paper, we take the opposite approach by introducing *STMlite*, a lean and efficient STM specifically customized for compiler parallelization. With our focus on compiler parallelization, the goal is managing a modest number of speculative threads (2-8) that a compiler can realistically expect to find in C and C++ applications. Further, we focus on tightly integrating the STM with the compiler parallelization framework to ensure low overhead. Some requirements of more generic STMs such as strong atomicity [36] and special handling of local variables are not needed in this setting. Locks are removed by centralizing the TM bookkeeping on a single, perhaps idle, core. In this manner, bookkeeping tasks occur in parallel with transaction execution and the overhead on each work thread is minimized. Most importantly, centralized control obviates the need for locks and their associated overhead. The obvious downside of centralized control is the lack of scalability, but for a modest number of threads, large increases in efficiency are possible for both parallelized and multithreaded applications.

This paper is organized as follows. In Section 2, we discuss challenges in STM systems and customization opportunities based on our main goal – exploiting loop-level parallelism. Section 3 describes STMlite, our proposed STM model. We discuss our parallelization framework and the interaction between the compiler-generated code and STMlite in Section 4. In Section 5, we present our experimental results. Finally, Section 6 discusses related work and Section 7 concludes the paper.

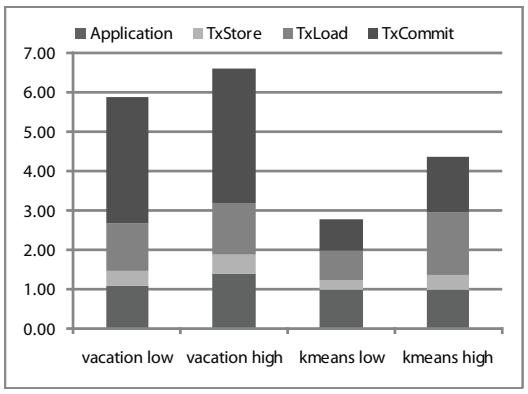

Figure 1. Single-threaded runtime breakdown of a state-of-art STM system on two STAMP transactional benchmarks.

2. Motivation

2.1 Challenges in Software Transactional Memory Systems

STMs have the advantage of requiring no additional hardware to run. However, since it is implemented entirely in software, it entails a large runtime overhead in maintaining transactional state. The high overheads of an STM are due to several reasons. The largest bottleneck in STMs is the maintenance and validation of read sets in read-write transactions. These sets keep track of every address read by a transaction, and are used to maintain coherence between transactions. For each load, the STM has to execute at least one transactional load and revalidate its timestamp when the transaction commits. As transactions read larger amounts of data, this overhead becomes substantial.

Secondly, global locks are necessary for transactions to write back their final "correct" data. During a transactional store, the address and value are stored into a write set, deferring any change in memory until commit. This allows transactions to remain coherent with each other, but adds a considerable overhead during commit time for obtaining the locks on these addresses and writing them back to their final location. The use of locks in the data write back is expensive as it involves atomic instructions.

In order to get a better understanding of what the major sources of overhead are in an advanced STM system, we performed an experiment on two STAMP benchmarks [28] using a state-of-art STM system - Sun's Transactional Locking 2 (TL2) [10]. We measured the time spent in each TM component of a single threaded transactional execution of these benchmarks using the TL2 library. A similar analysis has also been done in [29]. Figure 1 shows the result of this experiment. The vertical axis in these charts shows the execution time normalized to the sequential runtime. The vertical bars show the fractions of runtime spent in the main application, transactional commits (TxCommit), transactional stores (TxStore), and transactional loads (TxLoad).

The chart clearly shows the large overhead of read set maintenance in the Vacation benchmark, which has large transactions with many transactional reads. Keeping track of the read set causes considerable overhead, as depicted by the TxLoad portion of each bar. Additionally, the checks required during commit to maintain read set coherence are extremely costly [38], representing over half the runtime in Vacation with high contention. For the Kmeans benchmark, the overheads are not as severe because its read sets are smaller, but it still exhibits similar behavior.

2.2 Speculation Requirements for Loop Parallelization

There are several aspects of STM models that are crucial for correctness in general. However, we can loosen some of these limita-

tions and requirements in the loop parallelization domain to make the software-based speculation more efficient.

1. One of the shortcomings in STM models is the lack of strong atomicity guarantees, which raises correctness issues in parallel programs. Previous works [1, 36, 33] have addressed the issue of strong atomicity in STMs. While being effective, these approaches incur a non-trivial amount of complexity or performance overhead on the system. However, using STM for speculation in loop parallelization obviates the need for strong atomicity, because the execution consists of at most a single in-flight parallel loop at each point. Since all the code in the loop is running inside transactions, there can be no non-transactional code running at the same time as transactional code.
2. Special handling of local variables in a STM is not required for loop parallelization, because the loop iterations are not supposed to share any local variables on stack. Otherwise, they cause unresolvable cross iteration dependences, which prevent loop parallelization to begin with. Therefore, there is no need to have specialized transactional loads and stores for local variables.
3. Zombie transactions are transactions that have read a stale value or pointer from memory and have taken an incorrect code path which might lead to an infinite loop. One of the main sources of zombie transactions are loops with complicated linked-list operations. These loops are generally not parallelizable and therefore, we do not need to provide efficient and complicated ways for handling zombies in a STM for loop parallelization. However, to ensure correctness in other cases, we provide a mechanism for handling zombies in later sections that does not affect normal execution of transactions.

With these challenges in mind, we aim to tackle the two main sources of STM overhead: read-set maintenance and lock-based writeback mechanism. In addition, based on the specific speculation requirements in loop-level parallelism, we make simplifications to STMlite that makes it even more efficient.

3. STMlite

In this section, we describe our proposed STM model, STMlite. As was mentioned in Section 2, in traditional STM models, a considerable part of the execution time is spent in maintaining auxiliary data structures needed for providing correctness guarantees. In particular, one of the major bottlenecks is construction, maintenance, and frequent checking of read logs. The read log structure keeps track of the addresses (or objects in object-based implementations) read by each transaction. At transaction commit, these logs are walked over, and each address is checked for consistency. In addition, although the programmer does not have to deal with the subtleties of lock-based programming, thanks to the usage of atomic blocks and TM primitives, the performance of the underlying runtime system still suffers from the downsides of using locks in many implementations. In order to address these problems, and as a step towards stylized customization for speculation used in loop-level parallelization, we developed a new software-based model that eliminates the need for read log maintenance during transaction execution and explicit locking during memory writebacks.

We assign a dedicated software thread for managing the execution of the transactions involved in the main computation. This thread, which runs on an individual core, is referred to as the Transaction Commit Manager (TCM). Having a central commit manager provides an environment in which the manager is responsible for ensuring that, at any given time, at most one transaction is writing to a particular memory location. With higher numbers of transactions, there can be several coordinating TCMs with each TCM

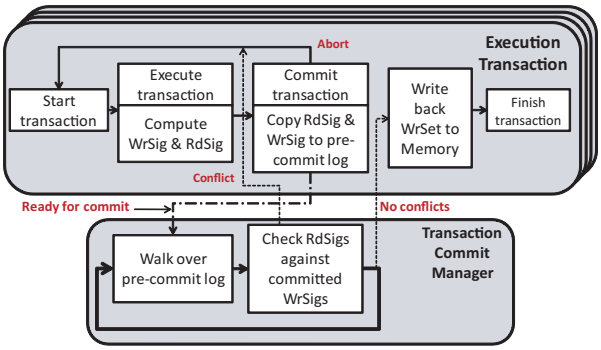

Figure 2. STMlite execution model. Solid lines denote execution flow. Dashed lines denote passing messages by signals or memory polling. The dash-dot line shows an indirect write/read relation (each transaction writes to a precommit log entry which is later read by the TCM).

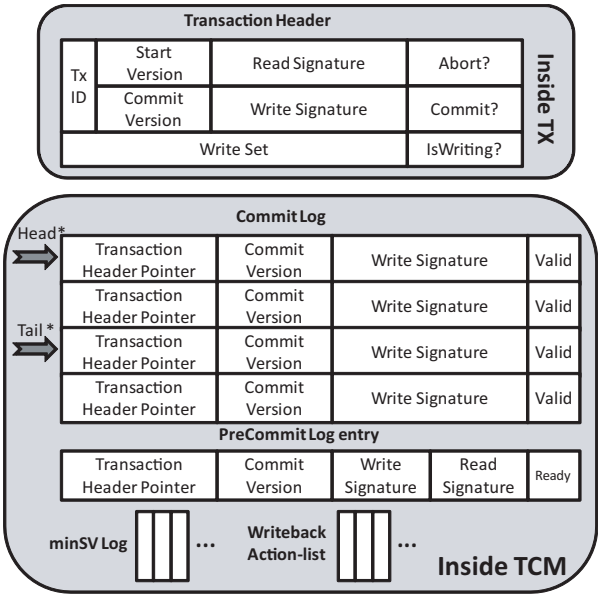

Figure 3. STMlite data structures. Each transaction has an individual header. The TCM has a single commit log and there is a precommit log for each execution core inside the TCM.

managing a group of execution transactions (TCM virtualization). In this way, we can avoid having a single point of serialization in highly parallel applications.

The STMlite model essentially consists of several execution cores for running individual transactions and a TCM core for maintaining transactional consistency in the system. In the following subsections, we explain in more detail how each step works.

3.1 Overview

Figure 2 summarizes the operation of STMlite. The top rectangle shows the execution flow inside each transaction. The bottom part is a summary of what happens inside the TCM.

Centralized management of individual transactions is made possible by using transactional read and write signatures, which are essentially hash-based representations of all reads and writes performed during execution. Using signatures in hardware was first proposed in [6]. However, unlike hardware, hash-based computations can become quite expensive in software. Therefore, choosing

```
TxLoad(Addr){
  if SignatureFind(Addr, Self->wrSig)
    Load the correct value from the wrSet
  else
    Load from memory
    SignatureInsert(Addr, Self->rdSig)
}
TxStore(Addr, Data){
  Store Data to the WriteSet
  SignatureInsert(Addr, Self->wrSig)
 }
```

Figure 4. Pseudocode for transactional loads and stores.

the right set of hash functions and the proper size for signatures is crucial in software systems to ensure minimal overhead and few false positives at the same time. In [18], hashing schemes are used to remove duplicates in the read-log and undo-logs of the "same" transaction. However, in order to use signatures for conflict detection between different transactions, a central manager is needed to check signatures against each other. In signature-based HTMs [6, 43], this is done in the coherence protocol. Here, it is done by the TCM.

Each transaction maintains a transaction header which is shown in Figure 3. The transaction header contains some information gathered during transaction execution and is used during the commit process. The main idea behind STMlite is that all transactions compute read and write signatures during their execution. At commit, they copy these signatures to a list called the precommit log (Figure 3). This log is basically a single-reader/single-writer buffer that is read by the TCM and written by transactions. Its operation is inspired by the reservation station in traditional out-of-order processors. Committed transactions reside in another data structure called commit log (Figure 3). The commit log is only updated and read by the TCM.

The TCM goes through precommit log entries and checks whether their read signatures have conflicts with the write signatures of overlapping already-committed transactions in the commit log. If there is no hash collision, the transaction is notified to start writing back its write set. Otherwise, the transaction aborts and restarts its execution. During the write back process, the TCM is responsible for preventing concurrent writes to the same addresses in memory. TCM operation is detailed in Section 3.3.

In order to keep track of the relative start and commit times of transactions, we use a global clock mechanism similar to [9]. The TCM increments the global clock value whenever a writing transaction commits. We define the start version for each transaction as the value of the global clock at transaction start. Likewise, the commit version is the value of the global clock at commit time.

3.2 Transactional Loads and Stores

Figure 4 shows the pseudocode for STMlite's transactional load and store functions. `TxLoad` first checks the transaction's write signature (`wrSig`) to see if this transaction has previously written to `Addr`. If so, it reads the data from the write set (`wrSet`) and returns. In order to avoid walking through the entire write set when the number of store-to-load forwarding instances is high, we added a hash map to each transaction that caches the latest stored addresses and values for quick retrieval. Therefore, if `Addr` is found in the write signature, this hash table is checked before walking through the write set. This helps to lower transactional load overhead in many cases. If the transaction hasn't written to `Addr`, data is loaded from memory and `Addr` is inserted into the read signature (`rdSig`). `TxStore` stores `Data` to the write set and inserts `Addr` to the write signature.

As can be seen, the only major extra overhead in transactional loads and stores is due to the signature insert and find operations,

```
TCM() {
  for entry precommitTX in PrecommitLogs
    if (precommitTX.Ready)
      if (ConflictCheck(precommitTX))
        Grant commit permission to precommitTX
      else
        Abort precommitTX
}

ConflictCheck(precommitTX) {
  for entry committedTX in CommitLog {
    if (precommitTX.startVersion
                < committedTX.commitVersion)
      if HashCollision(precommitTX.rdSig,
                       committedTX.wrSig)
        return 0;
  }
  if !(precommitTX.readOnly){
    Go through WBActionList
    wait for concurrent conflicting WBs to finish
  }
  return 1;
}
```

Figure 5. Commit management in the TCM.

though they remain low-cost for moderately sized signatures. Furthermore, the signature operations can be inserted in a decomposed fashion, separate form transactional loads and stores, enabling more aggressive compiler optimizations such as hoisting the signature calculations out of the loops with the aid of pointer alias analysis.

3.3 Transaction Commit Manager

As mentioned before, the TCM has two main data structures: the precommit log and the commit log (Figure 3). The commit log keeps track of committed transactions, and the precommit log contains transactions waiting to be served by the TCM. In order to reduce contention among transactions, a separate precommit log is assigned to each core. Figure 5 provides a summary of what happens in the TCM during runtime.

The TCM constantly polls `Ready` flags of precommit log entries (first `for` loop in the figure). When it detects a `Ready` is set, it reads the transaction's start version and checks it against the commit versions of commit log entries (in the `ConflictCheck` function). If the start version of the committing transaction is less than the commit version of a commit log entry, we know that their execution has overlapped at some point in time. Therefore, they should be checked for possible conflicts (in the `HashCollision` function). In case of a hash collision between the signatures, the committing transaction is instructed to abort by setting the *Abort* flag in its header. If the committing transaction passes the check against all overlapping commit log entries, it is safe to be committed. This is all that needs to be done for read-only transactions. Therefore, the TCM sets the *Commit* flag in the transaction header. It is not necessary to copy any information about read-only transactions to the commit log.

However, the mechanism is more subtle for writing transactions. Since we want to avoid having individual locks for writing back the write set to memory, the TCM needs to make sure no concurrent writes are happening to the same address during writeback. The TCM uses a secondary structure called the writeback action-list (`WBActionList`) for this purpose. The action-list has the same number of entries as the active threads in the system. At any given time, it contains the write signatures of the transactions that have passed the commit check in the TCM and are writing back their write set to the memory. When a transaction is ready to commit, the commit manager checks its write signature against all write signa-

tures in the writeback action-list. If there is no collision, the commit manager sets the *Commit* flag in the transaction header and writes the transaction's write signature to the action-list. Otherwise, it keeps checking the list until the colliding entry has finished writing back. An extra bit is added to the list to make sure that TCM does not repeatedly keep checking the signatures that have passed the collision test with the current committing transaction before. These checks could potentially become the TCM's bottleneck, though we did not notice any considerable busy waiting in our experiments. Subsequently, the TCM writes the necessary information about the committed transaction to the commit log, and moves on to checking the next entry in the precommit log.

Since commit log entries are no longer needed after all overlapping transactions have finished, a clean up mechanism is required to remove unnecessary entries. For this purpose, we maintain a minimum start version (minSV) log which contains the start versions of all in-flight transactions. Each transaction adds an entry to this log at start time and removes it at commit or abort. After each transaction commit or abort, the TCM starts from the commit log head entry and checks it against the start versions in the minSV log. If there are no overlapping in-flight transactions with the commit log head entry, that entry is removed and the head pointer is incremented. We keep doing this until the head entry in the commit log has an overlapping in-flight transaction. The reason we decided to use a circular buffer for the commit log (as opposed to a linked-list buffer) is to avoid the extra overhead of maintaining a linked list. Our commit log model only allows us to remove entries from the head of the log and add entries to the tail.

3.4 Individual Transaction Commits

When a transaction reaches the commit point, it fills up an entry in its precommit log with a pointer to its transaction header and sets the entry's `Ready` flag. Subsequently, it keeps polling *Commit* and *Abort* fields, waiting for them to be filled by the TCM. In order to avoid busy waiting at this point, we can relinquish the core[1] which is particularly useful when we have a larger number of threads than cores.

After a transaction receives commit permission from the TCM, it walks through its write set and writes back the actual values to memory. Because the TCM has already made sure that there are no concurrent transactions writing to the same locations, the committing transaction does not need to lock any memory locations. We chose to use a lazy version management strategy, because an eager version management system without locks introduces many complications in rolling back updates to memory locations after a conflict.

To minimize the overhead of individual transactional loads, a lazy conflict detection scheme is employed. This works particularly well for speculation support in loop parallelism, because minimum transactional load overhead is important for gaining performance from parallelizing loops. Furthermore, conflicts are rare due to the smart loop selection, and trying to detect conflicts eagerly at each transactional load provides no extra benefit. In eager conflict detection mechanism, since transactions are checked for conflicts at each load and store, the possibility of having zombie transactions is really low. However, eager conflict detection incurs substantial overhead on individual transactional operations.

Lazy conflict detection makes STMlite vulnerable to zombie transactions. These transactions may never reach the commit point and the commit manager normally does not get the chance to force them to abort. As a matter of fact, zombie transactions are particularly bad for our implementation because their corresponding entries remain valid within the minSV log and prevent the other commit log entries from being cleaned up. However, we can exploit the minSV log to resolve the zombie transaction issue. Each time we go through the commit log reading the minSV entries, if the difference between the start version of a particular transaction and the global clock is more than a threshold, the TCM identifies the corresponding transaction as a potential zombie. Subsequently, the TCM checks the suspicious transaction's read signature against write signatures of the commit log entries (although it has not reached the commit point yet). If there is a conflict, the TCM forcibly aborts the zombie transaction by sending an abort signal. We have a signal handler in each transaction that calls the abort function whenever it receives the TCM's abort signal. Otherwise, the TCM concludes that the suspicious zombie was just a long running transaction and avoids aborting it. In this work, since we do not parallelize loops with complicated linked list operations (which are the main sources of zombies transactions), the possibility of having zombies is quite low in our framework.

[1] In Linux, this can be done using `sched_yield` function.

4. Loop Parallelization Using STMlite

In this section, we introduce our loop parallelization framework and customizations made to STMlite for parallelizing speculative DOALL loops. Our framework successfully handles loops with cross iteration control dependences (e.g., while loops) as well as normal counted loops.

The general structure of our parallelization framework follows the code generation schema used in [44]. However, using that framework without the extra hardware support imposes a large overhead on the execution time. At the same time, STMlite gives us the opportunity to simplify the parallelization framework by exploiting some of its underlying features that are already used for providing transactional correctness.

4.1 Baseline Parallelization Framework

The purpose of the parallelization framework is to distribute loop execution across multiple cores. In this framework, DOALL loops are categorized into DOALL-counted and DOALL-uncounted types. In DOALL-counted loops, the number of iterations is known at runtime, whereas in DOALL-uncounted loops, this number is dependent on the loop execution (e.g. while loops). In these cases, starting every iteration is dependent on the outcome of exit branches in previous iterations (cross iteration control dependence).

Figure 6 shows the detailed implementation of the framework. In this scheme, loop iterations are divided into chunks. The operating system passes the number of available cores to the application and the framework is flexible enough to use any number of cores for loop execution. An outer loop is inserted around the original loop body to manage parallel execution between different chunks. The main thread (`THREAD_0`), which runs the sequential parts of the program, spawns the required number of threads at the start of the application. When a parallel loop is reached, a function pointer containing the proper loop chunk along with necessary parameters is sent to each spawned thread and they start the execution of loop chunks.

In order to capture the correct live-out registers after parallel loop execution, we use a set of registers called `last_upd_idx`, one for each conditional live-out (i.e., updated in an if-statement). When a conditional live-out register is updated, the corresponding `last_upd_idx` is set to the current iteration number to keep track of the latest modifications to the live-out values. If the live-out register is unconditional (i.e., updated in every iteration), the final live-out value can be retrieved from the last iteration and no tracking by `last_upd_idx` is needed. It should be noted that loop chunks in the framework do not share any local memory variables on stack. Otherwise, the loop would have unresolvable cross iteration dependences and would be unparallelizable. This leads to one of

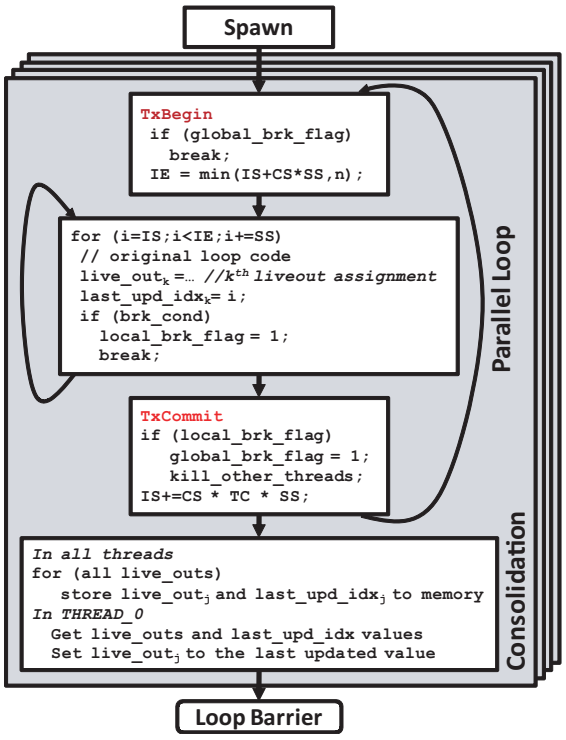

Figure 6. Overview of the parallelization framework (CS: chunk size, IS: iteration start, IE: iteration end, SS: step size, TC: thread count).

the simplifications we made in STMlite which is the elimination of the handling mechanism needed for speculative local memory variables. Following is a description of the functionality of each segment in Figure 6.

Spawn: `THREAD_0`, the main thread, sends the function pointer pointing to the start of loop chunks to the in-flight threads through memory. It also sends along the necessary parameters (chunk size, thread count, etc.) and live-in values.

Parallel Loop: The program stays in the parallel loop segment as long as there are some iterations to run and no break has happened. In this segment, each thread executes a set of chunks. Each chunk consists of several iterations starting from IS (iteration start) and ending at IE (iteration end). The value of IS and IE are updated after each chunk using the chunk size (CS), thread count (TC), and step size (SS). Each chunk is enclosed in a transaction using `TxBegin` and `TxCommit` function calls. In order to ensure correctness, an abort signal is sent to transactions running higher iterations if a conflict is detected.

One important requirement for parallelizing loop chunks is to force in-order chunk commit. This is necessary for maintaining correct execution and enabling partial loop rollback and recovery. The TCM in STMlite already provides the means to enforce ordering among transactions in the commit log. The same infrastructure can be used for in-order chunk execution as well. Therefore, there is no need for send/receive instructions and a scalar operand network as was used in [44]. However, some extra book-keeping data is required both for STMlite and the parallelization framework. Since this is mostly done in STMlite and it is almost transparent to the generated code, we explain these necessary steps in the next subsection detailing the interaction between STMlite and loop parallelization.

For uncounted loops, if a break happens in any thread, higher transactions are not aborted immediately because thread execution is speculative and the break could be false. Instead, the `local_brk_flag` variable in each thread is used to keep track of breaks in individual chunks. If a transaction commits successfully with its `local_brk_flag` set, the break is no longer speculative, and a transaction abort signal is sent to all threads. In addition, a `global_brk_flag` is set, so that all threads break out of the outer loop after restarting the transaction as a result of the abort signal. The reason for explicitly aborting higher iterations is that, if an iteration is started by misspeculation after the loop breaks, it could produce an illegal state. The execution of this iteration might cause unwanted exceptions or might never finish if it contains inner loops. This procedure of explicit handling of breaks has the benefit of avoiding zombie transactions, and although STMlite can handle zombies, this explicit handling has much lower cost.

Consolidation: After all cores are done with the execution of iteration chunks, they enter the consolidation phase. Each core sends its live-outs and `last_upd_idx` array to `THREAD_0` through memory. `THREAD_0` picks the last updated live-out values. All other threads keep waiting for chunks from other parallel loops later in the program.

Since the goal is to provide a low-cost software-based parallelization mechanism, most of the extra code is kept outside the main loop body, and is executed only once per chunk.

4.2 Interaction of Parallel Loops with STMlite

As mentioned in the previous subsection, in-order commit of individual loop chunks is crucial for correct parallel execution. In order to enforce that requirement, we add another log structured called the loop chunk commit log (LCCL) to the TCM. This log contains the loop ID of the last committing parallel loop and the chunk ID of the last committed chunk in that loop. The loop ID is assigned to each loop statically at compile time. It should be noted that our model allows only one in-flight parallel loop at a time by including a lightweight barrier at end of each chunk. Thus, there will be no problem if a parallel loop is invoked twice, because there is guaranteed to be no previous instances of this loop running. This is important, because if two in-flight loops have the same loop ID, they can completely distort each other's execution. The only problem is the case of loops in recursive functions. In this work, we do not parallelize loops with recursion. However, even in that case, a hash value based on the call site trace of the loop can be used to uniquely identify individual loops [34].

We reuse the initial value of IS (iteration start) which is computed at the beginning of each loop chunk as the chunk ID. When a loop chunk reaches the commit instruction, it writes its loop ID, chunk ID, chunk size, and the loop's first chunk ID to the precommit log. After the TCM reads in an entry from the precommit log, it performs one of the following two operations:

1. If the loop ID in the precommit log does not match the LCCL's committing loop ID, it infers that a new loop has started committing. Subsequently, it writes the new loop ID and the loop's first chunk ID to the LCCL. If the committing chunk is the first chunk in the loop, the TCM proceeds with the commit process. Otherwise, it just moves on to checking the next precommit log entry. This is because a chunk's commit process should not be started until all earlier chunks have been committed (i.e. have got commit permission from the TCM and started the writeback process).
2. If the loop ID of the committing chunk matches the entry in the LCCL, the TCM checks to see if the current chunk is right after the last committed chunk. If so, it proceeds with the chunk's commit process. Otherwise, it starts checking the next precommit log entry.

The above mechanism provides low-cost commit ordering by adding minimal complexity to the STMlite library. This integration of loop parallelization with STMlite leads to an efficient parallel loop execution platform.

5. Results

We set up two sets of experiments. First, we evaluated how STMlite performs in a typical transactional environment using the STAMP transactional benchmarks [28]. In the second set of experiments, we implemented the code generation part of the parallelization framework in the LLVM compiler [24]. Using this framework, a set of SPECfp benchmarks and several kernel benchmarks are parallelized. All benchmarks were written in C or converted from Fortran to C[2]. While the original Fortran applications can be parallelized using compilers such as SUIF [15], Fortran to C conversion introduces a large number of pointer variables, thus compiler analysis alone was insufficient to parallelize all applications. For SPECint benchmarks, as previous works have shown [44, 25], the level of loop-level parallelism is quite low, thus the overhead of using an all-software parallelization approach is too large to yield meaningful performance gains. More sophisticated parallelization techniques for integer applications are possible, such as those proposed by [4], and can lead to substantial gains. However, we have not implemented these transformations within our compiler system, yet they are orthogonal to what we are doing here.

5.1 STMlite on STAMP

We measured the performance of the STAMP benchmarks on a SunFire T2000 with an 8-core UltraSPARC T1 processor, running Solaris 10. We compare our performance with an implementation of a state-of-art STM - Transactional Locking 2 (TL2) [9]. Figure 7 shows the benchmark speedups on STMlite and TL2, both normalized to sequential execution. The number of cores in the STMlite results include the one extra core used for the TCM. For example, the 8 core results in STMlite have 7 computation cores and one TCM core. Thus, STMlite results start from two cores on the horizontal axis.

As can be seen, STMlite noticeably outperforms TL2 in both high and low contention executions of the Vacation benchmark. This is mainly because this benchmark has long transactions with a large number of loads. Therefore, the traditional STM performs poorly due to the high overhead of transactional loads and it can hardly achieve speedup over sequential even with 8 cores. However, using STMlite is particularly beneficial in these types of benchmarks. The overhead of transactional loads in our model is minimal due to the complete elimination of the read set. Furthermore, long length transactions and relatively low contention amortize the slight serialization effect that happens at commit time. Therefore, our model achieves about 2.5x and 3.1x speedup over TL2 with 8 cores, which is quite close to the speedup achieved by previous hybrid schemes [29].

STMlite follows the performance of TL2 in Kmeans, Labyrinth and Bayes. First, it should be noted that poor scalability from 4 to 8 cores in Kmeans and from 2 to 8 cores in Bayes is mainly due to the fact that these benchmarks contain heavy floating point computations. Since the UltraSPARC processor only has a single floating point unit that is shared by all processors, these floating point computations become the sequential bottleneck of parallel execution, especially with higher number of threads. We did not have any 8-core x86 processors available to investigate the scalability in a more fair environment. However, the Bayes benchmark scales fine from 2 to 4 cores on a quad-core x86 machine.

[2] Fortran to C conversion was done using the *f2c* tool with *-a* flag.

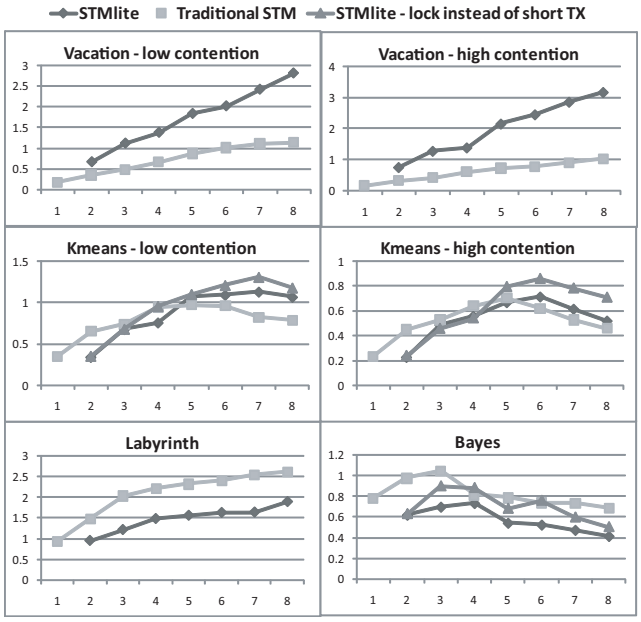

Figure 7. STMlite performance on STAMP benchmarks. The vertical axis shows the speedup compared to the sequential execution and horizontal axis is the number of cores. The number of cores in STMlite includes one core that is used for the TCM.

```
/* Original Kmeans Code*/ | /* Lock-based Kmeans Code */
TxBegin;                  | pthread_mutex_lock(&mutex1);
start = TxLoad(global_i); | start = global_i;
TxStore(global_i,         | global_i = start + CHUNK;
  (start + CHUNK));       |
TxCommit();               | pthread_mutex_unlock(&mutex1);


TxBegin();                | pthread_mutex_lock(&mutex2);
TxStore_f(global_delta,   | global_delta =
  TxLoad_f(global_delta)  |   global_delta + delta;
    + delta);             |
TxCommit();               | pthread_mutex_unlock(&mutex2);
```

Figure 8. Small transactions in Kmeans working on global data and their equivalent lock-based implementation.

The main reason STMlite performs similarly to TL2 in these benchmarks is the short length of transactions in Kmeans and relatively high rate of contention in Bayes and Labyrinth. So, the savings STMlite gets in transactional loads, transactional stores, and writebacks gets offset by the extra overhead of communications between execution transactions and the TCM. However, STMlite is still about 15% to 30% faster than TL2 in Kmeans for 4 and 8 threads. An interesting issue we found while looking through the performance bottlenecks of STMlite in Kmeans, is that there is a small transaction in the source code towards the end of the program that increments a global variable in all transactions (Figure 8). This part of the code causes a large number of transaction aborts in STMlite, which incurs a high cost considering the short transaction lengths. Whereas in TL2, since the library is acquiring locks for each address during writeback and uses a back-off mechanism if the lock is not free, there are fewer transaction aborts. In order to validate this observation, we placed a global lock around the transaction in Figure 8 and changed the transactional loads and stores to normal ones. The performance of the resulting execution

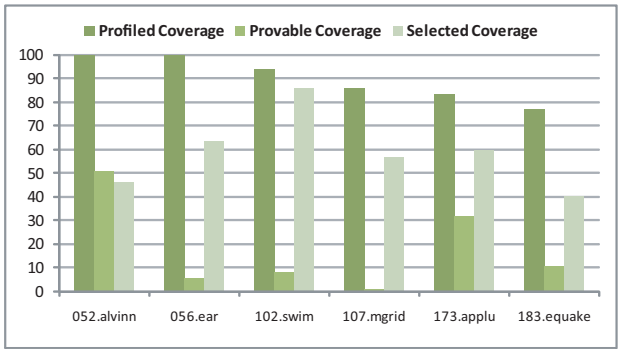

Figure 9. Profiled DOALL, provable DOALL and selected parallel loop coverage. The vertical axis shows fraction of sequential execution.

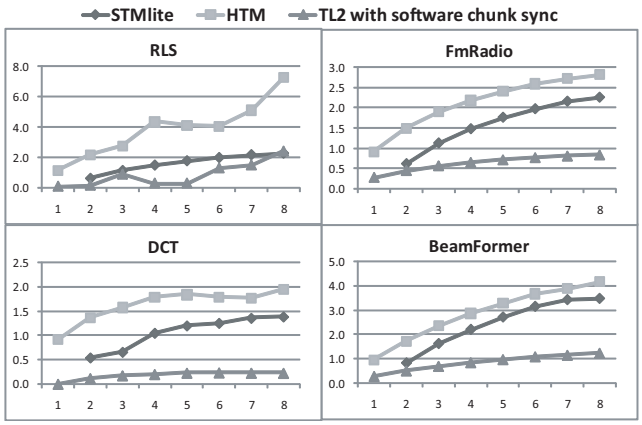

Figure 10. STMlite performance on automatically parallelized kernel benchmarks. The vertical axis shows the speedup compared to the sequential execution and horizontal axis is the number of cores. The number of cores in STMlite includes one core that is used for the TCM.

is also shown in Figure 7. This change in the benchmark did not affect the runtime for TL2 – since TL2 essentially does the same thing in short transactions. As can be seen in the figure, although STMlite still suffers from lack of enough floating point units, it performs better after replacing the small transaction with locks in Kmeans and Bayes.

5.2 STMlite on Parallelized Sequential Programs

Figure 9 shows the fraction of dynamic sequential execution that can be parallelized in several SPECfp benchmarks.[3] The first bar, profiled coverage, shows the fraction of sequential execution in loops identified as DOALL after profiling. The second bar, provable coverage, is the fraction of sequential execution spent in loops that could be statically identified as DOALL at compile time using LLVM's memory dependence analysis. As can be seen, a non-trivial percentage of DOALL coverage is obtained only after profiling, Finally the third bar, selected coverage, shows fraction of loops that were eventually parallelized.

It should be noted that not all the loops included in the coverage numbers are suitable for parallelization. There are many DOALL loops in these applications that do not contain any computation,

[3] These applications are a subset of SPECfp92/95/2000 that had moderate to high amount of loop level parallelism.

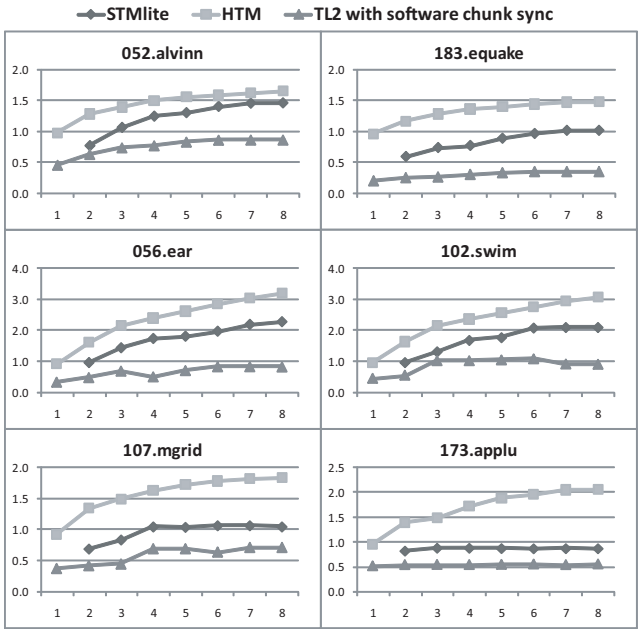

Figure 11. STMlite performance on automatically parallelized SPECfp benchmarks. The vertical axis shows the speed up compared to the sequential execution and horizontal axis is the number of cores. The number of cores in STMlite includes one core that is used for the TCM.

or the computation is not substantial. For instance, parallelizing a loop which initializes an array's elements to zero or increments all elements in an array, can not provide much benefit, since the overhead of parallelization would be more than the actual work in these loops. Therefore, we added a loop selection heuristic in our compiler which, according to the profile data, computes a "parallelizability" metric based on the total number of dynamic operations in the loop, number of iterations and total number of loop invocations in the program. The last bars in Figure 9 shows the total coverage of DOALLs that passed this metric.

We have parallelized all these loops using the framework introduced in Section 4.1. During the code generation pass, according to the static memory dependence analysis data, we performed a selective replacement of the loops' loads and stores with `TxLoad` and `TxStore` function calls. We essentially avoid changing loads and stores that can be proved to cause no cross iteration dependences.

As a step towards showing the effectiveness of our approach, we first tried the parallelization framework and STMLite on four kernel benchmarks: RLS, FMradio, DCT and beamformer. RLS is an implementation of recursive least squares filter which is used in system identification problems and time series analysis. DCT performs a discrete cosine transform and is used in image processing applications. FMradio and beamformer are two streaming applications from the StreamIt benchmark suite [41]. All these benchmarks have very high profiled DOALL coverage. Figure 10 shows the achieved speedup using STMlite and TL2. The STMlite results include the resource used for the TCM (1 extra core). Furthermore, since TL2 doesn't have any primitives for supporting chunk commit serialization, we implemented a software-based send/recv mechanism similar to [44]. Lastly, we estimated the results on a similar system with HTM support by replacing all transactional loads and stores with normal ones. This would represent a best-case HTM, and since we're only doing this for performance measurement, we ignore the possibility of incorrect execution due to the lack of proper spec-

ulation and we only take into account the performance numbers for executions that complete successfully. As can be seen, STMlite outperforms TL2 with software based chunk synchronization by as much as a factor of 3x in FMradio. In beamformer and DCT, STMlite follows the HTM results quite closely. For RLS, STMlite performs poorly compared to HTM results due to high number of transactional operations, yet it still achieves 2x speedup over sequential for 8 threads.

Returning to SPECfp, Figure 11 shows the speedup for these benchmarks. Runtime values are normalized to the sequential execution of the program. The figure shows that we achieve 0.6x to 2.2x speedup compared to sequential by going from two to eight cores.

One of the reasons for performance degradation in TL2 with software synchronization is the lack of library support for enforcing commit ordering in TL2. Adding this explicit software synchronization has a noticeably negative impact on the performance. Performance degradation would be even more in traditional TM systems with eager conflict detection, like [39]. As previous works have also suggested [38], workloads with transactions that have large readsets and low contention (similar to our parallelized sequential workloads), perform poorly with eager conflict detection. This is because eager conflict detection adds extra overhead to transactional loads and stores, but since conflicts are rare, it does not help improving the performance.

STMlite achieves decent speedup compared to HTM results and outperforms TL2 with software chunk synchronization in 052.alvinn, 056.ear and 102.swim. This is due to the lower overhead of transactional operations in STMlite which makes it quite efficient with moderate number of these operations. However, the relative STMlite achieved speedup, while being noticeably higher than TL2 with software synchronization, is quite low compared to HTM in other benchmarks. In SPECfp benchmarks, the parallelized loops contain a large number of memory operations that may cause cross iteration dependences based on the static analysis and therefore need to be transactified. Changing these operations to transactional versions causes the parallelized versions to become slow in some cases. Software-based speculation mechanisms are useful for parallelization in cases that the number of speculative variables is low, otherwise, the speculation mechanism amortizes the benefit caused by parallelization.

5.3 Effects of static memory analysis and signature sizes

To better understand the tradeoffs involved in compilation and execution parameters, we ran two other experiments. In the first experiment, we measured the achieved speedup with and without selective replacement of loads and stores with transactional versions. As mentioned before, LLVM's memory dependence analysis is used to avoid transactifying memory instructions that provably do not cause cross iteration dependence. Figure 12 shows the result of this experiment on the 052.alvinn benchmark. As can be seen, filtering out unnecessary transactional operations, while keeping the necessary ones, has a great impact on performance in both STMlite and TL2. This result further proves that software speculation systems are best suited for applications in which speculation is applied to a limited number of memory variables.

Our second experiment involves changing the signature size and studying the resulting performance impact. The effect of changing signature sizes on STMlite's performance is interesting. There is a subtle tradeoff involved in determining the right signature size. Larger sizes reduce the number of false positives and thereby reduce re-execution of correct transactions. However, at the same time, they lead to more time consuming signature operations. Since STMlite is dependent on these operations in several parts of the implementation, this can cause a noticeable performance degradation. Figure 13 illustrates this effect on the RLS kernel benchmark. Speedup values keep increasing up to signature sizes of 32, after which they start going down.

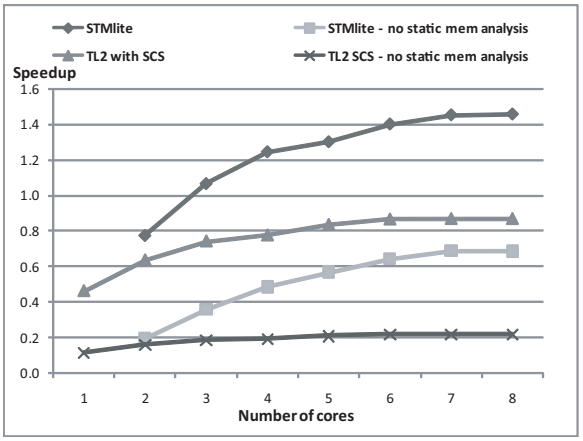

Figure 12. Effect of using static pointer analysis on speedup for 052.alvinn.

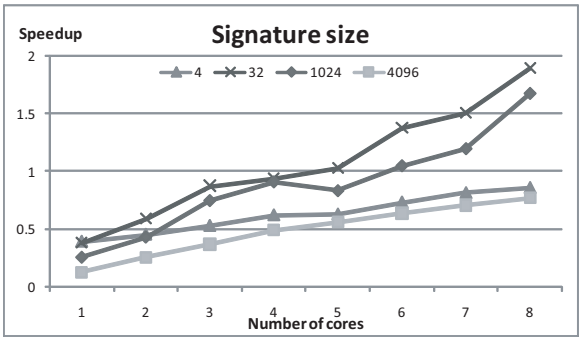

Figure 13. Effect of varying signature size on speedup for RLS.

6. Related Work

There is a significant amount of previous efforts in the area of transactional memory. Larus and Rajwar go through a detailed survey of different transactional memory techniques in [23].

In particular, Shavit *et al.* proposed the first implementation of software transactional memory in [35]. Several other works such as DSTM [19] and OSTM [17] proposed non-blocking STM implementations. A major part of non-blocking STMs is maintenance of publicly shared transaction structures which contain the undo information. In our implementation, the transaction structures only need to be visible to the TCM and individual executing transactions, keeping contention on those structures to a minimum. The authors in [18, 2] proposed a lock-based approach where write locks are acquired when an address is written. Also, they maintain a read set which needs to be validated before commit. In our STMlite design, no locks are required and correctness is guaranteed by the commit manager. Furthermore, we eliminate the need for the read set, which reduces the overhead of transactional loads and transaction commits. [10] proposes the Transactional Locking implementation which maintains a read set and a write set during transaction execution. Subsequently, at commit time, it acquires locks for each individual write set entry and writes back the data after the lock is secured. Also, the read set is checked during commit to ensure consistency.

There is also a large body of work in parallelization of sequential applications. Hydra [16] and Stampede [40] were two of

the earlier efforts in the area of general purpose program parallelization. The POSH compiler [25] uses loop-level parallelization with TLS hardware support. The authors in [44] proposed compiler transformation to extract more loop level parallelism from sequential programs. The compiler transformation part of that work is orthogonal to what we are doing and can be applied simultaneously here. Speculative decoupled software pipelining [42] is another approach that focuses on extracting parallelism from loops with cross iteration dependencies. In that work, they distribute a single iteration of the loop over several cores. The SUDS framework [13] performs automatic speculative parallelization of applications for the RAW processor. This system relies on the special architectural features in RAW to accomplish efficient speculative state management and synchronization, such as the scalar operand network. However in all these works, hardware TLS or transactional memory support and additional hardware mechanisms for synchronization are required. Whereas in this work, we are looking at a software-only solution and although our achieved speed up in some cases is lower than these works, we have the advantage of running our system on commodity hardware.

Ceze *et al.* [6] proposed the idea of using Bloom filters to represent read and write sets for transactions. They showed how, with specialized hardware, transaction state can be maintained through signatures with less overhead. This technique was extended in LogTM-SE [43] and SigTM [29], which are hybrid TM systems requiring no modifications to hardware caches. Our work uses the idea of storing Bloom filter-based read and write sets in software data structures, alleviating the need for the extra hardware. Authors in [18] use software hashing to remove duplicates in the read-log and undo-log of the ”same” transaction, whereas in STMlite, it is used for conflict detection between different transactions.

The most similar speculation management mechanism to ours is RingSTM [39] that uses a global ring structure to organize committing transactions. They use Bloom filters to represent read and write sets for transactions. However, because the ring is global, all threads face contention for ownership of the ring during commit, and prioritization is required to prevent starvation. Meanwhile, STMlite has thread local precommit logs and can relinquish the cores while the corresponding transaction is waiting for the commit manager to validate the transaction. Our commit log works in a round-robin fashion, ensuring all threads waiting to commit are serviced equally. Furthermore, in [39], the read signature is checked against several write signatures at each transactional load (eager conflict detection), which adds considerable overhead. However, in STMlite, transactional load overhead is minimal because the only extra operation added is insertion of the address in the read signature. This makes our model more prone to zombie transactions, but as mentioned in Section 3.4, the possibility of having zombies in parallelized loops is quite low, though STMlite can still handle them successfully.

Furthermore, we have customized STMlite to work for loop parallelization. This customization would be more complicated in RingSTM. The reason is that transaction commit is done by individual transactions after checking against the write signatures of ring elements. Therefore, if a loop chunk does not get a chance to commit in the first try (due to an unfinished previous chunk), there would be no efficient way of checking again later in the execution. The only way would be to use a back off mechanism and check back from time to time, which is inefficient. Whereas in STMlite, since the TCM is in charge of ordering loop chunks for commit, even if a chunk misses its chance, the TCM makes sure that it would be checked again in a timely manner.

An interesting, recently-proposed transactional memory model called FlexTM [37] adds mechanisms in hardware to coordinate read and write signature checking, speculative updates to caches and eager notifications to transactions about coherence events. They propose software mechanisms for deciding how to manage conflicts and for choosing appropriate conflict management and commit protocols.

7. Conclusion

As we move further into the multicore era, a major challenge in both hardware and software communities is exploiting the abundant computing resources made available by technology advancements. Automatic parallelization of applications is an appealing solution for utilizing these resources; however, parallelization efforts are commonly dependent on complex hardware changes such as adding speculation support. These changes are not yet popular among hardware manufacturers. On the other hand, software-based speculation support is still quite expensive in terms of performance to be widely used in parallel and parallelized applications. In this work, we try to tackle these issues from two closely related angles. First, we try to minimize the overheads of software based transactional memory models by decoupling and centralizing the commit stage in STMlite. We also eliminate the need for maintaining a read set during loads and checking them during commit. Secondly, we are able to lower the overhead of loop parallelization by reusing some of the underlying structures of STMlite. We have shown that our work outperforms the state-of-art transactional memory implementations on transactional benchmarks with large transactions while achieving similar performance in smaller transactions. Furthermore, we show that achieving real speculative speedup on sequential applications is possible without extra hardware support. We believe the value of this work lies in the idea that we make parallelization of sequential applications feasible on commodity hardware.

Acknowledgments

We would like to thank Dr. Tim Harris for his useful feedback on earlier drafts of this paper. We extend our thanks to anonymous reviewers for their excellent comments. We also thank Ganesh Dasika, Shuguang Feng, Shantanu Gupta and Amir Hormati for providing feedback on this work. This research was supported by the National Science Foundation grant CCF-0811065, Semiconductor Research Corporation (Task 1789.001), and the Gigascale Systems Research Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. Equipment was kindly provided by Sun Microsystems and Intel Corporation.

References

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 185–196, 2009.

[2] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proc. of the SIGPLAN '06 Conference on Programming Language Design and Implementation*, pages 26–37, 2006.

[3] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[4] M. J. Bridges et al. Revisiting the sequential programming model for multi-core. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 69–81, Dec. 2007.

[5] B. D. Carlstrom et al. The Atomos transactional programming language. In *Proc. of the SIGPLAN '06 Conference on Programming Language Design and Implementation*, pages 1–13, June 2006.

[6] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[7] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *Proc. of the 7th International Conference on Parallel Architectures and Compilation Techniques*, page 176, Oct. 1998.

[8] K. Cooper et al. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, Feb. 1993.

[9] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proc. of the 2006 International Symposium on Distributed Computing*, 2006.

[10] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *Proc. of the 2007 International Symposium on Code Generation and Optimization*, pages 21–33, 2007.

[11] Z.-H. Du et al. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proc. of the SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 71–81, 2004.

[12] W. Eatherton. The push of network processing to the top of the pyramid, 2005. Keynote address: Symposium on Architectures for Networking and Communications Systems.

[13] M. Frank. *SUDS: Automatic parallelization for Raw Processors*. PhD thesis, MIT, 2003.

[14] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, June 1998.

[15] M. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.

[16] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, Oct. 1998.

[17] T. Harris and K. Fraser. Language support for lightweight transactions. *Proceedings of the OOPSLA'03*, 38(11):388–402, 2003.

[18] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. *Proc. of the SIGPLAN '06 Conference on Programming Language Design and Implementation*, 41(6):14–25, 2006.

[19] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 339–353. Springer-Verlag, 2002.

[20] H. P. Hofstee. Power efficient processor design and the Cell processor. In *Proc. of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, Feb. 2005.

[21] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proc. of the SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 59–70, June 2004.

[22] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, Feb. 2005.

[23] J. Larus and R. Rajwar. *Transactional Memroy*. Morgan & Claypool Publishers, 2007.

[24] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[25] W. Liu et al. POSH: A TLS compiler that exploits program structure. In *Proc. of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 158–167, Apr. 2006.

[26] V. J. Marathe, W. N. Scherer, and M. L. Scott. Adaptive software transactional memory. In *Proc. of the 2005 International Symposium on Distributed Computing*, pages 354–368, Sept. 2005.

[27] P. Marcuello and A. Gonzalez. Thread-spawning schemes for speculative multithreading. In *Proc. of the 8th International Symposium on High-Performance Computer Architecture*, page 55, Feb. 2002.

[28] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of IISWC08*, 2008.

[29] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, pages 69–80, New York, NY, USA, 2007. ACM.

[30] J. Nickolls and I. Buck. NVIDIA CUDA software and GPU parallel computing architecture. In *Microprocessor Forum*, May 2007.

[31] E. Nystrom, H.-S. Kim, and W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proc. of the 11th Static Analysis Symposium*, pages 165–180, Aug. 2004.

[32] B. Saha, A. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *Proc. of the 39th Annual International Symposium on Microarchitecture*, pages 185–196, Nov. 2006.

[33] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *Proceedings of the OOPSLA'08*, pages 181–194, 2008.

[34] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, Oct. 1998.

[35] N. Shavit and D. Touitou. Software transactional memory. *Journal of Parallel and Distributed Computing*, 10(2):99–116, Feb. 1997.

[36] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *Proc. of the SIGPLAN '07 Conference on Programming Language Design and Implementation*, pages 78–88, 2007.

[37] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible Decoupled Transactional Memory Support. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, pages 139–150, 2008.

[38] M. F. Spear, V. J. Marathe, W. N. S. Iii, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proc. of the 2006 International Symposium on Distributed Computing*, 2006.

[39] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. pages 275–284, 2008.

[40] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proc. of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, 1998.

[41] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of the 2002 International Conference on Compiler Construction*, pages 179–196, 2002.

[42] N. Vachharajani, R. Rangan, E. Raman, M. Bridges, G. Ottoni, and D. August. Speculative Decoupled Software Pipelining. In *Proc. of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 49–59, Sept. 2007.

[43] L. Yen et al. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 261–272, Feb. 2007.

[44] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *Proc. of the 14th International Symposium on High-Performance Computer Architecture*, Feb. 2008.

[45] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 85–96, Nov. 2002.