Unifying Thread-Level Speculation and Transactional Memory *

João Barreto[1], Aleksandar Dragojevic[2], Paulo Ferreira[1], Ricardo Filipe[1]**, and Rachid Guerraoui[2]

[1] INESC-ID/Technical University Lisbon, Portugal
[joao.barreto,paulo.ferreira]@inesc-id.pt, rfilipe@gsd.inesc-id.pt
[2] EPFL, Switzerland
[aleksandar.dragojevic,rachid.guerraoui]@epfl.ch

Abstract. The motivation of this work is to ask whether Transactional Memory (TM) and Thread-Level Speculation (TLS), two prominent concurrency paradigms usually considered separately, can be combined into a hybrid approach that extracts untapped parallelism and speed-up from common programs.
We show that the answer is positive by describing an algorithm, called TLSTM, that leverages an existing TM with TLS capabilities. We also show that our approach is able to achieve up to a 48% increase in throughput over the base TM, on read dominated workloads of long transactions in a multi-threaded application.

1 Introduction

Multicore architectures are already the norm for most commodity computing devices. This trend calls for concurrent programs that expose enough parallelism to maximize the utilization of such increasing computational resources. Yet, concurrent programs are significantly more difficult to code than sequential ones.

In recent years we have witnessed increasing efforts from the research community to develop new emerging paradigms that ease the challenge of extracting parallelism from non-trivial programs. Thread-Level Speculation (TLS) [1, 2] and Transactional Memory (TM) [3] are perhaps the most prominent examples of such efforts. State-of-the-art solutions from both paradigms have already proved to extract considerable parallelism from a wide range of programs, while hiding complex concurrency issues away from the programmer [4, 5].

However, more than easily coding concurrent programs that yield *some* parallelism, we want concurrent programs that expose *as much parallelism as the ever increasing hardware thread count*. This goal becomes dramatically more challenging as affordable multicore machines include more and more cores each year. While 4-core processors supporting up to eight simultaneous hardware threads are already regarded as commodity hardware, 8-core, 16-core and even chips with tens or hundreds of cores promise to be an affordable reality soon [6].

* This work was supported by national funds through FCT - Fundação para a Ciência e a Tecnologia under projects PEst-OE/EEI/LA0021/2011 and specSTM (PTDC/EIA-EIA/122785/2010).

** Contact author.

Unfortunately, when examined individually, both TLS and TM have crucial limitations that hinder one's ability to extract high parallelism from most sequential programs.

On the one hand, TLS departs from a sequential program, breaks it into fine-grained tasks, and tries to automatically parallelize such tasks in a speculative fashion. For the sake of correctness, TLS ensures that any data dependencies stemming from the original sequential program order are respected in the speculatively parallelized execution. However, experience from the TLS systems proposed so far suggests that, for most programs, such data dependencies severely restrict the number of tasks that any TLS can parallelize effectively (i.e., without incurring in expensive rollbacks) [7]. Recent results show that even the most successful TLS systems rarely go beyond a relatively modest horizon of parallelization depth without rollback (e.g. less than 6 parallel tasks with SpLIP TLS [4]).

On the other hand, TM involves the programmer in the parallelization effort, by requiring him to explicitly fork the program into multiple threads. By carefully reasoning about the semantics of the application being parallelized, the programmer can thereby eliminate many data dependencies that were originally implicit across the original sequential program. Hence, in theory, higher levels parallelism are now attainable.

However, hand-parallelizing a program into many fine-grained threads is far from trivial. It requires a careful reasoning about the semantics of the application being parallelized, since the programmer must assert if the work performed by the parallelized tasks is actually commutative. Furthermore, it demands a thrifty understanding of the actual overheads of thread creation and management, so that the programmer can determine whether fine-grained tasks will actually introduce speed-up if parallelized. Hence, the programmer will typically choose a monolithic organization of coarse-grained threads. This is evident in the most representative TM benchmarks [8–10] and applications [11, 12]. In other words, the programmer is dissuaded from exposing the full fine-grained parallelism that the underlying application effectively contains.

Therefore, when facing the challenge of parallelizing a sequential program to run on a next-generation multicore machine, the programmer will most likely get disappointing results with either approach separately, TLS or TM.

While the research community places its efforts in exclusively improving one approach alone, we advocate that the time has come to question a hybrid direction: *Can TLS and TM be combined into a unified solution that would extract untapped parallelism (and speed-up) from our common applications?* If this hybrid approach proves to be feasible, programmers would first be asked to hand-parallelize their programs into coarse-grained threads using the TM paradigm. Each thread in the multi-threaded program would then be further parallelized into finer-grained parallel tasks, in a TLS fashion.

To the best of our knowledge, this paper is the first to give a positive answer to the above question, proving that TM and TLS do add up. We take a middleware approach, focusing on Software Transactional Memory (STM) and Software Thread-Level Speculation (STLS). Our main contribution is a unified STM+STLS middleware called *TLSTM*. *TLSTM* relies on standard techniques, such as compile time code inspection, to speculatively break each transaction

in a multi-threaded STM program into multiple tasks that will run in parallel. If no conflicts arise among the multiple tasks, then the transaction can commit earlier. *TLSTM* can even be more optimistic and speculatively execute future transactions of a thread, even when the current transaction in that thread is still active. If the speculation proves to be successful and every transaction commits, then further parallelism is accomplished.

TLSTM extends an existing STM, SwissTM [13]. The key insight is that a SwissTM transaction is used as speculative execution unit that supports two concepts: STM transactions (defined by the user) and TLS speculative tasks (automatically created at compile or run-time). An STM transaction is seen as a sequence of one or more TLS speculative tasks, which can run out-of-order in a speculative fashion, until they commit sequentially.

Our implementation of TLSTM[3] achieves up to a 48% speedup over SwissTM, when running on a multi-threaded benchmark of long transactions, with three speculative tasks inside each transactional memory thread. Furthermore, we also study scenarios where STLS does not provide any help to the STM runtime.

The remainder of the paper is organized as follows. Section 2 defines the STM+TLS model we wish to support. Section 3 then describes the TLSTM algorithm. We evaluate TLSTM on Section 4. Section 5 surveys related work on STM and STLS. Finally, Section 6 draws conclusions and discusses future work.

2 A Unified TM+TLS Model

We start by defining the novel model we want to support. Programmers can manually fork and join *user-threads* in their programs. Since critical sections might exist due to shared memory locations, programmers are also responsible for hand delimiting such critical sections as *user-transactions.* Together, the user-threads and their user-transactions comprise the hand-parallelized program. For presentation simplicity, we assume that user-transactions are flat (i.e., non-nested); however, the model can easily be extended to consider user-transaction nesting.

When executed, each user-thread's program will be further decomposed into speculative tasks, which will run in parallel in a speculative fashion. A task's boundaries lie either outside of a user-transaction's code, or inside a user-transaction's code. In case the task's boundaries lie inside of a user-transaction's code, they can either be the same as the user-transaction's boundaries or they can represent just a fraction of that user-transaction.

The life cycle of a successful task goes through a number of states: initially, the task is *running;* once the task has executed its last instruction, it is said to be *completed;* finally, it becomes *committed* when the task's effects become visible to all other tasks and cannot be undone. We say that a task is *active* if it is either running or completed. If the speculative execution of some task *tsk* is found to be inconsistent with the expected outcome of the sequential execution of *tsk*'s user-thread (causing an intra-thread conflict), or *tsk*'s execution is inconsistent

[3] Open source available at http://www.gsd.inesc-id.pt/project-pages/specSTM

with the execution of other user-threads (inter-thread conflict), then *tsk* must rollback, and is said to have *aborted*.

Hereafter, when a task tsk_1 runs code that precedes (in program order) the code executed by task tsk_2, we say that tsk_1 *is from the past of* tsk_2 (whereas tsk_2 *is from the future of* tsk_1). Within the collection of active tasks of a user-thread, we distinguish one *current task,* which corresponds to the earliest running task of the user-thread. This corresponds to the task that is running the code that the user-thread would be running if executing with no thread level parallelism. All the active tasks in the future of the current task are called *out-of-order tasks*. As soon as the current task completes, the next task in program order becomes the new current task.

The accesses performed by tasks belonging to the same user-thread must behave as if they ran sequentially. More precisely, our model ensures that any read from a task tsk_1 observes all the writes that tasks from tsk_1's past should perform and does not observe values written by future tasks.

Our model ensures that user-transactional correctness (more concretely, the opacity criteria [14]) is preserved across user-transactions, even when user-transactions are actually executed by multiple tasks running out of order. Only after every task belonging to the same user-transaction has completed its execution can the user-transaction commit.

3 TLSTM, A First Unified STM+TLS Middleware

A first naive solution to the STM+TLS problem that one might consider would be to simply run TLS on top of each thread of an existing multi-threaded STM application (either software-based or hardware-based), with no modifications on any of the two components. However, the correctness of conventional TLS algorithms relies on the assumption that the underlying (single-threaded) program exclusively accesses thread-local variables. Clearly, this no longer holds in the STM+TLS model.

Hence, we must look towards an integrated approach, i.e. a single runtime that fully supports the unified TM+TLS model that Section 2 introduced. TLSTM is a hybrid runtime that extends an existing STM, SwissTM [13], with TLS capabilities in order to support the unified STM+TLS model we described. SwissTM is a state-of-the-art STM system that supports optimistic read-write conflict detection and pessimistic write/write conflict detection, which has been shown to outperform other relevant STMs.

Therefore, before presenting TLSTM, Section 3.1 starts by describing the baseline SwissTM algorithm. Section 3.2 then discusses the hard challenges that we needed to tackle when leveraging SwissTM with support for TLS. Section 3.3 finally introduces the TLSTM algorithm.

3.1 The baseline STM: SwissTM

In SwissTM, a global commit counter, called *commit-ts*, is used as a wall clock that is incremented by every non-read-only user-transaction on commit. SwissTM maintains a global lock table. Each location is mapped to a pair of locks, *r-lock* (read) and *w-lock* (write) from the global table. *r-lock* can either hold a

version number or the *locked* value. *w-lock* can either hold a write-log entry or the *unlocked* value. Any user-transaction wishing to write must first obtain the location's *w-lock*. This eagerly prevents write/write conflicts between user-transactions.

Writes are performed in temporary copies, and only applied on the actual location once the associated user-transaction commits. During commit, the user-transaction acquires the *r-lock* of each location that the user-transaction wrote to. This prevents other user-transactions from reading the written locations and, as a result, observing inconsistent states. Upon successful commit, the *r-lock* is unlocked and contains the new *commit-ts* value, hence denoting the instant where the new value of the location was made visible to every other user-transaction.

SwissTM uses lazy counter-based validation [15, 16] to detect read/write conflicts. Each user-transaction maintains a version timestamp, *valid-ts*, denoting a point in the logical commit time for which all the values that the user-transaction has observed so far are guaranteed to be valid. Whenever the user-transaction reads some location that has a higher version than its *valid-ts*, the user-transaction needs to extend its *valid-ts* to the version being read. This requires traversing the user-transaction's read-log to validate that each version read so far remains valid at the new *valid-ts*, i.e. it has not been overwritten in the meantime.

3.2 Leveraging SwissTM with Thread-Level Speculation: Main Challenges

The key insight is that what used to be a SwissTM transaction is now used as a task in TLSTM. A user-transaction will now consist of a sequence of one or more tasks, which are automatically/manually created at compile or runtime, and can run out-of-order in a speculative fashion.

However, extending an STM (such as SwissTM) with TLS support is far from trivial. In the following, we discuss all the main challenges and give an intuitive overview of how TLSTM tackles each of them.

Ensure low overhead. A major part of the unified runtime's overhead comes from conflict detection. Besides the inter-thread conflicts of SwissTM, TLSTM must also detect and resolve the intra-thread conflicts resulting from TLS. There are several TLS techniques we can employ, but bluntly doing so would incur unacceptable overheads in conflict detection. Thus we must ensure that the overhead of the unified runtime is much smaller than the overhead of the sum of its parts. This requires that TLSTM reuses most of SwissTM's data structures and procedures, and adds minimal complexity to conflict detection.

Conceptually, two types of intra-user-thread conflicts may arise: *write-after-read* (WAR), where a task writes to a location that a future task already read from; and *write-after-write* (WAW), where a task wants to write to a location already written by a future task.

WAR conflicts are discovered through a new task validation procedure that starts by validating the read-log inherited from SwissTM, which records the reads performed from committed state. Then this procedure validates a new task-read-log, similar to SwissTM's read-log, which records the reads performed

from writer tasks of the task's past. First, this validation checks if any of the values read from committed state were speculatively written by running tasks from the task's past. Second, this validation must ensure that each value the task has read from a past writer task has not been updated by a task from the writer task's future. If any of these situations has occurred, we abort the task performing validation. We check the need for this validation at read, write and commit time.

In the case of WAW conflicts we cannot rely on SwissTM's write-write conflict handling alone. If we did so, we could easily have intra-thread deadlocks when a future task wrote to a location and waited for its past tasks to complete in order to commit. This task might be stuck waiting forever if a past task wishes to write to that same location, which in turn would be indefinitely waiting for the location's write-lock to be released.

This problem requires a very small addition to SwissTM's write-write conflict handling. If a task wishes to acquire a write-lock that is held by a past task from the same user-thread, the task wishing to acquire the lock aborts. If, otherwise, it was a future task that write-locked the location, that future task will be signaled to abort. By following this task contention management approach we have only one running task writing on a certain location at a time.

Before committing a task, TLSTM must also ensure that all past tasks have completed and cannot be aborted because of intra-thread conflicts. TLSTM achieves this by *serializing commits* of tasks belonging to the same user-thread, along with the previously explained intra-thread conflict detection.

We ensure this by associating each task with a monotonically increasing *serial number* in the scope of the task's user-thread, and once the commit step of some task starts, the task waits for tasks from the same user-thread with lower serial numbers to complete before committing.

Transaction commit. The transaction commit procedure in TLSTM needs to take into account the reads and writes performed by every single task of the user-transaction, in order to preserve atomicity. Thus, transaction commit differs substantially from SwissTM's, since it is performed by the last task of the user-transaction in program order, which we call the commit-task.

When committing a user-transaction, the commit-task validates the reads of all tasks of the user-transaction. When committing to memory the values of a write user-transaction, the commit-task needs to update all values written by all tasks of the user-transaction. Intermediate tasks take no part in validating reads or updating writes of the user-transaction, while waiting for the commit-task to commit the user-transaction.

Transaction abort. As in commits, transaction abort in TLSTM involves a coordinated effort from the multiple parallel tasks that comprise the user-transaction. This challenge is especially difficult as some of such tasks might still be running when the abort decision is taken.

When a task receives the abort transaction signal it waits until all tasks from its user-transaction have received that signal. Then, the last task of the aborting user-transaction clears every write-lock of all tasks in its user-transaction and

resets the tasks' state to their last known correct values. Finally, the last task signals every past task of its user-transaction to restart, before restarting itself.

Preventing inter-thread deadlocks. Since TLSTM supports multiple threads, TLSTM must ensure that there are no deadlocks between tasks of different user-threads writing to several locations.

Imagine the scenario of an application with two user-threads running two tasks each ($T_{A,1}$ represents task 1 from thread A and so on): $T_{A,1}$, $T_{A,2}$, $T_{B,1}$, $T_{B,2}$. $T_{A,2}$ holds the write-lock to location X and $T_{B,2}$ holds the write-lock to location Y. Assume that $T_{A,1}$ wants to write to Y and $T_{B,1}$ wants to write to X and that TLSTM inherits the inter-thread contention manager from SwissTM. Hence, when a task holds the write-lock of a location and tasks from other user-threads want to write to that location, they have to wait for the current writer to commit.

Both tasks $T_{A,1}$ and $T_{B,1}$ will be blocked waiting for the lock owners to abort or commit, but the contention manager will not signal the lock owners to abort and the lock owner tasks will not commit because they are waiting for their past tasks to complete (as a consequence of *serializing commits*).

In order to solve this problem, the inter-thread contention manager must be task-aware, so that it makes decisions according to the user-thread's set of tasks and not for each task individually.

Whenever an inter-thread conflict is detected between two tasks, the contention manager aborts the more speculative one, i.e. the one that has fewer tasks from its past that are still running. Not only does this strategy favor tasks with higher probability of completing successfully, but it also prevents starvation. If contending user-transactions have the same number of completed tasks, then TLSTM employs traditional STM contention management algorithms.Currently, TLSTM implements the two phase greedy contention manager for this case.

Inconsistent Reads. TLS and STM can induce out of order reads that may trigger undesirable effects. For example, picture $T_{A,1}$ writing NULL to location X and then allocating a new object to X. If $T_{A,2}$ reads the intermediate value of X it will crash because of a NULL pointer exception.

While in STM these are prevented through atomicity, as read operations only read values from the user-transaction itself or from committed state. In TLS values can be read from running tasks, which may result in reading intermediate and inaccurate values [4].

Therefore, in a unified runtime it is not possible to prevent all inconsistent reads, so TLSTM needs to detect and take care of those coming from TLS. In TLSTM, when a task reads a location the task needs to check if the location it is reading from is valid. Unfortunately, this validation also takes a toll on correct read operations.

3.3 Algorithm

We now describe in detail how TLSTM overcomes the challenges discussed in the previous section, thereby leveraging SwissTM with TLS support. Algorithms

1 to 3 present the pseudo-code of TLSTM. The following sections explain each aspect of the algorithm in detail.

For each user-thread, the runtime supports up to a fixed number of simultaneously active tasks, called *speculative depth* (SPECDEPTH). Each task is assigned a unique user-thread identifier and a unique serial number which represents the task's position in program order.

Any technique for decomposing each user-thread into tasks can be employed, which is orthogonal to our model and out of the scope of this paper, as long as it ensures that a task does not span across the boundaries we presented earlier. Several standard techniques can be used for user-thread decomposition, from loop iteration speculation (e.g. spec-DOALL and spec-DOACROSS [17]), to procedure fall-through speculation [18], at either compile-time and/or execution-time.

Task, User-Transaction and User-Thread State. Each task maintains the following state inherited from SwissTM:

- *valid-ts*, a timestamp denoting the instant where the read accesses performed by this task are guaranteed to be valid;
- *read-log* and *write-log* tables, each one used to store location entries that were read (resp. written) by the task;

Furthermore, each task also maintains the following new state:

- *tid*, the task's user-thread identifier;
- *serial*, the program order of this task within its user-thread;
- *tx-start-serial* and *tx-commit-serial*, which denote the first and last task, respectively, from the task's user-transaction in program order;
- *try-commit*, a flag that indicates whether this is the last task in the user-transaction;
- *last-writer*, which holds the serial of the last known writer task of the user-thread. Used to check if task validation is required;

Each user-thread maintains the following state, shared by every task running on behalf of this user-thread:

- *completed-task* and *completed-writer*, denoting the serial identifiers of the last completed task and last completed writer task of the user-thread;
- $owners[SPECDEPTH]$, an array of pointers to the state of each task in the user-thread.

For a given task tsk of user-thread thr, its state can be obtained at index $[tsk.serial \bmod SPECDEPTH]$ of the $thr.owners$ array.

Starting a Task. By definition, a task can only start when the number of active tasks in the given user-thread is lower than the $SPECDEPTH$ limit. Once that condition is satisfied, the task is assigned the next serial number in its user-thread and its initial state is saved in the corresponding position of the

owners array (line 2 alg. 1). If the task belongs to an user-transaction, its start and commit-serial are assigned.

The *last-writer* of the new task is assigned the value of the last completed writer of that task's user-thread (line 3 alg. 1). The *valid-ts* of the new task is initialized with the current value of the global counter *commit-ts* (line 4 alg. 1).

Reading. Before reading, the task consults the location's write-lock. In TLSTM, a location's write-lock is either unlocked or points to the location's redo-log. In sum, the location's redo-log has the last speculative write-log entry for that location. TLSTM's write-log entry complements that of SwissTM with the serial number and user-thread identifier of the task that owns the write-log entry, as well as links to entries from past tasks which also wrote to that entry's location.

There are two possible branches for a read operation on a location, depending on whether the location is write-locked by the task's own user-thread or not. If the location is not write-locked by the task's user-thread, i.e. the location is either write-locked by another user-thread or unlocked, TLSTM follows the same procedure as SwissTM: the task reads the location's committed value from memory (line 16 alg. 1).

If, otherwise, the location has been write-locked by the task's user-thread, the task needs to read from the most recent speculative value. This value was either written by the task itself or a past task. The task traverses the redo-log until it finds the entry the task itself wrote to, or a past task wrote to (line 8 alg. 1). If it was the task itself to write to that location, the task can simply return the written value (line 10 alg. 1), since the task's reads from its own writes do not need to be validated.

If, instead, the last speculative value was written by a past task, the task first checks if that past task has already completed. If the past writer task has not completed yet, the task waits until the past task has completed in order to proceed (line 11 alg. 1). TLSTM implements this restriction to simplify the WAR conflict validation procedure. This procedure would have to additionally check the number of writes a past task had performed on the location, in order to validate intra-thread reads done from running tasks.

Afterwards, the task performs validation looking for WAR conflicts, which may have occurred between the current task and the past task that just completed (line 13 alg. 1). If all went well, the task creates a new entry in the task-read-log and adds the location and the validation information (task's serial number) to that log (line 14 alg. 1).

In order to detect dangerous inconsistent reads that could crash the application TLSTM uses several known techniques from previous STLSs [4].

Writing. The task starts by checking if the location has been write-locked by the task itself (line 36 alg. 2). If it has, it just needs to update the logged value, like SwissTM does.

If this is not the case, three situations may occur, depending on whether the lock is:

Write-locked by another task from the same user-thread. If the location is write-locked by a future task, the future task is signaled to abort, since

the task is from the past of the location's current writer in program order (line 47 alg. 2).

If the location is write-locked by a past task, TLSTM needs to check if that past task has already completed (line 45 alg. 2). If the past task is still running, the task will rollback since it is from the future of the location's current writer in program order. If the past task has completed, TLSTM locks the location for writing and adds a new entry to the redo-log which previously owned the write-lock (line 51 alg. 2).

Write-locked by a task from another user-thread. In this case (line 41 alg. 2), the task calls the contention manager in order to decide whether the writer task or the current owner of the write-lock must abort. If the contention manager decides the owner of the write-lock must abort, the writer task waits until the write-lock is eventually unlocked.

Unlocked. This means the present task is the only active task writing to that location. Here the task atomically locks the location's write-lock by $compare-and-swap$, creates a new redo-log that owns the location, assigns the redo-log to the write-lock and continues. Finally, the task performs inter-thread validation, just like SwissTM. Additionally, the task performs intra-thread validation looking for WAR conflicts that may have occurred in the meantime.

Commit. The commit of a user-transaction is carried out by its last task (in program order), called the *commit-task* , once the commit-task and all preceding tasks have completed (line 66 alg. 3). The commit step is very similar to SwissTM, with a few modifications. The commit-task must now consider the read-logs and write-logs of every task of the user-transaction (and not just its own logs) when validating read-logs or committing writes.

Every user-transaction now needs to check for possible validation at commit time, whereas on SwissTM only write user-transactions needed to (line 78 alg. 3). The reason why read user-transactions can no longer proceed without checking for validation is because each task of the user-transaction may have completed at different points in time. This means some tasks of the same user-transaction may have different valid-ts values, thus TLSTM cannot rely on the commit-task's valid-ts alone. If all tasks of a user-transaction have the same valid-ts, then the commit-task can skip this validation.

The commit of the user-transaction then proceeds as in SwissTM, locking the write-logs' read-locks (line 82 alg. 3), incrementing the commit timestamp and validating the user-transaction (line 85 alg. 3). Then, the commit-task updates the values in main memory with the new values from the write-logs of all the user-transaction's tasks (line 89 alg. 3). At the end, the commit-task releases the read and write locks associated with the updated values (line 92 alg. 3).

Finally, the commit-task updates the completed-writer counter if it belongs to a write user-transactions, and updates the completed-task counter to signal the completion of the task and user-transaction (line 93 alg. 3).

Intermediate tasks of a user-transaction just have to update the completed-writer counter, if they have written anything, and the completed-task counter (line 74 alg. 3). Then, they start waiting until all future tasks of the user-transaction have completed, and thus the user-transaction has committed, so that the task can exit safely.

Algorithm 1: Pseudo-code representation of TLSTM

```
1  function start(serial, program-thread-id, try-commit, start-serial, commit-serial)
2      task-init(serial, ptid, try-commit, start-serial, commit-serial);
3      last-writer ← uthread[tsk.ptid].completed-writer;
4      tsk.valid-ts ← commit-ts;

5  function read-word(tsk, addr)
6      (r-lock, w-lock) ← map-addr-to-locks(addr);
7      if is-locked-by-my-thread(w-lock, tsk) then
8          while w-lock and w-lock.serial > tsk.serial do
9              w-lock = w-lock.previous-entry;
10         if w-lock.serial = tsk.serial then return read(addr);
11         while uthread[tsk.ptid].completed-task < w-lock.serial − 1 do
12             if abort-transaction then rollback(tsk);
13         if uthread[tsk.ptid].completed-writer > last-writer and not
           validate-task(tsk) then rollback(tsk);
14         add-to-task-read-log(tsk, w-lock, w-lock.serial);
15         return read(addr);
16     return SwissTM-read-commited-value(addr);

17 function validate-task(tsk)
18     for log-entry in tsk.task-read-log do
19         if is-locked-by-my-thread(log-entry.w-lock) then
20             w-lock = log-entry.w-lock;
21             if w-lock.serial = tsk.serial then
22                 w-lock = w-lock.previous-entry;
23             if w-lock = NULL or log-entry.serial ≠ w-lock.serial then
24                 return false;
25         else return false;
26     for log-entry in tsk.read-log do
27         if is-locked-by-my-thread(log-entry.w-lock) then
           w-lock = log-entry.w-lock;
28         while w-lock do
29             if w-lock.serial >= serial then
30                 w-lock = w-lock.previous-entry;
31             else return false;
32     return true;
```

Algorithm 2: Pseudo-code representation of TLSTM

```
33 function write-word(tsk, addr, value)
34     if aborted-internally then rollback(tsk);
35     (r-lock, w-lock) ← map-addr-to-locks(addr);
36     if is-locked-by-my-task(w-lock, tsk) then
37         update-log-entry(w-lock, addr, value);
38         return;
39     while true do
40         if abort-transaction then rollback(tsk);
41         if is-locked-by-other-thread(w-lock) then
42             if cm-should-abort(tsk, w-lock) then rollback(tsk);
43             else continue;
44         if w-lock.serial < tsk.serial then
45             if uthread[tsk.ptid].completed-task < w-lock.serial then rollback(tsk);
46         else
47             owners[w-lock.serial].aborted-internally = true;
48             continue;
49         previous-entry = w-lock;
50         log-entry ← add-to-write-log(tsk, w-lock, addr, value , ptid, serial,
           previous-entry);
51         if compare&swap(w-lock, w-lock, log-entry) then break;
52     if read(r-lock) > tsk.valid-ts and not extend(tsk) then rollback(tsk);
53     if uthread[tsk.ptid].completed-writer > last-writer and not validate-task(tsk)
       then rollback(tsk);

54 function cm-should-abort(tsk, w-lock)
55     task-progress = uthread[tsk.ptid].completed-task - tsk.start-serial;
56     owner-progress = uthread[w-lock.ptid].completed-task −
       w-lock.owner.start-serial;
57     if task-progress > owner-progress then
58         w-lock.owner.abort-transaction = true;
59         return false;
60     if task-progress < owner-progress then return true;
61     if cm-task-stronger-than-owner(tsk, w-lock.owner) then
62         w-lock.owner.abort-transaction = true;
63         return false;
64     return true;
```

Algorithm 3: Pseudo-code representation of TLSTM

```
65 function commit(tsk)
66     while uthread[tsk.ptid].completed-task < tsk.serial - 1 do
67         if aborted-internally then rollback(tsk);
68     if abort-transaction then rollback-transaction(start-serial);
69     if uthread[tsk.ptid].completed-writer ≠ last-writer then
70         if validate-task(tsk) = false then rollback(tsk);
71     if not tsk.try-commit then
72         if not is-read-only(tsk) then
73             uthread[tsk.ptid].completed-writer = serial;
74         uthread[tsk.ptid].completed-task = serial;
75         while uthread[tsk.ptid].completed-task < tsk.commit-serial do
76             if abort-transaction then rollback(tsk);
77         return;
78     if (abort-serial = validate(tx)) > 0 then
79         rollback-transaction(abort-serial);
80     if not is-read-only(tx) then
81         for write-log in tx do
82             for log-entry in write-log do
83                 write(log-entry.r-lock, locked);
84         ts ← increment&get(commit-ts);
85         if (abort-serial = validate(tx)) > 0 then
86             rollback-transaction(abort-serial);
87         for write-log in tx do
88             for log-entry in write-log do
89                 write(log-entry.addr, log-entry.value);
90                 if log-entry.w-lock = log-entry then
91                     write(log-entry.r-lock, ts);
92                     write(log-entry.w-lock, unlocked);
93         uthread[tsk.ptid].completed-writer = serial;
94     uthread[tsk.ptid].completed-task = serial;
95 function rollback-transaction(start-serial)
96     for write-log in tx do
97         for log-entry in write-log do
98             write(log-entry.w-lock, log-entry.previous-entry);
99         write-log.clear();
100    uthread[tsk.ptid].completed-writer = start-serial-1;
101    uthread[tsk.ptid].completed-task = start-serial-1;
102    abort-transaction =false;
103    for i=start-serial TO serial-1 do
104        owners[i].abort-transaction = true;
105    rollback(tsk);
```

Aborts. Aborting a single task follows the same procedure of SwissTM's user-transaction abort. TLSTM needs to abort a single task when an intra-thread WAR or WAW conflict is detected. Intra-thread WAW conflicts are checked for in two distinct places. First, WAW conflict verification is performed when a task wishes to write to a location (line 34 alg. 2). TLSTM could perform this verification on read operations too, but that would incur in more overhead for the most common read operation.

Second, TLSTM checks for intra-thread WAW conflicts at commit time, while waiting for all past tasks to complete. When all past tasks have completed and the task has not aborted due to a WAW conflict, the task needs to be validated for previously undetected WAR conflicts (as explained in Section 3.2). If WAR conflict validation fails, the task must be individually aborted.

There are also situations where a task may need to abort its entire user-transaction (every single task of the user-transaction to which the aborting task belongs to) because of an inter-thread write-write conflict. The first situation occurs at commit time, if the task passed WAR conflict validation (line 68 alg. 3). The second situation occurs also at commit time, while an intermediate task waits for the future tasks of its user-transaction to commit (line 76 alg. 3).

We chose to abort every single task of the aborting user-transaction because of the simplicity of this approach. The alternative would be to abort only the user-transaction's tasks that wrote to the location that triggered the write-write conflict, and the user-transaction's tasks that read those speculative values. Discovering all these tasks would be very complex, since TLSTM would need to traverse the write-log of each task in the user-transaction in search of the location that triggered the abort and mark those tasks for abort. Then TLSTM would need to traverse the task-read-log of each task of the user-transaction in search for the locations written by the tasks marked to abort and also mark the tasks where TLSTM finds those locations.

4 Evaluation

This section evaluates a TLSTM prototype, which was implemented in C++, based on the C++ implementation of the SwissTM STM, using POSIX threads. The measurements discussed next were obtained using a quad AMD Opteron 6272 with 64 cores total for the STAMP Vacation application and a SPARC Enterprise T5120 server with up to 64 hardware threads for the remainder of the benchmarks. Each result measures the throughput of the respective runtime in operations per second and is the average of three repeated experiments.

We want to determine answers to two main questions: *1) can our unified TLS+STM approach effectively achieve speed-up from simple STM programs?*; and *2) in which kind of applications is TLSTM more advantageous, and in which applications is it not a good approach?*

We started by looking at a modified version of the traditional Red-black Tree micro-benchmark in order to figure out if task size had any impact on the unified runtime's performance. In this modified version each thread runs a transaction that performs a number of lookup operations, which are read-only, to the Red-Black Tree. We can easily split those transactions into several tasks that execute

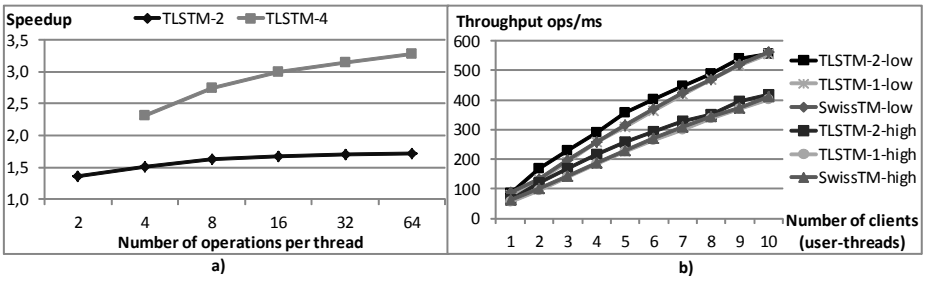

Fig. 1. a) Speedup in the red-black tree's throughput for TLSTM with 2 and 4 tasks and 1 thread vs SwissTM with 1 thread; b) Throughput of TLSTM with 1 and 2 tasks per thread vs SwissTM, with an increasing number of threads on STAMP's Vacation

fewer operations each, e.g. if a transaction runs four operations in total, we can split it into two tasks that run two operations each.

From this experiment we can see that task size does indeed have an impact on the runtime's performance (Figure 1.a). For larger task sizes we obtain a better throughput ratio, for both two and four tasks per user-thread, from which we can deduce that our approach has better performance in applications with large transactions which can be split into large tasks.

Therefore, we started looking at the STAMP application suite [9] in search of applications with large transactions that could be easily split into several parallel tasks. However, most of STAMP's applications had either very small transactions or no further parallelization potential. One application stood out though, the Vacation application which implements an online transaction processing system for travel reservations. A client can issue several operations to the system, e.g. reserve a plane ticket, reserve an hotel or rent a car, and each operation is encapsulated in a transaction.

Since these operations are quite small, similar in size to the red-black tree micro-benchmark operations, we modified the Vacation benchmark taking into account the red-black tree results. We picture each client issuing eight operations at a time, which now incorporate an application server transaction and can be easily split into two tasks, executing four operations each. We still mimic the low and high contention scenarios of the original Vacation application.

The results of this experiment (Figure 1.b) show us that a unified TLS+STM runtime using two task per user-thread improves the throughput of applications with a self-imposed limit to the number of spawned user-threads, in this case the number of concurrent clients being served. Interestingly, both low and high contention scenarios of this application show the same behavior, which we assume to occur because of the very low contention between operations, even in the higher contention scenario. We can also see that TLSTM with one task per user-thread has a very similar throughput to SwissTM, with both lines overlapping most of the time, on both scenarios. This suggests to us that there are applications where TLSTM can be used as a replacement STM.

Another interesting reference benchmark for STMs that includes large transactions is STMBench7 [8], which has a wide range of operations on a very large

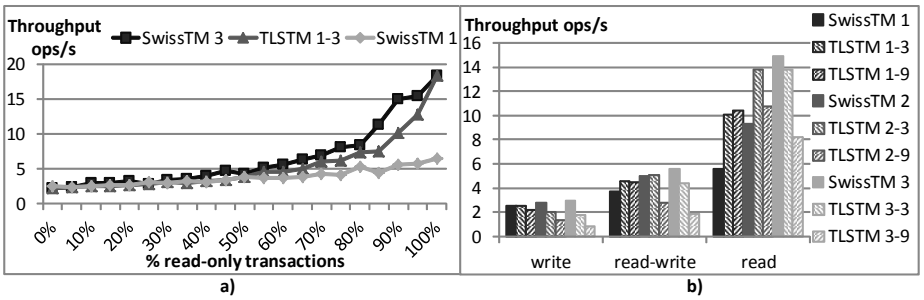

Fig. 2. a) Throughput of TLSTM with 1 thread and 3 tasks vs SwissTM with 1 thread and 3 threads; b) Throughput of SwissTM vs TLSTM with 3 and 9 tasks per thread, with up to 3 threads; Both on STMBench7 long traversals only

shared data structure. From these operations we targeted those which could be automatically split into tasks by a compiler or runtime, in which the set of "Long Traversals" operations stood out. It was also the most computationally intensive set of operations, which made it a perfect candidate to parallelize even further. Most of the remainder operations were either non-divisible or very short, so they would not benefit from parallelization too much. The shared data structure of STMBench7 is built as a tree of objects, with three branches departing from the root, each with arbitrary depth. Therefore, it made sense to split the "Long Traversals" which traverse the whole tree in multiples of three tasks.

The experiment on figure 2.a compares the performance of running one and three user-threads in SwissTM to one user-thread in TLSTM with three tasks. By comparing TLSTM with three tasks and one user-thread to SwissTM with three user-threads we can see how much does the programmer gain if he is capable of hand-parallelizing more code into transactions, instead of relying on automated code division and a unified TLS+STM runtime. But if he does rely on such a runtime, we can see that TLSTM is most beneficial for read-dominated workloads. In fact, for 100% read-only transactions TLSTM achieves practically full speedup. In contrast, TLSTM already performs worse than the base STM for write-dominated workloads.

The problem lies within STMBench7's write "Long Traversals". These write-transactions have a high intra-thread conflict rate (several tasks writing to the same location). Such conflicts translate into the observed decrease in performance, since these transactions will execute almost serially. This is the worst case scenario for TLSTM. In write-transactions with a low conflict rate, such as those of the STAMP's Vacation application, we see that TLSTM performs close to the observed behavior on read-only transactions.

For the last of our experiments, we consider the default settings that STMBench7 originally defines [8]: *write-dominated workload* (10% read operations); *read-write workload* (60% read operations); and *read-dominated workload* (90% read operations).

This last experiment aims at studying how TLSTM behaves as the number of user-threads grows, and how such performance compares to SwissTM running

the same number of user-threads. We can see in figure 2.b that TLSTM with three tasks decreases its performance when going from two to three user-threads, whereas SwissTM scales quite acceptably on read-write and read-dominated workloads. This is an effect of the increased contention in the workload. The inter-thread abort procedure is substantially more complex in TLSTM than in SwissTM, thus hindering its performance in scenarios where contention is higher (more conflicts and rollbacks).

However, we can see that for read-dominated workloads TLSTM with three tasks outperforms SwissTM by 80% on one user-thread and 48% on two user-threads. By increasing the number of user-threads we increase the level of contention even further, thus providing diminishing returns. When executing an inter-thread abort, all of the user-thread's tasks must be aborted. Thus, the inter-thread abort procedure's performance is directly influenced by the number of tasks in the user-thread. In order to measure this influence, we experimented on TLSTM with nine tasks and up to three user-threads (Figure 2.b).

In the case of one user-thread in the read-dominated workload, we can achieve even more speedup with nine TLSTM tasks than with three. But as soon as we get to use two user-threads, the inter-thread contention becomes high enough to harm TLSTM's performance. We can see this is a trend for increasing numbers of user-threads, on any type of workload. This fact suggests that the inter-thread abort procedure is one of the major bottlenecks in the unified approach of TLSTM.

We conclude that each application using TLSTM will have to find a sweet spot between the number of user-threads and tasks in use. Too many user-threads may prevent scalability of the application, while too many tasks may dramatically hinder the performance of the hybrid runtime. For STMBench7's "Long Traversals" this spot seems to be two user-threads with three tasks each, in order to achieve maximum performance.

5 Related Work

Originally introduced in the seminal paper from Herlihy and Moss [3], the interest and advancement in the STM area has grown dramatically in recent years, incited by the advent of affordable multicore processors. Still, most STM programs are still organized as a monolithic collection of a relatively small number of coarse-grained threads. Evidence of this is found in most benchmarks (e.g. [8, 9, 19, 20]) and representative applications of STM.

A distinct research direction that has similar goals as STM is automatic parallelization of sequential programs. Classically, this approach focused on automatically identifying tasks that have no data dependencies (e.g. independent loop iterations) and executing them in parallel. On the other hand, the approach of Thread-Level Speculation (TLS) developed over the last decade has a more aggressive technique for extracting parallelism from sequential programs [1, 2, 21–23]. Rather than parallelizing only provable independent tasks, TLS executes tasks in parallel speculatively and relies on the runtime detection of violations to the sequential semantics of the original program to discard the changes to the program state and restart the affected tasks.

While the first proposed TLS solutions relied on hardware support, recently there has been a growing focus on software approaches. Solutions such as [17, 24–27, 4] are examples of successful efforts towards Software TLS that can yield substantial speed-ups from sequential programs. Still, the conservative nature of TLS constitutes a key limitation to the level of parallelism that it can extract. While proposed TLS systems have been shown to achieve considerable speed-ups (e.g. 77% on average according to Oancea et. al. [4]), most non-trivial programs that do not fit in the category of embarrassingly parallel problems have relatively low bounds on the level of conflict-free speculation.

Most of the run-time support of Software TLS has close resemblance with an STM run-time. For instance, writes are speculative, as they may need to be undone; accesses must be validated for conflicts; tasks have a commit stage; and tasks can be aborted, and restarted. Departing naturally from such an observation, a number of recent Software TLS solutions rely on an underlying simplified STM run-time to offer TLS to single-threaded programs [28]. These solutions are radically different than our proposal: while TLSTM combines STM and TLS, allowing each thread in a transactional multi-threaded programs to be automatically parallelized, TLS solutions relying on STM only address the case of single-threaded programs.

To the best of our knowledge, the only work that addresses TLS support on multi-threaded programs is due to Martinez and Torrelas [29]. However, their approach is fundamentally different from ours, as it only tries to speculatively execute and synchronize threads that would otherwise be blocked waiting on a barrier, lock or flag. Unifying STM and TLS in a common run-time implies solving a number of fundamentally different problems.

In the context of replicated STMs there are some recent examples of systems that employ automatic speculative parallelization to hide the expensive latency of distributed transaction commit [30, 31]. In contract to our contribution, these solutions are proposed in a distinct context (distributed STMs) and limit speculation to one transaction that runs in parallel while the preceding transaction is awaiting commitment.

6 Concluding Remarks

The rapidly increasing core count of commodity machines is demanding highly parallel programs. We claim that the time has come to question a hybrid direction that unifies two prominent research directions of parallel programming that, up to now, have been working (almost) separately with very similar goals.

This paper shows that, although unifying TLS and TM in a hybrid middleware introduces hard challenges, they can be overcome and untapped parallelism potential can be discovered. We describe our experience with a first proof of concept, the TLSTM algorithm. Results obtained with a non-trivial benchmark confirm that STM and STLS do add up successfully for some workloads. Our results also show that there is still a considerable amount of improvement to be made towards devising a unified STM+STLS solution that scales gracefully with both the number of hand-parallelized threads and the number of automatically spawned speculative tasks.

Our preliminary work shows that issues such as transaction rollback and commit are now much more complex (due to the multiple tasks that may comprise each user-transaction), and future work should focus on their negative impact on the overall throughput. The location redo-logs have also showed to add substantial overhead. Hence, different approaches for handling speculative writes (e.g. in-place writes [4]) should be studied.

References

1. G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, (New York, NY, USA), pp. 521–532, ACM, 1998.
2. L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," *SIGPLAN Not.*, vol. 33, pp. 58–69, October 1998.
3. M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993.
4. C. E. Oancea, A. Mycroft, and T. Harris, "A lightweight in-place implementation for software thread-level speculation," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pp. 223–232, 2009.
5. A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui, "Why stm can be more than a research toy," *Commun. ACM*, vol. 54, pp. 70–77, Apr. 2011.
6. J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson, "A 48-core ia-32 message-passing processor with dvfs in 45nm cmos," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pp. 108 –109, feb. 2010.
7. J. T. Oplinger, D. L. Heine, and M. S. Lam, "In search of speculative thread-level parallelism," in *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, 1999.
8. R. Guerraoui, M. Kapalka, and J. Vitek, "Stmbench7: a benchmark for software transactional memory," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 315–324, 2007.
9. C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.
10. M. Ansari, C. Kotselidis, I. Watson, C. C. Kirkham, M. Luján, and K. Jarvis, "Lee-tm: A non-trivial benchmark suite for transactional memory," in *ICA3PP*, 2008.
11. F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero, "Atomic quake: using transactional memory in an interactive multiplayer game server," in *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 25–34, 2009.
12. N. Carvalho, J. a. Cachopo, L. Rodrigues, and A. R. Silva, "Versioned transactional shared memory for the fénixedu web application," in *Proceedings of the 2nd workshop on Dependable distributed data management*, pp. 15–18, 2008.
13. A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pp. 155–165, ACM, 2009.
14. R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

15. D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, (Berlin, Heidelberg), pp. 194–208, Springer-Verlag, 2006.
16. P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 237–246, 2008.
17. H. Kim, A. Raman, F. Liu, J. W. Lee, and D. I. August, "Scalable speculative parallelization on commodity clusters," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, 2010.
18. M. K. Chen and K. Olukotun, "Exploiting method-level parallelism in single-threaded java programs," in *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, 1998.
19. V. Gajinov, F. Zyulkyarov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero, "Quaketm: parallelizing a complex sequential application using transactional memory," in *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, (New York, NY, USA), pp. 126–135, ACM, 2009.
20. I. Watson, C. Kirkham, and M. Luján, "A study of a transactional parallel routing algorithm," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques – PACT 2007*, pp. 388–398, 2007.
21. J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pp. 1–12, 2000.
22. J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry, "The stampede approach to thread-level speculation," *ACM Trans. Comput. Syst.*, vol. 23, pp. 253–300, 2005.
23. W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, "Posh: a tls compiler that exploits program structure," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, (New York, NY, USA), pp. 158–167, ACM, 2006.
24. C. E. Oancea and A. Mycroft, "Software thread-level speculation: an optimistic library implementation," in *Proceedings of the 1st international workshop on Multicore software engineering*, IWMSE '08, pp. 23–32, 2008.
25. S. Devabhaktuni, "Softspec: Software-based speculative parallelism via stride prediction," in *Master's thesis, M.I.T*, 1999.
26. M. Cintra and D. R. Llanos, "Toward efficient and robust software speculative parallelization on multiprocessors," in *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 13–24, 2003.
27. P. Rundberg and P. Stenström, "An all-software thread-level data dependence speculation system for multiprocessors," *Journal of Instruction-Level Parallelism*, vol. 3, p. 2002, 2001.
28. M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke, "Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pp. 166–176, 2009.
29. J. F. Martínez and J. Torrellas, "Speculative synchronization: applying thread-level speculation to explicitly parallel applications," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, (New York, NY, USA), pp. 18–29, ACM, 2002.
30. B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *IEEE Trans. on Knowl. and Data Eng.*, vol. 15, pp. 1018–1032, July 2003.
31. R. Palmieri, F. Quaglia, and P. Romano, "Osare: Opportunistic speculation in actively replicated transactional systems," in *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems*, SRDS '11, (Washington, DC, USA), pp. 59–64, IEEE Computer Society, 2011.