STM^2: A Parallel STM for High Performance Simultaneous Multithreading Systems

Gokcen Kestor* †, Roberto Gioiosa*, Tim Harris§, Osman S. Unsal*, Adrian Cristal* ‡, Ibrahim Hur* and Mateo Valero* †
* Barcelona Supercomputing Center
† Univesitat Politecnica de Catalunya
‡ IIIA - Artificial Intelligence Research Institute - CSIC - Spanish National Research Council
§ Microsoft Research
(gokcen.kestor, roberto.gioiosa, osman.unsal, adrian.cristal, ibrahim.hur)@bsc.es
tharris@microsoft.com, mateo@ac.upc.edu

***Abstract*—Extracting high performance from modern chip multithreading (CMT) processors is a complex task, especially for large CMT systems. Programmers must efficiently parallelize performance-critical software while avoiding deadlocks and race conditions. Transactional memory (TM) is a promising programming model that allows programmers to focus on parallelism rather than maintaining correctness and avoiding deadlock. Software-only implementations (STMs) are especially compelling because they run on commodity hardware, therefore providing high portability. Unfortunately, STM systems usually suffer from high overheads, which may limit their usage especially at scale.**

In this paper we present *STM*^{2}, a novel parallel STM designed for high performance, aggressive multithreading systems. *STM*^{2} significantly lowers runtime overhead by offloading read-set validation, bookkeeping and conflict detection to auxiliary threads running on sibling hardware threads. Auxiliary threads perform STM operations in parallel with their paired application threads and absorb STM overhead, significantly improving performance.

We exploit the fact that, on modern multi-core processors, sets of cores can share L1 or L2 caches. This lets us achieve closer coupling between the application thread and the auxiliary thread (when compared with a traditional multi-processor systems). Our results, performed on an IBM POWER7 machine, a state-of-the-art, aggressive multi-threaded system, show that our approach outperforms several well-known STM implementations. In particular, *STM*^{2} shows speedups between 1.8x and 5.2x over the tested STM systems, on average, with peaks up to 12.8x.

I. Introduction

Chip Multithreading (CMT) processors promise to deliver higher performance by running more than one stream of instructions in parallel rather than by increasing the processor's frequency. CMT processors may come with different architectures: Chip Multi-Processor (CMP), Simultaneous Multi-Threading (SMT), or a combination of them. To exploit CMT's capabilities, users have to parallelize their applications. Unfortunately, efficiently parallelizing applications is not trivial. Several proposals focus on how to reduce the effort of parallelizing applications on CMT machines. Fine-grained locking techniques provide good performance but pose challenges to programmers. Consequently, automatic parallelization techniques [12], [24] and innovative programming models [11] have been proposed to reduce programmers' effort.

Transactional Memory [14] (TM) is one of such novel programming models. The principal goal of TM is to simplify synchronization by raising the level of abstraction, breaking the connection between semantic atomicity and the means by which that atomicity is achieved. Programmers indicate atomic section in the source code (e.g., using language constructs such as *atomic* blocks, or using macros such as `BEGIN_TRANSACTION` and `END_TRANSACTION`) without explicitly locking individual shared memory locations. An underlying TM system executes such transactions concurrently whenever possible, generally by means of speculation – optimistic but checked execution, rollbacking when conflicts arise. Transactions commit or abort *atomically*, i.e., either all memory locations modified during the transaction are updated (commit) or nothing is modified (abort). Transactions are allowed to commit only if they have no conflicts or all their conflicts are resolved positively.

Unfortunately, the performance of the current TM systems is not always satisfactory, especially for STM proposals: the overheads introduced by the STM runtime system may well outweigh the parallelism gained [5]. Some authors report drastic slow-downs when using STM (e.g., only breaking even with optimized sequential code after using 8 cores [5]). Even state-of-the-art TM systems typically require at least two threads to achieve performance that matches optimized sequential code [9], [13]. In any parallel program, Amdahl's law [3] limits the extent to which parallel execution can achieve speedups. With TM, if large sections of parallel code run within transactions, there is a risk that the speedup possible via Amdahl's law will never be enough to recover the overheads of using TM. Transactions execute optimistically in parallel but may be forced to abort and rollback when conflicts arise. Moreover, STMs that use bookkeeping may introduce considerable slowdown due to read- and write-set validation and transaction state management [5].

In this paper we tackle the problem of reducing the overhead in STM systems. Performance of several applications among the most common benchmarks suites, such as STAMP [4], are limited by STM overhead and provide performance degradation beyond a certain number of threads [5]. On the other

hand, 4- or 8-core architectures with tens of hardware threads are already available [25], [27] and this count is expected to increase in the coming years. Since not all applications are able to effectively use all cores and/or hardware threads, we propose to perform time-consuming STM operations on those computing elements that do not provide measurable performance improvement. The general idea is that using additional cores/hardware threads to speed up STM operations may provide higher performance than using these processing elements to run additional application threads. Specifically, we offload read-set validation, bookkeeping, transaction state management and conflict detection to an *auxiliary thread* running on a *sibling* core/hardware thread, i.e., a processing element that shares some levels of hardware resource (like the L1 or L2 cache) with the *application thread*.[1]

In order to demonstrate our proposal we implemented *Software Transactional Memory for Simultaneous Multithreading* systems (*STM*^{2} - pronounced *STM-squared*). To the best of our knowledge, *STM*^{2} is the first parallel STM system that uses secondary hardware threads to leverage STM overhead. *STM*^{2} is essentially a parallel STM system where transactional operations are divided between *application threads* (computation) and *auxiliary threads* (STM management). With *STM*^{2}, application threads optimistically perform their computation with minimal support from the underlying STM system. All synchronization and STM management operations are performed by the paired auxiliary threads. This means that application threads experience minimal overhead. Auxiliary threads, instead, validate read-sets, maintain transaction states and detect conflicts in parallel with the application threads' computation. *STM*^{2} detects conflicts as soon as they occur (eager conflict detection). If a conflict is detected, the auxiliary thread interrupts its corresponding application thread and aborts the transaction. If no conflicts arise during a specific transaction, the auxiliary thread commits the transaction and updates the modified shared memory location (lazy update). Communication between application threads and their corresponding auxiliary threads is performed through a lock-free circular buffer and simple atomic state variables.

We tested *STM*^{2} on an aggressive, high performance SMT processor, an IBM POWER7 system with a total of 32 hardware threads. To the best of our knowledge, this is the first study that tests transactional memory on a POWER7 processor. Our results show that by overlapping computation and STM management operations *STM*^{2} obtains performance improvement, outperforming modern well-known STM systems, namely TinySTM, NOrec, TML and TL2, for several STAMP benchmarks. Our experiments show that *STM*^{2} achieves, on average, between 1.8x and 5.2x speedups over state-of-the-art STM systems, with peaks up to 12.8x.

This paper makes the following contributions:

- Introduces *STM*^{2}, a novel parallel STM implementation that reduces the runtime overheads by offloading time consuming TM management operations to auxiliary threads running on sibling hardware threads.
- Tests several state-of-the-art STM systems, namely TinySTM, TL2, NOrec and TML, on an aggressive multithreading processor designed for high performance.
- We show that, perhaps surprisingly, it is often better to use hardware threads to parallelize the STM implementation, than to devote those hardware threads to running additional application threads.

The rest of this paper is organized as follows: Section II motivates our work. Section III describes the design of *STM*^{2} and provides internal details of the implementation. Section IV and Section V describe our experimental setup and results, respectively. Section VI summarizes related work. Finally, Section VII concludes this paper.

II. Motivation

STM management operations are time-consuming and may introduce considerable overheads that increase with the number of threads running in parallel and the size of the read- and write-set [15]. The result is that STM systems may not be able to provide satisfactory performance at scale [5]. In order to understand the overhead introduced by STMs, we run preliminary experiments instrumenting TinySTM, a widely used STM system. Figure 1 shows the per-transaction overhead introduced by TinySTM on STAMP benchmarks running on an IBM POWER7 system when varying the number of threads from 1 to 32.[2] Since the total number of transactions in STAMP benchmarks is constant, the number of transactions per thread decreases with the number of concurrent threads. We, hence, report the per-transaction overhead of each benchmark normalized to the overhead introduced by the STM when running the same application with one thread. For example, *Vacation-Low* presents large, mostly-read transactions, therefore, each transaction spends most of its time in the `stm_read()` function. When going from 1 to 32 concurrent threads, the commit rate increases and more read-set validations need to be performed. For the specific case of *Vacation-Low*, the STM overhead with 32 concurrent threads increases by 5.7x with respect to the instrumented single thread execution. While *Vacation-Low*'s and *Vacation-High*'s transactions are dominated by read time, *Genome* and *SSCA2* have short transactions, thus their per-transaction overhead breakdown is completely different. In particular, Figure 1 shows that the relative overhead of `begin_transaction()` and `commit()` have a higher impact on *SSCA2* than on *Vacation-Low* in the instrumented, single thread version. Figure 1 also shows that for applications with high contention, such as *Labyrinth*, the overhead introduced by aborts increases at scale.

Our experiments, in accordance to what was previously observed [5], [15], [19], show that per-transaction STM overhead increases with the number of concurrent threads, which may limit scalability substantially. Note that the actual impact

[1]Two hardware threads in a core or two cores sharing the L2 cache are examples of sibling hardware thread/core, respectively.

[2]In these experiments and in the rest of the paper, we use the default configuration for TinySTM, eager conflict detection and lazy data versioning.

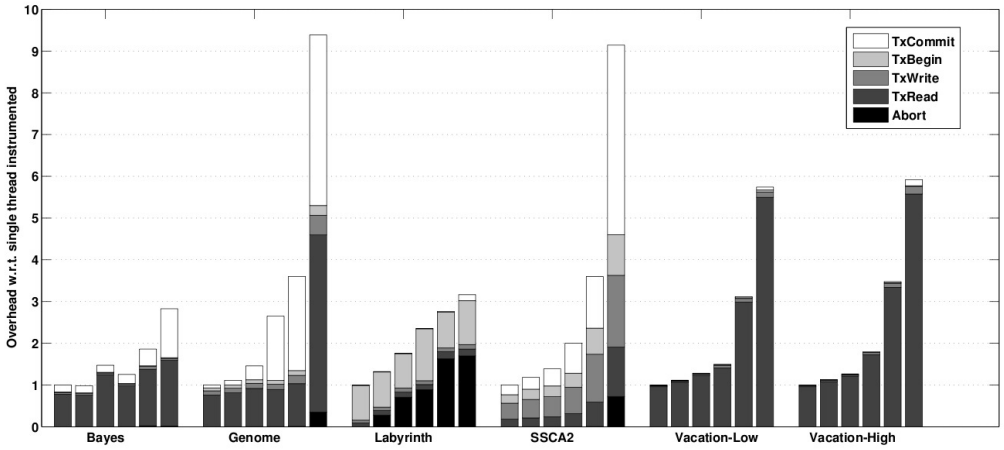


Fig. 1: TinySTM per-transaction overhead breakdown for STAMP applications with respect to instrumented single thread version. We report per-transaction overhead because the number of transactions per thread for STAMP applications decreases with the number of threads. We instrumented TinySTM and obtained per-transaction overhead breakdown for single thread, 2, 4, 8, 16 and 32 concurrent threads versions. STM per-transaction overhead increases with the number of threads because of the higher pressure on internal STM data structure, more frequent read-set validations and higher contention.

on the applications' performance may vary depending on the amount of time each application spends inside transactions. Section V discusses the impact of STM overhead on each application's performance.

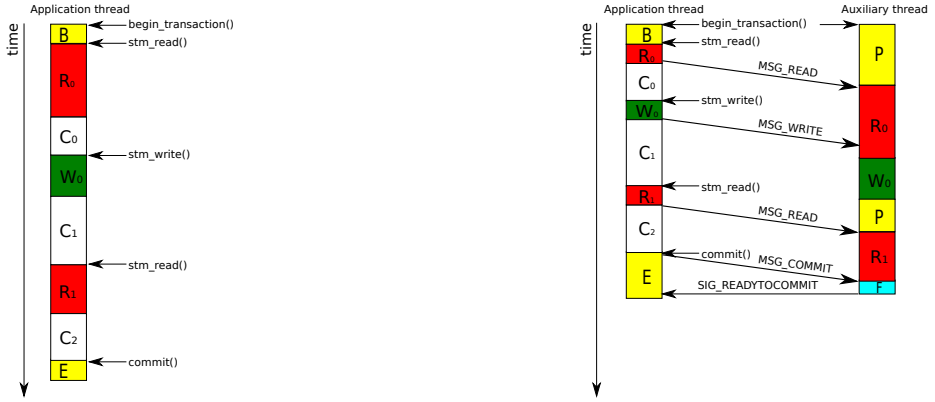
III. STM^2 Design and Implementation

Recent studies [16] show that future scientific problems with large data sets will require higher computational power (i.e., higher number of cores) than what is currently available. Processing elements that do not directly provide performance improvement should be used in a better way, for example, to leverage the work performed by overloaded cores. *STM*^{2} is a novel implementation that goes in this direction: time-consuming operations (such as read-set validation and conflict detection) are offloaded to *auxiliary threads* running on separate hardware threads. Application threads have fewer STM management operations to perform and can spend their cycles on more useful work.

Figure 2 shows how offloading operations to auxiliary threads may reduce STM overhead, therefore improving performance. Figure 2a shows how a typical eager conflict detection, lazy data versioning STM system works. Before actually accessing any memory location, the application thread performs an `stm_read()` when reading a memory location, or an `stm_write()` when attempting to modify a shared variable. These two functions notify the STM runtime about which locations should be inserted into the read-set and the write-set of that thread. Whenever the STM runtime system takes control, it may check whether a conflict has occurred and, if so, abort a conflicting transaction. As Figure 2a shows, the application thread often runs STM library code rather than performing its computation, especially if the STM operation triggers time-consuming activities, such as read-set validation.

We propose to move time-consuming STM operations to another hardware thread and perform them in parallel with its application thread. Figure 2b shows our approach: Whenever an application thread accesses a memory location (either reading or writing), it simply sends a message to its corresponding auxiliary thread and then keeps performing its computation. The auxiliary thread, in turn, waits for messages coming from its corresponding application thread and performs read-set validation, transaction state management and conflict detection. Whenever an auxiliary thread detects a conflict, it aborts its corresponding application thread. As Figure 2b shows, offloading STM operations to auxiliary threads and performing transaction management in parallel reduce the transaction's execution time, therefore improving performance.

STM^2 is an eager conflict detection, lazy update STM system. Given that we are using an auxiliary thread that runs in parallel with the application thread, it makes sense to perform as many operations as possible in parallel. In this scenario, lazy conflict detection would delay most of the work at commit stage, serializing the execution of the application thread (that would mainly run during the transaction) and the auxiliary thread (that would be idle during the transaction and overloaded at commit time) and indeed invalidate most of the benefit of our approach. On the other hand, the main drawback of eager conflict detection is the extra overhead caused by the STM management operations (Figure 2a). This overhead is exactly what *STM*^{2} reduces. Eager conflict detection increases the parallelism of *STM*^{2} and decreases the amount of work to be done at commit stage, which is a synchronization point and critical for the *STM*^{2} performance (see Figure 2b). Eager updates, instead, would require extra communication among the auxiliary threads. Memory locations modified by aborted transactions must rollback to their original values, hence, all transactions that have read those invalid values may have to rollback too. In *STM*^{2}, memory updates and aborts are handled by auxiliary threads, hence they should also take care of restoring memory locations modified by



(a) Standard STM system

(b) *STM*^{2}

Fig. 2: *STM*^{2} offloads time-consuming STM operations to sibling hardware threads. In this Figure the application thread performs two read (R_0 and R_1) and a write (W_0) operations. *C* denotes computational phases that do not access shared memory locations. `begin_transaction()` and `commit()` are marked with *B* and *E*, respectively, while *P* denotes polling and *F* denotes commit phase at the auxiliary thread's side. Offloading STM operations to auxiliary threads reduces the overall execution time.

aborted transactions. This, in turn, would require auxiliary threads to exchange messages among themselves. Although eager updates is a possible solution, lazy updates minimize communication among auxiliary threads.

Offloading time-consuming STM operations to a secondary processing element is particularly appealing for multithreading architectures (like IBM POWER, Intel with Hyper-Threading or SUN Niagara). In *STM*^{2}, application and auxiliary threads are paired on the same core, i.e., they are pinned to two separate hardware threads of the same core. While application and auxiliary threads could run on different cores, running on the same core is advisable for the following reasons: 1) the cost of a hardware thread (in terms of space, resources and power consumption) is lower than that of a core; 2) even though extra cores may improve performance linearly, extra hardware threads usually provide only between 1.2x and 1.6x speedup [2], and 3) application and auxiliary threads running on the same core share more resources (for example, the L1 cache), which allows lower-latency communication.

In the current implementation, *STM*^{2} supports a basic TM programming model in which a transaction that aborts does not necessarily see a consistent view of memory, and in which there is no conflict detection between transactional and non-transactional memory accesses. Consequently, the programmer or compiler using *STM*^{2} must sandbox the effects of "zombie" transactions, and must ensure that data is accessed in a consistent way (e.g., using the fence techniques of Spear et al. [22], or using the memory protection isolation mechanisms of Abadi et al. [1]). This is the programming model typically used in STAMP and other TM applications (e.g., *Labyrinth* explicitly restarts inconsistent transactions) therefore no extra support is required to run STAMP benchmarks.

The following subsections describe with more detail the main components of *STM*^{2}: application and auxiliary threads synchronization (Section III-A), transactional write (Section III-B) and read (Section III-C) operations.

A. Application/Auxiliary Thread Synchronization

Application and auxiliary threads communicate through a communication channel and atomic status variables. Application threads send messages to their paired auxiliary threads to notify read and write operations. These operations require extra parameters and cannot be implemented by a simple shared variable (see in following subsection). Auxiliary threads, instead, only need to send two signals[3] to application threads: `SIG_READYTOCOMMIT` and `SIG_ABORT`. We thus implemented a single-producer/single-consumer, circular, lock-free queue where the application thread (producer) posts read and write messages that the auxiliary thread (consumer) retrieves and processes. The `SIG_READYTOCOMMIT` and `SIG_ABORT` signals do not need extra information and are implemented through atomic status variables shared between application and auxiliary threads. These variables are accessed and modified using atomic operations. An extra signal (`SIG_START`) and message (`MSG_COMMIT`), are sent to auxiliary threads when a transaction begins or ends. When an application thread is not involved in a transaction, its corresponding auxiliary thread waits in a spinning loop. As the application thread enters a transaction (`begin_transaction()`), the auxiliary thread receives the `SIG_START` signal and starts polling the communication channel for incoming messages. When an application thread reaches the end of a transaction and attempts to commit (`commit()`), it sends the `MSG_COMMIT` message and waits for the auxiliary thread to complete its work by spinning on the `SIG_READYTOCOMMIT` signal. If the auxiliary thread succeeds resolving all conflicts and validating its read-set, it commits the transaction by updating all shared memory locations modified by the application thread and sets the `SIG_READYTOCOMMIT` signal.

Finally, all shared atomic variables are modified only by one of the two threads during a transaction. For example, auxiliary threads set `SIG_READYTOCOMMIT` and `SIG_ABORT` signals while application threads only read the status of these vari-

[3]Note that these are different from operating system signals.

```
void stm_write(Addr, Val)
{
  if (writeset.insert(Addr, Val) == present)
     return;
  else
     channel.send(MSG_WRITE, Addr, Val);
}
```

(a) Application thread transactional write

```
void aux_stm_write(Addr, Val)
{
  if (validate() == fail) abort();
  if (acquire_ownership(Addr))
      ownedlist.add(Addr);
  else
      ret = cm.onconflict();
      if (ret == CM_ABORT)
         abort();
}
```

(b) Auxiliary thread transactional write

Fig. 3: Pseudo-code for application and auxiliary thread STM write

ables. Similarly, only application threads set the `SIG_START` signal, while auxiliary threads only poll on its value. The result is that the communication involved is minimal and we believe that a small extra hardware buffer between two hardware threads may eliminate the need of using the L1 and increase performance. In this paper we take a software-based approach using off-the-shelf hardware and we leave extra hardware support for future work.

B. Writing to a shared memory location

Memory locations modified by application threads during transactions are not visible to other threads until the transaction commits. On the contrary, conflicts are detected as soon as they occur, avoiding unnecessary computation for transactions that will be aborted and reducing the overhead at commit time.

To guarantee correctness, only one application thread at a time is allowed to change the value of a particular shared memory location, although several threads can modify different memory locations at the same time. Before altering a memory location, application threads need to be sure that no other thread is currently attempting to modify the same location. *STM*^{2} uses *ownership records* to identify which thread is entitled to change the value of a given shared memory location. Once a thread owns a location, it is allowed to modify its content. Any other thread that needs to alter the content of the same location and, therefore, tries to acquire its ownership, will fail (conflict) and will restart the transaction. *STM*^{2} maintains a per-thread *write-set* buffer to temporarily store values modified during a transaction but not yet committed. If the transaction commits successfully, *STM*^{2} will publish its write-set. The updated values will then become visible to the other application threads. *STM*^{2} uses versioning based on extendable timestamps to detect conflicts [21]: every time a shared memory location is updated with a new value, the current timestamp is used as version number and associated with that location. A conflict arises when an application thread has read a value from a memory location whose version number is lower than the current one.

Whenever an application thread wants to modify a shared memory location, it issues an `stm_write()` call, passing the address of the memory location and the new value as arguments. Figure 3 shows the pseudo-code for `stm_write()` on both application and auxiliary thread sides. On the application thread side (Figure 3a), `stm_write()` checks whether the location is already in the write-set, in which case the application thread simply updates the value and returns. If the location is not in the write-set, the application thread will still optimistically write the new value to its write-set but it will also send an `MSG_WRITE` message to its corresponding auxiliary thread. Upon receiving an `MSG_WRITE` message, the auxiliary thread first validates its read-set and then tries to acquire the ownership of the target memory location (Figure 3b). If both operations are successful, the auxiliary thread adds the location to its list of owned shared memory locations. At commit stage, these locations will be updated in memory and the new values will become visible to the other application threads. Note that, on success, no other message is sent to the application thread because it had optimistically already proceeded with the transaction. If the auxiliary thread detects a conflict while trying to acquire the ownership of the location, the contention manager will decide which transaction has to abort. In case the contention manager returns `CM_ABORT`, the auxiliary thread notifies its corresponding application thread by setting the status of the transaction to aborted. The auxiliary thread then removes all entries in the read-set, releases all owned locations, and rolls back. Whenever an STM operation is issued, application threads check their transaction's status and restart the transaction if they find out that the transaction has been aborted by their paired auxiliary threads. Note that, besides resetting the write-set, no other actions are required from the application thread on abort.

We minimized synchronization overhead by using a lock-free data structure for the communication channel described in Section III-A, and by clearly dividing data structures between application and auxiliary threads. Auxiliary threads own the read-set and the list of owned locations. Application threads, on the other hand, own the write-set. Since application threads never access auxiliary threads' data structures (and vice versa) there is no need to protect them with locks.

C. Reading from a shared memory location

Application threads read shared memory locations by calling the `stm_read()` function and passing the address of the target memory location as argument. The `stm_read()` has three main goals: 1) locate the current version of the shared value to return, 2) insert the address of the shared location in

```
void stm_read(Addr)
{
  found = writeset.find(Addr);
  if (found)
    return from writeset;
  else
    channel.send(MSG_READ, Addr);
    return from memory;
}
```

(a) Application thread transactional read

```
void aux_stm_read(Addr)
{
  if (is_owned(Addr))
     ret = cm.onconflict();
     if (ret == CM_ABORT) abort();
  if (validate() == fail)
     abort();
  else
     reads.insert(Addr);
}
```

(b) Auxiliary thread transactional read

Fig. 4: Pseudo-code for application and auxiliary thread STM read

the transaction's read-set (unless it is already present), and 3) perform read-set validation, if required. These operations are divided between the application and the auxiliary threads.

Figure 4 shows how application and auxiliary threads operate when reading a memory location. The application thread (Figure 4a) locates the current version of the value to be read. The current value is either stored in the transaction's write-set or in the original memory location. In the former case, the application thread has already issued at least one write operation on that location at the time of reading the value. In this case STM^2 returns the value modified by the last write operation contained in the transaction's write-set. If the address is not found in the write-set, STM^2 returns the current version from memory and sends an `MSG_READ` message to the auxiliary thread together with the address of the target memory location. Note that other threads may be modifying the same memory location but those threads have not committed their transactions yet, hence those modifications are not visible to the current thread.

When the auxiliary thread receives the `MSG_READ` message, it performs conflict detection. A conflict occurs when 1) the memory location is locked by another thread or 2) the version read by the application thread is different from the current version, i.e., some other thread has committed a new version (`validate()` returns `fail`). In the former case, the auxiliary thread calls the contention manager which may decide to abort either the current transaction or the one that has locked the location. In the latter case, the transaction aborts. If no conflicts are detected, the auxiliary thread inserts the memory location's address into the read-set and moves to the next message.

IV. Experimental Setup

This section describes the setup environment, the benchmarks and the STM systems used in our experiments.

We performed our experiments on an IBM POWER7 [2], [27], an out-of-order, 8-core design where each core is 4-way SMT (32 hardware threads in total). Each core region (or "chiplet") contains a 32 KB 4-way set associative L1 I-cache and a 32 KB 8-way set associative L1 D-cache, a private per-core 256 KB L2 cache and a 4 MB portion of the shared 32 MB L3 cache. Since POWER7 is capable of running 32 threads concurrently, we limit our experiments to 32 threads without over-provisioning the system (i.e., we run as many threads as available hardware threads). Each POWER7 core can run in single-thread (ST) mode, SMT2 (two threads executing on a core concurrently) or SMT4 (three or four threads executing on a core) mode. For capacity computing (i.e., multiple independent, serial jobs running in parallel), both SMT2 and SMT4 modes are expected to provide benefits. For capability computing (i.e., parallel applications with high degree of parallelism), SMT4 may not show extra benefits [2]. STM^2 uses the SMT4 mode and offloads time-consuming TM operations to secondary hardware threads that, otherwise, may not provide extra performance improvement.

We compare STM^2 to several well-known, publicly available and mature STM proposals, namely TML [23], NOrec [9], TinySTM [21], and TL2 [10], using the STAMP benchmark suite [4] compiled with gcc 4.3.4 and `-O3` settings.

TML is an eager conflict detection, eager versioning system with a single sequence lock [17]. TML allows concurrent read-only transactions with no logging overhead, but only one writer, system-wide, is allowed. This approach is effective in workloads where reads are the common case. However, using a single sequence lock without logging means that conflict detection is extremely conservative: any writer conflicts with any other concurrent transaction.

TL2 is a lazy versioning system. A transaction begins by reading the value t in a global "clock." Ownership records (orecs), found by address hashing, indicate the last time at which one of the corresponding locations was modified. If a transaction encounters a location that was written after t, it assumes it is inconsistent, aborts, and retries. At commit time, the transaction locks the orecs for all locations that need to be modified, checks to make sure that all of the locations it read still have a timestamp earlier than t, increments the global time, stores the new time into all the locked orecs, writes out all the updates, and then unlocks the orecs.

TinySTM is an eager conflict detection, lazy versioning system with extendable timestamps. Extendable timestamps avoid false positives in which a transaction is aborted despite having seen a consistent view of memory. If a transaction accesses a location that was written after start time t, it checks to see whether any previously read location has been modified since t. If not, it re-reads the global clock and continues, pretending it started at this new time t' instead of t.

NOrec extends TML with lazy updates and value-based conflict detection. NOrec uses a single sequence lock, but unlike TML, it acquires the lock only when updating memory at commit time, which increases concurrency. **NOrec** uses a single-writer commit protocol, which may limit its scalability in workloads with many writers. Moreover, value-based conflict detection requires to store both the address and the value of the read location.

We selected these STM systems because they reflect popular but divergent points in the STM design space. Several of these STM systems have not been officially ported on POWER architectures (e.g., TL2, NOrec). We ported those STM systems on POWER processors[4] to be able to fairly evaluate *STM*^{2} but some of the STAMP benchmarks (namely *Intruder*, *Kmeans* and *Yada*) did not execute correctly with some of the tested STMs due to bugs in STAMP code [6]. We omit these results for those benchmarks for fairness. Finally, in order to evaluate the effect of increasing the read-set size on the performance of the STMs, we run two versions of *Vacation* (i.e., *Vacation-Low* and *Vacation-High*).

V. Experimental Results

In this Section we analyze the performance of *STM*^{2} and the other tested STM systems. Figure 5 shows performance of STAMP benchmarks running on the IBM POWER7 system previously described. We report the execution time of each STAMP benchmark when varying the number of threads from 2 to 32. In the first set of experiments, we compare STM systems running STAMP benchmarks when using the same number of application threads: we, thus, compare STMs with N threads to *STM*^{2} running N application threads plus N auxiliary threads (N+N), for N=2, 4, 8, 16. While in these experiments *STM*^{2} uses double the number of threads (N+N) than the other STMs (N), the extra hardware threads are available and there is no reason why they should be left idle if an STM can take advantage of them. Moreover, in this set of experiments, the number of transactions per application thread is the same. In the second set of experiments, we analyze the performance of *STM*^{2} and the other STMs using the same amount of hardware resources: we compare STM systems with 32 threads to *STM*^{2} with 16 application threads and 16 auxiliary threads (16+16). We report *STM*^{2} speedups for this experiment (32 threads versus 16+16 threads) in Figure 6.

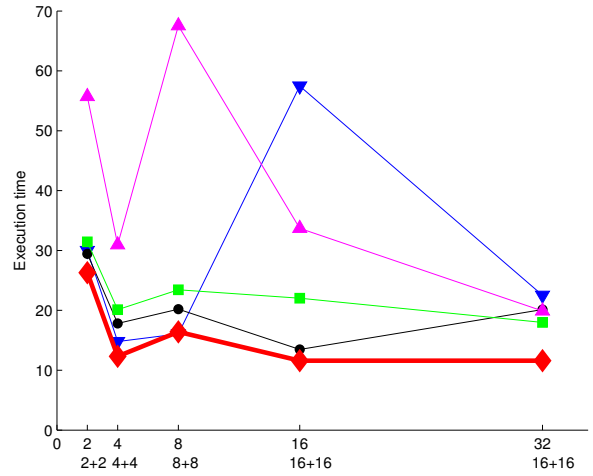
As we can see from Figures 5 and 6, *STM*^{2} reduces runtime overhead by offloading time-consuming operations to dedicated hardware threads. The reduced overhead directly translates to better performance (lower execution time).

TinySTM Both TinySTM and *STM*^{2} use eager conflict detection and lazy versioning. TinySTM is, thus, the ideal STM system to be compared with in order to analyze the effect of offloading transaction state maintenance, read-set validation and conflict detection to secondary hardware threads. As Figure 5 shows, *STM*^{2} performs better or equal than TinySTM in all cases. If the level of contention is low and the read-set size are small (*Genome* and *SSCA2*), *STM*^{2} and TinySTM behave similarly, especially at small scale (N=2 or N=4 threads). When the read-set becomes larger, *STM*^{2} clearly outperforms TinySTM. For example, *STM*^{2} performs considerably better when running *Vacation-Low* (7x faster) and *Vacation-High* (12.3x faster) with 32 hardware threads. *Vacation-Low* exhibits large, mostly-read transactions, thus its read-set size is considerably larger than other applications. Eager conflict detection requires scanning read-sets to identify possible conflicts during the execution of each transaction. In this scenario, larger read-sets introduce higher runtime overhead. Moreover, as reported by Cascaval et al. [5] and confirmed by our experiments (Figure 1), runtime overhead increases with the number of concurrent threads. *STM*^{2} is able to absorb transaction state maintenance, read-set validation and conflict detection overheads with the secondary hardware threads. In our experiments, TinySTM is not always able to scale beyond N=16 threads: *Vacation-Low* takes about 24.24 seconds with N=16 threads and 96.88 seconds with N=32 threads. *STM*^{2} instead is able to make a better use of the last 16 hardware threads by accelerating STM operations and reducing the execution time to 13.79 seconds (7x faster) when using 16 applications threads and 16 auxiliary threads (16+16). Moreover, *STM*^{2} is also faster than the best TinySTM performance obtained with N=16 threads (1.8x). We conclude that the STM overhead introduced by TinySTM on *Vacation-Low* is completely absorbed by the auxiliary threads in *STM*^{2}. The effects of offloading STM operations to secondary hardware threads become more evident when increasing the number of read operations performed during each transaction or the level of contention in the application. *Vacation-High* performs the same algorithm as *Vacation-Low* but its transactions operate on more items (i.e., larger read-sets). Figure 5f shows that TinySTM does not provide performance improvement beyond N=4 threads (in fact, performance constantly reduces with the number of threads). *STM*^{2}, instead, efficiently scales up to 32 hardware threads (16+16), providing a final speedup of 12.3x over TinySTM with 32 hardware threads.

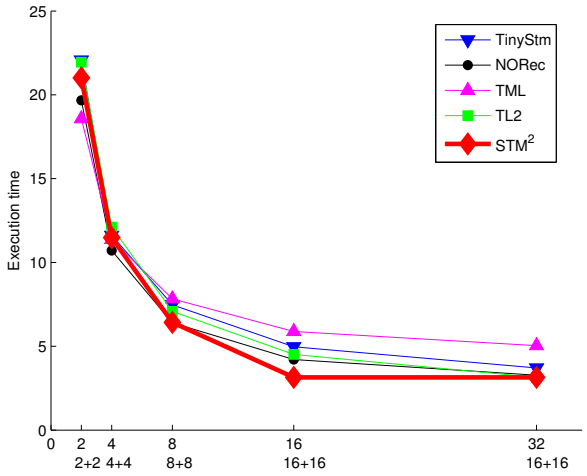
Bayes and *Labyrinth* exhibit a high level of conflict, even though their read- and write-sets are not as large as in *Vacation-Low*. *STM*^{2} performs better than any other STM in these two cases and, in particular, shows a 1.9x and 1.1x speedup over TinySTM with 32 hardware threads for *Bayes* and *Labyrinth*, respectively.

For *SSCA2* eager conflict detection, multiple-writers STMs (*STM*^{2} and TinySTM) perform considerably better than the other STMs. This seems to indicate that early detection of conflicts reduces the STM overhead for this application. *SSCA2* differs from the other benchmarks in that it shows a bursting and irregular behavior with higher number of short, read-write transactions per second (high commit rate). Lazy conflict detection STMs (TL2 and NOrec) fail to acquire all required locks at commit time because other transactions commit in the meantime. Even if these commits do not generate actual conflicts, NOrec still needs to re-validate the elements in

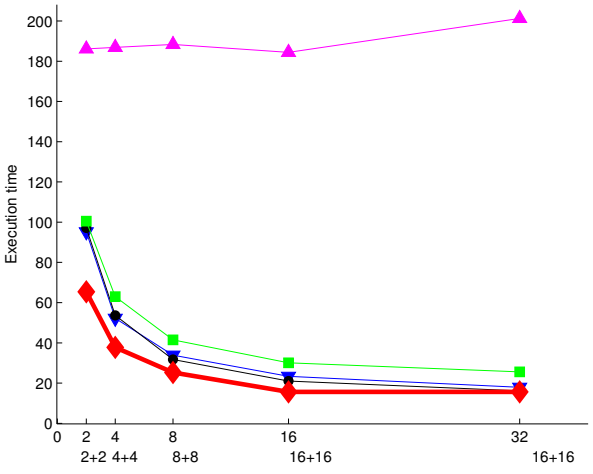
[4]No further modifications to the original implementations have been applied.



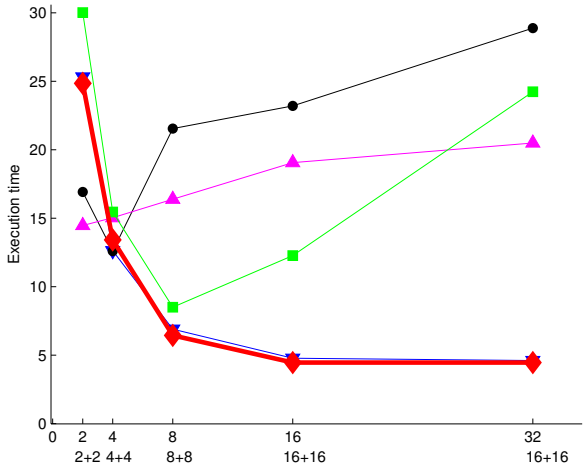
(a) Bayes



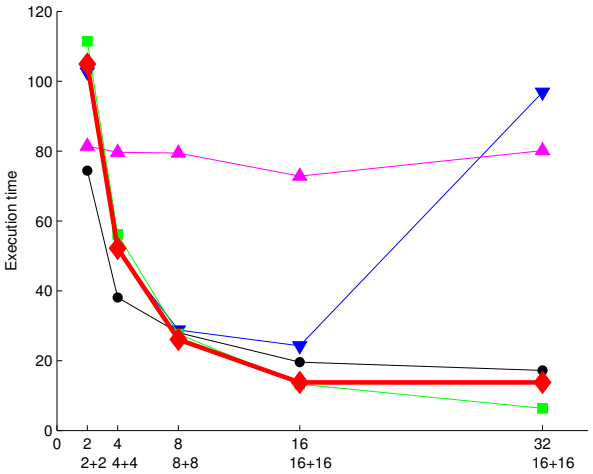
(b) Genome



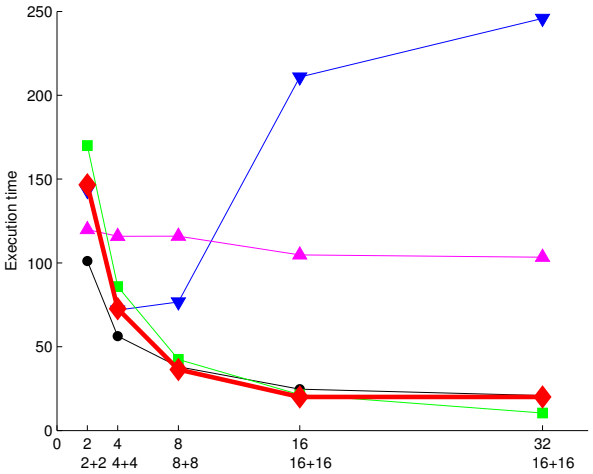
(c) Labyrinth



(d) SSCA2



(e) Vacation-Low



(f) Vacation-High

Fig. 5: STAMP benchmarks with different STMs. The x-axis reports the number of used threads, which is N for the standard STMs and N+N for STM^2, for N=2,4,8,16. For N=32, we compare STMs performance to STM^2 using 16+16 threads (we repeat this value in correspondence of N=16 and N=32 to facilitate comparison with the other STMs having equal hardware resources). In the graphs, lower is better.

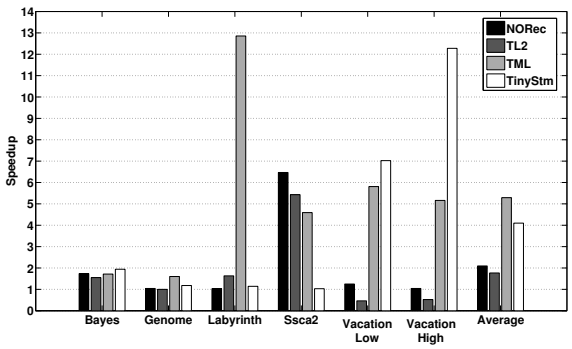


Fig. 6: Speedups of STM^2over tested STMs for STAMP applications using the same amount of hardware resources (32 hardware threads).

the read-sets. Increasing the number of concurrent threads also raises the probability that a transaction commits while another thread is validating its read-sets. The result is that commit time increases with the number of threads. Conversely, TinySTM and STM^2 acquire ownership of shared memory locations when a thread issues a write operation and maintain it throughout the execution of the short transaction, which proves to be a good choice for this particular case.

NOrec Unlike TinySTM, TL2 and STM^2, NOrec does not perform any bookkeeping. Runtime overhead is negligible and limited to the initialization and finalization of transactions. However, NOrec only allows one active writer transaction in the system at a time. We, thus, expect NOrec to perform better than STMs with bookkeeping when the level of contention is limited, but to gradually reduce performance when the number of writers per transaction increases (which depends on the application and the number of concurrent threads). Indeed, NOrec scales nicely for all applications except *Bayes* and *SSCA2*. On the other hand, bookkeeping allows STM^2 to support several concurrent writer transactions at a time. The result of combining concurrent writers and reduced runtime overhead is that STM^2 usually performs better or equal than NOrec. For applications with limited contention (*Genome*) or with a limited number of concurrent writers (*Vacation-Low* and *Vacation-High*), STM^2 and NOrec perform similarly. When the level of contention increases or there are several writers per transaction, STM^2 outperforms NOrec. This happens with *Bayes* (high contention) with N=32 threads (1.7x speedup) and with *SSCA2* (high number of concurrent writers) beyond N=4 threads (up to 6.4x speedup). In these cases, STM^2 keeps scaling up to 32 hardware threads, while rollbacks and re-validation limit NOrec's performance.

NOrec and STM^2 perform similarly for *Labyrinth*, which presents large transactions and a high conflict rate. For this kind of application, lazy conflict detection STMs usually have a disadvantage with respect to eager conflict detection STMs. NOrec, however, is able to make up for this disadvantage with its value-based conflict detection. The results is that NOrec introduces fewer false aborts than TL2.

STM^2 provides, on average, 2.1x speedup over NOrec (see Figure 6). Since NOrec has a low runtime overhead, our results prove that eager conflict detection and bookkeeping overhead is effectively absorbed by the auxiliary threads.

TL2 Both TL2 and STM^2 perform lazy data versioning, though TL2 detects conflicts at commit stage while STM^2 detects conflicts during a transaction's execution. Lazy conflict detection STMs introduce negligible validation runtime overhead but they may suffer from higher abort overhead, caused by the "wasted" time spent executing transactions that will abort, and high commit overhead (lock acquisition). Our experiments show that, indeed, TL2 performs well for applications with low contention, like *Vacation-Low* and *Vacation-High* (which perform mainly read operations) or *Genome* (limited contention). While STM^2 and TL2 are essentially equivalent for *Genome*, TL2 performs better than STM^2 when running *Vacation-Low* and *Vacation-High*. Applications with high contention or high commit rate, instead, pose challenges to TL2 due to frequent modifications of a centralized data structure [18]. For these kinds of applications, STM^2 outperforms TL2: with 32 hardware threads, STM^2 achieves 1.5x speedup over TL2 for *Bayes*, 1.6x speedup for *Labyrinth*, and 5.4x speedup for *SSCA2* (Figure 6). Note that, while STM^2 performs significantly better than TL2 for high contention applications, TL2 does not substantially outdistance STM^2 for applications with low-contention or applications with mostly-read transactions. The results show that, on average, STM^2 shows a 1.8x speedup over TL2.

TML STM^2 performs consistently and substantially better than TML for all STAMP benchmarks. While a global lock provides low runtime overhead and intrinsically guarantees serialization, performance is usually poor for applications with high contention and/or large transactions. Our experiments show that the serialization overhead induced by the use of a global lock with a high number of threads considerably reduces overall performance. As Figure 6 shows, STM^2 exceeds TML for applications with high contention, like *Bayes* (1.7x speedup) and *Labyrinth* (12.8x speedup), large read-sets, like *Vacation-Low* (5.8x speedup), and read-write transactions, like *SSCA2* (4.6x speedup).

Summary Our results show that, on average, STM^2 outperforms all tested STMs. For applications with high contention (*Bayes* and *Labyrinth*) or bursting and irregular transactions with a high number of concurrent writers (*SSCA2*), STM^2 provides high speedups over lazy conflict detection STMs (up to 6.4x) or single global lock STM (12.8x). For applications with low contention and mostly-read transactions (*Vacation-Low*, *Vacation-High* and *Genome*), STM^2 performs well with respect to lazy conflict detection and no bookkeeping STMs: Only TL2 outperforms STM^2 when running *Vacation-Low* and *Vacation-High*, while STM^2 still outperforms NOrec and TML for *Vacation-Low* and *Vacation-High* and NOrec, TML and TL2 for *Genome*. STM^2 provides the same performance, or even outperforms, lazy conflict detection and no-bookkeeping STMs for applications where lazy conflict detection provides advantages. Finally, STM^2 exceeds TinySTM with all the applications and provides speedups up to 12.3x over TinySTM for applications that are critical for eager conflict detection

STMs, such as *Vacation-High*.

Our proposal largely overlaps computation and STM management operations and effectively reduces runtime overhead. *STM*^{2} remarkably improves performance and provides the advantages of eager conflict detection STMs with the limited runtime overhead of lazy conflict detection STMs. Note that, given that all STMs run the same number of transactions and that *STM*^{2} is faster than the other STMs (between 1.8x and 5.2x with 32 hardware threads, on average), it follows that *STM*^{2}'s throughput (measured in number of transactions per second) is higher, despite the use of dedicated hardware threads to run STM operations.

VI. Related Work

The use of extra threads to help the computation of main threads has been previously proposed, though for different goals. Auxiliary threads are usually employed to resolve unpredictable branches or cache misses that the main threads would have to stall upon otherwise [7], [8], [26] or to prefetch data from memory. Zilles et al. [28] explore using separate threads in a multithreading processor for exception handling to avoid squashing in-flight instructions.

Mehrara et al. [19] and Milovanovic et al. [20] propose the use of an auxiliary thread in lazy conflict detection STMs. Both proposals, however, use a centralized dedicated thread. Mehrara et al. [19] present STMlite, a software transactional memory that aims to automatically parallelize sequential applications. In this work, all the application threads send their memory modifications to the auxiliary thread, which, at commit time, serially performs the updates. This approach provides benefits when the lock contention is high by serializing the memory updates in one thread. Milovanovic et al. [20] propose a combined OpenMP and STM runtime system based on an STM library, which performs lazy conflict detection and lazy versioning management. The authors introduce an additional separate thread for asynchronous eager conflict detection that aims to detect conflicts before the commit time and, therefore, reduce wasted time for doomed transactions. However, the authors did not implement an advanced synchronization mechanism between transactions and the associated dedicated thread. This unnecessarily forces the system at commit phase to repeat several checks already performed during the eager conflict detection phase. Both proposals suffer from a lack of scalability: the centralized auxiliary thread may become a bottleneck, especially for a high count of threads.

Casper et al. [6] use an FPGA connected to the AMD HyperTransport bus to accelerate conflict detection using bloom filters. Conflict detection is performed at commit phase by the accelerator and it is synchronous with the threads running on the normal cores which have to wait for the accelerator to complete conflict detection.

In contrast to previous work, *STM*^{2} is a fully parallel STM: *STM*^{2} assigns a dedicated auxiliary thread to each application thread for managing validation and bookkeeping involved in the main computation. These threads run on dedicated cores/hardware threads. Since each application thread has its own auxiliary thread for their transactional operations, unlike STMlite [19] and the approach proposed by Milovanovic et al. [20], we avoid having a single point of serialization. Finally, *STM*^{2} and the work proposed by Casper et al. [6] are orthogonal: *STM*^{2}'s auxiliary threads could be accelerated through dedicated hardware, such as FPGAs.

VII. Conclusion and Future Work

In conclusion, we have presented *STM*^{2}, a parallel STM system that offloads STM time-consuming management operations to auxiliary threads running on separate hardware threads. To the best of our knowledge, *STM*^{2} is the first parallel STM that takes advantage of secondary hardware threads to accelerate STM functions and reduces overall overhead.

We tested *STM*^{2} on an IBM POWER7 system, an aggressively multithreading processor designed for high performance. By overlapping computation and STM operations, *STM*^{2} generally outperforms current, state-of-the-art STMs, namely TinySTM, TL2, NOrec and TML. Our experiments show average speedups between 1.8x and 5.2x over the tested STMs, with peaks up to 12.8x, with 32 hardware threads. We conclude that auxiliary threads effectively absorb the overhead of transactional bookkeeping and conflict detection, considerably improving the overall performance.

As future work we plan to investigate hardware (dedicated communication buffer) and software (message packing) techniques to reduce the communication overhead between application and auxiliary threads.

Acknowledgment

Our thanks to the anonymous reviewers and to Robert W. Wisniewski from IBM, Nehir Sonmez and Vasileios Karakostas from BSC for their helpful comments. This work is supported by the cooperation agreement between the Barcelona Supercomputing Center and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIN2007-60625, JCI-2008-3688, 2009501052, and TIN2008-02055-E, by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (IST-004408) and by the European Commission FP7 project VELOX (216852). Gokcen Kestor is also supported by a scholarship from the Government of Catalonia.

References

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

[2] J. Abeles, L. Brochard, L. Capps, D. DeSota, J. Edwards, B. Elkin, J. Lewars, E. Michel, R. Panda, R. Ravindran, J. Robichaux, S. Kandadai, and S. Vemuganti. Performance guide for HPC applications on IBM power 755 system, 2010.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, pages 483–485, 1967.

[4] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of The International Symposium on Workload Characterization*, 2008.

[5] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software Transactional Memory: Why is it only a research toy? *ACM Queue*, pages 46–58, 2008.

[6] J. Casper, T. Oguntebi, S. Hong, N. G. Bronson, C. Kozyrakis, and K. Olukotun. Hardware acceleration of transactional memory on commodity systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 27–38, 2011.

[7] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 186–195, 1999.

[8] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 14–25, 2001.

[9] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 67–78, 2010.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the International Symposium on Distributed Computing*, pages 194–208, 2006.

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

[12] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating systems*, pages 58–69, 1998.

[13] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 14–25, 2006.

[14] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[15] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. RMS-TM: A comprehensive benchmark suite for transactional memory systems. In *Proceeding of the International Conference on Performance Engineering*, pages 335–346, 2011.

[16] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. A. Yelick. ExaScale computing study: Technology challenges in achieving exascale systems. Technical Report DARPA-2008-13, DARPA IPTO, September 2008.

[17] C. Lameter. Effective synchronization on Linux/NUMA systems. In *Proceedings of the Gelato Federation Meeting*, 2005.

[18] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *Workshop on Transactional Computing*, 2009.

[19] M. Mehrara, J. Hao, P. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 166–176, 2009.

[20] M. Milovanović, R. Ferrer, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, and M. Valero. Multithreaded software transactional memory and OpenMP. In *Proceedings of the Workshop on MEmory performance: DEaling with Applications, systems and architecture*, pages 81–88, 2007.

[21] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the Annual Symposium on Parallel Algorithms and Architectures*, pages 221–228, 2007.

[22] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *Proceedings of the Annual Symposium on Principles of Distributed Computing*, pages 338–339, 2007.

[23] M. F. Spear, A. Shriraman, L. Dalessandro, and M. L. Scott. Transactional mutex locks. In *Workshop on Transactional Computing*, 2009.

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

[25] SUN Microsystem. OpenSPARCTM T2 Core Microarchitecture specification.

[26] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating systems*, pages 257–268, 2000.

[27] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. Rubio, F. Rawson, and J. Carter. Architecting for power management: The IBM POWER7 approach. In *Proceedings fo the IEEE International Symposium on High Performance Computer Architecture*, 2010.

[28] C. B. Zilles, J. S. Emer, and G. S. Sohi. The use of multithreading for exception handling. In *Proceedings of the Annual International Symposium on Microarchitecture*, pages 219–229, 1999.