A Comparative Analysis of STM Approaches to Reduction Operations in Irregular Applications

Manuel Pedrero[a], Eladio Gutierrez[a], Sergio Romero[a], Oscar Plata[a]

[a] *Dept. Computer Architecture*
Universidad de Málaga
29071 Málaga, Spain

Abstract

As a recently consolidated paradigm for optimistic concurrency in modern multicore architectures, Transactional Memory (TM) can help to the exploitation of parallelism in irregular applications when data dependence information is not available up to runtime. This paper presents and discusses how to leverage TM to exploit parallelism in an important class of irregular applications, the class that exhibits irregular reduction patterns. In order to test and compare our techniques with other solutions, they were implemented in a software TM system called ReduxSTM, that acts as a proof of concept. Basically, ReduxSTM combines two major ideas: a sequential-equivalent ordering of transaction commits that assures the correct result, and an extension of the underlying TM privatization mechanism to reduce unnecessary overhead due to reduction memory updates as well as unnecesary aborts and rollbacks. A comparative study of STM solutions, including ReduxSTM, and other more classical approaches to the parallelization of reduction operations is presented in terms of time, memory and overhead.

Keywords: Transactional Memory, Concurrency, Reduction operations, Irregular applications

1. Introduction

The availability of multiple cores sharing a global memory in modern commodity computers is having a strong influence in how applications need to be designed so that they can benefit from all this available computational power. When decomposing a problem into a number of concurrent tasks, the achievable performance is subject to the right resolution of data and control dependencies. In general, dependencies are managed in a conservative way, specially when they are solved at compilation time. Such is the case for applications exhibiting irregular memory patterns whose dependences could not be known until execution time. In this context, transactional memory (TM) [1] can provide an optimistic concurrency support on multicore architectures, helping programmers to exploit parallelism in irregular applications when data dependence information is not easily analyzable or even available before runtime.

TM has emerged as an alternative way to coordinate concurrent threads. TM provides the concept of transaction, a construct that enforces atomicity, consistency and isolation to the execution of a computation wrapped in the transaction. Transactions are allowed to run concurrently, but in a way that the results are the same as if they were executed serially.

In a TM system, transactions are speculatively executed and their changes are tracked by a data version manager. If two concurrent transactions conflict (write/write, read/write the same shared memory location), one of them must abort. After restoring its initial state, the aborted transaction retries its execution. When a transaction finishes its execution, it commits, making its changes in memory definitive. The design of the version manager is eager if changes are immediately translated into memory and a undo-log is used to store the old values (to be used in case of abort). By contrast, in a lazy version manager, updates are held in a write-buffer and not written in memory until commit takes place. In a similar way, the conflict manager may detect the conflict when it occurs (eager), or may postpone the conflict check until commit time (lazy). Many TM systems have been proposed in the last two decades, implemented either in software (STM), in hardware (HTM) or, hybrid, a combination of both (HyTM) [2].

Encouraged by TM benefits, efforts have been devoted to leverage TM for extracting parallelism from sequential applications. In fact, many basic operations in a TM system, like detecting and solving memory conflicts, buffering of memory updates, and execution rollbacks, are also required by speculative multithreading (SpMT), or thread-level speculation (TLS) [3]. These techniques have been shown useful for finding parallelism in single-threaded programs.

In general, extracting thread parallelism from a sequential program requires decomposing the program into tasks and the correct computation of dependencies between these tasks. Computing such dependencies statically (at compile time) is often not possible for many complex applications. In these cases, SpMT/TLS could be useful to find parallelism, as no static data dependence analysis is demanded (see for instance, [4]). In this way, the optimistic concurrency exploited by TM systems

Email addresses: mpedrero@uma.es (Manuel Pedrero), eladio@uma.es (Eladio Gutierrez), sromero@uma.es (Sergio Romero), oplata@uma.es (Oscar Plata)

may help to parallelize irregular applications. Tasks, defined as code sections out of the sequential program, may be executed as concurrent transactions so as the TM system is in charge of tracking memory accesses at runtime in order to detect and solve conflicts (data dependencies) between transactions. Note, however, that some ordering constraints amongst transactions must be fulfilled to avoid violations of data dependencies and to assure correct results. In general, transactions must commit in an order that preserves the sequential semantics.

This paper presents and discusses how to leverage TM to exploit parallelism in an important class of irregular applications, the class that exhibits irregular reduction patterns. In order to test and compare our techniques with other solutions, they were implemented in a system called ReduxSTM, that acts as a proof of concept. Basically, ReduxSTM takes advantage of both, a transaction commit ordering and the TM privatization mechanism, in order to avoid unnecessary aborts and rollbacks and reduce overhead due to memory updates.

The rest of the paper discusses irregular reduction patterns and the most common techniques to extract parallelism from them. Next, some related work is introduced. Section 5 presents our proposal to support full and partial irregular reductions using TM, and the design of the ReduxSTM system. Next section discusses an experimental evaluation of our proposal and a comparison with other TM systems available. Finally, some conclusions are drawn.

2. Reduction patterns in irregular applications

A reduction statement is a pattern of the form $O = O \oplus \xi$, where $\oplus$ is a commutative and associative operator applied to the memory object O, and ξ is an expression computed using objects not depending on O. A reduction or histogram loop is a computational pattern that includes one or several reduction statements with the same or different memory objects but with the same operator for each object. In addition, no references to those memory objects can occur in other parts of the loop outside the reduction statements [5].

Reductions are found in the core of many scientific and engineering applications such as sparse matrix computations or finite element solvers, and they are frequently associated with irregular access patterns. Examples of reduction loops are shown in Fig. 1, where the reduction operation is the sum which is applied to scalars or elements of reduction arrays. The irregular nature of the operation comes from the access through the indirection arrays, acting as subscripts of the reduction array, which, in turn, are subscripted by the loop index.

From the data dependence viewpoint, memory accesses inside the reduction loop could give rise to loop-carried dependencies. An optimistic situation occurs when the only true dependencies are caused by the reduction statements. In such a case, the iterations of the loop can be arbitrarily reordered without altering the final result, as a consequence of the commutativity and associativity of the reduction operator. Hence the loop can be executed in parallel in spite of the reduction dependencies. On the other hand, there are situations where the reduction conditions are not completely fulfilled. Examples of these situations are when the reduction object is accessed in the loop outside the reduction statements, when several different reduction operators are applied to the same reduction object, or when the loop includes other true cross-iteration dependencies apart from the reduction statements [6, 7]. In the first two cases, it is possible that the reduction conditions are fulfilled for some of the memory accesses to the reduction object, but not for all of them. This situation is referred to as partial reduction (see an example in Fig. 2).

As a common problem, a not small number of solutions to the parallelization of reductions can be found in the literature, each one focused on improving specific aspects of the parallel programming (performance, memory requirements, complexity, etc). We distinguish three major approaches to parallelizing reduction loops: (a) methods that guarantee the mutual exclusion between accesses to the reduction array [8], (b) methods that accumulate in a private storage during the original loop execution and update the reduction array with the partial accumulations of each thread in a final stage [9, 10, 11], and (c) methods that split the reduction array by using an inspection phase that is in charge of determining the computation assigned to each thread [12, 13]. The third group drops out of the scope of this work due to its low programmability, as they require a substantial effort and knowledge about the application.

Mutual exclusion. One obvious way of avoiding race conditions is transforming the original sequential loop into a DOALL, and to assure that only a single thread accesses to the shared data each time by enclosing such data accesses in a *critical section*. Drawbacks of this technique are the degree of serialization and the cost of synchronization in typical multi core processors. The degree of serialization is basically determined by the number of conflicting iterations and by the particular implementation.

The serialization issues in critical sections can be improved by using *fine-grained locks*. Basically, this method associates one lock with every element in the reduction array. In this way, two concurrent threads will be able to simultaneously access to different elements, but not the same one. Although it is an efficient approach, supporting such a large number of locks can result in high memory requirements depending on the size of the reduction array.

Also *atomic operations* can be used to guarantee the mutual exclusion between reduction operations. These operations block the memory bus until operations are completed, avoiding other processes to access shared data simultaneously. The use of atomic operations is usually a more efficient approach and exhibits a negligible memory consumption, however, its availability is strongly dependent on the hardware architecture. For example, most of the commodity multiprocessors do not include floating point support for atomic operations, which are the basis of a high proportion of scientific applications. Furthermore, they present other limitations like the inability to group several operations in a single atomic block.

Task-based models, which can be suitable to tackle the parallelization of reduction loops, can be also included in this group. For example, using task supporting dependencies (like

```
Read subscript arrays:
    edge(1,*), edge(2,*)

do itime=1,nTimes
 do i=1,nEdges
   n1 = edge(1,i)
   n2 = edge(2,i)

   Compute ζ1, ζ2, ζ3

   vel(1,n1)=vel(1,n1)+ζ1
   vel(2,n1)=vel(2,n1)+ζ2
   vel(3,n1)=vel(3,n1)+ζ3

   vel(1,n2)=vel(1,n2)-ζ1
   vel(2,n2)=vel(2,n2)-ζ2
   vel(3,n2)=vel(3,n2)-ζ3
 enddo
  .....
enddo
```

(a)

```
Compute subscript arrays:
     m(*),
     mbeg(*),
     mend(*)

do irow=1, nRows
  do i=1,jdt+1
   im =m(i)
   imb=mbeg(i)
   ime=mend(i)
   do is=imb,ime,2
     Compute ζ1, ζ2...
      do ilev=1,2*jdlev
       f1(ilev,im)=f1(ilev,im)+ζ1
       f2(ilev,im)=f2(ilev,im)+ζ2
       .....
      enddo
   enddo
   .....
enddo
```

(b)

```
do ihop=1, nHops
  Update subscripts: B1(*),B2(*)
  do itime=1,nTimes
    do ih=1,nParticles
      i=B1(ih)
      j=B2(ih)
      Compute r(i,j), ζ(i,j), η(i,j), θ(i,j)

      if (r .lt. CutOff) then
       AX(i)=AX(i) + ζ
       AX(j)=AX(j) - ζ
       AY(i)=AY(i) + ζ
       AY(j)=AY(j) - ζ
       U = U + η
       P = P + θ
      endif
    enddo
    .....
  enddo
enddo
```

(c)

Figure 1: Outline of some reduction kernels of interest: (a) Unstructured, (b) Legendre transform, and (c) 2D Molecular dynamics.

OpenMP task depend or [14, 15]) reduction variables can be expressed as an input-output dependency of the task. In irregular codes, the main limitation of these approaches derives from their capacity of expressing complex dependencies with enough precision, like indirections or subscripted subscripts; or when these dependencies cannot be known when the task is created because indirections are computed inside the task.

Privatization. One simple solution to avoid race conditions is to distribute the iteration space of the loop into threads, each of which performs its reductions over a local reduction space. A preamble is necessary in order to initialize the private reduction space to the identity (neutral) element of the reduction operator (copy-in). Similarly, a final reduction phase must accumulate all private reduction values into the (global) reduction array (copy-out).

Two representative approaches are Replicated Buffer and Array Expansion. While the first one proposes a local copy of the reduction array per thread, in the Array Expansion method, the reduction array is extended by adding a new dimention for the number of concurrent threads. In this way, the copy-out phase can be fully parallelized.

The privatization technique is quite direct to implement with no additional work during the loop execution, and works very well if the access pattern is dense (most of the elements in the reduction array are written during the execution). If access pattern is sparse, then many elements may stay unmodified and too much time can be wasted by reducing with the neutral element. Furthermore, its work increases linearly with the number of threads. However, the main drawback of privatization-based techniques is the memory requirements because the reduction memory space is multiplied by the number of threads.

3. TM approach to reduction patterns

When considering the parallelization of a full reduction loop, a TM straightforward approach is to replace critical sections by TM sections as shown in Fig. 3. It is a simple solution from the viewpoint of programmability, and potential conflicts

```
for (i=0; i<N; i++){
  A[K[i]] = ...;
  ... = A[L[i]];
  A[R[i]] = A[R[i]] ⊕ ξ;
}
```

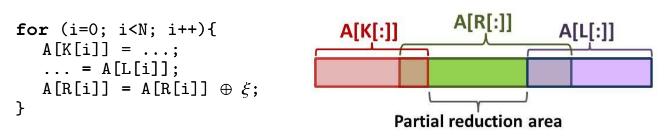

Figure 2: Example of loop with a partial reduction area. Indirect subscripts, K, L and R, constrain the accesses to array A as shown on the right. Observe that areas A[K[:]] and A[L[:]] do not overlap.

```
for (i=0; i<NInd; i++){
  Compute ξ1, ξ2
  #pragma omp critical{
    ...
    A[idx1[i]] =
                A[idx1[i]] ⊕ ξ1
    A[idx2[i]] =
                A[idx2[i]] ⊕ ξ2
    ...
  }
}
```

(a)

```
for (i=0; i<NInd; i++){
  Compute ξ1, ξ2
  BEGIN_XACT()
     ...
     TM_WRITE(A[idx1[i]]],
              TM_READ(A[idx1[i]] ⊕ ξ1))
     TM_WRITE(A[idx2[i]]],
              TM_READ(A[idx2[i]] ⊕ ξ1))
     ...
  END_XACT()
}
```

(b)

Figure 3: TM straightforward approach (b) as replacement of critical section (a).

resulting from indirect accesses are managed by the TM system. In case of low contention scenarios, TM will keep a good concurrency level in contrast to the serialization derived from the used of critical sections (mutex/spin locks, atomics, etc.).

If certain transaction order is introduced in a TM system, in such a way that commits take place as in the sequential code, then it can be used as a support for TLS. This is specially interesting when classical techniques are not suitable because the reduction condition is not fully satisfied.

One example of this situation is shown in Fig.2, where a loop contains a reduction sentence that may conflict with other reads and writes. Nevertheless there may be a subset of iterations fulfilling the reduction condition if the subscripts are subject to the pattern shown in the figure [6]. Fig. 4 sketches another example where some reads and writes through pointers can be aliases of the reduction variables. This fact prevents the

```
for (termptr = ... ; termptr = termptr->nextterm) {
    ...
    for (netptr = ... ; netptr=netptr->nterm) {
        ...
        *costptr += ...; // Reduction sentence
    }
    ...
    rowsptr = tmp_rows[net] ;
    for (row = 0 ; rowsptr[row] == 0 ; row++ ){
        ...
    }
    ...
    tmp_num_feeds[net] = f ;
    ...
    tmp_missing_rows[net] = -m ;
    ...
    delta_vert_cost += ( ... ); // Reduction sentence
}
```

Figure 4: Sketch of a loop from the procedure *new_dbox_a()* found in the code *300.twolf* (benchmark SPEC CPU2000). Here pointer aliases make unable to know statically if it is a full reduction loop, although reduction sentences are included.

loop from being recognized statically as a reduction loop [7].

These patterns are known as *partial reductions* which have been addressed in the literature by means of speculative approaches, as detailed in the next section.

4. Related work

The idea of leveraging reductions in speculative parallelization of loops is not new. In [16], LRPD is proposed, featuring a run-time data dependence test that is launched joined with a speculative *do-all* parallel execution of a loop. If the test fails, the loop execution is discarded and must be re-executed serially. This test can detect and privatize reduction statements avoiding the need of serialization. Privateer [5] extends LRPD idea handling dynamic and recursive data structures with an inspector/executor scheme. Another TLS approach for reduction patterns is presented in [7] focused on partial reduction variables (PRV). By means of a PRV detection algorithm, speculative tasks are created, which are executed in parallel. Threads stall when a reduction variable is accessed by a non reduction operation. Conflicts between reductions are not causing stalls as they operate on private replicas that must be committed eventually. Specific architectural support is needed.

A solution to exploit reduction properties in an STM system is proposed in [17, 18]. This work is restricted to histogram loops in which iterations can be safely reordered, as the only allowed dependences come from reduction variables. This proposal disables the conflict detection for reduction memory operations, leverages privatization to accumulate locally to each transaction partial reduction updates, and adapts the commit phase to accumulate partial reduction updates to global memory.

Other more general TM-based solutions that support speculative execution that can be applied to irregular reductions are proposed in [19, 20, 21]. IPOT [19] features a programming model aimed to parallelize sequential code. A master thread spawns chunks of instructions in parallel using TM-like structures with additional annotations that enable the relaxation of consistency constraints. IPOT requires additional compiler and architectural support for speculative execution, ordering and conflict detection. Similarly, ALTER [20] proposes another TM-style TLS scheme. Variables can be annotated in order to use a more permissive consistency checking like *out-of-order* (TLS with no ordering) or *stale read* (ignoring read dependences). A variable can be declared as reduction according to a reduction operator. ALTER uses a fork-join scheme for the annotated loops, executing chunks of iterations in TM-like structures and validating results at the end of the chunks. Finally, an automatic parallelization technique using ordered hardware transactions is proposed in [21]. The model relies in hardware performance counters to detect loops and identify candidate instructions to parallelize, and generates parallel code that relies in TM to detect and resolve any conflict.

More recently an STM approach called RMW (Read-Modify-Write without aborts [22]) adds specific TM support for read-modify-write patterns, in which reductions can be included. Nevertheless this technique does not guarantee any transaction ordering and its scalability is limited to few RMW variables, which excludes large vectors as commonly happen in reduction loops.

5. ReduxSTM Design

In addition to be a straightforward substitute for critical sections, TM carries out an underlying selective privatization of those tracked shared variables. The idea we introduce here is to harness this underlying privatization as a mechanism to tackle reduction variables allowing skipping dependencies. The transaction commit phase will be in this case the moment when the transactional reduction private values are accumulated into the shared version of the variables in memory. Commutative and associative properties guarantee that this can be done safely. In this way the programmer or compiler only needs to recognize reduction sentences but not whether the whole loop verifies the condition of reduction. Besides, with this strategy, full or partial reductions can be treated with the same degree of programmability. Reduction sentences that conflict with other memory accesses to the same position will be treated as transactional conflicts by aborting if necessary.

Our proposal, ReduxSTM, supports patterns that cannot be determined statically whether they are full or partial reductions. Hence, it overcomes the limitations of previous works [17, 18]. ReduxSTM is conceived as a proof of concept to test TM techniques to support partial reduction patterns. As such, it could be implemented as an extension of an existing STM system, or as a stand-alone system. It does not need to rely on special compiler or architectural support. A feature comparison of ReduxSTM with the other aforementioned approaches is summarized in Table 1.

5.1. Features

Reduction support. One key feature of ReduxSTM is the ability to exploit the underlying TM selective privatization mechanism for reduction operations. Although not strictly mandatory for speculative parallelization, this feature becomes useful to remove a fraction of accesses to shared memory, hence reducing contention between transactions. Exploiting this implicit

Table 1: Trade off among parallelization techniques.

	Memory Requirements	Potential Parallelism	Synch. Overhead	Merging Overhead
Privatization	very high	very high	very low	yes
Critical Sections	very low	very low	high	none
Fine-grained Locks	high	high	high	none
Atomic Ops	none	very high	high	none
TM straightforward	low	high/medium	low	yes
ReduxSTM	low	very high	low	yes

privatization allows to eliminate unnecessary transaction conflicts, improving concurrency.

For this purpose, a new primitive for memory reduction has been added to the existing ones. This primitive is handled by the conflict, version and commit managers as a third basic memory operation: read (R), write (W) and reduction (Rdx). The semantics associated with Rdx is the combination of two operations: a read followed by a write on the same memory location of the reduced value. In this way, the new primitive $Rdx(add, val, \oplus)$ is functionally equivalent to $W(add, R(add) \oplus val)$, but it enables an improved execution.

Ordered transactions. A second important feature that we demand to ReduxSTM is to keep ordering constraints between transactions. When decomposing an application in speculative tasks, the concurrent execution of these tasks must preserve the sequential semantics to ensure the correctness of the final result. Besides, although fully-reduction loops may be safely reordered, reduction operations may coexist with other reads or writes on the same memory locations, invalidating reduction conditions, thus depending on these ordering constraints to ensure correct results.

Because of that, ReduxSTM commits transactions in accordance with the sequential execution order. This constraint is added to the commit manager at the expense of a certain performance cost due to delays in commit phases, and also means that the memory updates performed by each transaction during its commit phase are serialized following this order.

Notwithstanding, potential aborts and rollbacks caused by false dependences can be avoided thanks to this ordering knowledge [3, 18], as shown in Table 2. The first column specifies two memory operations executed by two different transactions where the first one must commit before the second one (in sequential order). For instance, R−W represents an anti-dependence. The second column corresponds to the behavior of a standard TM system, while the last column consider additional support for order and reductions.

Version management. Supporting reduction operations (that may conflict with reads and writes) together with ordered commits leads the design to a lazy approach, that is, updates to memory are held in private buffers during the execution of the transaction, and they are consolidated into memory during commit. Two kinds of private buffers, that store disjoint information, are considered: the write buffer (typically used by lazy-versioned STMs to store written values) and the reduction buffer (a new space to store the values partially accumulated by the transaction). Note that a reduction buffer is required for each different reduction operator under consideration.

Table 2: Possible transactional conflicts.

	Standard STM	ReduxSTM (Order + Reductions)
R−W	abort	no conflict
W−R	abort	abort
W−W	abort	no conflict
Rdx−R	as R-W-R	abort
R−Rdx	as R-R-W	no conflict
Rdx−W	as R-W-W	no conflict
W−Rdx	as W-R-W	no conflict
Rdx−Rdx	as R-W-R-W	no conflict

Every time a transactional reduction is issued, the write buffer is searched for the memory address. If it is found, the value specified in the reduction operation is reduced with the value stored in the buffer. However, this new computed value is kept in the write buffer because the reduction condition is not fulfilled. Otherwise, the reduction value is operated with the corresponding accumulated value in the reduction buffer (or with the neutral element in case of the first reduction on this memory position) in order to update it in the reduction buffer. If a position stored in the reduction buffer is written later in the same transaction, it must be withdrawn from the reduction buffer and inserted in the write buffer because the reduction condition is broken. Notice that if a transaction performs a transactional read in a variable previously reduced within the transaction, the value must be obtained as a combination of the data in memory with the associated value in the reduction buffer.

Conflict detection. Transactions in ReduxSTM perform validation (or invalidation) during its commit phase making conflict detection lazy [23].

Commit management. As only one transaction can be committing at a time, such transaction is responsible for checking conflicts and for validating itself or invalidating other transactions conflicting with it. Ordered finalization guarantees the atomic nature of the commit phase acting as the main synchronization mechanism.

The commit phase is subject to the particular characteristics of the implementation strategy, which are discussed in subsection 5.2. Independently of its implementation the commit phase must be in charge of: (a) Waiting for the turn (order) to commit; (b) Checking conflicts and trigger aborts suitably; (c) Updating main memory consolidating the values stored in the write and reduction buffers (reduction values need to be accumulated with those in memory).

5.2. Implementation

Two well-known STM algorithms, Commit Time Invalidation and Time-Based Validation, were selected to implement two versions of ReduxSTM. These algorithms were chosen because they are suitable to implement straightforwardly the described features to support reductions. These two versions were implemented from scratch.

Commit Time Invalidation (CTI). In this approach the committing transaction will mark as killed (invalidate) those subse-

quent transactions that have conflicts with it following a similar scheme to the one described in [24]. Data conflicts are detected based on memory addresses. At the beginning of its commit phase, each transaction must check if it has been marked as killed (validation). If so, it must abort. If not, it can start consolidating its written and reduced values into memory (only one transaction can be in this phase). After consolidating values, the committing transaction proceeds to check its data sets against other transactions' data sets, invalidating those transactions according to rules in table 2. Three data sets are defined: read, write and reduction sets. In our implementation, these sets have been represented by Bloom filters that are accessed through a hash of the memory addresses.

Time-based validation (TS). In contrast to the previous strategy, this one uses time stamps to register *when* read and updates take place [25]. In our implementation these time stamps are represented by the transaction order number. Conflicts are detected by validating the data accessed by each transaction in terms of this information.

Each transaction maintains a private array for its read time stamp. A global write time stamp structure keeps track of the time stamp when this position was updated (by any write and reduction from any transaction). In its commit phase, the transaction validates its reads by comparing its private reads' time stamp with global write time stamp. If validation is successful, the transaction consolidates its writes and reductions into memory, updating the global write time stamp accordingly. To limit the memory requirements, these structures have a fixed size and they are accessed by applying a hash function to the associated memory address.

6. Experimental Evaluation

The goal of this section is to evaluate experimentally how STM techniques perform in codes dominated by reduction operations, specially when special reduction support is incorporated to the STM system, such as the proposed ReduxSTM. Experiments were conducted on a server with 256 GB RAM and a 16-core/32-threads Intel Xeon E5-2698 processor at 2.3 GHz. The system runs Linux kernel 3.13 (64 bits) and all programs were compiled using GNU GCC 4.8.2 (optimization option `-O2`).

TinySTM (v.1.0.5) [25] was used as the baseline state-of-the-art STM system. Both the standard and the ordered versions of TinySTM have been tested. Note that if the STM system does not preserve the sequential order, correct results are obtained only for those codes that can be safely reordered, such as full reduction loops (e.g. histograms). Nevertheless, in codes where reduction sentences may conflict with other read or write operations the order may need to be preserved.

6.1. Experimental Setup

Several representative codes have been used for evaluation. These codes, briefly described in table 3, spend a significant execution time in irregular reductions. Fluidanimate, MD2, Unstructured and Legendre kernels correspond with full reduction loops, as shown in Fig. 1. Additionally, 300.twolf includes reduction patterns combined with other potentially conflicting reads and writes, as sketched in Fig. 4.

Table 3: Tested benchmarks.

Fluidanimate [26]	Part of PARSEC benchmark suite. This application uses the Smoothed Particle Hydrodynamics (SPH) method for fluid simulations for interactive real-time animations such as computer games. Two different configurations were tested. The *simmedium* input simulates 100K particles during 5 time frames and *native* configuration simulates 500K particles during 500 time frames.
MD2 [27]	This application simulates 2D molecular dynamics for very large systems when those molecules have short-range interactions. It involves the determination of the forces that affect each particle. To reduce the complexity of this problem when the interactions between the particles are short-scoped, *MD2* uses a neighbor list that limits the interactions.
Unstructured [28]	Unstructured solves the Euler equations for physical simulations involving fluid dynamics and material deformations. Each time step, the application computes the forces and velocities for each node of a mesh.
Legendre [29]	This kernel uses Legendre transforms used in numerical weather prediction. It uses 3d arrays to represent different conditions. In each time step, two subroutines are called to perform inverse an direct transformations, involving irregular reductions due to indirections in the array accesses.
300.twolf [30]	This is a place and route simulator code included in the SPEC 2000 benchmark suite. It contains some interesting reduction patterns like those in routine `new_dbox_a()` in `dimbox.c`, where potential conflicts may appear between reduction and non-reduction variables due to aliases.

Table 4: Benchmarks features and workload.

Benchmark	**S**	**Acc**	**Loops**	**Rdx-in-loop**
Fluidanimate	104M	182M	2	2, 6
MD2	320K	572M	1	6
Unstructured	40K	9M	6	6, 6, 6, 9, 9, 6
Legendre	390K	2M	2	8, 8
300.twolf	43K	30M	1	2

Table 4 summarizes some features of the tested workload for the selected benchmarks. Column labeled *S* corresponds to the combined number of elements in all the reduction arrays. It acts as an indicator of the extra memory requirements. *Acc* is the total number of accesses to the reduction arrays. *Loop* shows the number of the analyzed reduction loops in the benchmark. *Rdx-in-loop* is the number of reduction statements in each of the reduction loops. If a code has more than one reduction loop, the sentences in each loop are displayed separated by commas.

Parallelization techniques have been applied to the reduction loops contained in the codes Fluidanimate, MD2, Unstructured and Legendre and to the loop in function `new_dbox_a()` of 300.twolf. These techniques have been:

- *Coarse-grained locks* (CG Locks), which correspond with critical sections;
- *Fine-grained locks* (FG Locks), where we use an individual lock variable associated to each element of the shared reduction array;
- Full privatization of the reduction variables, implemented via array expansion [10];

- TinySTM used as reference state-of-the-art STM with the default configuration;
- TinySTM-ordered, the version of TinySTM with ordered commits;
- ReduxSTM: both the *timestamp-based* (ReduxSTM-TS) and the *commit-time-invalidation* (ReduxSTM-CTI) strategies.

For all STM sytems, the parallelization of the reductions loops has been carried out by decomposing the loops into chunks of consecutive iterations and assigning a chunk of iterations to a transaction. Note that for the STM algorithms analyzed, both ReduxSTM variants and TinySTM-ordered have ordering constraints while TinySTM has not.

Due to the necessary instrumentation in STM approaches, the number of iterations enclosed by a single transaction becomes a relevant parameter to be tuned as a trade-off, which is given by the chunk size. Larger transactions can reduce the overhead due to the additional instrumentation, as we need fewer transactions to complete the loop. However, larger transactions involve also large datasets which increases the probability of conflicts, and consequently the number of aborts. On the other hand, although the cost and probability of abort is lower for smaller transactions, the total overhead of starting and committing transactions (e.g., reset of data structures) has a higher impact for a larger number of small transactions.

For privatization, and both lock-based techniques, loops are equally partitioned among threads, without grouping iterations into chunks. Remember that privatization requires an initialization and a final reduction phase.

Locks has been implemented using POSIX spinlocks. It should be mentioned that for FGL the number of locks has been optimized by using a single lock per reduction array subscript instead per reduction array element. In this way, groups of reductions sharing the same subscript (e.g., `vel(*,n1)` in Fig. 1(a) for Unstructured) has been protectedwith the same lock. This reduces the number of the required lock variables and the number of lock/unlock pairs, which translates into a lower overhead.

6.2. *Performance comparison*

Next, a comparative of different techniques in terms of speed-up is provided. The observed speedup is calculated with respect to the non-instrumented sequential code, using as execution time the minimum wall-clock time among ten executions at least.

In experiments, the independent variables are the number of threads and the transaction size (iterations per chunk) for the techniques where applicable. In such cases, speedup plots display the speedup for the transaction size (chunk) exhibiting the best speedup for a given number of threads. The plots showing speedup in function of transaction size are referred to experiments with 16 threads.

Fig. 5 shows speedups for Fluidanimate using two datasets corresponding to meshes of 100K and 500K particles. As expected, CG Lock does not accelerate at all due to the use of a single global lock. This method can be suitable for applications where the time spent in reduction sentences are negligible in comparison with the computation time outside the critical section, which is not the case. Best results are obtained using FG Locks or privatization because of the low contention associated with the problem. Observe that for the largest mesh, privatization performance stops scaling well with a high number of threads. This is caused by the large memory requirement of the technique and the NUMA characteristics of the target machine architecture. All STM systems exhibit a similar behavior as long as the abort rate remains very low, so ReduxSTM does not take advantage of its reduction support. In this scenario, STM approaches obtain a better speedup and become a good trade-off between performance and memory requirements. Regarding the influence of the transaction size on performance, the behavior of STMs is very dependent on the input data features. In this way, for the smallest mesh, the penalty of large transactions is more significant than in the case of the largest mesh, for which smaller transactions involve a high penalty.

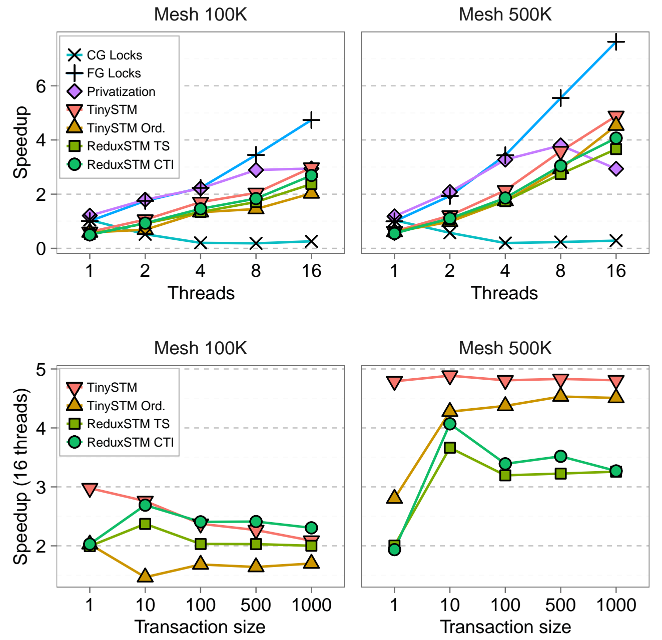

Figure 5: Fluidanimate: speedup for the best transaction size in each case, and influence of the transaction size for 16 threads.

MD2 includes both scalar and array reductions in the reduction loop as shown in Fig. 1(c). Two different parallelizations have been carried out (Fig. 6). In the first one, all shared reduction variables (scalars and vectors) have been treated transactionally. In the second one, scalar variables have been privatized, in such a way that the number of reduction statements on shared objects decreases to 4 (from 6) inside the iteration. This is a common optimization found in the literature [31] for this class of codes, that requires further knowledge about the application. Without scalar privatization, the contention caused by scalars makes both TinySTM versions worsen their performance due to the a high abort rate. In contrast, ReduxSTM is able to filter conflicts caused by reduction operations. No-

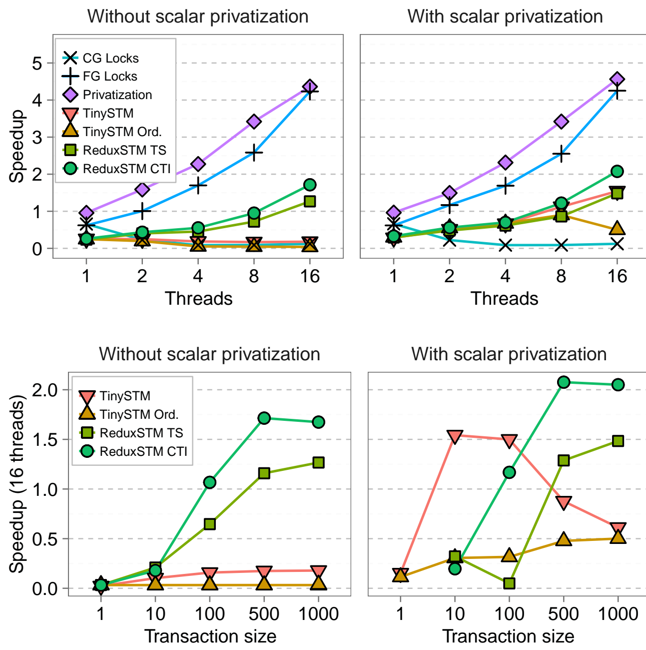

Figure 6: MD2: speedup for the best transaction size in each case, and influence of the transaction size for 16 threads.

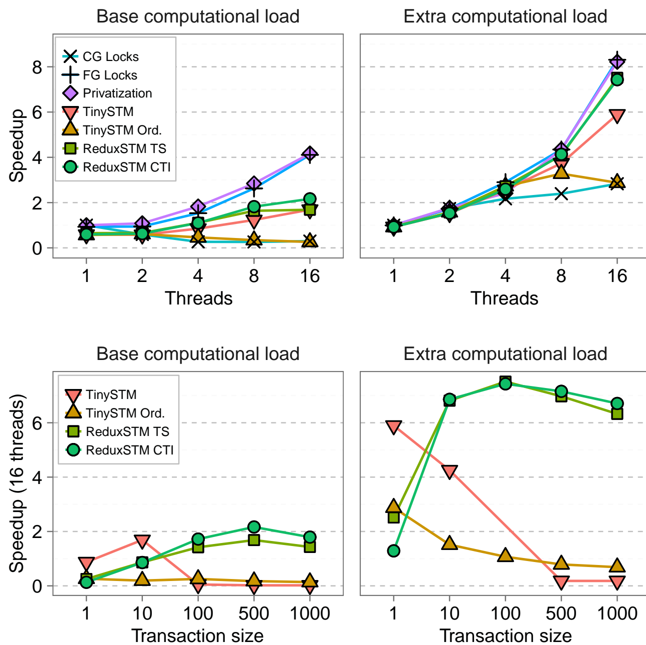

Figure 7: Unstructured: speedup for the best transaction size in each case, and influence of the transaction size for 16 threads.

netheless, the lower size of the reduction arrays allows the privatization technique to obtain the best results, outperforming FG Locks.

Fig. 7 shows results for Unstructured. In these experiments the reduction support makes ReduxSTM gain advantage with respect to TinySTM. TinySTM performance degrades quickly with large transactions due to conflicts. Nevertheless,

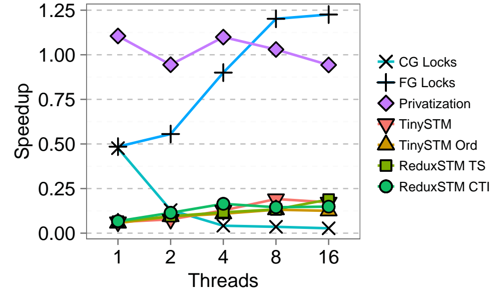

Figure 8: Legendre: speedup (one iteration per transaction considered).

ReduxSTM can leverage the absence of conflicts to increase transaction size and obtaining better results. As in previous experiment, both FG Locks and privatization get the best results. In addition to the base workload, experiments including a synthetic extra computational load have been conducted. The synthetic load is ten times larger than the base one. It is remarkable that ReduxSTM is able to reach similar speedups than privatization when the problem exhibits a high computational load.

Legendre results are shown in Fig. 8. Different transaction sizes have been not analyzed due to the small number of iterations in the outermost loop. None of the tested algorithms achieves good results in this benchmark as the low computational workload (memory-bounded code) is making the additional instrumentation costs much more noticeable.

Fig. 9 shows results obtained with 300.twolf. Experiments are focused on the routine `new_dbox_a()`. Experiments were carried out using the medium-sized workload provided with 300.twolf. This code features a high contended memory access pattern and also the amount of exploitable parallelism is limited by its memory-bound nature. Only one and two iterations per transactions have been considered as performance decreases quickly for larger transactions. Note that only methods that guarantee the original sequential order can be tested because reduction operations coexist with other potentially conflicting reads and writes. For this reason only ordered STM systems ensure a correct execution. Although TinySTM does not fulfil this condition, it has been included for comparison as an optimistic reference. ReduxSTM performs significantly better than TinySTM for every configuration, as ReduxSTM can deal with the conflicts related with reduction patterns. Despite the nature of the benchmark, which makes it difficult to exploit parallelism, ReduxSTM is able to obtain speedup up to a relatively high number of threads. Nevertheless, for the analyzed workload, beyond 8 threads conflicts raise the number of aborts degrading the performance.

6.3. Memory overhead

In this subsection memory overhead of the different techniques is analyzed (extra memory usage over the sequential version) and the results are depicted in Fig. 10. An 8-byte size for both variables and pointers, and 4-byte size for lock structures (POSIX spinlocks) has been assumed.

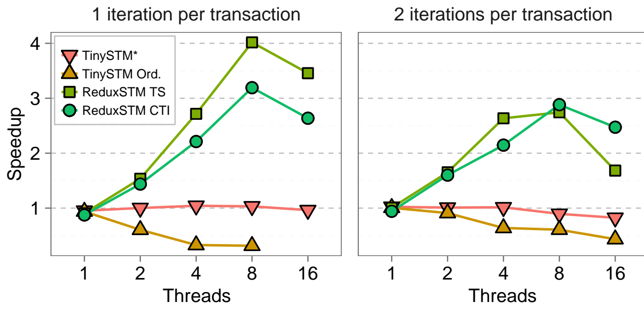

Figure 9: 300-Twolf: speedup for transactions containing one and two iterations. Only ordered STM methods that guarantees correct result are shown; TinySTM (not ordered) is shown with reference purpose.

As privatization needs to allocate an additional private reduction array for each thread, the memory requirements grow with the number of threads and eventually it becomes the most memory expensive method. Total memory overhead can be obtained as $S \cdot T$, where S is the size of all used reduction arrays, and T is the number of threads.

Fine-grain locks have a memory overhead comparable to privatization, as it needs a lock array of size S. However, this overhead remains the same independently of the number of threads. We can see how memory costs are slightly better than 1-thread privatization due to the memory costs for lock structures. Coarse-grain lock parallelization needs only a single lock, so it has the lowest memory requirements.

Regarding STM approaches, as data buffer entries are recycled between transactions the needs of memory depend on the distinct memory accesses performed within a transaction. To calculate this occupation, the number of different addresses acceded by each transaction has been monitored. It has been assumed that each data buffer entry involves a minimum of two 8-byte elements: the address and its associated value.

In addition to data buffers, TinySTM declares a constant 2^{20} elements 8-byte lock array for handling conflict detection and an additional 24 bytes per write-buffer entry to hold metadata needed for versioning. Although read-buffer entries in TinySTM do not need these additional metadata, we have included them in the calculations to obtain a conservative upper bound of the memory requirements. This way, the extra memory can be estimated as $8 \cdot 2^{20} + \eta\,(2+3)\,8\,T$, where η represents the maximum distinct memory addresses accessed during a single transaction in each code, and $8 \cdot (2+3)$ stands for two 8-byte elements per entry to hold address and value and three 8-byte elements for additional metadata.

ReduxSTM-CTI employs two 2^6 bit (8-byte) per-transaction signatures to perform conflict detection besides the per-transaction write-buffer and two 2^8 2-byte arrays for fast-access to the buffer. Its memory requirements are obtained as $(8 \cdot 2 + 16\,\eta + 2 \cdot 2^8 \cdot 2)\,T$. ReduxSTM-TS requires a single per-thread read timestamp and the global update timestamp, having both 2^6 8-byte elements. It also uses the 2-byte arrays for fast access to write-buffer. The needed memory is calculated as $(2^6 \cdot 8 + 16\,\eta + 2 \cdot 2^8 \cdot 2)\,T + 2^6 \cdot 8$. Note that in highly contended workloads, potential aliases in STMs consequence of small metadata (false positives) might harm the performance. In this case, larger signatures (ReduxSTM-CTI) or timestamps (ReduxSTM-TS), should be configured according to the memory requirements of the problem.

While in section 6.2 we have shown that often privatization and fine-grain locks obtain the best results in pure performance terms, its memory requirements may be too high compared with all analyzed STM systems as shown in Fig. 10. We can conclude that STM approaches can be a useful method that allows to extract significant parallelism with relatively low memory requirements.

7. Conclusions

It is known that many applications that exhibit irregular memory access patterns are very hard to parallelize. In fact, it is usual that no data dependence information is available. In this context, the optimistic concurrency support provided by transactional memory (TM) may be useful to extract parallelism from such applications. TM can be leveraged to exploit thread level parallelism. This work has presented ReduxSTM, a software TM (STM) with specific support for reduction operations, a pattern found very frequently in the core of irregular applications. Commit/version management and conflict detection were tailored to take advantage of both, transaction sequential ordering to assure correct results and the privatization of reduction patterns. Both facts were used to avoid unnecessary transaction rollbacks. In this way, ReduxSTM allows managing reduction sentences safely without further knowledge about the loop enclosing them. Compared with classical parallelization of reduction loops, experiments showed that STM approaches are a good trade-off between exploited parallelism and memory overhead, and their advantages in irregular codes are extended when support for reductions is added, specially in highly contended workloads.

Acknowledgements

This work has been supported by the Government of Spain under project TIN2013-42253-P and Junta de Andalucía under project P12-TIC-1470.

References

[1] M. Herlihy, J. Moss, Transactional Memory: Architectural support for lock-free data structures, in: 20th Ann. Int'l. Symp. on Computer Architecture (ISCA'93), San Diego, CA, USA, 1993, pp. 289–300.

[2] T. Harris, J. Larus, R. Rajwar, Transactional Memory, 2nd Ed, Morgan & Claypool Publishers, USA, 2010.

[3] L. Porter, B. Choi, D. Tullsen, Mapping out a path from hardware transactional memory to speculative multithreading, in: 18th Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'09), Raleigh, NC, USA, 2009.

[4] M. Mehrara, J. Hao, P.-C. Hsu, S. Mahlke, Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory, in: 30th ACM Conf. on Programming Language Design and Implementation (PLDI'09), Dublin, Ireland, 2009.

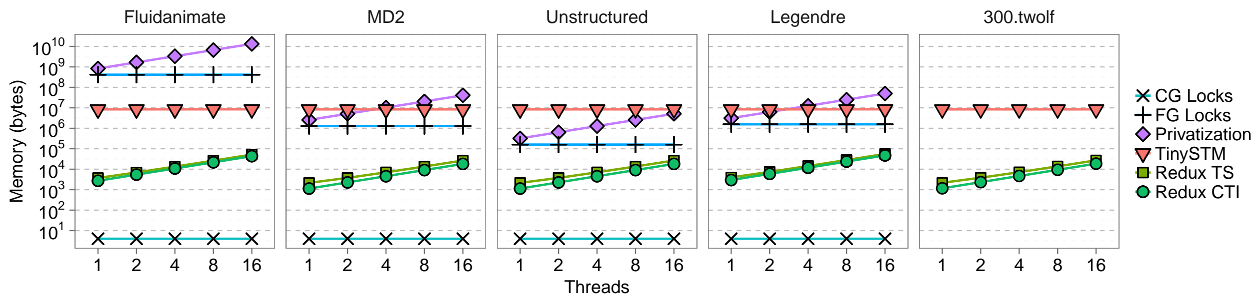

Figure 10: Memory overhead in different parallelization approaches.

[5] N. Johnson, H. Kim, P. Prabhu, A. Zaks, D. August, Speculative separation for privatization and reductions, in: 33rd ACM Conf. on Programming Language Design and Implementation (PLDI'12), Beijing, China, 2012, pp. 359–370.

[6] L. Rauchwerger, Speculative parallelization of loops, in: Encyclopedia of Parallel Computing, Springer, 2011, pp. 1901–1912.

[7] L. Han, W. Liu, J. Tuck, Speculative parallelization of partial reduction variables, in: 8th Ann. IEEE/ACM Int'l. Symp. on Code Generation and Optimization (CGO'10), Toronto, Canada, 2010, pp. 141–150.

[8] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, Parallel programming with Polaris, IEEE Computer 29 (12) (1996) 78–82.

[9] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bu, Maximizing multiprocessor performance with the SUIF compiler, IEEE Computer 29 (12) (1996) 84–89.

[10] P. Feautrier, Array expansion, in: 2nd Int'l Conf. on Supercomputing (ICS'88), Saint Malo, France, 1988, pp. 429–441.

[11] H. Yu, L. Rauchwerger, An adaptive algorithm selection framework for reduction parallelization, IEEE Trans. on Parallel and Distributed Systems 17 (10) (2006) 1084–1096.

[12] H. Han, C. Tseng, Exploiting locality for irregular scientific codes, IEEE Trans. on Parallel and Distributed Systems 17 (7) (2006) 606–618.

[13] E. Gutiérrez, O. Plata, E. Zapata, A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors, in: 14th Int'l. Conf. on Supercomputing (ICS'00), Santa Fe, NM, USA, 2000, pp. 78–87.

[14] J. M. Perez, R. M. Badia, J. Labarta, A dependency-aware task-based programming environment for multi-core architectures, in: 10th IEEE Int'l Conf. on Cluster Computing, 2008, pp. 142–151.

[15] C. H. González, B. B. Fraguela, A framework for argument-based task synchronization with automatic detection of dependencies, Parallel Computing 39 (9) (2013) 475–489.

[16] L. Rauchwerger, D. A. Padua, The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization, IEEE Trans, on Parallel and Distributed Systems 10 (2) (1999) 160–180.

[17] M. Gonzalez-Mesa, R. Quislant, E. Gutierrez, O. Plata, Dealing with reduction operations using transactional memory, in: 25th Int'l. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD'13), Porto de Galinhas, Brasil, 2013, pp. 128–135.

[18] M. Gonzalez-Mesa, E. Gutierrez, E. Zapata, O. Plata, Effective transactional memory execution management for improved concurrency, ACM Trans. on Architecture and Code Optimization 11 (3) (2014) 24:1–24:27.

[19] C. von Praun, C. Ceze, C. Cascaval, Implicit parallelism with ordered transactions, in: 12th ACM Symp. on Principles and Practice of Parallel Programming (PLDI'07), San Diego, CA, USA, 2007, pp. 79–89.

[20] A. Udupa, K. Rajan, W. Thies, ALTER: Exploiting breakable dependences for parallelization, 32nd ACM Conf. on Programming Language Design and Implementation (PLDI'11) (2011) 480–491.

[21] M. DeVuyst, D. Tullsen, S. Kim, Runtime parallelization of legacy code on a transactional memory system, in: 6th Int'l. Conf. on High Performance and Embedded Architectures and Compilers (HiPEAC'11), Heraklion, Greece, 2011, pp. 127–136.

[22] W. Ruan, Y. Liu, M. Spear, Transactional Read-Modify-Write without aborts, ACM Trans. on Architecture and Code Optimization 11 (4) (2015) 1–24.

[23] R. Quislant, E. Gutierrez, E. L. Zapata, O. Plata, Conflict detection in hardware transactional memory, in: Transactional Memory. Foundations, Algorithms, Tools, and Applications, Springer, 2015, pp. 127–149.

[24] J. Gottschlich, M. Vachharajani, J. Siek, An efficient software transactional memory using commit-time invalidation, in: 8th Ann. IEEE/ACM Int'l. Symp. on Code Generation and Optimization (CGO'10), Toronto, Canada, 2010, pp. 101–110.

[25] P. Felber, C. Fetzer, P. Marlier, T. Riegel, Time-based software transactional memory, IEEE Trans. on Parallel and Distributed Systems 21 (12) (2010) 1793–1807.

[26] C. Bienia, S. Kumar, J. Singh, K. Li, The PARSEC benchmark suite: Characterization and architectural implications, in: 17th Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'08), Toronto, Canada, 2008, pp. 72–81.

[27] J. Morales, S. Toxvaerd, The Cell-Neighbour table method in molecular dynamics simulations, Computer Physics Communications 71 (1992) 71–76.

[28] I. Foster, R. Schreiber, P. Havlak, HPF-2, scope of activities and motivating applications, Tech. rep., Technical Report CRPC-TR94492, Rice University (1994).

[29] N. Mukherjee, J. Gurd, A comparative analysis of four parallelisation schemes, in: 13th Int'l. Conf. on Supercomputing (ICS'99), Portland, OR, USA, 1999, pp. 278–285.

[30] M. K. Prabhu, K. Olukotun, Exposing speculative thread parallelism in SPEC2000, in: 10th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'05), Chicago, IL, USA, 2005, p. 142.

[31] M. Dai Wang, M. Burcea, L. Li, et al., Exploring the performance and programmability design space of hardware transactional memory, in: ACM Workshop on Transactional Computing (TRANSACT'14), Salt Lake City, UT, USA, 2014.