This space is reserved for the Procedia header, do not use it

Improving Transactional Memory Performance for Irregular Applications

Manuel Pedrero, Eladio Gutierrez, Sergio Romero, and Oscar Plata

Universidad de Malaga, Andalucia Tech, Dept. Computer Architecture, 29071 Malaga, Spain
{mpedrero,eladio,sromero,oplata}@uma.es

Abstract
Transactional memory (TM) offers optimistic concurrency support in modern multicore architectures, helping the programmers to extract parallelism in irregular applications when data dependence information is not available before runtime. In fact, recent research focus on exploiting thread-level parallelism using TM approaches. However, the proposed techniques are of general use, valid for any type of application.

This work presents ReduxSTM, a software TM system specially designed to extract maximum parallelism from irregular applications. Commit management and conflict detection are tailored to take advantage of both, sequential transaction ordering to assure correct results, and privatization of reduction patterns, a very frequent memory access pattern in irregular applications. Both techniques are used to avoid unnecessary transaction aborts.

A function in 300.twolf package from SPEC CPU2000 was taken as a motivating irregular program. This code was parallelized using ReduxSTM and an ordered version of TinySTM, a state-of-the-art TM system. Experimental evaluation shows that ReduxTM exploits more parallelism from the sequential program and obtains better performance than the other system.

Keywords: Irregular application, Transactional memory, Thread-level speculation, Reduction pattern

1 Introduction and Related Work

The availability of multiples 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) [8] 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 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 in a combination of both (HyTM) [7].

Encouraged by TM benefits, efforts have being devoted to leverage TM for extracting parallelism from sequential applications. In fact, many basic operations in a TM system, like detection and solving of memory conflicts, buffering of memory updates, and execution rollbacks, are also required by speculative multithreading (SpMT), or thread-level speculation (TLS) [13]. 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. In this way, the optimistic concurrency exploited by TM systems may help parallelizing 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 assure correct results. In general, transactions must commit in an order that preserves the sequential semantics.

Research focused on combining or exploiting TLS using TM approaches can be found in the literature [17, 11, 10, 3, 2, 15, 12, 5]. Most of these techniques are of general use, not tailored for specific types of applications. However, speculative techniques are specially useful for irregular applications, as discussed above. In this work, a transactional memory system, ReduxSTM, is presented. The system is specifically developed to extract maximum parallelism from irregular applications, considering that no data dependence information is available before runtime. To maximize parallelism between tasks, the system is designed to take advantage of both, the sequential transaction ordering and the privatization of irregular reduction patterns, to avoid unnecessary aborts and rollbacks (only true dependencies are really enforced).

A code from SPEC CPU2000 was selected as a motivating application. This code exhibits an irregular memory behavior that makes it suitable for being sped up by TLS, as dependencies are not easily analyzable. This type of applications demands an important effort from the parallel programmer to develop an optimized parallel version of the code. The aim of this paper is to show how ReduxSTM may help to simplify greatly the work of the programmer in parallelizing the code, extracting more parallelism than other (ordered) TM systems.

```
1  new_dbox_a(...) {
2    ...
3    for ( termptr=antrmptr; termptr; termptr=termptr->nextterm ) {
4      ...
5      new_mean = dimptr->new_total / dimptr->numpins;
6      old_mean = dimptr->old_total / dimptr->numpins;
7      for (netptr=dimptr->netptr; netptr; netptr=netptr->nterm ) {
8        oldx = netptr->xpos;
9        if ( netptr->flag == 1 ) {
10         newx = netptr->newx;
11         netptr->flag = 0;
12       } else { newx = oldx; }
13 // (1) Potential reduction sentence
14       *costptr += ABS(newx - new_mean) - ABS(oldx - old_mean);
15     }
16     ...
17     tmp_num_feeds[net] = f;   // Potential alias with *costptr and delta_vert_cost
18     ...
19     tmp_missing_rows[net] = -m; // Potential alias with *costptr and delta_vert_cost
20 // (2) Potential reduction sentence
21     delta_vert_cost += ((tmp_num_feeds[net] - num_feeds[net]) + (tmp_missing_rows[net] - missing_rows[net])) * 2 * rowHeight;
22   }
23   return;
24 }
```

Figure 1: `new_dbox_a()` function in 300.twolf code from SPEC CPU2000

2 Motivating Irregular Application

As case study and motivating application, the TimberWolfSC placement and global routing package (300.twolf) was selected from the SPEC CPU2000 suite. This package determines the placement and global connections for standard cells in a microchip, using simulated annealing to speed-up the exploration of the state space.

Our focus is on the `new_dbox_a()` function included in `dimbox.c`, one of the files in 300.twolf. This function is shown in Figure 1. The code has an outer loop that includes two sentences with a reduction pattern, using `costptr` and `delta_vert_cost` as reduction variables. However, as `costptr` is a pointer, it may alias with the other reduction variable or with other global variables in the loop, like `tmp_num_feeds[]` or `tmp_missing_rows[]`. As a consequence, the compiler is not able to analyze dependences and extract parallelism from this loop.

Although no conventional parallelization techniques can be applied to the reduction sentences, the reduction condition may be valid for some of the accesses to the reduction variable (this is called a partial reduction [14, 6]). If this fraction of accesses is high enough a wealth of parallelism can be exploited.

3 ReduxSTM Design

The main objective in the design of ReduxSTM is to leverage the TM basic mechanisms to exploit speculative parallelism from sequential applications (specially, irregular ones), considering that no data dependence information is statically available.

Ordered transactions. A feature that must have ReduxSTM is to keep ordering constraints between transactions. The concurrent execution of the speculative tasks in which the application has been decomposed must preserve the sequential semantics to ensure the correctness of the

final results. Because of that, ReduxSTM must commit transactions in accordance with the sequential execution order. This ordering constraint is added to the commit manager but it may cause performance degradation due to delays in committing transactions that finish out of order. To cope with this problem, conflict detection uses the ordering knowledge to filter some conflicts and avoid unnecessary aborts and rollbacks [13, 5].

Reduction patterns. A second feature added to ReduxSTM is a privatization mechanism. This is not mandatory for speculative parallelization but it is useful to remove a fraction of accesses to shared memory and hence reduce contention between transactions. Privatization allows to eliminate unnecessary transaction conflicts, improving parallelism exploitation. In fact, privatization has been found as a key technique to enable parallelism [16, 9].

In particular, privatization is implemented in ReduxSTM to extract parallelism from reduction memory patterns. Scientific applications present these patterns frequently and, in the case of irregular problems, with abundance of parallelism exploitable using privatization techniques. A reduction pattern is characterized by a reduction sentence in the program of the form $A[] = A[] \oplus \xi$, where $A[]$ is the reduction variable (array, in general), ξ is an expression not including $A[]$, and $\oplus$ is a commutative and associative operation. This sentence is located in the body of a loop.

Reduction patterns are usually hard to parallelize when occurring in irregular programs because the reduction variable (array) is often subscripted and/or some locations comprised by such array are used (read or write) outside the reduction sentence. This is the case of a partial reduction, where the reduction definition is fulfilled in a portion of the reduction variable.

To support privatization of reductions in ReduxSTM, a primitive for memory reduction was defined. This primitive is handled by the conflict, version and commit managers as a third basic memory operation: read (R), write (W) and reduction (Rdx). Rdx represents a combination of two operations, a read followed by a write on the same memory location of the reduced value, $Rdx(add, val, \oplus) := W(add, R(add) \oplus val)$.

Version management. In order to support privatization of reduction operations, version management in ReduxSTM is lazy, that is, updates to memory are held in a private buffer during the execution of the transaction. The write buffer, used to hold new values from transactional writes, was extended to privatize transactional reduction values. The extension consists in a state tag associated to each entry of the buffer, that specifies if such entry holds a written value or a reduced value (and the corresponding reduction operation).

Every time a transactional reduction is issued, the write buffer is searched for the memory address. If it is found and the state tag corresponds to a write, the value specified in the reduction operation is reduced with the value stored in the buffer. However, the state tag is kept as a write because the reduction condition is not fulfilled. Otherwise, if the state tag is a reduction, a similar operation is carried out and the state tag remains as a reduction. Similarly, a transactional write fires a searching of the address in the write buffer. If it is found, the entry is updated and the state tag is set to write, independently on the original state.

Conflict management. Conflict detection in ReduxSTM is also lazy, that is, at commit time, as this allows to filter out most of the conflicts, improving concurrency between transactions. Table 1 shows which conflicts are avoided thanks to the ordering of transactions and the privatization of reduction operations. 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

Table 1: Transactional conflicts in different scenarios

	TM	**TM + Order**	**TM + Order + Reduction**
R−W	abort	no conflict	no conflict
W−R	abort	abort	abort
W−W	abort	no conflict	no conflict
Rdx−R	–	–	abort
R−Rdx	–	–	no conflict
Rdx−W	–	–	no conflict
W−Rdx	–	–	no conflict
Rdx−Rdx	–	–	no conflict

corresponds to the behavior of a standard TM system, while the last two columns consider additional support for order and reductions. Conflict manager uses read and write sets to detect conflicts (note that some ordered TM systems do not filter out conflicts). These sets were implemented as signatures based on Bloom filters.

Commit management. In order to minimize the overhead of the transactional memory operations and to support a lazy-lazy behavior, ReduxSTM is based on a full commit invalidation algorithm, that is, conflicts are resolved at commit time.

The commit phase is in charge of the three following tasks: (1) Check if the ordering condition is fulfilled (otherwise, wait for that); (2) check if the transaction was aborted (killed) by a previously committed one; (3) update main memory with the values stored in the write buffer, reducing in memory the corresponding ones; (4) compare the write Bloom filter with the read Bloom filter of all subsequent (in the established order) active transactions, marking as killed those conflicting ones.

4 Experimental Evaluation

In this section, we evaluate the performance of ReduxSTM system when used to parallelize the `new_dbox_a()` function from 330.twolf. The goal is to show how ReduxSTM allows to extract more parallelism from the sequential code than other state-of-the-art (ordered) STM system, thanks to the design of our conflict manager that avoids unnecessary aborts.

4.1 Experimental Setup

Experiments were conducted on a server with 8 GB RAM and one quad-core Intel Core i7-3770 processor at 3.4GHz that supports 8 concurrent threads. The server runs Linux kernel 3.2.0-75 (64 bits) and all programs were compiled using GNU GCC (optimization option `-O2`). The ordered version of TinySTM (v.1.0.5) [4] was used as the baseline state-of-the-art STM system.

The following two-step methodology was used for the evaluation. First, the original sequential program was instrumented to obtain a trace of the relevant memory accesses, those that can cause a conflict (shared accesses). Figure 2 shows the instrumented version of the `new_dbox_a()` function. Macros `MARKREAD()`, `MARKWRITE()` and `MARKREDUX()` specify which memory addresses must appear in the trace and what type of operation is made (read, write or reduction). Reduction points out a memory location subjected to a potential reduction operation. It is a "potential" reduction because the conflict manager in ReduxTM will determine at runtime if it is the case or not. In a second step, a simulator is fed with the resulting memory trace which simulates the original `new_dbox_a()` function recreating the instrumented memory pattern, as

```
1  new_dbox_a(...) {
2    ...
3    for ( termptr=antrmptr; termptr; termptr=termptr->nextterm ) {
4      ...
5      MARKREAD(&dimptr->new_total);
6      MARKREAD(&dimptr->numpins);
7      MARKREAD(&dimptr->old_total);
8      new_mean = dimptr->new_total / dimptr->numpins;
9      old_mean = dimptr->old_total / dimptr->numpins;
10     MARKREAD(&dimptr->netptr);
11     for (netptr=dimptr->netptr; netptr; netptr=netptr->nterm ) {
12       MARKREAD(&netptr->xpos);
13       oldx = netptr->xpos;
14       MARKREAD(&netptr->flag);
15       if ( netptr->flag == 1 ) {
16         MARKREAD(&netptr->newx);
17         newx = netptr->newx;
18         netptr->flag = 0;
19         MARKWRITE(&netptr->flag,0);
20       } else { newx = oldx; }
21 // (1) Potential reduction sentence
22       *costptr += ABS(newx - new_mean) - ABS(oldx - old_mean);
23       MARKREDUX(&(*costptr),ABS(newx - new_mean) - ABS(oldx - old_mean));
24       MARKREAD(&netptr->nterm);
25     }
26     ...
27     tmp_num_feeds[net] = f;   // Potential alias with *costptr and delta_vert_cost
28     MARKWRITE(&tmp_num_feeds[net],f);
29     ...
30     tmp_missing_rows[net] = -m; // Potential alias with *costptr and delta_vert_cost
31     MARKWRITE(&tmp_missing_rows[net],-m);
32     MARKREAD(&tmp_num_feeds[net]);
33     MARKREAD(&num_feeds[net]);
34     MARKREAD(&tmp_missing_rows[net]);
35     MARKREAD(&missing_rows[net]);
36 // (2) Potential reduction sentence
37     delta_vert_cost  += ((tmp_num_feeds[net] - num_feeds[net]) + (tmp_missing_rows[net] - missing_rows[net])) * 2 * rowHeight;
38     MARKREDUX(&(delta_vert_cost),((tmp_num_feeds[net] - num_feeds[net]) + (tmp_missing_rows[net] - missing_rows[net]))...);
39     MARKREAD(&termptr->nextterm);
40   }
41   return;
42 }
```

Figure 2: Instrumented `new_dbox_a()` function

well as the original workload. This simulation is carried out in parallel and in a transactional way. This is accomplished by partitioning the outer loop of the simulated function into blocks of consecutive iterations. Each one of these blocks is executed as a transaction. Transactions are mapped to threads in a round-robin way.

Experiments were executed using a medium-sized workload (the *training workload* included in 300.twolf). The complete execution of the application is long. In particular, the outer loop in the `new_dbox_a()` function executed almost 12 million iterations. The resulting memory trace contained almost 550M reads, 46M writes and 70M reductions. The results shown in this paper, however, are obtained from a section of 500K iterations taken from the middle of the execution trace. That represents around 24M reads, 2M writes and 3M reductions in the memory trace. An important observation is that in all 29M memory accesses in the selected trace there are only about 43K different memory locations (less than 0.15% of the total memory references are different). That means a high contented transactional execution. In addition, the benchmark

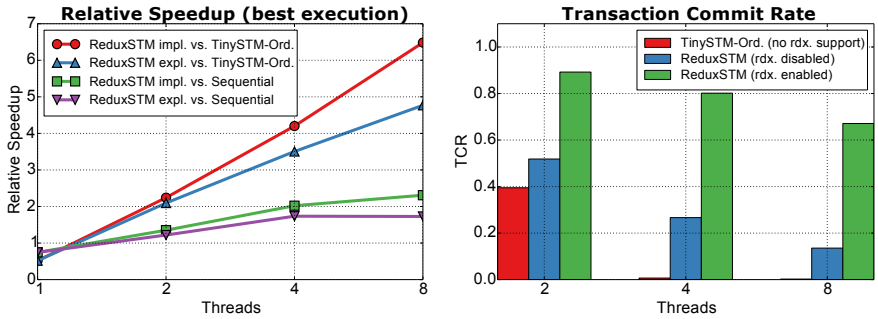

Figure 3: Speedup of ReduxSTM over TinySTM and sequential version (left), transaction commit rate (TCR) of ReduxSTM and TinySTM (right)

code is memory-bound, that also limits the amount of exploitable parallelism.

Experiments include the following versions of the application: (1) The original sequential version (executed in one thread); (2) a parallel version based on the ordered TinySTM system, where the commit order is given by the starting time of the transaction (first to start, first to commit); (3) a parallel version based on ReduxSTM with implicit order given when the transaction reaches the commit phase (first to finish execution, first to commit); (4) a parallel version based on ReduxSTM with explicit order given by an ordinal number specified by the programmer (this order is the same as the sequential order). ReduxSTM with explicit order is the only version that assure correct results (identical to the sequential version). The other two TM versions, with a more relaxed ordering, are used for comparison.

4.2 Relative Speedup

The speedup obtained with both versions of ReduxSTM (implicit and explicit order) related to the speedup when using ordered TinySTM and regarding the sequential version is shown in Figure 3 (left). As the performance of the transactional codes depend on the size of transactions, experiments were conducted for different sizes (from 1 to 8 iterations), selecting the best result.

Speedup for ReduxSTM is significantly better than that for TinySTM thanks to the ability of the former system to avoid unnecessary aborts and rollbacks. Implicit order is better that explicit one, as expected, as it is more relaxed. ReduxSTM is also able to exploit parallelism from the sequential version, despite it is memory bound.

The number of transactional conflicts increases with the number of threads. However, ReduxSTM improves its performance better than TinySTM thanks to its ability to filter out many of the conflicts. With 8 threads, ReduxSTM is 6.5 (implicit order) or 4.8 (explicit order) times faster than ordered TinySTM.

4.3 Transaction Commit Rate (TCR)

A comparison of both TM systems in terms of the transaction commit rate (TCR) [1] is depicted in Figure 3 (right). TCR is defined as the percentage of committed transactions out of all executed ones, and it is a suitable parameter to measure the exploited concurrency. In the figure two versions of ReduxSTM are shown, one with the support for privatizing reductions and the other with such feature disabled. Note that for our application such feature allows to exploit much more parallelism, specially if the thread count increases. For instance, for 8 threads,

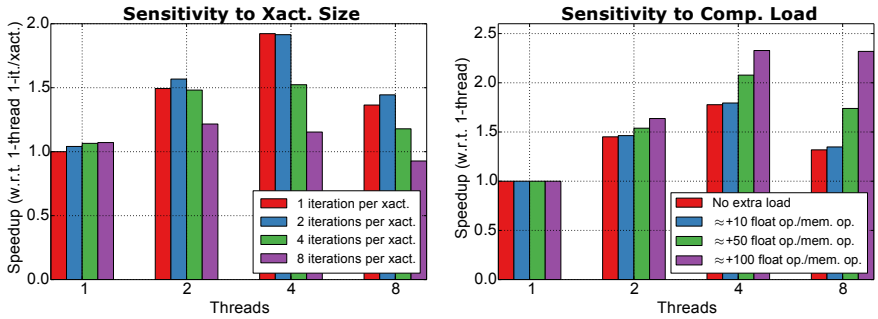

Figure 4: Sensitivity of ReduxSTM to transaction size (left), and to computational load (right)

the complete ReduxSTM obtains a TCR of almost 70% (almost 70% of the transactions were executed and committed without rollbacks). Without reduction support, however, TCR drops below 20%, while TCR for ordered TinySTM is almost zero (basically, most of the transactions execute serially, one after another).

From the results it can be seen that, in a speculative context, when order between transactions is mandatory, the potential to filter out conflicts thanks to such order is beneficial, allowing to keep a high TCR while increasing the number of threads. In addition, the extra conflict elimination allowed by the privatization of reduction patterns is also very relevant. Besides, by design, ReduxSTM limits the number of rollbacks that a transaction can suffer to the thread count minus one.

4.4 Sensitivity to Transaction Size

The impact of the transaction size (in terms of the number of iterations of the outer loop in `new_dbox_a()`) on performance for ReduxSTM is depicted in Figure 4 (left). The graph shows the speedup obtained using ReduxSTM with respect to the execution in 1 thread with 1-iteration transactions.

With one thread, speedup increases with the transaction size, as the number of transactions reduces and so the overhead to manage them. When the number of threads increases, the probability of conflicts increases with the size of transactions, decreasing the obtained performance. The figure shows modest speedups due to the memory bound nature of the application.

4.5 Sensitivity to Computational Workload

In order to test the ability of ReduxSTM to extract parallelism from situations where there is a more balanced combination of computation and memory accesses, experiments were conducted for different workloads added to the transactions. This extra load was specified in terms of a number of floating-point operations per memory access. Figure 4 (right) shows the obtained results, measured as the speedup with respect to the execution using 1 thread.

As expected, the performance of ReduxSTM for 4 and 8 threads improves significantly with the increase in the computational workload, as the relative performance impact of the transaction management overheads drop.

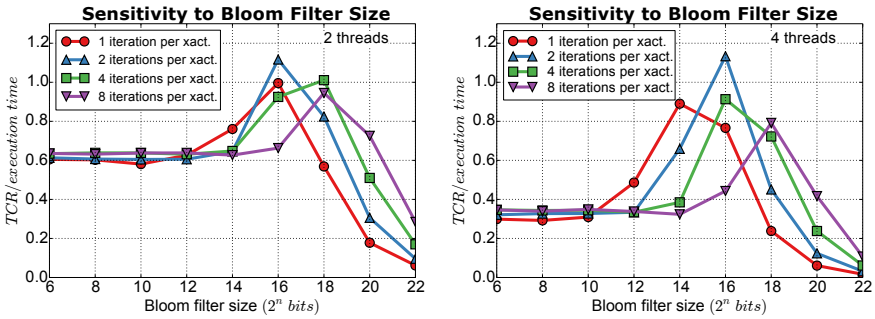

Figure 5: Sensitivity of ReduxSTM to Bloom filters size

4.6 Sensitivity to Bloom Filters Size

ReduxSTM uses Bloom filters (signatures) as the data structures to store the read and write sets of a transaction. These hash structures allow fast insert and test operations for an unbounded set of addresses, but at the cost of some probability of false positives (due to aliasing). Increasing their size reduces such probability but at the expense of occupying more cache space (for fast accesses) and increasing reset time (required after an abort or a commit). We need to find a tradeoff between the filters size and the probability of false positives that maximizes performance.

Figure 5 shows the performance of ReduxSTM for different Bloom filter sizes (both, read and write filters, are of the same size). Performance is measured as the ratio between TCR and execution time, because maximizing this ratio means maximum concurrency exploitation at minimum execution time.

From the figure it can be concluded that the best filters size is in the range between 2^{14} and 2^{18} bits, for different transaction sizes and thread counts. For smaller Bloom filters, the performance remains constant as the number of aborts for a given transaction in ReduxSTM is limited by the number of threads. For larger Bloom filters, on the other hand, performance drops fast due to the overhead of storing the filters out of the first levels of the cache hierarchy, as well as the overhead of resetting them.

5 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 presented ReduxSTM, a software TM (STM) system specially designed to extract maximum parallelism from 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. Experimental results using a motivating benchmark from the SPEC CPU2000 suite prove that ReduxSTM allows important improvements in performance regarding a state-of-the-art STM system.

Acknowledgements

The authors want to aknowledge Universidad de Malaga, Campus de Excelencia Internacional Andalucia Tech, for its support.

References

[1] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. Advanced concurrency control for transactional memory using transaction commit rate. In *14th Int'l. Euro-Par Conference (Euro-Par'2008)*, pages 719–728, 2008.

[2] J. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui. Unifying thread-level speculation and transactional memory. In *ACM/IFIP/USENIX 13th Int'l. Middleware Conf.*, pages 187–207, 2012.

[3] Matthew DeVuyst, Dean M. Tullsen, and Seon Wook 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)*, pages 127–136, 2011.

[4] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Trans. on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.

[5] M.A. Gonzalez-Mesa, E. Gutierrez, E.L. Zapata, and O. Plata. Effective transactional memory execution management for improved concurrency. *ACM Transactions on Architecture and Code Optimization*, 11(3), 2014.

[6] Liang Han, Wei Liu, and James M. Tuck. Speculative parallelization of partial reduction variables. In *8th Annual IEEE/ACM Int'l. Symp. on Code Generation and Optimization (CGO'10)*, pages 141–150, 2010.

[7] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd.* Morgan & Claypool Publishers, USA, 2010.

[8] M. Herlihy and J. Moss. Transactional Memory: Architectural support for lock-free data structures. In *20 Ann. Int'l. Symp. on Computer Architecture (ISCA'93)*, pages 289–300, 1993.

[9] Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. Speculative separation for privatization and reductions. In *33rd ACM Conf. on Programming Language Design and Implementation (PLDI'12)*, pages 359–370, 2012.

[10] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'09)*, pages 166–176, 2009.

[11] K. Nikas, N. Anastopoulos, G. Goumas, and N. Koziris. Employing transactional memory and helper threads to speedup Dijkstra's algorithm. In *38th Int'l. Conf. on Parallel Processing (ICPP'2009)*, pages 388–395, 2009.

[12] Rei Odaira and Takuya Nakaike. Thread-level speculation on off-the-self hardware transactional memory. In *IEEE Int'l. Symp. on Workload Characterization (IISWC'2014)*, pages 212–221, 2014.

[13] Leo Porter, Bumyong Choi, and Dean M. 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)*, pages 313–324, 2009.

[14] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2), 1999.

[15] M.M. Saad, M. Mohamedin, and B. Ravindran. HydraVM: Extracting parallelism from legacy sequential code using STM. In *4th USENIX Workshop on Hot Topics in Parallelism (HotPar'12)*, pages 1–7, 2012.

[16] Peng Tu and David Padua. Automatic array privatization. In *6th Int'l. Workshop on Languages and Compilers for Parallel Computing (LCPC'94)*, pages 500–521, 1994.

[17] C. von Praun, C. Ceze, and C. Cascaval. Implicit parallelism with ordered transactions. In *12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'2007)*, pages 79–89, 2007.