Proceedings of the 16th International Conference
on Computational and Mathematical Methods
in Science and Engineering, CMMSE 2016
4–8 July, 2016.

Critical Sections and Software Transactional Memory Comparison in the Context of a TLS Runtime Library

Sergio Aldea[1], Diego R. Llanos[1] and Arturo Gonzalez-Escribano[1]

[1] *Departamento de Informática, Universidad de Valladolid*

emails: sergio@infor.uva.es, diego@infor.uva.es, arturo@infor.uva.es

Abstract

Transactional Memory (TM) is a technique that aims to mitigate the performance losses that are inherent to the serialization of accesses in critical sections. Some studies have shown that the use of TM may lead to performance improvements, despite the existence of management overheads. However, the relative performance of TM, with respect to classical critical sections management depends greatly on the actual percentage of times that the same data is handled simultaneously by two transactions. In this paper, we compare the relative performance of the critical sections provided by OpenMP with respect to two Software Transactional Memory (STM) implementations. These three methods are used to manage concurrent data accesses in ATLaS, a software-based, Thread-Level Speculation (TLS) system. The complexity of this application makes it extremely difficult to predict whether two transactions may conflict or not, and how many times the transactions will be executed. Our experimental results show that the STM solutions only deliver a performance comparable to OpenMP when there are almost no conflicts. In any other case, their performance losses make OpenMP the best alternative to manage critical sections.

Key words: Software Transactional Memory, STM, Thread-Level Speculation, TLS, OpenMP, ATLaS

1 Introduction

Current multicore processors offer an opportunity to speed up the computation of sequential applications. To exploit these parallel technologies, the software needs to be parallelized, that is, transformed in order to correctly distribute the work among different threads. This process usually involves synchronizing the accesses to certain memory areas that are

 ISBN: 978-84-608-6082-2

shared by the concurrent threads, with the aim of avoiding potential data races. This synchronization is usually performed by using critical sections that protect shared memory structures.

To simplify this process, parallel programming models such as OpenMP [1] offer compiler directives, not only to parallelize the code, but also to synchronize accesses and define and manage critical sections. Despite their simplicity, these solutions present a problem: Critical sections introduce performance losses, not only because they serialize the code, but also because of the cost associated to locking management.

Software Transactional Memory (STM) [2] arises as a possible solution to the first problem, allowing programmers to transform critical sections in transactions that are concurrently and atomically executed. This is based on the optimistic assumption that the code inside the transaction will access to different locations of the shared memory being protected. In these cases, accesses are carried out concurrently. If this is not the case, conflictive transactions should be rolled back and executed one at a time.

Works such as [3] have shown that STM can outperform OpenMP critical sections, despite the relatively high overheads of STM. However, the relative performance of STM versus OpenMP critical sections is highly dependent on the running profile of each particular application. Different patterns of accesses to the same critical section may lead to different performance figures.

This paper compares the OpenMP critical sections approach with two STM libraries, using them to handle the critical sections that appears in the runtime library of ATLaS [4], a state-of-the-art, software-based Thread Level Speculation (TLS) system. Our goal is to study the relative performance of both approaches when managing concurrent accesses in such a complex piece of code.

The rest of this paper is structured as follows: Section 2 briefly describes the fundamentals of software TLS. Section 3 details how our TLS runtime library handles the speculative execution of a source code, and how critical data structures are protected to ensure correctness. Section 4 describes how this protection can be ensured using OpenMP and two different STM libraries. Section 5 shows the performance results obtained by each OpenMP and the STM libraries considered. Finally, Sect. 6 concludes this paper.

2 Thread-Level Speculation in a Nutshell

Speculative parallelization (SP), also called Thread-Level Speculation (TLS) or Optimistic Parallelization [5, 6, 7, 8, 9, 10, 11], is a technique that allows the parallel execution of fragments of code (typically blocks of iterations of a loop) without the need for a compile-time analysis, which guarantees that the fragments do not present data dependences between them. Instead, TLS solutions assume that the loop can be optimistically executed in parallel, and rely on a runtime monitor to ensure that no dependence violations appear. TLS

 ISBN: 978-84-608-6082-2

solutions can be implemented in software or hardware. From here on, we will focus on software-based TLS.

A dependence violation appears when a given thread generates a datum that has already been consumed by a thread executing a subsequent set of iterations with respect to the original sequential order. In this case, the results calculated so far by the successor (called the offending thread) are not valid and should be discarded. Early proposals [5, 6] stop the parallel execution and restart the loop serially. Other proposals stop the offending thread and all its successors, re-executing them in parallel [7, 8, 9, 10].

Figure 1 shows an example of thread-level speculation. The figure represents four threads executing one out of four consecutive iterations, and the sequence of events that occurs when the loop is executed in parallel. All threads access certain data elements from the `SV` vector. If the values of `x` are not known at compile time, the compiler is not able to ensure that accesses to the `SV` structure do not lead to dependence violations when executing them in parallel. However, the indexes of the data elements being accessed are known at runtime, so dependence violations can be detected and corrected while the program is running.

Speculative parallelization works as follows. If the programmer labels the SV vector as speculative, the code should be instrumented at compile time to monitor at runtime that all uses of `SV` follow sequential semantics. At runtime, each thread maintains a version copy of the elements of the `SV` vector being accessed. All read operations to `SV` are replaced by a function that performs a *speculative load*. This function obtains the most up-to-date value of the element being accessed. This operation is called *forwarding*. If a predecessor (that is, a thread executing an earlier iteration) has already read or written that element then the value is forwarded (as Thread 2 does in Fig. 1). If not, then the function obtains the value from the main copy of the speculative data structure (as Thread 3 does in the figure).

Regarding modifications to the speculative data structure, all write operations are replaced at compile time by a *speculative store* function. This function writes the datum in the version copy of the current processor, and ensures that no thread executing a subsequent iteration has already consumed an outdated value for this structure element, a situation called "dependence violation". If such a violation is detected, the offending thread and its successors are stopped and restarted, in a so-called *squash* operation.

If no dependence violation arises for a given thread, it should *commit* all the data stored in its version copy to the main copy of the speculative structure. Note that commits should be done in order, to ensure that the most up-to-date values are stored. After performing the commit operation, a thread can assign itself a new iteration or block of iterations to continue the parallel work.

 ISBN: 978-84-608-6082-2

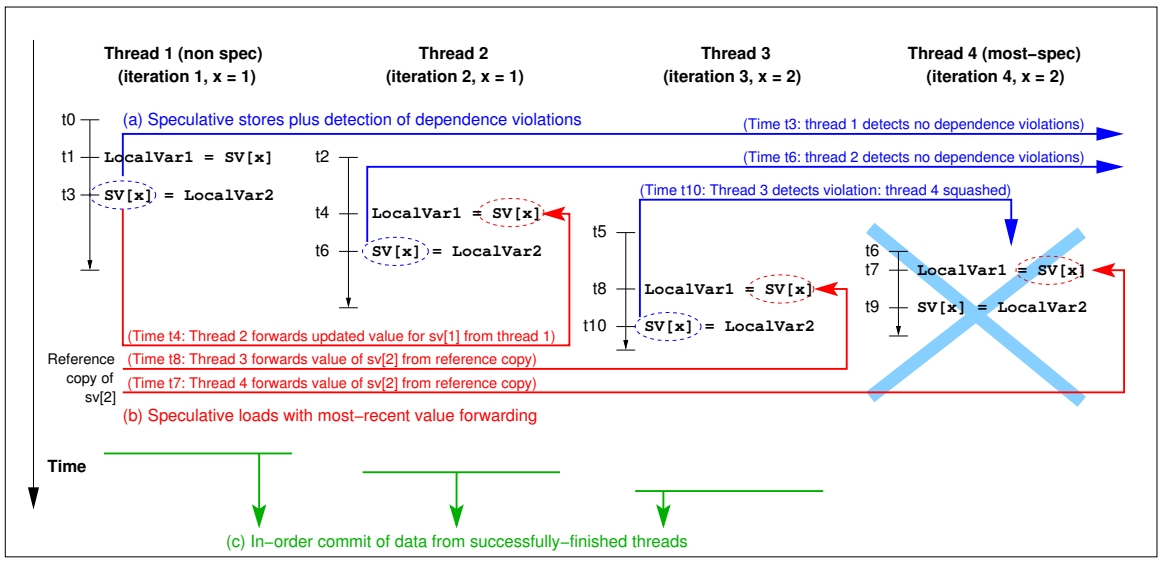

Figure 1: Example of speculative execution of a loop and summary of operations carried out by a runtime TLS library.

3 The ATLaS framework and runtime library

We have developed an extension to OpenMP that incorporates Thread-Level Speculation support. The ATLaS framework [4] allows any loop to be executed in parallel without the need of a prior dependence analysis. This is done by defining a new OpenMP variable classification clause, namely speculative. If the user is unsure about whether the access in parallel to a variable or structure inside a given loop may lead to a dependence violation, he/she may simply classify it as speculative, instead of labeling it as private or shared. In this case, the source code is instrumented at compile time to add TLS execution support. This is done with the help of a GCC compiler plugin [12] that transforms the code, inserting calls to the ATLaS TLS runtime library. When running in parallel, the runtime library ensures that all the accesses to all data elements classified as speculative follows sequential semantics.

The ATLaS runtime library [13] supports all the operations described in the previous section. It follows the design principles of the speculative parallelization library developed by Cintra and Llanos [7], with several improvements that allow, for example, the speculative parallelization of loops that use pointer arithmetic or complex data structures.

One of the key advantages of this library over previous designs is that the ATLaS runtime library is almost free of critical sections. The only critical section needed is the one that manages the data structure that maintains the assignment of chunks of iterations

 ISBN: 978-84-608-6082-2

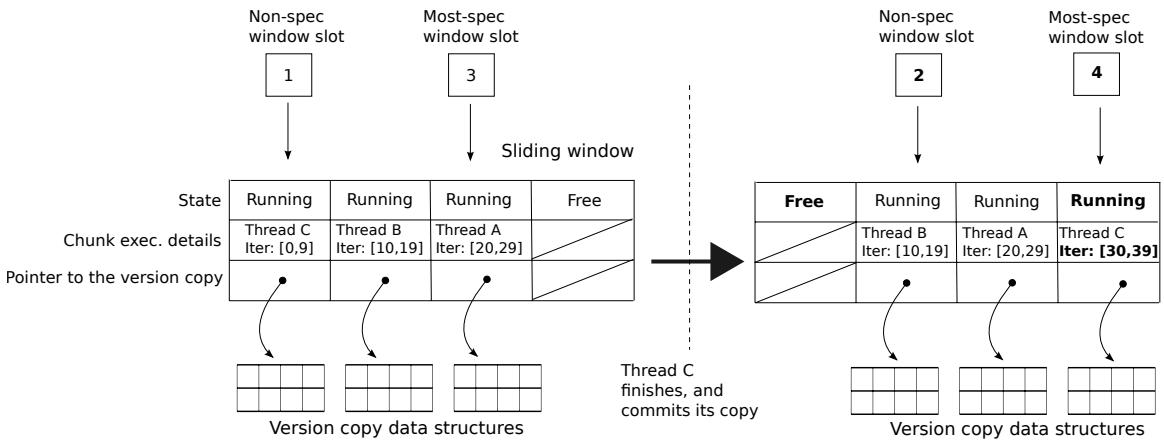

Figure 2: Updating the sliding window that handles the parallel, speculative execution. At a given moment (left), the thread C working in slot 1 is running. When Thread C finishes, it frees its slot and gets a new one, updating non-spec and most-spec pointers (right).

to each thread. ATLaS handles the parallel execution of each chunk of iterations through a sliding window mechanism, which is implemented by a matrix with W columns representing W window slots. Figure 2 depicts a simplified version of the sliding window implementation (see [4, 13] for more details). The figure represents a sliding window with four slots, hosting the execution of three parallel speculative threads.

The thread executing the earliest chunk of iterations (Thread C in our example) is called *non-speculative*, since it has no predecessors that may squash it. Conversely, the thread executing the latest chunk is called the *most-speculative* thread. As can be seen in Fig. 2, two pointers indicate the slots where the non-speculative and most-speculative threads are being executed. The part of the window being used is always the one from the non-spec pointer to the right, up to the most-spec pointer.

The only critical section in the ATLaS runtime library is the one that protects this sliding window. If two or more threads finish at the same time, they could be assigned to the same `Free` slot, resulting in an incorrect execution. Therefore, in order to ensure the correct operation of the ATLaS runtime library, it is necessary to protect the accesses to these shared structures, including the matrix that implements the sliding window mechanism, and the variables that point to the non- and most-speculative slots.

Figure 2 shows what happens when a non-speculative thread successfully finishes its execution. Suppose that Thread C, the one executing the non-speculative thread, finishes its execution and commits its data (the commit operation is not shown in the figure). After this, it enters the critical section to perform several actions. It marks slot 1 as Free; it advances the non-speculative pointer to slot 2; after checking that the slot past the most-speculative one is Free, it assigns it to itself, setting the most-speculative pointer to 4 and

 ISBN: 978-84-608-6082-2

changing its state to Running; and finally, after getting the following chunk of iterations to be executed (iterations 30 to 39 in our example), it exits the critical section. Note that the implementation of the sliding window works in a circular way: When Thread B eventually finishes, it will assign itself the slot that follows the one used by Thread C, in our case the leftmost slot.

The sliding window is modified in three different locations within the ATLaS runtime library. Therefore, the same lock is used in three different parts of the code to protect the access to these data structures. As will be seen, the place from where the access is performed has a noticeable impact in the performance of the protecting system being used. These places are the following:

- **(A) Each time a dependence violation is detected.** In the case of a write to a speculative variable, the thread in charge should update its version copy, and check whether a successor has consumed an outdated value of this variable. If this is the case, a *dependence violation* has happened, so the offending thread should be restarted in order to consume an updated version of the variable. This is done in several steps. First, the thread that has detected the situation should enter the critical section to change the state of the offending thread, from `Running` to `Squashed`, and the most-speculative pointer should be moved backwards to the last Running thread. After these changes, the thread exits the critical section and resumes its normal operation. The offending squashed thread will eventually discover its new state and will enter the critical section (see below).

- **(B) Each time a thread finishes its work,** either because the chunk has been successfully executed or because the thread discovers that it has been squashed. In both cases, the thread enters the critical section to change its own state from Running (resp. Squashed) to Free. After this operation, if the slot following the most-speculative one is Free, the thread assigns it to itself, and advances the most-speculative pointer by one. Otherwise, it means that the following slot is occupied either by a Running thread (this means that the window is full) or by another Squashed thread. In both cases our thread should exit the critical section and attempt to re-enter again, in order to give the thread that is using the slot the opportunity to free it[1] (see below).

- **(C) Each time a thread should wait for a free slot.** If a thread is not able to get a free window slot to work, because the following slot is not Free yet, it should get out and try to gain access again to the critical section to assign itself the following slot and to advance the most-speculative pointer.

[1]Our thread cannot simply wait inside the critical section, because it should get out in order to let the thread using that slot to get in and change its own state.

 ISBN: 978-84-608-6082-2

4 Protecting data accesses with OpenMP and STM

The original TLS runtime library uses the OpenMP critical directive to guarantee exclusive access of the threads to the three parts of the code mentioned above. Because the same data structures are accessed from three different places, the same lock is used to protect them in all cases. Recall that a block of code marked with an OpenMP critical directive is only executed by one thread at a time, whilst the rest of the threads that have reached the same point in the code have to wait. This procedure ensures that the sliding window is always in a consistent state, thus avoiding multiple threads concurrently updating this structure with the potential loss of consistency.

It is easy to see that the serialization of operations described above should imply a noticeable overhead in the performance of the speculative runtime library. A possible way to reduce this performance penalty would be to replace the strict, OpenMP critical construct with the more optimistic constructs that offer the Transactional Memory paradigm. The goal of STM is precisely to help in explicit parallel programming by reducing the costs of the locks required to avoid race conditions in critical sections [14, 15]. While OpenMP critical constructs only allow one single thread at a time inside the critical section, a transactional-based implementation allows several threads inside it, permitting their concurrent execution as long as consistency is not compromised.

However, the optimism of STM, as well as that of TLS, comes at the cost of some overheads, because of the extra instrumentation needed to handle the transactions, as well as the cost associated to the extra runs of particular transactions when a conflict appears. As can be seen, both OpenMP and STM approaches to protect data integrity have identified overheads. It is extremely difficult to predict which approach will be better for a particular problem, since it depends on the application, its running profile, and how often the benchmark accesses the potentially conflictive shared variables, among other factors.

Regarding the programmability, OpenMP has been designed to simplify, to a great extent, the process of parallelization, while the direct use of STM libraries involves a non-trivial instrumentation of the source code, from the definition of the transactional region to monitoring each access to speculative variables. This effort is mitigated by the existence of STM solutions that rely on the compiler to replace STM constructs with calls to the STM library. Some STM approaches propose language extensions or new constructs to declare transactional code regions that comprises statements that must be executed atomically. Then, either an ad-hoc compiler, or an existing compiler modified for this purpose, parses these new constructs, and generates all the instrumentation, in the same way as compilers process OpenMP constructs.

As we said above, OpenMP allows the user to delimit the critical sections with the construct `omp critical`. To declare a transactional region, STM libraries rely on different alternatives, such as new constructs (e.g. GCC-TM's `transaction_atomic{}` [16], the Intel's `tm_atomic{}` [17], or the more generic `transaction{}`), new compiler directives (such

 ISBN: 978-84-608-6082-2

Application	% target loop	Max. speedup P = 64 (Amhdahl)	% of iterations that present dep. violations	# of potentially speculative scalar variables	Size of chunks issued	Critical Sections accessed
FAST	100	64	0.001%	2	25	A, **B**
TREE	95.17	15.84	0%	259	100	**B**
2D-MEC	43.75	1.76	0.009%	10	1 800	A, **B**
2D-Hull, Kuzmin	100	64	0.0008%	1 206	11 000	A, **B**, C
2D-Hull, Square	100	64	0.0032%	3 906	3 000	A, **B**, C
2D-Hull, Disc	100	64	0.0219%	26 406	1 250	**A, B**, C
Delaunay	97.60	25.47	0.5%	12 030 060	2	**A, B, C**

Table 1: Percentages of potentially parallelism for the benchmarks and loops considered, together with some benchmarks' characteristics. Chunk sizes were selected to obtain maximum speedups.

as IBM's [18] `tm_atomic{}`), or even new OpenMP pragmas, such as `omp transaction`, defined by OpenTM [19]. Unfortunately, Intel STM compiler and OpenTM are not currently available, while the IBM compiler's transactional built-in memory functions are only valid for Power8 architecture and Blue Gene/Q.

In this work, we have used OpenMP, the GCC-TM, and the TinySTM libraries [20, 21] to protect the accesses to the sliding window described previously. These three approaches simplify the parallelization process with the mentioned constructs and directives. Moreover, GCC-TM defines a specification for transactional language constructs that other STM libraries can leverage, and hence, changing the underlying STM library is just a process of proper linking. In fact, TinySTM is compatible with GCC-TM, allowing programmers to use the same interface and save some programming effort.

Handling the critical sections with OpenMP is straightforward: The programmer should simply delimit the region by using the defined `omp critical` directive. This process is similar when using the GCC-TM specification. However, to ensure that the transaction is atomically executed, there may be certain functions inside the transaction that must *not* be executed. Since the compiler is not able to detect this issue for the functions called within a transaction, it is also necessary to annotate their declaration and specify whether they are safe to be called, with the `transaction_safe` attribute.

The following section describes the performance results obtained by ATLaS when using these three solutions to execute a set of real-world and synthetic benchmarks.

5 Experimentation

Experiments were carried out on a 64-processor server, equipped with four 16-core AMD Opteron 6376 processors at 2.3GHz and 256GB of RAM, which runs Ubuntu 12.04.3 LTS. All threads had exclusive access to the processors during the execution of the experiments, and we used wall-clock times in our measurements. We have used the OpenMP implementation

 ISBN: 978-84-608-6082-2

from GCC 4.8.2, and the transactional libraries from GCC-TM 4.8.2, and TinySTM 1.0.5.

To perform the experiments, we used both real-world and synthetic benchmarks. The real-world applications include the 2-dimensional Minimum Enclosing Circle (2D-MEC) problem [22], the 2-dimensional Convex Hull problem (2D-Hull) [23], the Delaunay Triangulation problem [24, 25], and a C implementation of the TREE benchmark [26]. We have also used a synthetic benchmark called Fast [4], which presents almost no dependences between iterations, and which was designed to test the overheads of the ATLaS runtime library.

Table 1 summarizes the characteristics of each benchmark, including the percentage of execution time consumed by each target loop, an estimation of the maximum speedup attainable (applying Amhdahls Law), the percentage of iterations of the target loop that lead to runtime dependence violations, the number of speculative variables within the loop, and the size of the chunk of consecutive iterations speculatively executed. I/O time consumed by the benchmarks were not taken into account. We also give an indication of which accesses to the sliding window protected by the critical section are more frequent in the benchmark (bold letters indicate that the corresponding call is more frequent). The performance results obtained by each benchmark and library used are summarized in Fig. 3.

The Fast benchmark was designed to test the efficiency of the speculative scheduling mechanism, with few iterations leading to a dependence violation, although they are enough to prevent a compiler from parallelizing the loop. This benchmark has very few dependence violations, so the critical section is primarily accessed to get the following chunk of iterations to be executed (access of type B in our library). As can be seen in the corresponding performance plot, OpenMP and the STM libraries handle the critical sections equally well, delivering almost identical performances, with a speedup of up to 37× with 64 processors.

Unlike the rest of the benchmarks, TREE does not suffer from dependence violations, but it is still not parallelizable at compile time because the compiler is not able to ensure that there are no data dependencies. Since it does not present dependence violations, the code that accesses the critical section is primarily B. Again, OpenMP and STM solutions deliver the same performance, with a peak speedup of 6× when running this benchmark with a 4096-point input set. As can be seen in the figure, the overheads of the TLS runtime library lead to a performance loss when using 48 threads or more, regardless of the implementation chosen to handle critical sections.

The 2D-MEC benchmark is a tricky code which has only 10 speculative variables that are frequently accessed. This benchmark calls the speculative loop many times with a very different number of iterations each time, making threads access the sliding window system frequently to get the following chunk. As long as it presents some dependence violations, the critical sections are accessed by codes A, but mostly B (C is rarely accessed in this benchmark). For this benchmark, the use of the OpenMP critical sections leads to the best performance, while the STM libraries leading to much poorer results. OpenMP gets a peak

 ISBN: 978-84-608-6082-2

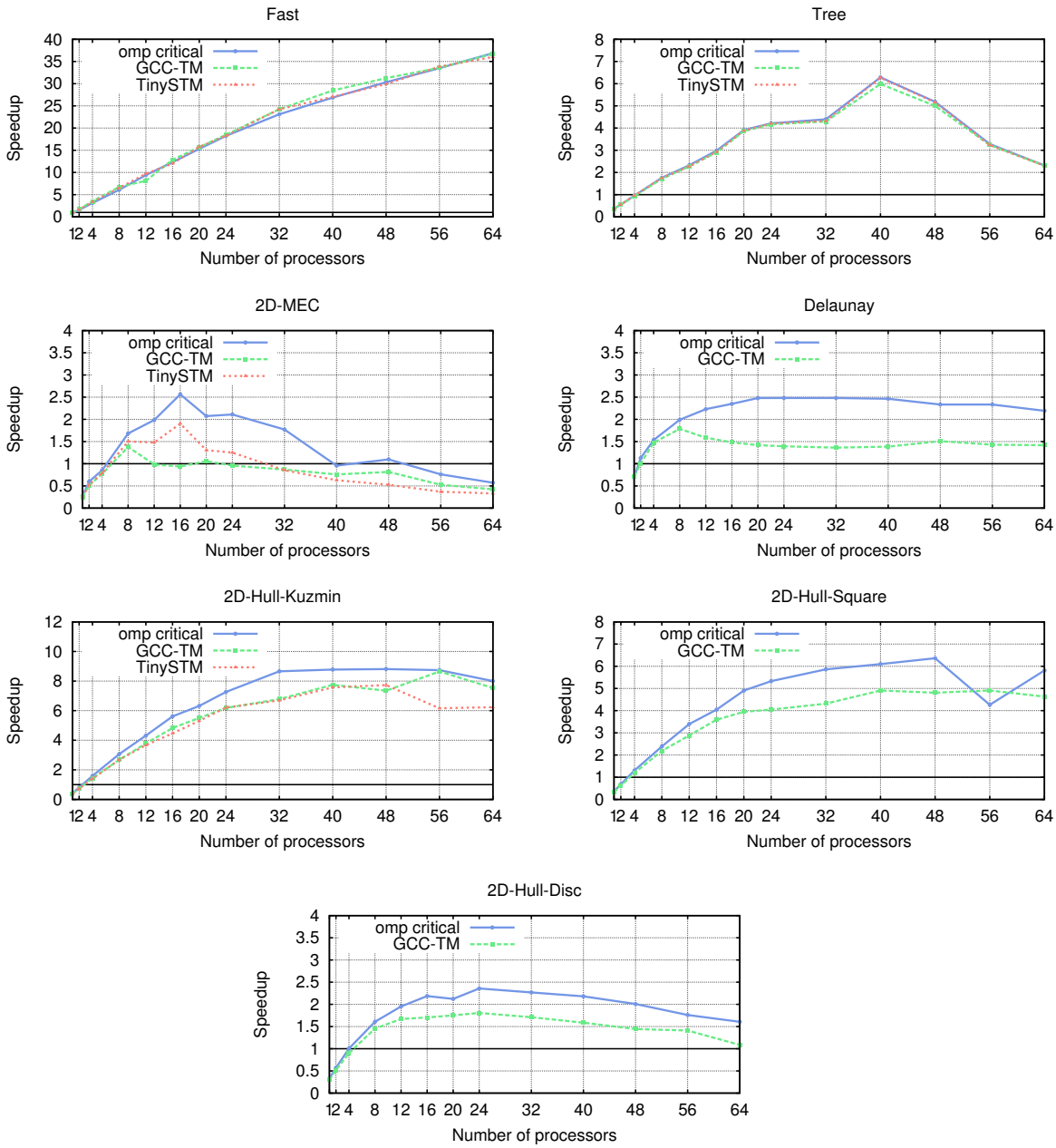

Figure 3: Speedups by number of processors for each tested benchmark, comparing the performance obtained by using OpenMP critical sections, and GCC-TM and TinySTM transactional libraries

ISBN: 978-84-608-6082-2

speedup of 2.5×, outperforming the speedup of TinySTM (1.9×) and GCC-TM (1.3×).

The overheads of TinySTM and GCC-TM are even more noticeable for the Delaunay problem, the benchmark with the highest number of dependence violations (0.5%) and the smallest optimum chunk size (just two iterations). Therefore, critical sections are accessed quite frequently, because of the detection of dependence violations (code A), the need to schedule the execution of many chunks (code B), and some degree of load imbalance which frequently makes the window to be full, leading to contention (code C). The stress on the exclusive access management is so high that, in fact, the TinySTM library is not able to properly handle the accesses to the protected sliding window when running this problem with two or more threads. Hence, we cannot present performance figures for this library in this case. Meanwhile, GCC-TM leads to very modest speedups compared with the use of OpenMP critical sections.

A similar issue occurs with the execution of the 2D-Hull problem with different input sets. We have found that runs of the 2D-Hull problem, with datasets whose execution involves a larger amount of conflicts and dependence violations (as happen when the Square and Disc input sets are used), do not finish when using TinySTM, regardless of the number of parallel threads. TinySTM also fails when using 2 threads and the Kuzmin dataset, producing different, unexpected outcomes on each execution. Besides this, although both STM libraries perform similarly for each dataset, their performance is consistently worse than the one offered by the OpenMP critical sections.

From the results described above we can make the following observations:

1. Not surprisingly, the relative performance of the lock implementations depends, to a great extent, on the running profile and the particular characteristics of each benchmark. In our case, this profile not only depends on the number of dependence violations (that requires accesses through code A to squash the offending threads), but the number of chunks of iterations that should be scheduled (requiring accesses to codes B and C).

2. In general terms, the more frequent the accesses to the protected data structures, the poorer the performance of the STM libraries with respect to the OpenMP critical sections. In fact, if the number of accesses is high enough, TinySTM starts to fail. In this sense, we have found that the GCC-TM implementation is more robust.

3. While we expected that the performance of the STM implementations would decay when the number of accesses to the protected structures increases, we also expected that, when the number of accesses were relatively low, the performance obtained with STM would be better than using the OpenMP critical sections. As our performance figures show, this is not the case, the use of the OpenMP critical sections being the best choice for our problem.

 ISBN: 978-84-608-6082-2

6 Conclusions

The aims of this study were to test whether the use of STM might lead to an improvement in the performance of our software-based, thread-level speculation system, and to assess the efficiency and maturity of STM libraries. Our experimental results show that, in general, STM solutions deliver poorer performance than the use of the classical OpenMP critical sections for our problem. Regarding the maturity of STM libraries, in our experience, the GCC-TM implementation is mature enough to be used for production purposes, while TinySTM still presents room for improvement. Our present and future work includes testing the performance of hardware TM solutions in this same context.

7 Acknowledgments

This research has been partially supported by MICINN (Spain) and ERDF program of the European Union: HomProg-HetSys project (TIN2014-58876-P), CAPAP-H5 network (TIN2014-53522-REDT), and COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

References

[1] Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J.: Parallel Programming in OpenMP. 1 edn. Morgan Kaufmann (October 2000)

[2] Shavit, N., Touitou, D.: Software transactional memory. Distributed Computing **10** (1997) 99–116

[3] Wong, M., Bihari, B.L., de Supinski, B.R., Wu, P., Michael, M., Liu, Y., Chen, W.: A case for including transactions in OpenMP. IWOMP'10 Proceedings (2010) 149160

[4] Estebanez, A., Aldea, S., Llanos, D.R., Gonzalez-Escribano, A.: An OpenMP extension that supports thread-level speculation. IEEE Transactions on Parallel and Distributed Systems, to appear.

[5] Gupta, M., Nim, R.: Techniques for speculative run-time parallelization of loops. In: Proc. of the 1998 ACM/IEEE Conference on Supercomputing. (1998) 1–12

[6] Rauchwerger, L., Padua, D.: The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In: PLDI'95 Proceedings. (1995) 218–232

[7] Cintra, M., Llanos, D.R.: Toward efficient and robust software speculative parallelization on multiprocessors. In: PPoPP'03 Proceedings. (June 2003) 13–24

[8] Dang, F.H., Yu, H., Rauchwerger, L.: The R-LRPD test: Speculative parallelization of partially parallel loops. In: 16th IPDPS Proceedings. (2002) 20–29

[9] Xekalakis, P., Ioannou, N., Cintra, M.: Combining thread level speculation helper threads and runahead execution. In: ICS 2009 Proceedings. (2009) 410–420

 ISBN: 978-84-608-6082-2

[10] Gao, L., Li, L., Xue, J., Yew, P.C.: SEED: A statically greedy and dynamically adaptive approach for speculative loop execution. IEEE Transactions on Computers **62**(5) (2013) 1004–1016

[11] Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. In: PLDI Proceedings. (2007) 211–222

[12] Aldea, S., Estebanez, A., Llanos, D.R., Gonzalez-Escribano, A.: A new gcc plugin-based compiler pass to add support for thread-level speculation into openmp. In: Euro-Par 2014 Proceedings. LNCS 8632 (2014) 234–245

[13] Estebanez, A., Llanos, D.R., Gonzalez-Escribano, A.: New data structures to handle speculative parallelization at runtime. In: HLPP '14 Proceedings. (2014)

[14] Ceze, L., Tuck, J., Torrellas, J., Cascaval, C.: Bulk disambiguation of speculative threads in multiprocessors. ACM SIGARCH Computer Architecture News **34**(2) (2006) 227–238

[15] Barreto, J., Dragojevic, A., Ferreira, P., Filipe, R., Guerraoui, R.: Unifying thread-level speculation and transactional memory. In: Middleware '12 Proceedings. (2012) 187–207

[16] Riegel, T.: Transactional Memory in GCC. `https://gcc.gnu.org/wiki/TransactionalMemory` (2012)

[17] : Intel C++ STM Compiler, Prototype Edition (2012)

[18] IBM: Thread-level speculative execution for C/C++ (2012) Tech. report.

[19] Baek, W., Minh, C.C., Trautmann, M., Kozyrakis, C., Olukotun, K.: The OpenTM transactional application programming interface. In: 16th ISCA Proceedings, IEEE Computer Society (2007) 376–387

[20] Pascal Felber, Christof Fetzer, P.M., Riegel, T.: Time-based software transactional memory. IEEE Transactions on Parallel and Distributed Systems **21**(12) (December 2010) 1793–1807

[21] Pascal Felber, C.F., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: PPoPP '08 Proceedings . (2008) 237–246

[22] Welzl, E.: Smallest enclosing disks (balls and ellipsoids). In: New results and new trends in computer science. Volume 555 of Lecture Notes in Computer Science., Springer-Verlag (1991) 359–370

[23] Clarkson, K.L., Mehlhorn, K., Seidel, R.: Four results on randomized incremental constructions. Comput. Geom. Theory Appl. **3**(4) (1993) 185–212

[24] Mücke, E.P., Saias, I., Zhu, B.: Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. In: SoCG '96 Proceedings. (1996) 274–283

[25] Devroye, L., Mücke, E.P., Zhu, B.: A note on point location in Delaunay triangulations of random points. Algorithmica **22** (1998) 477–482

[26] Barnes, J.E.: TREE. Institute for Astronomy. University of Hawaii. ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode/ (1997)

 ISBN: 978-84-608-6082-2