Transactional Memory For Distributed Systems

Michael Schöttner, Marc-Florian Müller, Kim-Thomas Möller, Michael Sonnenfroh

michael.schoettner@uni-duesseldorf.de

Heinrich-Heine-Universität Düsseldorf, 40225 Düsseldorf, Germany

1. Introduction

Because of physical constraints CPU clock rates are no longer growing as fast as in the past. As a consequence there is a paradigm shift in computer architecture moving to multi and many core CPUs. Only concurrent (multi-threaded) programs can exploit the potential of such CPUs pushing parallel programming into mainstream.

Programmers are faced with the challenge to properly and efficiently synchronize their threads which is not an easy task. Proper synchronization means that all accesses to shared data must be protected using locks in order to ensure correct program results. This requires the programmer to reason about thread access patterns and to label all critical sections using locks of parallel programs. Efficiency of synchronization refers to maximization of concurrency which is essential to gain any speedup by utilizing multiple cores. The optimization of concurrency is often a frustrating task because suddenly achieved speedups often cause incorrect results. Another class of painful errors include deadlocks, which sometimes occur nondeterministically depending on input parameters, machine speed etc.

One recent approach to reduce the burden of dealing with concurrency is to use a transactional memory [1]. The idea of providing hardware support for transactions originated in 1986 [6] and the idea was popularized in the early 90s [5]. Transactional memory has a lot of similarities with data base transactions combined with optimistic synchronization [4]. As known from the database world using optimistic concurrency control avoids deadlock situations but prefers short TAs and rare conflicts. In contrast to database TAs, transactions in the context of transactional memory (TM) are not queries to databases, are mostly short TAs, and do not follow all ACID properties (A=atomicity, C=consistency, I=isolation, D=durability). Typically, TAs in a TM support atomicity, isolation, and consistency but not persistence [2]. In the following text we discuss TAs in the context of TM, only.

Atomicity defines that either all changes to TM locations within in one TA are performed ore none. Atomicity requires the programmer to define the begin and end of a TA, e.g. by inserting calls to BOT and EOT functions into the source code. Typically, critical sections are marked by BOT and EOT. Some TM-implementations reuse existing language constructs. For example Java-based TM implementations imply TA-boundaries by the brackets of the synchronized statement [3].

Isolation implies that changes of TAs to TM become visible after a successful commit, only. Typically, intermediate modifications are not visible to overlapping TAs. Thus, if there are conflicts between overlapping TAs all conflicting TAs are automatically serialized by the the TM. Aborting a TA requires to undo all its changes which is easy for memory but difficult for I/O, see Section 2.

Consistency implies that if the TM is in a consistent state any TA will move it to another consistent state. Consistency is is ensured by a serializeable commit order.

Any implementation needs to record read- and write-sets, provide restartability (in case of an abort caused by a conflict), and need a validation phase to detect conflicts during commit time. TM can be implemented in hardware or software, targeting single CPUs (with multiple cores) or distributed systems (multiple machines, each node potentially having many cores). There are also hybrid approaches realized in hard- and software.

Hardware TM systems may have modifications in processors, cache and bus protocol to support TAs. For example, Transactional Consistency/Coherence (TCC) replaces Intels MESI-protocol [7]. SUN has announced TM-support in future server CPUs. HW-based approaches benefit of the system bus offering high bandwidth and low latency allowing fast data transfers and commits.

Software transactional memory provides transactional memory semantics in a software runtime library or the programming language, e.g. Haskell STM [8] or Intel STM Compiler [29]. Software implementations impose a higher overhead but do not require specific hardware.

An integrated approach of TM spawning from the hardware to the application is currently studied within the Velox project [10].

While most of the TM efforts focus on single nodes there are also projects trying to adopt TM ideas and concepts to distributed systems. The Plurix OS (Ulm University) was one of the first projects using speculative transactions for data sharing in clusters [11]. Recently, TM has also been successfully studied on large clusters [12]. But also within large scale applications, e.g. game server clusters, speculative transactions are used to synchronize servers [13].

Finally, so called in-memory data grids allow remote memory access within grid environments [14]. In this context the Object Sharing Service (OSS) [16] developed within the EU-funded project XtreemOS studies speculative transactions for grids [15].

2. Implementation aspects

Conflict Detection

One transaction at a time is allowed to validate using a backward or forward strategy. Both compare the validating TA_v with overlapping transactions. Backward validation compares TA_v with TAs that have already committed and forward validation with those TAs still in progress. Committed TAs cannot be aborted, such that forward validation is more flexible as it allows to select which TA to abort and is the base for enhanced fairness strategies. When using forward validation, two transaction TA_1 and TA_2 conflict if TA_1 wants to commit and has written a variable x that has been read by TA_2. If TA_1 commits before TA_2 than TA_2 has read an outdated value of x and thus needs to be restarted to read the correct (new) value of x.

The consistency unit size for conflict detection is a design question. It can be variables, cache lines, objects, memory pages [17] etc. like known from Distributed Shared Memory (DSM) systems [9].

Obviously, if there are many cores or nodes executing many short transactions, they all compete to be the next to be validated. This mandatory serialization of the validation phase limits scalability even if all transactions are conflict free. The latency of the network is a sensitive factor here. When implementing fairness strategies the validation phase will get more complicated and will consume even more time.

Most TM systems use a first-wins strategy. Here it is up to the system designer to define when conflicts are detected by overlapping transactions. It can be enforced synchronously during the commit requiring the committing TA to wait for acknowledgments from all other nodes. An alternative is to postpone conflict detection to commit time, which is acceptable for short transactions but not for long running ones. Finally, this can also be done concurrently in-between. The committing transactions sends its write-set to overlapping transactions but does not wait for acknowledgments. Validation on all affected nodes takes place concurrently but the system must ensure that data or commit requests of affected nodes to not interfere with commits in transit.

Finally, it is also important to mention that concurrent conflict resolution interrupts transactions even if they are not in conflict. At a larger scale this might cause significant overhead.

Restartability

When a conflicting TAs needs to be aborted, all its changes need to be rolled back. For caches and memory it is easy to implement shadow images by copying the old version before the first write access. Depending on the write-set size of a TA, the time to copy large amounts of data may consume considerable CPU time and virtual memory.

For third-party libraries and OS system calls, restartability is more difficult to implement. As these code is not designed to be restartable nor to be interrupted at all locations, aborts might need to be postponed to transaction-safe code. But the restarting itself might require compensating operations [19], or some TM implementations prohibit to access third-party code within TAs.

Another aspect is restartable I/O which is hard or sometimes even impossible. If a message has been sent over the network it cannot be grabbed back but some systems proposed to sent anti-messages. For file systems one could imagine to make a shadow copy of a file before modifying it within a TA but this might introduce an unacceptable high overhead for very large files. Therefore, some TM systems prohibit device access within TAs. Others allow device access but postpone writes until the commit, e.g. smart buffers [18].

3. Distributed Transactional Memory

Object Sharing Service – Overview

The Object Sharing Service (OSS) implements a TM for grid environments integrated within the Linux-based operating system XtreemOS. OSS is a multi consistency approach also implementing a software-based TM in C. Memory accesses are detected using virtual memory functionality. A program may allocate objects/memory in local or in shared address space (64-Bit). Shared data may contain references to other shared data. References are 64-Bit addresses that can be swizzled to any other virtual address to support heterogeneous setups similar to the Interweave approach [21]. It is possible to allocate shared data within TAs and also to call system functions. However, aborts caused by conflicts may occur any time and are delayed until the application is back in transaction-safe code [22].

By implementing a similar approach like millipage [20] we avoid false sharing and support an object-based access detection. The millipage approach statically maps different logical memory pages onto one physical frame, each logical page containing one object. Although no physical memory is wasted this approach consumes considerable logical memory which is no problem for 64-Bit architectures. However, we pollute the TLB but the overhead but is a magnitude lower than accessing memory objects over the network. Currently, we extend the millipage approach by adaptively changing the cache consistency unit size during runtime according to monitoring information. If there is no false sharing we can allow access to all objects mapped to the same physical frame when one of these objects is accessed. This corresponds to a page-based access control which can be changed to an object-based access detection when false sharing is detected during runtime. A more sophisticated approach will try to dynamically detect object access groups without false sharing within a single page frame. Typical programming language objects are rather small, e.g. 32-64 Byte thus we expect to have 64-128 objects per page frame.

Replicas are important for performance and reliability. When a replicated object is updated is defined by the underlying consistency model, in case of TM it is at commit time. Depending on the monitored access patterns it is useful to invalidate some replicas (on nodes having not accessed these objects for a longer time) and to update those on nodes frequently accessing these objects. From a reliability perspective it is also important to scatter replicas in the network to ensure application progress also in case of node failures and partitioned networks.

In order to avoid unnecessary network communication, the replica management detects if a TA has modified objects that are not accessible on other nodes. In such a situation the TA can just commit locally. For reliability reasons we enforce backup replicas on other nodes that cannot directly be accessed in order to allow local commits.

Scalability aspects
In order to improve scalability we have implemented a super-peer overlay network where each peer is connected to its assigned super-peer and all super-peers are connected in a mesh. For these network connections we use TCP. Data transfers between nodes are processed directly using TCP, too. In the future we plan to study a reliable UDP protocol. Node distances are estimated using ping round trip times (RTT). To avoid flooding the network with ping messages we plan to implement a network coordinate system similar to Vivaldi [23]. Due to node failures the overlay network must be dynamically reorganized.

As stated before, only one peer can validate a TA at a given time. This serialization is implemented by a circulating token. In order to reduce the number of peers competing to commit TAs, each peer commits through its associated super peer. When a super peer has the token it can commit a bunch of TAs of its peers but it is allowed to hold the token only for a given time to avoid monopolization of the network. Those super peers waiting for the token can validate pending TAs of their peer group locally and in case of conflicts decide which TA to abort. Thus unnecessary network traffic can be avoided, too. There is one ultra peer responsible for token passing. Although this central solution may limit scalability we want to avoid broadcast messages between super peers and race conditions. Furthermore, node failures and token loss detection can be simplified with this approach. When objects are accessed for the first time peers do not know peers holding these objects. Thus they request them from their super peer which either forwards this request to a peer stored in its cache. If there is no information in the cache of the super peer or if the information is out-dated the data is searched using a CAN-based DHT [24].

Committing through super peers introduces some delay, which can be alleviated by locally starting the next transaction before the pending commit is finished. The idea is to take the risk of cascading aborts but to be optimistic and try to proceed than just waiting for a commit to finish. The implementation requires version management for the shared data and also the programmer to mark pipelined TAs. If there is a chain of TAs on a peer all but the last one can be pipelined. The last TA in this chain must be blocking because the non-transactional code after the last TA cannot be restarted. If all actions would be performed within TAs, like for example in Plurix, the pipelined TA approach would be more transparent but we want to avoid the transactional overhead wherever unnecessary.

Another technique to reduce the number of peers involved in TA serialization is to introduce multi-consistency domains. Here the transactional memory is divided into different sections/segments each synchronized separately and the programmer needs to define consistency domain affiliation during object allocation. This is not transparent to the programmer but for some applications rather straightforward. For example multi-user virtual worlds are typically separated in different islands and an avatar can be on one island at a given time, only. When sharing the game state using transactional memory is seems to be natural to associate with each virtual island one consistency domain. We prohibit TAs from modifying data belonging to different consistency domains within one TA. From a technical point of view this could be implemented but it is complicated and might end up in slow TA processing.

Further optimizations include read-access to transactional memory outside of transactions. This is useful for regularly updated or converging values and reduces the burden of strong consistency implied by TM.

4. Lessons learned

We have implemented a running prototype of OSS tested with a parallel ray tracer in a cluster and the multi-user virtual world Wissenheim [25] in a small-scale setup between Rennes, France and Duesseldorf, Germany. Wissenheim was originally designed to run under the Plurix OS. The latter executes all actions within TAs. Wissenheim shares the scene graph and user interactions through TM, and has been adapted to Linux and OSS within the XtreemOS project.

In OSS, not all code is executed within TAs and thus the programmer must mark TA-boundaries using BOT and EOT. OSS allows to start a TA in one function and to commit it in another one. From a user point of view it seems to be better to allow such a situation to be able to support the information hiding principle of object-oriented programming-models. In Wissenheim for example, BOT and EOT are encapsulated to automatically switch memory allocators before BOT and right after EOT. It is up to the programmer to define the granularity of transactions, whether to split a function into one or several TAs. Due to the underlying optimistic synchronization the programmer should define short transactions in order to reduce conflict probability.

So, depending on the conflict pattern it might be necessary to split one TA into several smaller ones (if possible) to reduce conflicts. Obviously, in these situations the programmer has to reason about data access patterns and potential conflicts. A monitoring runtime provides feedback to the programmer giving useful hints which TAs conflict often. But it is also important to keep in mind that too many very short TAs and might be painful because of the commit overhead.

As we had expected scalability of TM is limited by several factors: the network latency and bandwidth. Network latency is not a problem for multi-core TM but is painful for distributed TM systems. Although bandwidth is much lower in distributed systems it is not a major issue when using at least DSL connections combined with compression and diffs.

Furthermore, even for perfectly programmed transactional programs two other factors limit scalability: the requirement of serializing the validation phase and the strong consistency enforced by TM. Both factors are especially painful in distributed environments and force system designers to provide further optimizations as presented in Section 3 which reduce the transparency of TM.

Experiments with our parallel ray tracer on a 16 node cluster (Gigabit Ethernet network) show almost linear speedup after doing some optimizations. Of course the ray tracer can be easily parallelized and can be seen as a best case for a TM. The costs for restartability and validation were not noticeable in this small scale. Further experiments on Grid'5000 [26] with more nodes are planned.

As expected it was much more difficult to port Wissenheim to a WAN-based TM and explicit TAs. It is worth to mention that Wissenheim was from the beginning designed for transactional processing but all code executed within TAs. Nevertheless, conflicts were much more painful in a WAN and we had to carefully define TA-boundaries in order to keep conflicts at a minimum. But even these changes were not sufficient and we needed to introduce read accesses to TM outside TAs, e.g. for motion vectors of avatars. From time to time these vectors are updated using TAs but in between avatar positions are animated estimating their position using dead reckoning. This is a typical approach in professional online games.

Still there are not enough real applications using TM rather small benchmarks, e.g. [28], but some publications show it is neither easy to convert existing applications into transactional processing nor does this approach easily lead to scalable programs. For example in [27] the authors ported the quake server engine to TM and evaluated the application on a eight core CPU showing scalability problems.

5. Conclusions

TM aims at simplifying concurrency control for the programmer in parallel and distributed programs. For a smaller scale and single nodes this promise can be fulfilled. However, for many core CPUs and distributed TM scalability is limited by several factors forcing the programmer to reason about transaction boundaries, access patterns, etc. Like expected, experiences show that this is not a trivial task especially for applications that have not been designed from the beginning for transactional processing. Nevertheless, TM avoids deadlocks, and optimization in TM is a step-by-step process simplified by providing monitoring information from applications runs.

More real applications need to be developed for or ported to TM in order to assess its usefulness. But due to the growing interest in TM we expect more and more transactional applications to show up in the near future.

Regarding scalability, we believe that TM must be extended by several optional features which make concurrency less transparent for the programmer. The TM community working on many-core systems will certainly soon face similar limitations like we did with our distributed TM.

Extensions for operating systems to support TM would improve performance and could provide restartability for system calls and I/O.

Overall we believe it is worth to continue TM research on all levels including distributed systems to study how far this concept can be pushed without burdening the programmer to much.

Our work is funded by the EC within the FP6-XtreemOS project.

References:

[1] J. Larus and C. Kozyrakis, “Transactional Memory”, Communications of the ACM, Vol. 51, Issue 7, July 2008.

[2] P. Felber, C. Fetzer, R. Guerraoui, T. Harris, “Transactions are back---but are they the same?”, ACM SIGACT News, Vol. 39, Issue 1, March 2008.

[3] B. D. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. C. Minh, L. Hammond, C. Kozyrakis, K. Olukotun, “Executing Java programs with transactional memory”, Science of Computer Programming, Volume 63, Issue 2, December 2006.

[4] H. T. Kung and J. T. Robinson, “On optimistic methods for concurrency control”, ACM Transactions on Database Systems 6(2), 1981.

[5] M. Herlihy, J. Moss, B. Eliot, “Transactional memory: Architectural support for lock-free data structures”, International Symposium on Computer Architecture (ISCA), 1993.

[6] T. Knight, “An architecture for mostly functional languages”, ACM conference on LISP and functional programming, 1986.

[7] L. Hammond, V. Wong, M. Chen, B.D. Carlstrom, J.D. Davis, B. Hertzberg, M.K. Prabhu, W. Honggo, C. Kozyrakis, K. Olukotun, “Transactional memory coherence and consistency”, International Symposium on Computer Architecture, 2004.

[8] A. Discolo, T. Harris, S. Marlow, S. Peyton Jones, S. Singh, “Lock Free Data Structures using STMs in Haskell”, Intl. Symposium on Functional and Logic Programming, Fuji Susono, JAPAN, April 2006.

[9] A. Judge, P.A. Nixon, V.J. Cahill, B. Tangney, and S. Weber, “Overview of Distributed Shared Memory”, technical report, Trinity College Dublin, 1998.
[10] http://www.velox-project.eu
[11] http://www.plurix.org
[12] R. L. Bocchino, V. S. Adve, B. L. Chamberlain, “Software transactional memory for large scale clusters”, ACM SIGPLAN Symposium on Principles and practice of parallel programming, 2008.
[13] http://www.projectdarkstar.com
[14] http://www.gigaspaces.com/edg
[15] http://www.xtreemos.eu
[16] K.-T. Möller, M.-F. Müller, M. Sonnenfroh, and M. Schöttner, “A Software Transactional Memory Service for Grids”, International Conference on Algorithms and Architectures for Parallel Processing, Taipei, Taiwan, 2009.
[17] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. Van Biesbrouck, G. Pokam, D. Brad Calder, O. Colavin, “Unbounded page-based transactional memory”, ACM SIGPLAN Notices, Vol. 41, Issue 11, 2006.
[18] T. Bindhammer, R. Göckelmann, O. Marquardt, M. Schöttner, M. Wende, and P. Schulthess, “Device Programming in a Transactional DSM Operating System”, Asia-Pacific Computer Systems Architecture Conference, Australia, 2002.
[19] H. Volos, A. Jaan Tack, N. Goyal, M. M. Swift, and A. Welc, “xCalls: Safe I/O in Memory Transactions”, European Conference on Computer Systems, Nuremberg, Germany, 2009.
[20] A. Itzkovitz and A. Schuster, “Multiview and millipage – finegrain sharing in page-based dsms”, Symposium on Operating systems design and implementation, Berkeley, CA, USA, 1999.
[21] C. Tang, D. Chen, S. Dwarkadas, and M. L. Scott, “Integrating Remote Invocation and Distributed Shared State”, Intl. Parallel and Distributed Processing Symp., Apr. 2004.
[22] M.-F. Müller, K.-T. Möller, M. Sonnenfroh, M. Schöttner, “Transactional Data Sharing in Grids”, International Conference on Parallel and Distributed Computing and Systems, Orlando, USA, 2008.
[23] R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris, “Practical, distributed network coordinates”, SIGCOMM Comput. Commun. Rev., 34(1), 2004.
[24] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, “ Scalable Content-Addressable Network”, SIGCOMM’01, 2001.
[25] http://www.wissenheim.de
[26] http://www.grid5000.fr
[27] F. Zyulkyarov, V. Gajinov, O. Unsal, A. Cristal, E. Ayguadé, T. Harris, M. Valero, “Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server”, 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2009.
[28] C. C. Minh, J. Chung, C. Kozyrakis, K. Olukotun, “STAMP: Stanford Transactional Applications for Multi-Processing”, In IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization, Sept. 2008.
[29] A. Adl-Tabatabei, B. T. Lewis, V. Menon, B.R.Murphy, B. Saha and T. Shpeisman, “Compiler and runtime support for efficient software transactional memory”, ACM SIGPLAN Programming language design and implementation, 2006