Memory Management for Transaction Processing Core in Heterogeneous Chip-Multiprocessors

Ferad Zyulkyarov[12], Osman S. Unsal[1], Adrian Cristal[1], Milos Milovanovic[12], Eduard Ayguade[12], Mateo Valero[12], Tim Harris[3]

[1]Barcelona Supercomputing Center, Jordi Girona 29, 08034 Barcelona, Spain
[2]Universitat Politècnica de Catalunya, Departament d'Arquitectura de Computadors, Jordi Girona, 1-3, D6 Campus Nord, 08034 Barcelona, Spain
[3]Microsoft Research Ltd, J J Thomson Avenue, Cambridge, CB3 0FB, UK
[1]{ferad.zyulkyarov, osman.unsal, adrian.cristal, milos.milovanovic, eduard.ayguade, mateo.valero}@bsc.es
[3]{tharris}@microsoft.com

Abstract. Transactional Memory is a promising technique for easy coding of efficient multi-threaded applications that would increase the utilization of the current and next generation Chip-Multiprocessors. Execution of a transaction in any Transactional Memory system involves memory management as the system creates tentative working copies from original objects and later deletes them when the transaction commits or aborts. Transactional Processing Core is a simple dedicated core for accelerating the slowest and most common transactional operations in Software Transactional Memory which are conflict detection and address translation. In this paper we propose a specialized memory management for the Transaction Processing Core in heterogeneous Chip-Multiprocessors. With the introduced enhancement, the transactional core is able to perform the auxiliary operations related to the tentative working copies of the object. Thus off-loading the Software Transactional Memory library and making it more simpler.

1 Introduction

The advent of shared-memory Chip Microprocessors (CMP) has created a new opening to exploit thread-level parallelism. Microprocessor manufacturers are packing more processing cores on die with each technology node. A new Moore´s law is proposed which postulates that the number of cores per CMP will double every two years [1]. However, programming those many-cores will be a challenge with the existing frameworks. Transactional Memory (TM) is a promising key technology for tackling this problem by abstracting some of the complexities associated with concurrent access to shared data [2]. In TM, transactions replace locking with atomic execution units, so that the programmer can focus on determining where atomicity is necessary, rather than on the mechanisms that enforce it. For example, the following code segment shows an example atomic region that debits money from one account and deposits to another.

```
atomic {
  account1.debit(100);
  account2.deposit(100);
}
```

With this abstraction, the programmer identifies the operations that form a critical section, while the TM implementation determines how to run that critical section in isolation from other threads.

Typical TM implementations optimistically run transactions in parallel, assuming that the transactions won't perform conflicting memory accesses, keeping tentative updated versions. When a conflict occurs, it is detected and one or more of the conflicting transactions is then aborted, undoing the tentative updates. On the other hand if the TM system determines that a transaction does not have any conflicts, the transaction can commit its tentative changes to main memory.
There are two main variants of TM, Hardware (HTM) [3, 4, 5, 6] and Software (STM) [7, 8, 9, 10]. HTM is fast but suffers from resource constraints. STM, on the other hand, is comparatively much slower but is more flexible and offers a rich expandable set of primitives. Also there exists hybrid solutions that either try to accelerate STM [17, 18] or to virtualize HTM [19, 20, 21]. Transaction Processing Core (TxPC) [14], is a simple hardware core designated to accelerate the slow and the most frequent STM library operations which are respectively conflict detection and address translation (from global shared address to transaction local address) [11, 12, 13]. In order to do its task, TxPC maintains registry with the transaction's access set[1]. To keep the TxPC's registry current, the STM library explicitly tells TxPC about the objects it accesses for the first time by invoking instructions from the TxPC ISA extension [14]. Accessing an object (reading or updating) for the first time involves creating a local copy of the shared object that the transaction will further operate on [6] or backing up the original value of the object [5]. In STM, the procedure of copying the object in a private place and later freeing this place when transaction commits or aborts is the task of the STM library. To distinguish it from the regular memory management that is not related to the tentative transactional copies, we refer it as *transactional memory management*. In this paper we propose a further enhancement on TxPC with a custom transactional memory management. In the proposed new feature of TxPC, the operating system

1 Transaction read and write set.

allocates a special memory area – *transactional heap*, which is managed (allocated and freed) by the TxPC only. This way the task of managing the memory allocated for transaction local objects is off-loaded from STM library to the transactional core.

The rest of this paper is organized as follows: In Section 2, we introduce the design and architecture of the Transactional Processing Core. In Section 3, we describe the transactional memory management of TxPC and conclude with Section 4.

2 Transaction Processing Core

The Transaction Processing Core (TxPC) is an on-die simple core with the task to execute the slow and frequent STM operations in hardware [14]. The STM operations that TxPC handles in hardware are *eager conflict detection* and *address resolution*. The connection between the processing core and the other cores on-die is done through the front-side bus (Figure 1 (a) and (b)). To seamlessly integrate the TxPC with the other cores we add a special *Transactional Memory Register* (TMR) to each non-specialized core. This register is used for storing the results that TxPC produces. So, when a TxPC specific instruction is decoded by one of the non-specialized cores, it is forwarded to the TxPC. When the TxPC executes this instruction, it returns the result back to the associated core and this core places the result in its TMR register.

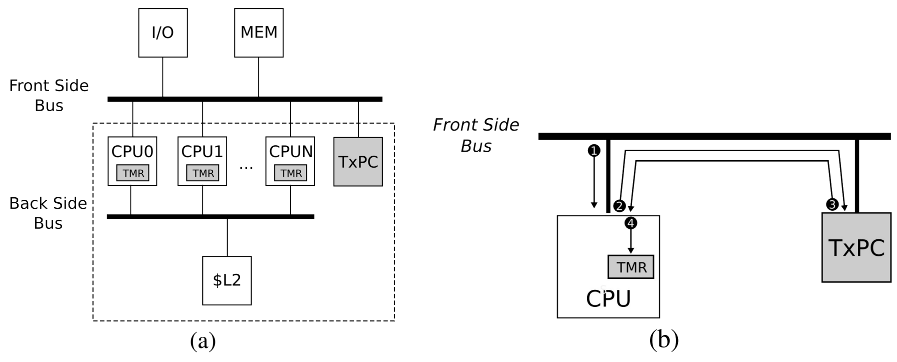

Figure 1. Connecting the Transaction Processing Core with the other cores on-die. The gray parts represent the micro architectural extensions. (a) TxPC is connected to the front-side bus. (b) When a transactional instruction is decoded (1) it is forwarded to the TxPC (2). When the result is produced by the TxPC, it is send back to the requesting core (3). Then the result is placed in TMR (4).

To accelerate conflict detection and address translation, TxPC indexes the transactional objects in a very fast low-latency content addressable memory [16]. An abstraction of the storage structure that the TxPC uses to track a transactional objects along with the meta data associated with each object is shown in Figure 2. For each object accessed in a transaction, TxPC keeps record about the address of the global shared object, the transaction that accessed it and the address of local copy of the object per each transaction that accesses it. For every object being accessed for the first time, the STM library creates a local copy of the object that is private for the transaction and supplies TxPC with the aforementioned information through the `OpenObjectForRead` and `OpenObjectForWrite` instructions. For the complete TxPC ISA extension reader is referred to the more detailed technical report of TxPC [14].

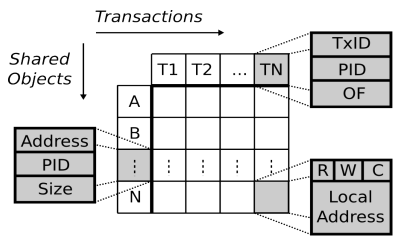

Figure 2. The Transaction Processing Core uses a matrix-like data structure to index the transactional objects. The big grayed boxes show the details about the associated group of cells. Each row contains information about a transactional objects. Transactional objects are distinguished by their virtual address and process ID. Each column contains information about the objects that a given transaction has read or written. Different transactions are distinguished by the transaction ID (TxID) and the process ID (PID). Every transaction is given an overflow (OF) bit that tells whether the read and write set of the transaction has overflowed. Each cell in the matrix tells whether a transaction has read (R) the object, written (W) the object, the object creates a conflict (C) within the transaction and the transaction-local address where the copy of the object is stored.

An important consideration in the baseline design of the Transactional Processing Core is that it has minimal overall CMP hardware re-design impact, requiring minimal changes to the other non-specialized cores and the interconnect. Thus, when the transistor budget becomes generous enough, TxPC can be easily built into the next generation heterogeneous Chip-Multiprocessors.

3 Transactional Memory Management

In every transactional memory system, execution of a transaction involves very specific memory management, necessary for creating private working copies of the objects the transaction operates on [6] or backing up the original values of the objects when they are updated in a transaction [5]. When transaction commits, all the memory allocated for the tentative updates is freed. This transactional memory allocation and the deallocation abides to a very simple pattern with two phases: (1) when transaction executes, it accumulates memory and (2) when it commits or aborts it returns the allocated memory back to the pool. In Software Transactional Memory, these memory management operations are handled by the STM library.

Because of the simplicity in managing the memory for the tentative object copies, we propose implementing the transactional memory management in the TxPC core. The benefit of this enhancement is that the STM library becomes simpler since it will be off-loaded from the task of handling auxiliary operations not related with the transaction execution. The implementation of this approach involves changes in the operating system (memory manager) and the TxPC's ISA extension (`OpenObject-ForRead` and `OpenObjectForWrite` instructions). The memory manager in the operating system is modified to manage a special heap – *transactional heap*. All the features of the transactional heap are similar to the normal heap [15] with three exceptions:

1. processes running on the system are prevented by the operating system to allocate and deallocate memory from this area and only the Transaction Processing Core can allocate and deallocate memory from the transactional heap;
2. threads that do not run any transaction cannot read or update the transactional heap; and
3. threads executing transactions cannot access other transactions' memory on the transactional heap.

To keep track of the free and allocated space in the transactional heap, TxPC uses a circular queue like data structure (Figure 3). The head of the queue points to the most recent chunk of memory that has been allocated in time and the tail to the oldest chunk of memory. The algorithm that TxPC uses to manage the transactional heap is to always move forward (advance) the head and tail pointer when memory is respectively allocated or freed. For example, Figure 3 (a) shows a transactional heap, which contains memory chunks allocated in time order by two different transactions. When transaction TX2 commits (Figure 3 (b)), its memory is marked as free (overwritten by a token symbol – zero). In this case the tail pointer does not change because the memory chunk that has been allocated first is not freed. At this point, the heap is fragmented which results to memory waste. After that (Figure 3 (c)), transaction TX3 allocates 2 memory chunks, which causes the head pointer to advance forward. In Figure 3 (d) transaction TX 1 aborts, memory allocated by this transaction is returned to the heap. In this case the oldest memory chunk allocated in time is also freed which causes the the tail to advance to the memory chunk which happens to be oldest. Now, when TX1 freed its memory, the created fragmentation in (b) has disappeared and the wasted memory gained back.

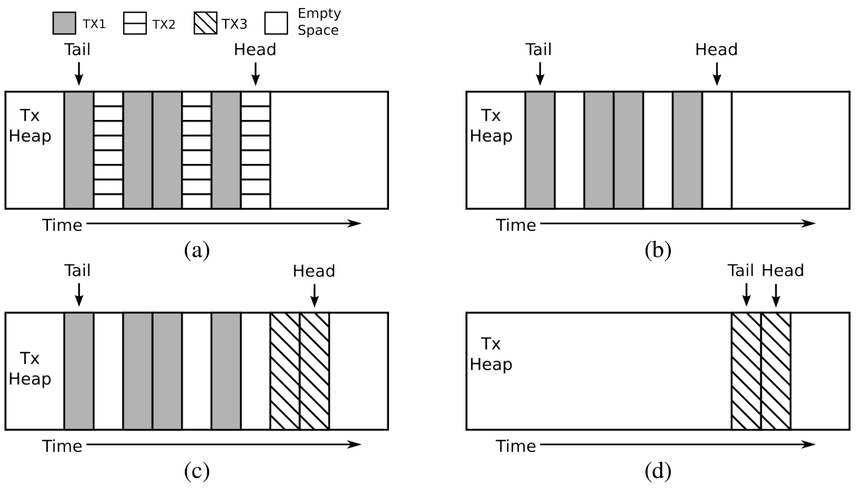

Figure 3. Transactional memory management using circular queue: (a) there are two transactions with allocated memory (TX1 and TX2); (b) TX2 commits and memory associated with TX2 is freed; (c) TX3 allocates 2 memory chunks; (d) TX1 aborts and frees its memory.

To benefit from this specialized memory management, we change the semantics of the `OpenObjectForRead` and `OpenObjectForWrite` instructions from the TxPC's ISA extension [14]. The old signature and the semantics of these instructions is:

```
- OpenObjectForRead TxID sharedObjectAddress localAddress objectSize
- OpenObjectForWrite TxID sharedObjectAddress localAddress objectSize
```

To properly index the shared objects, the STM library specifies the transaction ID, object's shared and private (local) addresses, and object's size. The TxPC registers the object into transactions read/write set and asynchronously checks for occurring conflicts. If a conflict is detected, TxPC raises an interrupt to acknowledge about that.

The new signature and semantics of these instructions is:

```
- OpenObjectForRead TxID sharedObjectAddress objectSize
- OpenObjectForWrite TxID sharedObjectAddress objectSize
```

The STM library does not any more deal with creating a private copy of the shared transactional objects. It just passes the address of the shared copy to the TxPC, then TxPC allocates memory from the transactional heap, creates a private copy of the object, registers the object into the transaction's read/write set and returns the address of the private copy by placing it into the TMR register. Again the object is asynchronously checked for any conflicts and if there is a conflict an interrupt is raised.

The introduced memory management for transactions, transfers the burden of routine memory management from the STM library into the hardware thus making the STM library simpler. Also, this approach decouples the logic of executing a transaction from the auxiliary operations for doing irrelevant bookkeeping.

4 Conclusion

In this paper we have introduced a memory management extension for the Transaction Processing Core. With the integrated memory management in TxPC, the STM library becomes functionally simpler as it is off-loaded from the task of performing irrelevant auxiliary operations. With the introduced enhancement, when an object is being accessed for the first time, the STM library provides the TxPC only with the address and size of the accessed object. Then TxPC allocates memory on a special transactional heap, creates a transaction private copy in this area and returns the address of the private copy though the TMR register. When the transaction commits or aborts the deletion of the tentative copies is again automatically done by the TxPC core.

Implementation of the transactional memory management, requires minimal changes in the Operating System and the TxPC core. The memory manager in the operating system should prevent the processes from allocating and deallocating memory from the transactional heap. To manage the heap the TxPC core should utilize a circular heap. The head of the queue points to the most recently allocated memory and the tail points to the oldest. The head and the tail pointers are always forward advanced respectively when TxPC allocates or frees memory.

References

1. P. P. Gelsinger, “ Microprocessors for the new millennium: Challenges, opportunities,and new frontiers”, Solid-State Circuits Conference, 2001. Digest of Technical Papers. ISSCC. 2001 IEEE International pp.22-25, February 2001

2. J. Larus and R. Rajwar, “Transactional Memory”, Morgan Claypool, 2006.
3. M. Herlihy and J. E. B. Moss, “Transactional Memory: Architectural Support for Lock-Free Data Structures”, 20th International Symposium on Computer Architecture, May 1993.
4. M. P. Herlihy and J. E. B. Moss, “Transactional Support for Lock-Free Data Structures”, Technical Report 92/07, Digital Cambridge Research Lab, December 1992
5. K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. “LogTM: Log-based transactional memory” In Proc. 12th Annual International Symposium on High Performance Computer Architecture (HPCA), 2006
6. L. Hammond, V. Wong , M. Chen , B. D. Carlstrom , J. D. Davis , B. Hertzberg , M. K. Prabhu , H. Wijaya , C. Kozyrakis , K. Olukotun, “Transactional Memory Coherence and Consistency”, Proceedings of the 31st annual international symposium on Computer architecture, p.102, June 2004
7. N. Shavit and D. Touitou, "Software Transactional Memory", 14th Annual ACM Symposium on Principles of Distributed Computing, August 1995.
8. B. Saha, Adl-Tabatabai , R. L. Hudson , Chi Cao Minh , Benjamin Hertzberg, “McRT-STM: a high performance software transactional memory system for a multi-core runtime”, Proceedings of the eleventh ACM SIGPLAN (PPoPP), March 2006
9. Adl-Tabatabai, A. Lewis, B.T. Menon, V.S. Murphy, R.B. Saha and Spheisman T., “Compiler and Runtime Optimizations for Efficient Software Transactional Memory”, PLDI 2006
10. M. Milovanović, O. S. Unsal, A. Cristal, S. Stipić, F. Zyulkyarov and M. Valero, “Compile time support for using Transactional Memory in C/C++ applications”, 11th Annual Workshop on the Interaction between Compilers and Computer Architecture INTERACT-11, Phoenix, Arizona, February 2007.
11. C. Perfumo, N. Sonmez, O.S. Unsal, A. Cristal, M. Valero and T. Harris, “Dissecting Transactional Executions in Haskell”, 2nd ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT), August 2007.
12. J Babba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, D. A. Wood, “Performance Pathologies in Hardware Transactional Memory”, Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA), June 2007
13. J. Chung, H. Chafi, C. Minh, A. Mc Donald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun 12th International Symposium on High Performance Computer Architecture (HPCA), Austin, Texas, USA, 11-15 February 2006.
14. F. Zyulkyarov, M. Milovanovic, O. S. Unsal, A. Cristal, E. Ayguade, M. Valero, T. Harris, “Transaction Processing Core for Accelerating Software Transactional Memory”, UPC, Departament d'Arquitectura de Computadors Technical Report UPC-DAC-RR-GEN-2007-5, August 2007
15. Donald Knuth. Fundamental Algorithms, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4. Section 2.4: Dynamic Storage Allocation, pp.435–456.
16. A. Krikelis, C. C. Weems, “Associative Processing and Processors”, IEEE Computer Science Press, 1997 ISBN 0-8186-7661-2

17. B. Saha, A.-R. Adl-Tabatabai, Q. Jacobson, “Architectural Support for Software Transactional Memory”, In 39th International Symposium in Microarchitecture, December 2006.
18. A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. Scherrer, M. F. Spear “Hardware Acceleration of Software Transactional Memory”, 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT), June 2006
19. R. Rajwar, M. Herlihy, K. Lai, “Virtualizing Transactional Memory”, 32nd Annual International Symposium on Computer Architecture (ISCA), June 2005
20. P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir and D. Nussbaum, "Hybrid Transactional Memory", 12th International Conference ovn Architectural Support for Programming Languages and Operating Systems, October 2006.
21. S. Kumar, M. Chu, C. J. Hughes, P. Kundu, A. Nguyen, “Hybrid Transactional Memory”, ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming , March 2006.