Configurable Transactional Memory

Christoforos Kachris[1,2], Chidamber Kulkarni[2]

Computer Engineering Lab[1]
Delft University of Technology
The Netherlands

Xilinx Research Labs[2]
Xilinx Inc.
San Jose, CA

Abstract

Programming efficiency of heterogeneous concurrent systems is limited by the use of lock-based synchronization mechanisms. Transactional memories can greatly improve the programming efficiency of such systems. In field-programmable computing machines, a conventional fixed transactional memory becomes inefficient use of the silicon. We propose configurable transactional memory (CTM) as a mechanism to implement application specific synchronization that utilizes the field-programmability of such devices to match with the requirements of an application. The proposed configurable transactional memory is targeted at embedded applications and is area efficient compared to conventional schemes that are implemented with cache-coherent protocols. In particular, the CTM is designed to be incorporated in to compilation and synthesis paths of either high-level languages or during system creation process using tools such as Xilinx EDK. We study the impact of deploying a CTM in a packet metering and statistics application and two micro-benchmarks as compared to a lock-based synchronization scheme. We have implemented this application in a Xilinx Virtex4 device and found that the CTM was 0-73% better than a fine-grained lock-based scheme.

Introduction

Programmable platforms are becoming increasingly critical to addressing the ever decreasing time-to-market, pushing designers away from risk time-consuming ASIC design process. An example of platform FPGA is the Xilinx Virtex-4 family of programmable logic devices that include hard IP cores such as the PowerPC microprocessor, RocketIO serial transceivers, digital signal processing blocks, and true dual-ported on-chip embedded block RAM (BRAM) memories. Current state-of-the-art design flow for platform FPGAs in the networking domain involves starting with HDLs, which are not suited for describing systems. Thus a productivity gap is widening with every new process generation. To close this gap, we need methods and tools that can transform higher-level concurrent semantics in to HDL (at register-transfer level) implementations. A primary goal of this work is to enable a higher abstraction for mapping applications in networking domain to platform FPGAs.

Programming concurrent heterogeneous platforms remains an art due to many challenges. One such challenge is the synchronization among concurrent threads or processes. In this paper we focus on shared memory abstraction for implementing synchronizations. In particular, we investigate efficient mapping of synchronization mechanisms on platform FPGAs for networking applications. Current shared memory abstractions based on locks and mutual exclusions are difficult to use, scale, and generally result in a tedious and error-prone design process.

Typical FPGA based system implementations comprise one or more soft processors and custom state machines in logic. In such a system, the soft processor(s) perform control and supervisory functions and the state machines implement custom functions. The use of multiple soft processors combined with dedicated logic to implement a concurrent system has created the need for simpler synchronization mechanisms. In the multi-processor context, the synchronization of the processors is based on locks that have significant performance limitations and increased programming complexity. One of the problems of using lock-based synchronization is to find the right granularity for each application. Fine-grained locks have better performance but increased programming complexity and are error prone. On the other hand, the programming complexity can be reduced using coarse-grained locks (one lock for each data structure) but with a major impact on the performance of the system. A much simpler abstraction for synchronization of multi-processors is the use of transactions. Transactions are programming operations that are executed atomically as seen by the processors; hence they provide the consistency of the shared memory. Using transactions, the programming of a parallel platform is much simpler. There is no need to lock the variables before modifying them or to unlock after the update. Furthermore, the order of the locks is very important to avoid deadlocks

and livelocks. Hence transactional schemes can be used to develop more robust code.

Many transactional memory schemes with hardware support have been proposed but most of them target high performance systems with cache coherence protocols. In many embedded applications, such as network processing, the target platforms incorporate devices with multiple simple RISC processor or micro-engines (e.g. network processors and FPGAs [10]). In this case the programming of the platform using locks is error prone and the adoption of complex transactional schemes with cache coherency adds significant area overhead. Hence, a simple transactional memory scheme is required. In this paper, we present a configurable transactional memory that is light weight and integrates in to existing tool chains and soft processors easily. Furthermore, in applications such as network processing, where the number of data conflicts and lock contention is low, the proposed scheme can also be used to improve significantly the performance of the system.

The main contributions of the proposed scheme are:

- A configurable transactional memory for FPGAs that fits to the application
- Small area overhead, thus ideal for embedded applications targeting FPGAs
- No need for cache coherence protocol
- No need for changes in the ISA, or the compiler
- Major speedup over locks in applications with low presence of data conflicts (e.g. network processing)

Section 2 presents a discussion of the related work in the area of transactional memories with hardware support. Section 3 presents the overall system architecture that we have used in this work and section 4 shows the performance evaluation of the proposed scheme for a network processing application. Finally, in section 4 we discuss our conclusions and present scope for future work.

Related work

Although lock-based synchronization schemes are most widely used method of synchronization of multi-threaded and multi-processors systems, the performance limitations and the increased programming complexity has created the need for research on hardware assisted transactional memories. The majority of the proposed schemes are implemented by modifying or upgrading standard multi-processor cache coherence protocols. One of the first implementations of transactional memories is presented in [1]. In this paper a transactional memory is implemented by modifying the cache coherence protocol. The main idea of the proposed architecture is that the same protocol that is used for cache coherency can be used for transactional conflict management with no extra cost. The proposed architecture introduces new instructions such as *load-transactional*, *store-transactional*, *commit* and *abort* and can be implemented for both bus-based (snoopy cache) and network-based (directory) architectures. In the proposed implementation, each processor maintains two caches: a regular cache for regular operations and a transactional cache for transactional operations. In the second cache each line contains information about the transaction such as invalid, valid, dirty or reserved.

A transactional memory with the same semantics has been proposed in [2],[3]. In these papers a unified model is proposed in which the conventional coherence and consistency technique has been completely replaced instead of being duplicated for regular and transactional accesses. In these works, only one cache per processor is used to store the data and the transaction information. Furthermore, a *write-buffer* is used to store the data that have been modified. In case that a commit is requested, all of the data of the *write-buffer* are transferred to the main memory without being interrupted and the addresses are also broadcasted to the other processor to preserve the consistency. The main drawback of the proposed scheme is that it increases the communication bandwidth of the multi-processor platform.

In [4][5][6], a new scheme is introduced called Transactional Lock Removal (TLR). The proposed scheme uses the lock semantics to identify and create the boundaries of the transactions. The main feature of the proposed scheme is that it tries to serialize the code only on conflicts while using speculative lock elision for the remaining part of the critical code. When there is a conflict it uses timestamps to eliminate livelocks and to serialize the transactions. The main advantage of the proposed scheme is that it does not require either instruction set changes or compiler changes; only the transactional hardware unit must be incorporated to the current processors to support transactional memories. The main bottleneck of the transactional memories is that they are often bounded by system constraints such as cache size. In [7], a new scheme is introduced to support unbounded transactional memories. The proposed scheme explains how to use the main memory to store the data of the transactional cache, hence to eliminate the bounding of the transactional memories by sacrificing the performance of the system.

Until now the FPGAs have been used only for prototyping of the proposed transactional schemes ([8], [9]). But as the FPGAs have evolved from simple logic devices to complete SoC platforms incorporating multiple processors (either hard or soft core) there is the need for efficient and robust concurrent programming abstraction without significant area overhead. Our scheme is mainly targeted at embedded applications in FPGAs and can be used on multi- processors with or without cache. Hence, the proposed scheme does not depend on cache coherence

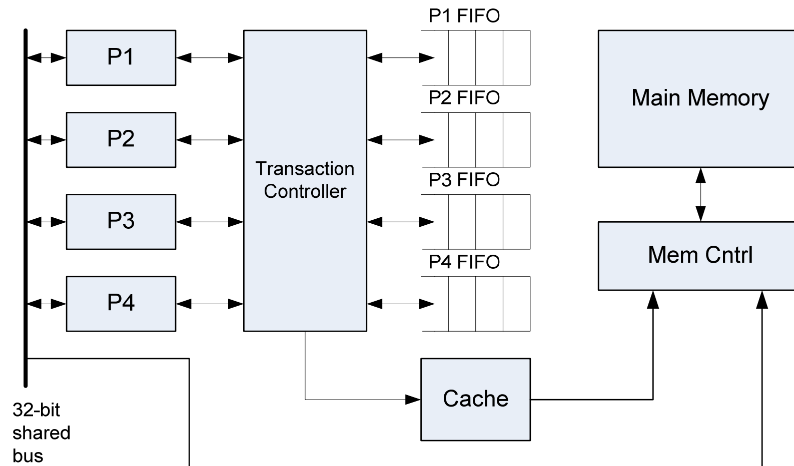

Figure 1. Top level architecture

protocols. Furthermore, the proposed scheme is targeting reconfigurable platforms hence it can be reconfigured to meet the application demands. The size of the cache and the buffers that are used for the memory consistency can be extracted by profiling or static analysis of the application code. The main feature of the proposed architecture is that it is centralized. It uses only one cache to store the shared data for all of the processors. This feature keeps the communication bandwidth of the platform low since the data of each processor does not have to be broadcasted to the other processors. The main limitation of this feature is that it is not scalable to a large number of processors. We believe that the proposed scheme can be used for platforms incorporating up to 8 processors.

System Architecture

The top level architecture of the proposed scheme is shown in Figure 1. This figure depicts an example with 4 processors that have access to a shared memory. The processors can access the shared memory either conventionally, using the shared bus, or transactional, using a direct interface with the transaction controller. The access to the shared memory using the bus can be used when the processor accesses the private sections of the memory. In case that the processors want to access shared portion of the memory, the transaction controller can be used. The transactional controller uses a small memory cache to store the shared data and small FIFOs (one for each processor) in which the addresses, of each shared variable that each processor has accessed, are stored. Note that we have multiple ports on the memory controller as opposed to a bus-based single port memory controller found in conventional processor based systems.

The organization of the cache is shown in Figure 2. For each entry there are 8 bits that are used to store what kind of access each processor had (one Read and one Write flag for each processor). When a processor issues a load, if the data is not stored in cache then the transaction controller accesses the main memory, stores the data to the cache, returns the value to the processor and asserts the corresponding Read flag in the cache. When the processor issues a store, the transaction controller checks if there are any collisions. A collision occurs if any of the other processors R or W flags is already asserted. If there is no collision the data are stored to the cache and the corresponding W flag is asserted. When a processor issue a commit then the transaction controller retrieves all the accessed addresses from the corresponding FIFO and copy the entries from the cache to the main memory. In case that an entry has been accessed only by this processor then the whole entry is cleared. In case that the entry has been also accessed by other processors then only the R and W flags of this processor are cleared and the entry remains valid.

In case there is a collision the same procedure is used but in this case the data are not stored back to the memory. All the entries of the processor that created the collision are removed from the cache and a collision signal is sent to the processor when it tries to commit. Thus, the processor does not have to poll or to be interrupted in case that there is a collision. The drawback of this scheme is that in case there is a collision the processor will keep executing the instructions and it will be notified only when it tries to commit. But if the number of instructions of the transaction is small (which is typical in network processing applications) the performance of this scheme is better than using polling or interrupts.

The current scheme uses the Cuckoo hashing to implement the hash collisions. The Cuckoo hashing [12] uses two hash functions instead of only one. When a new entry is inserted then it is stored in the location of the first hash key. If the entry is occupied the old entry is moved to

its second hash address and the procedure is repeated until an empty slot is found. This algorithm provides constant lookup time O(1) (lookup requires just inspection of two locations in the hash table) while the insert time depend on the cache size (O(n)). In case that the procedure enters an infinite loop the hash table is rebuild. But even in this case it can be shown that the performance of the system in the insertion is efficient as long as the number of keys is kept below half of the capacity of the hash table.

Tag	Data	R1	R2	R3	R4	W1	W2	W3	W4
0x1000	0x1	✓							
0x2000	0x2		✓	✓					
0x3000	0x3	✓	✓		✓	✓			

Figure 2. Cache organization

The main advantage of the proposed scheme is that the organization can be configured to meet the application requirements. Table 1 shows the parameters of the design that can be easily changed to meet the target application. The number of FIFOs depends on the number of the processors that are used to access the shared memory. The depth of the FIFOs depends on the application e.g. the maximum number of shared variables in the transactions. The size of the cache depends not only on the number of processor and the number of shared variables but also on the application requirements (e.g. collision probability). In summary, CTM is based on two basic hypothesis; first, for applications that require small number of operations per transaction (less than 10 instructions), CTM will have a small area overhead but will have higher performance and significantly lower design effort; second, for applications that perform lot of operations in each transaction (greater than 20-25 instructions), the lock based scheme might have marginally better performance and lower area overhead but the design effort required will be significantly larger than that based on CTM.

Table 1. Parameters

Parameter	Dependency
Number of FIFOs	Depends on the # of processors
Depth of FIFOs	Depends on the application (shared variables)
Cache Size	Depend on the # of processors, the application and the constraints

Furthermore, the proposed scheme can also be used to provide ordering between the transactions to maintain a specific program execution similar to the scheme used in [2]. The programmer can assign an identification number (common to a group of transactions) and a sequence number to each transaction. These two numbers can be forwarded to the transactional control with the commit command request (32-bit wide). The transactional controller checks the identification and the sequence number of the transaction to ensure consistent execution. If the sequence number for a specific transaction group is higher than the current sequence number then the transaction has to rollback. Otherwise, the transaction is committed. Figure 3 shows an example in which three transactions with common identification number have to be synchronized. The transaction with ID X, and Y do not have to be synchronized while the transaction with ID Z has to be synchronized.

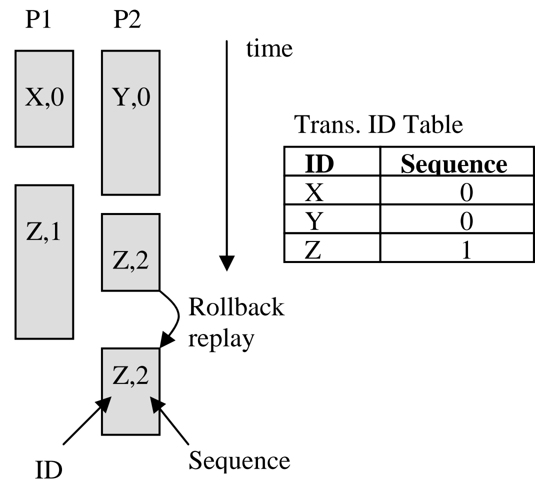

Figure 3. Transaction ordering

A small table is used in the configurable transactional controller to check if the sequence number of the current transaction is smaller than the last one stored in the table. In that case the transaction is rolled back and is executed again as it is shown for the transaction that is executed in processor 2.

We will now discuss the experimental set up used in our experiments with CTM on a packet metering application and two micro-benchmarks from related work and discuss the results of our experiments.

Implementation and System Evaluation

The system has been implemented in a Xilinx Virtex4 LX FPGA platform. We use the soft-core 32-bit Microblaze processors [11]. Each Microblaze can be configured with up to 8 32-bit dedicated interfaces called Fast Simplex Link (FSL). This interface has been used for the

Network Metering Application (Update of two counters)

```
do {
    x_counter = trans_load(x_addr);
    y_counter = trans_load(y_addr);
    x_counter++;
    y_counter++;
    trans_store(x_addr, x_counter);
    trans_store(y_addr, y_counter);

    temp = trans_commit();
} while (temp!=0);
```

Figure 4a. Transactional scheme

```
while (temp = lock_req(x_addr));
while (temp = lock_req(y_addr));

x_counter = lock_load(x_addr);
y_counter = lock_load(y_addr);
x_counter++;
y_counter++;
lock_store(x_addr, x_counter);
lock_store(y_addr, y_counter);

unlock_req(x_addr);
unlock_req(y_addr);
```

Figure 4b. Lock scheme

communication of the processor with the transaction controller. Each time a processor issues a transactional load, it sends the address to the FSL link and waits until it receives the value of the data. When the processor issues a store, the address and the data are sent to the FSL link. Furthermore, the FSL interface uses a dedicated signal for control information. When a processor tries to commit a transaction this control bit is asserted. The transaction controller uses the control signals to convey either acceptance or rejection of a particular transaction.

The key advantage of the above scheme is that at design time, based on profiling the code and static analysis the user can configure a particular transactional memory configuration. In our experiments this process was done manually. However this can be easily automated and is part of our ongoing research. In this study, we used a 64 entries cache, and the FIFOs are 32-bit wide with 16 entries deep. The main memory is 32x16K internal RAM (BRAM). Note that for larger memory size requirements we can use an external memory by connecting the transaction controller to a DRAM memory controller. In such a scenario, a multi-port memory controller becomes even more critical to provide efficient data path from the memory controller to the processor sub-system.

To evaluate the proposed scheme an application was developed. Since there is a lack of widely accepted benchmarks for transactional memories, we chose a networking application that is widely used for per-flow metering and statistics in network processing equipments (e.g. edge routers). Similar applications are used in programs for network billing and accounting, performance analysis and security management such as the Argus open source framework [13]. In this application, each time a packet arrives in the system, the header is extracted and forwarded to the next available processor. Each processor extracts some information from the header (Source IP, TCP Port, etc.) and uses this information to update specific counters. These counters can be used for billing and Quality of Service purposes. To evaluate the efficiency of the proposed scheme, a similar scheme has been developed in which the transaction controller has been replaced by a fine-grain lock controller. The lock controller does not use the FIFOs hence it consumes much less area. Each processor must first lock the addresses of the variables that are going to be updated, then update the counters and then unlock the addresses. Furthermore, the cache has been modified and instead of using R and W bits, only one bit is used for each processor to indicate which processor has locked the address. Although the lock controller consumes less area, the programmability of this scheme is much harder and error prone than using the transactions. The programmer has to perform detailed analysis of their implementation to avoid deadlocks or livelocks.

Figure 4 shows an example source code for MicroBlaze implementation of the transactional and the lock schemes. Here the transaction begins at the first transactional instruction (trans_load()) and ends at the last commit operation. This approach is similar to conventional transactional begin and end semantics. However in our approach these key words are not explicitly required. As shown, in the case of the transactional scheme, the programmer does not have to worry about issuing lock requests but instead issues transactional loads and stores. Note the difference compared to other efforts where in there is burden on the compiler to take code that are marked with transactional sections (using transaction begin and end constructs) and generate appropriate code. In our approach we used a library based scheme to work around the requirement of modifying the MicroBlaze compiler. Similarly, for the lock-based scheme, the user now has to issue lock requests and unlock requests to particular sections of the memory. Thus programming with transactional semantics entails less burden on the programmer, who is not required to worry about acquiring and releasing various sections of the memory, a function that is now performed by the hardware.

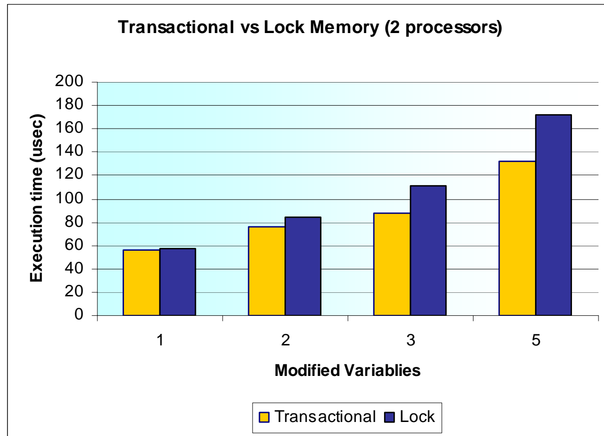

Figure 5a. Comparison for 2 processors

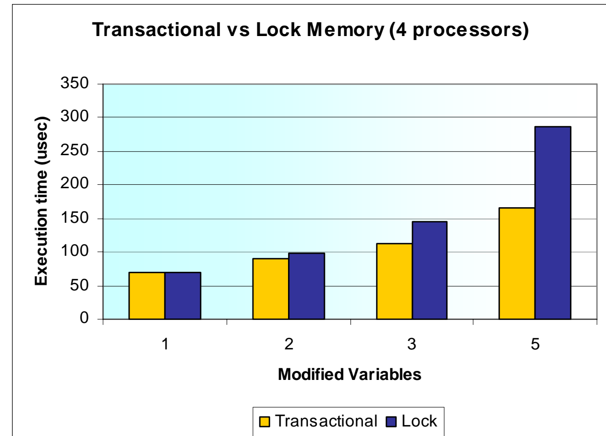

Figure 5b. Comparison for 4 processors

Figure 5 shows the performance evaluation of the transactional and lock scheme for 2 and for 4 processors depending on the number of modified shared variables for the packet metering application. We used EDK8.2i in our work. We constructed the configurable transactional memory with various parameters using VHDL and manually instantiated the different parameterized versions of the CTM in our experiments. We performed RTL simulation in EDK using ModelSim6.0 to obtain the cycle accurate data. We generated our timing information using Xilinx ISE 8.2i after place and route.

Figures 5a and 5b show the execution time to process 50 network packets (each processor). In the case of two processors the speedup of the transactional scheme is from marginal (2% using only 1 shared variable) to 30% using 5 shared variables. In the case of four processors the speedup is from 0% using one shared counter, to 73% using five shared counters. When five shared counters are used, a lot of time is wasted in the instructions that are used to lock and unlock the variables. On the other hand, the transactional scheme uses only one instruction to commit all of the shared variables, hence the execution time is much lower. Thus, the proposed scheme not only introduces a much simpler programming model for multiprocessor platform but can also be used to improve the execution time of such applications. In the current application, the range of the counters that are updated is very wide (1024 different counters that represent 1024 different network flows) hence the probability of collision between 2 processors is very small. In case that the shared variables refer to the same address the probability of collision will be higher thus the execution time will increase.

Figure 6 shows how the performance of the transactional memory depends on the number of rollbacks. The figure shows the time to process 200 network packets using four processors using the transactional and the lock scheme. The range of counters that had to be updated for each packet was artificially changed to create data dependencies. As it is shown the processing time is from 20% to 30% smaller using the transactional memory than the lock-based scheme even when the percentage of rollbacks (percentage of data conflicts) is 50% (one rollback for every commit transaction request). Note that the function that was used to update the counters was consuming almost one third of the total time to process a packet. Hence, the probability of collision has a smaller impact on the total execution time.

To further illustrate how the memory conflicts affect the performance of the system we implemented the micro-benchmarks from [5]. These benchmarks represent two extreme cases for the update of the counters. In the first benchmarks all of the processors update the same counter (same address) creating a high number of conflicts. In the second benchmark all of the processors update a different counter which means that there are no conflicts at all. The performance of the system for these two cases is depicted in Figure 7 for a system with four processors. As shown in the case of the single counter (high conflicts) the processing time to update 400 counts (100 for each

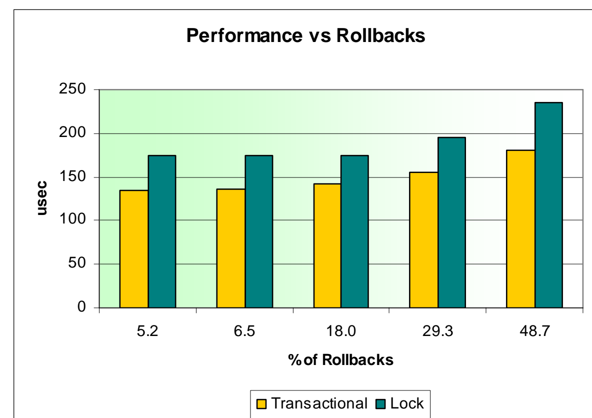

Figure 6. Performance over Rollbacks

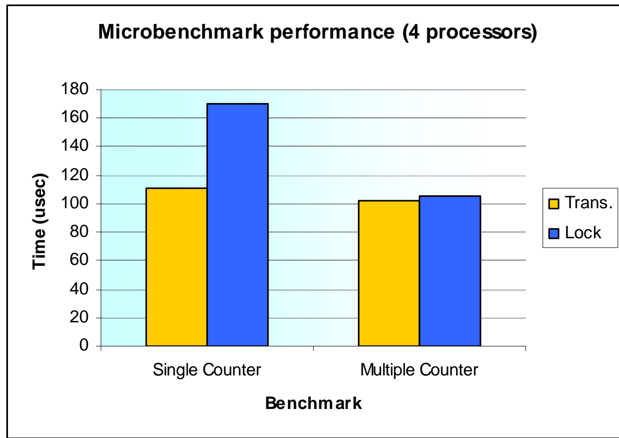

Figure 7. Microbenchmark results for 4 processors

processor) is almost 40% less than the processing time using the lock mechanism. The transactional scheme only rollbacks when there is a conflict, while in the case of the lock mechanism the processor is waiting for the other processors to unlock the specific memory. On the other hand, when there are no conflicts, then the performance of system is independent of the scheme.

The area and the maximum clock frequency of the transactional controller and the lock controller for several processors are shown in Table 2 and Table 3 respectively. As it is shown the current implementation of the transactional controller can support up to 8 processors using the same clock frequency as the processors and the bus (100MHz). The area overhead is quite small (less than 530 slices for 8 processors, which requires a total area of about 8000 slices). This is clearly an important benefit of the configurability of our scheme that tailors the transactional controller to the application needs. The number of occupied internal Block RAMs (BRAMs) that are used for the FIFOs and the cache depends on the number of processors and the application. The depth of the FIFOs and the cache is proportional to the number of shared variables among the processors. These are parameters that can be configured or customized for each application. As it is shown the transactional controller is larger than the lock controller and the maximum clock frequency is less than the lock controller. But the reduced program execution and the easy of programming that the transactional memory offers compensates for the area overhead.

Table 2. Area comparison

Processors	Area (slices)	
	Transact.	Lock
2	277	137
4	370	247
8	530	353

Table 3. Clock frequency comparison

Processors	Clock Freq. (MHz)	
	Transact.	Lock
2	151	182
4	137	156
8	107	149

Conclusions and future work

The ability to configure a transactional memory to the needs of a particular application is an important aspect that is explored in this paper. The proposed configurable transactional memory helps in increasing the programming efficiency of a multi-processor system on FPGA. In addition, for some applications it can also help in increasing the system performance, where the synchronization among processes is the performance

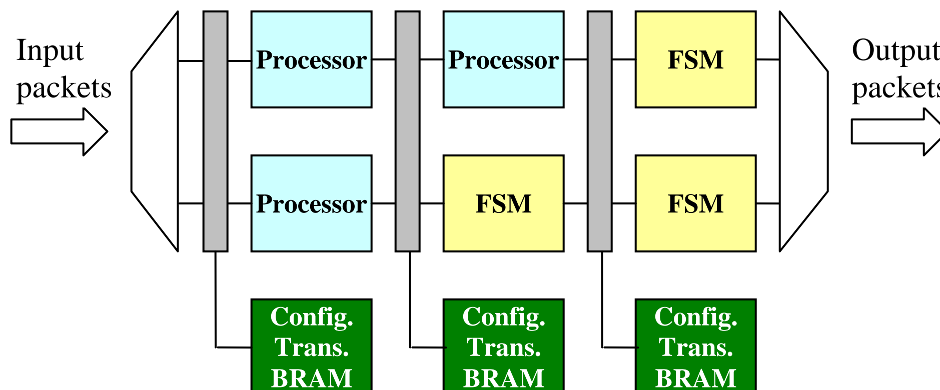

Figure 5. Configurable Transactional Memory in multi-processing

bottleneck. Our experiments have shown that for the packet metering application we could increase the system performance compared to a lock-based implementation for a small increase in the total area of the implementation.

Finally, the proposed scheme could be extended to support not only processors but also hardware accelerators modules. Hence, in this case the configurable transactional controller could be used to provide consistency between the processors and the hardware acceleration modules as it is shown in Figure 5. This figure shows a multi-processor platform in which the processing of the packet is performed in several stages that include both processors and hardware acceleration modules and consistency needs to be preserved. In addition, we have not explored leveraging compiler optimizations based on life-time analysis of data structures as well as investigation in to the impact of CTM on power consumption compared to a lock-based scheme. These are topics of our future research.

References

[1] M. Herlihy, J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures", Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, CA, USA, May 1993

[2] L. Hammond, et al., "Transactional Coherence and Consistency: Simplifying Parallel Hardware and Software", IEEE Micro, November-December 2004

[3] Austen McDonald, et al., "Characterization of TCC on Chip-Multiprocessors", Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, 2005

[4] R. Rajwar, J. R. Goodman, "Transactional Execution: Towards Reliable, High-Performance Multithreading", IEEE Micro, November-December, 2003

[5] R. Rajwar, J. R. Goodman, "Transactional Lock-Free Execution of Lock-Based Programs", Proc. Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS 02), ACM Press, 2002, pp. 5-17

[6] R. Rajwar, J. R. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution", Proceedings of the 34th International Symp. Microarchitecture (MICRO-34), IEEE CS Press, 2001, pp. 294-305

[7] Sean Lie, "Hardware Support for Unbounded Transactional Memory", Masters thesis, Massachusetts Institute of Technology, May 2004

[8] S. Grinberg, S. Weiss, "Investigation of Transactional Memory using FPGAs", Proc. 2nd Workshop on Architecture Research using FPGA Platforms, Austin, Texas, USA, February 2006

[9] N. Njoroge et al., "Building and Using the ATLAS Transactional Memory System", Proc. 2nd Workshop on Architecture Research using FPGA Platforms, Austin, Texas, USA, February 2006

[10] B. Moyer, "Packet Subsystem on a Chip", Xilinx Embedded Magazine, March 2006

[11] "MicroBlaze Processor Reference Guide", Datasheet, Xilinx, June 2006

[12] R. Pagh, F. F. Rodler, "Cuckoo Hashing", Proceedings of ESA 2001, Lecture Notes in Computer Science, vol. 2161, 2001

[13] Qosient Argus, Auditing Network Activity, http://www.qosient.com/argus