Investigation of Transactional Memory Using FPGAs

Simon Grinberg Shlomo Weiss
School of Electrical Engineering, Tel Aviv University, Tel Aviv 69978, ISRAEL
simongr1@post.tau.ac.il weiss@eng.tau.ac.il

ABSTRACT

The following outlines an effort to speedup the evaluation of a transactional memory system without loosing accuracy. Instead of using the traditional software simulation techniques we build our system within a large FPGA device. The system elements are a mix of commercially available IP cores and our own design. Together with appropriate runtime monitoring this approach yields a powerful substitute to simulation.

1. INTRODUCTION

Simulation has traditionally been the method for evaluation of new ideas and system architectures in the field of multiprocessors systems. The cost, complexity, and the time consuming effort of implementing a real system has driven researchers to revert to simulation. However a truly accurate simulation of such systems may take weeks per a single scenario, so researchers had to reduce considerably the accuracy of their simulations. Inaccurate simulations may result because of simplified CPU models or fixed delay networks. We address this in a different way, using Altera's largest StratixII FPGA device in order to implement a 16 processor system. This enables us after the initial setup of the system to run many scenarios in considerably shorter time then simulation. The setup time of such system is not much longer then the time to setup a simulation for such a system.

2. METHOD

The processors used are Altera's NiosII processors [5] which are simple and relatively small soft core RISC processors, each consuming less then one percent of the selected device resources. The use of these processors further decreases the time to set up the system since now the entire system except for the system memory can be implemented inside the programmable device. Using this off the shelf microprocessor saves considerable time since the design of a new processor is time consuming. It is not done even in simulations which usually adjust an existing processor model to meet the needs. We do the same here utilizing the custom instruction feature of the NiosII soft core. Since the system resides in a single FPGA we don't have to design and produce a board of our own, instead we use an existing board which features a suitable FPGA device with sufficient attached memory.

We chose Altera's DSP evaluation board [4] which accommodates a StratixII EP2S180 device with a total of 32MByte of SDRAM attached to it. We design the system using Altera's SOPC [7] builder to further simplify the system generation process. The SOPC Builder is a system-design tool which converts a system-description file into HDL files that can be synthesized into an FPGA. This tool supports both Altera's intellectual property system elements and user modules written in an HDL language. This enables us to define the building blocks of our system and then easily create multiple systems in case that a parameterized system is not sufficient.

3. TRANSACTIONAL MEMORY

We investigate the speculative transactional memory access approach [1,2,3] which addresses both consistency and coherency issues of such a system. This approach relies on statistical analysis of many programs, which shows that in many cases the synchronization in shared memory multiprocessors is not actually needed. Some reasons for this are:

• The conservative approach that programmers take when inserting synchronization elements. This approach introduces many unnecessary synchronization points assuming that any access to shared memory needs to be protected.

• The dynamic behavior of the program at run time which may or may not cause data contention at a particular point of time.

• Programmers must often make trade offs between the complexity and the time consuming effort of writing fine grained programs and the poor performance of coarse grained programs. Fine grained programs introduce synchronization elements exactly at the point they are needed and remove them as soon as possible while coarse grained programs use synchronization elements for large blocks. Often the coarse grained approach wins, which means other processes may wait idle until the entire block ends execution.

The transactional shared memory approach basically suggests chopping the program into atomic chunks of code called transactions. Transactions pass data to other transactions only through the shared memory. Each transaction is performed locally and writes are stored in an alternative storage to the shared memory whether it is a local write buffer [3], the local cache [2], or a shadow block in the directory table (like in our work), until the transaction execution ends. When execution ends, the transaction commits its data to the main memory. At this point any uncommitted transaction in the system, which used data that was modified by the recently committed transaction, must roll back and restart. Consistency is maintained by the semantics of the transactions, and coherency is guaranteed by the violation mechanism with roll back which causes re-fetch of the modified data.

Since the transactional memory itself is not the purpose of this document we will not go into further detailed description of the transactional memory system but rather concentrate on the proposed system and the methods we use to implement and evaluate a transactional memory system using FPGAs.

4. USING FPGAs TO IMPLEMENT TRANSACTIONAL MEMORY

Figure 1 illustrates a multiprocessor system that supports transactions. The target multiprocessor is a DSM (distributed shared memory) system in which each node is an SMP (symmetric multiprocessor). In order to achieve that, we compose our system as clusters of microprocessors representing an SMP, each with an attached shared memory. These clusters are interconnected to each other. Each of the processors in the system can access both its local shared memory or the memory of the other clusters paying the penalty of an inter-cluster programmed delay emulating the distributed system. This penalty may be reduced to none thus making the equivalent of one large SMP composed of all the processors in the system.

4.1 System Switch Fabric and Local Switch Fabric

The System Switch Fabric is the Altera's Avalon switch fabric [6] which is instantiated by the SOPC builder. This fabric comprises of address decoders, data-path multiplexers, and built in arbiters. It supports a variety of bus sizes, peripherals, clock-domain crossers, and most importantly variable-latency transactions and burst transactions. Since the Avalon interconnecting fabric supports variable latency, we utilize this feature by adding a delay element before each memory module and the fabric, thus emulating varying delay network typical to DSM. This delay is determined on a per node (processor) basis enabling us to use in fact a single fabric which can be programmed to divide the system to clusters. Zero added delay between a processor and a memory element means that they belong to the same cluster (local switch fabric) while any other delay represents the DSM delay (system switch fabric). This scheme enables any configuration of clusters from one processor per cluster and up to all processors on the same cluster (one SMP). The delay between clusters may be fixed or pseudo random depending on the chosen scenario.

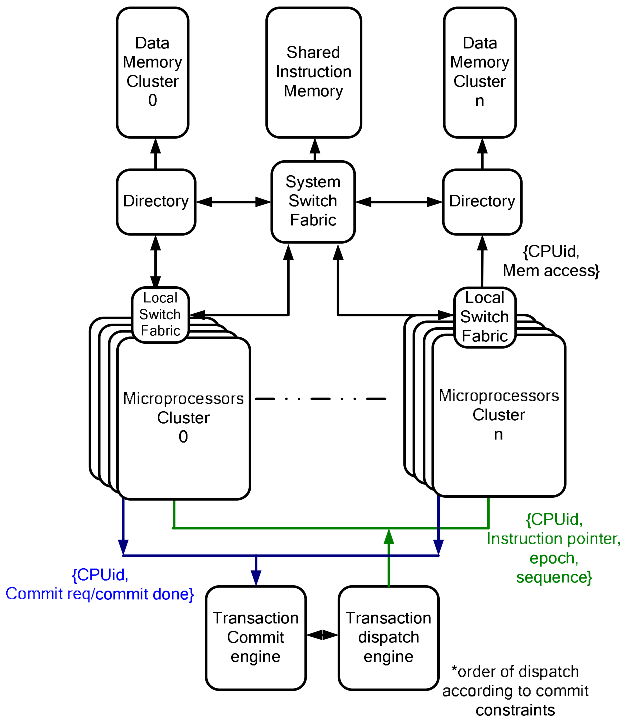
Data Memory Cluster 0
Shared Instruction Memory
Data Memory Cluster n
Directory
System Switch Fabric
Directory
{CPUid, Mem access}
Local Switch Fabric
Local Switch Fabric
Microprocessors Cluster 0
Microprocessors Cluster n
{CPUid, Instruction pointer, epoch, sequence}
{CPUid, Commit req/commit done}
Transaction Commit engine
Transaction dispatch engine
*order of dispatch according to commit constraints

Figure 1. Distributed shared memory system on an FPGA. Each microprocessor cluster represents a symmetric multiprocessor.

4.2 Microprocessor

As already noted, the processor used is Altera's soft core processor, NiosII. The version we use is the five pipeline stages processor. The interface of this processor to the system in our implementation consists of the data bus, the instruction bus, and the interrupt line. The data bus is used for the transaction data and also for accessing dedicated memory mapped locations over the switch fabric. These memory locations are used to dispatch transactions to the processors, pass information between the processors and the transaction control engines, and arbitrate for commit grant. The instruction bus is used for transaction instructions fetches. The interrupt line is used to roll back the current transaction. This interrupt line is connected to the Transaction Rollback Controller.

4.3 System Memory

In a real system, each cluster has an attached local memory with a Directory placed between the memory and the system fabric. However the evaluation board we use has a single 32Mbytes SDRAM which can be controlled by a single memory controller located inside the FPGA. To solve this issue we divide the memory into logical data memory units each serving one cluster. This partition is achieved by adding an adapting layer between the SDRAM controller and the rest of the logic. This layer has one slave port per directory such that each directory is attached to this layer as if it were connected directly to the SDRAM controller. In addition the connecting layer has another slave port emulating an additional memory holding the program code and serving all processors. The cost of this adapter layer is slower memory access since it actually shares the SDRAM bandwidth between all the directories.

4.4 Directory

The Directory placed between each logical data unit and the fabric is designed to support the transactional memory system. Its key features are:

• It keeps track of transactional accesses to each block. When a transaction from a specific processor has committed, the directory updates the rest of the processors about this event.

• It holds shadow blocks (copies of dirty blocks that a processor had to evict from the cache before the end of the transaction). When the processor needs the evicted block again the shadow block will be returned instead of the main memory copy of the block. Other processors always access the valid, committed copy of the block.

The second feature enables us to share a pool of relatively low cost memory block buffers residing in external memory dedicated for the directory among all the processors in the system. This enlarges the size of the maximum transaction available beyond the limited size of the local cache of each processor. In our limited board resources we have to use part of the logical memory allocated for each cluster for this use. When the pool of shadow blocks buffers is full then the directory requests the identity of the processor which is least likely to commit, from the commit engine and orders it to rollback. This frees the shadow resources acquired by this processor and enables the more likely to commit processors to advance. If required this process repeats itself until there is only a single processor which owns shadow block buffers in that directory. This sets an upper limit to the size of a single transaction in the system. It cannot be larger then the local cache size plus the size of the smallest directory shadow blocks pool in the system.

4.5 Transaction Rollback Controller

The rollback of a violated transaction is done using the interrupt lines to the cores. Each processor has an associated transaction rollback controller. These are mapped on the system switch fabric so they can be accessed by any of the directories. The directory informs the rollback controller that a transaction violation has occurred by writing a violation message to its location on the system fabric. The Rollback Controller decides whether a rollback is required and if so it sends an interrupt signal to the processor. This makes the rollback process itself straightforward. In the rollback interrupt routine the processor invalidates the entire cache, loads again the transaction start address to the instruction pointer and returns to transaction execution from the beginning. A possible performance enhancement may be to flush from the cache only the blocks which came from the directory that initiated the rollback plus the dirty blocks, instead of flushing the entire cache.

4.6 Instructions cache

The instruction cache is the standard instruction cache which comes with the NiosII processor. This cache is implemented as a simple one way cache.

4.7 Data cache

The standard NiosII data cache is replaced with our own data cache. We achieve that by generating the NiosII processor without a data cache but only with an Avalon master port for the data bus. Our cache is inserted between the fabric and the processor, connected as an Avalon slave to the processor's data bus and as an Avalon master to the system switch fabric. This cache is customized to support the Directory Based Transactional Memory System including support for the shadow write back of a block whenever eviction is necessary. Until a transaction commits all the transaction dirty blocks are kept in the cache or in a shadow block in the directory. When a transaction commits, all the dirty blocks in the cache are written to the main memory and if the directory holds shadow blocks for this cache they are also committed to the main memory.

Since our system may have up to sixteen processors, and in view of the limited internal memory resources of the FPGA, we implement a small one-way cache. The actual size is determined according to the number of processors used.

4.8 Transaction Dispatch Engine

The Transaction Dispatch engine is the heart of our system. It holds the program outline in the form of transactions. Without going into too many details each transaction is represented by its location in the code and its commit order. This program outline is prepared by a modified GNU compiler.

Each processor in the system is waiting in a loop over a local memory location which can be also accessed from the switch fabric. The Transaction Dispatch Engine initiates a new transaction by writing a new program pointer value to this local memory. The communication between the processors and the engines is done via the fabric to emulate a real system in which the Transaction Engines can reside only in one cluster or in a totally separated location. A programmable delay is added to any access between each processor and the engines to emulate this.

4.9 Transaction Commit Engine

The transaction commit engine is responsible for commit grants for the entire system. It only holds information about the transaction currently running on each of the processors and the relations between them, and grants commit according to that. Only one processor in the system is allowed to commit at any given moment. No concurrent commits allowed. When a transaction commit is done the

Commit Engine signals the Dispatch engine that the processor just committed is free to accept new transaction.

5. PERFORMANCE EVALUATION

In our evaluation board we use an attached one megabyte SRAM to collect the required data. This data is read at the end of the program execution in debug mode via the JTAG debug connector.

5.1 Performance calibration statistics

The transaction dispatch engine allocates a table in the SRAM to hold these statistics. The Transaction ID serves as a pointer to its entry in this table.

• Rollback due to data contention counters. When executing a transaction, each processor counts rollback events for this transaction in its rollback interrupt routine. At the end of the commit process this counter value is reported to the Transaction Commit Engine which updates the table. A large number of rollbacks may suggest that the transaction is too large and the programmer may want to reduce it to improve performance.

• Directory out of shadow blocks memory indication. A time stamp of this event is logged to each active Transaction entry. This helps identifying the set of transactions leading to this event that may involve substantial performance penalty.

• Directory shadow blocks usage. Each time the cache issues a shadow block write-back request this event is counted in the transaction statistics table. If a transaction has a high count of shadow block usage and also has an indication that a Directory out-of-shadow event occurred during the transaction execution, the programmer may need to split this transaction to improve performance.

5.2 Performance statistics

The following are counters implemented and saved inside the FPGA.

• Total program execution time, logged by the Transaction Dispatch Engine. It counts the number of cycles from the dispatch of the first transaction to commit end of the last one.

• Average idle time per CPU due to ordering issue. In ordered transactions this counter counts the wasted cycles spent waiting to preceding transactions to end execution.

• Average idle time per CPU due to commit bus load. It counts idle cycles spent when the bus is occupied by another committing transfer.

• Total average idle time per CPU. It counts all the delays, including wait for the cache to fetch data.

6. CONCLUSIONS

Using FPGAs to simulate a transactional memory system is fast and accurate. It can considerably reduce weeks of simulations to hours. While simulations have to reduce accuracy in order to deliver results in acceptable time the FPGA approach runs a real system and experiences accurate fabric behavior.

Performance monitoring is the weak spot of the entire FPGA system simulation concept. Unlike in software simulators, in an FPGA it is not possible to collect megabytes of data during run time. The resources are limited. We must carefully plan the statistics we require and add system monitors that collect this data during run time.

7. ACKNOWLEDGMENTS

We thank Altera for supplying us with the required evaluation board and product licensing.

8. REFERENCES

[1] Maurice Herlihy, J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. Proceedings of the 20th annual international symposium on Computer architecture, pages 289 – 300, 1993.

[2] Ravi Rajwar, James R. Goodman. Transactional lock-free execution of lock-based programs. Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, pages 5 – 17, 2002.

[3] Lance Hammond, Kunle Olukotun, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis. Transactional Memory Coherence and Consistency. Proceedings of the 31st annual international symposium on Computer architecture, pages 102 – 113, 2004.

[4] Altera. Stratix II EP2S180 DSP Development Board Reference Manual. August 2005.

[5] Altera. Nios II Processor Reference Handbook. October 2005.

[6] Altera. Avalon Interface Specification. May 2005

[7] Altera. Quartus II Version 5.1 Handbook, Volume 4, SOPC Builder. October 2005.