A Survey on Distributed Transactional Memory System with a Proposed Design for Parallel Processors

Dr.G.Muneeswari[1]

[1]*Professor, School of Computer Science and Engineering, VIT-AP University, Amaravati, Andhra Pradesh, India*

Abstract - *The main aim of this review is to investigate on a comprehensive hardware support to transactional computing. In the modern technology many research work is going on with respect to the transactional requirement like hardware support, system software support and runtime environment support etc., For the concurrency control in transactions, many lock based synchronization methods have been evolved but they are limited to the speed of the execution. A method which can be proposed to an alternate to the lock based approach is transactional memory which allows the transaction to execute concurrently and later resolves the conflicts. This survey, reviews several variants of transactional memory schemes and two new design mechanisms are proposed for the implementation of transactional memory.*

Key Words: Transactional Memory, Cache Coherency, Concurrency, Context Switching, Locking, Hardware architecture

1. INTRODUCTION

In the modern world, mostly speculation-based transaction processing and some programming language constructs and system software changes provides an alternate solution to the traditional concurrency control mechanisms. The objective of this proposal is to investigate on development of novel hardware architecture, related software techniques/algorithms and their respective implementations to support transactions on any concurrent environment as a whole, and as a fine grained technique applicable for many cores with course grain threaded, efficient distributed systems.

In a nutshell, for the multiple patallel architectures and for the multicore systems, inorder to provide concurrency support, locking techniques are basically used which leads to the complexity of the system software and overheads related to performance metrics. The main issues proposed for investigation are as follows:

- Investigation on hardware architectures for deploying the Hardware Transactional Memory leads to provide a best solution for concurrency problems and a suitable alternate to locks.

- To compare the proposed HTM system with lock-based systems from a performance and scalability points of view.

- The proposed HTM shall be enhanced to take into account issues like exception handling, paging and context switching.

In this paper section 2 describes about literature survey and section 3 elaborates on the proposed TM design. Section 4 concludes the paper.

2. LITERATURE SURVEY

This section presents some previous approaches to Transactional Memory as reported in the literature. The section also highlights the drawbacks of those in reference to the modern architectures. The actual representation of transactional memory was first introduced by Herlihy and Moss [1,4]. Their implementation was an extension to the cache-coherence protocol and cache mechanisms used in general-purpose architectures. The primary goal of this model was to provide a mechanism for implementing atomic operations with ease. However, this model imposes restrictions on the size of the transaction and cannot survive context switching.

Transactional Lock Removal (TLR) [2,5]: With the help of this idea, the concurrency control mechanism using locks can be freely executed without enabling locks. This can be successfully completed irrespective of the existence of some module conflicts or changes in the code or non-existence of programmer. This model already incorporating all the relevant features of current computing systems and its associated features. The main advantage of this system is scalability, extended programming features and performance. Another major problem with the existing critical problem solution is blocking behavior and that is totally avoided in this transactional lock removal mechanism.

Speculative Lock Elision (SLE) [3,6]: This method mainly focusses on multithreaded program execution. It is a hardware based approach where unwanted serializability using concurrent locks can be avoided in the execution phase. One of the vital part of the execution of threads here is that the read locks and write locks need not be obtained for the proper functionality of the code. Some of the instructions required for the concurrency control can be predicted and various threads can be executed parallel or concurrently in a critical section enforced by the similar locks. There is a chance for the misprediction and this may be identified using traditional caching mechanism and it at all failure occurs during transaction then rollback recovery mechanism will be enabled. This mechanism can be implemented as part of the micro architecture without any additional hardware or software modifications.

Large Transactional Memory (LTM): This mechanism is an extended version of Hardware Transactional Memory (HTM) that uses cache memory for its implementation. Inside the operating system code, a table is maintained which will be filled up with all the transaction information. The major disadvantage here is the physical or main memory size is very small and that leads to the complexity with respect to the detection of conflicts.

Unbounded Transactional Memory (UTM) [7]: The disadvantage of LTM is eliminated in UTM in which the transactions with context switches can be efficiently handled here. Actually in the case of LTM, the transaction is limited by the size of the physical memory but in UTM it is restricted only by the virtual memory which can handle huge number of transactions. Initially, if the transaction is not handled by the physical address space, then it can be handled in the thread virtual address space. The disadvantage of UTM is that for certain transactions with I/O operations are not supported.

Virtual Transactional Memory (VTM) [8]: It is an efficient method which handles the transaction execution in both hardware and software architecture module design. It supports full implementation of cache overflows and also handles transactional context switching. But this model does not have any implementation model for system calls or waited I/O or interrupts during the transaction execution. All the nested transactions are identified with virtual address space with an improved thrashing capabilities.

Transactional memory Coherence and Consistency (TCC)[9]: This model incorporates atomic transaction and it is a basic unit of concurrency and the transaction information is sent as a message with a designated packet and this packet can be broadcasted to the physical memory atomically into a large block. Ideally, this method eliminates the requirement of the traditional bus based coherence protocols. There may be some problem associated with a simultaneous read and write access to the shared data and this can be resolved using rollbacks that make use of hardware mechanism. The major problem here is the number of messages transferred here is more and increases the bandwidth of the processor messages.

Log-Based Transactional memory (LogTM) [10]: This transactional memory mechanism will make the operation of commit faster by updating the before image values in the log maintained in the virtual memory. This enables the after image value to be updated in the actual original place. This method identifies the conflict by extending the MOESI directory protocol.
Only commit operation will be handled in hardware but abort operation will be handled in software only. The major problem with this mechanism is other kind of I/O operation, context switching and run time OS interactions can't be addressed. Unrestricted Transactional Memory (UrTM)[11]: The major contribution in this model is a design that can handle operating system calls, I/O operations of a transaction in an efficient way. This mechanism handles two types of transaction operation. One is restricted and another one is unrestricted. These transactions can be implemented on the hardware which makes the entire system faster. Each process is permitted to execute only one unbounded transaction and optimization [12] is achieved with respect to the memory blocks. Serializability is incorporated on top of the transaction.

Our proposed Goals try to address several key issues in transactional memory system:

- To Design, implement and test Hardware Transactional Memory (STM) systems that will facilitate automatic program parallelization
- To Evaluate Hardware Transactional Memory Performance on traditional hard-to-parallelize applications (E.g., SPEC 2000 CPU INT, Splash, Splash-2, etc.)
- Explore the opportunities brought by Transactional Memory and study their impact on traditional spectrums of parallelism
- Evaluating the fundamental approaches in development of HTM platforms for the new class of multi-core machines
- Building the support needed by the transactions in hardware, compilation, library support, and the interoperation HTM and STM systems
- Measuring the tradeoffs in handling overflow in hardware vs handling in software. Fine grain locking must be compared against HTMs and STMs in this issue.
- Investigating what programming language transactional constructs would best help in programming, and what are the issues with their implementation.
- Addressing the operating system challenges (context switching, I/O, Interrupts) in HTM and providing solutions for it
- Efficient workload creation that clearly identify the advantages and disadvantages of providing TM support at different layers of the software/hardware stack

- Integrating a TM system into a legacy system, such that the user achieves at least the same level of performance with a TM system as opposed to a system without one

3. PROPOSED DESIGN

An RTL model for a simple Hardware Transactional Memory with multi core support has been developed. In this model we have essentially used a centralized directory based MSI (Modified, Shared, Invalid) cache coherency protocol and also the De-centralized MOESI (Modified, Owned, Exclusive, Shared, Invalid) directory based cache coherency protocol is used for conflict detection. The memory model for this simple TM consists of L1 level Cache (local for every processor), a common L2 level cache and a main memory.

Multicore System Design - I:

L1 level Cache with inclusive two-level cache hierarchy is implemented using verilog HDL. L1 level cache is designed with write-back, LRU replacement policy. The centralized directory is built with the support of full bit vector scheme sharer list. The L2 level cache is common for all the processors and it uses round robin replacement policy. The policy that is used for handling cache writes is Write-Back policy. We Basically use the Write-back caching because it saves the system from performing many unnecessary write cycles to the system RAM, which can lead to noticeably faster execution. Main memory is designed in such a way that it will transfer only one word per clock cycle. It requires 20 clock cycle latency for a block of 16 words. Apart from this memory model a simple state machine based processor is designed and it can handle one instruction at a time.

Automation using Perl:

Perl is used to generate the multicore design in verilog HDL taking the parameters like Number of processors, Main memory size, Cache block size, L1 size its set associativity and L2 size its set associativity.

Multicore System Design - II:

L1 level cache is implemented with write-back, LRU replacement policy. The De-centralized MOESI (Modified, Owned, Exclusive, Shared, Invalid) directory based cache coherency protocol is built with the support of full bit vector scheme sharer list. The L2 level cache and interconnection network yet to be implemented. Similar to design-I, a simple state machine based processor is designed and it can handle one instruction at a time.

Feeds from Simics:

The simulation framework uses *Virtutech Simics* [9], a full-system functional simulator, accurately models the Ebony processor. A cross compiler of the Ebony processor is used to create the executables from a given 'c' code. From the given C code with lock instructions, the support for our SimpleTM interface will be added using Simics "magic" instructions. A Python script captures the load/store requests made by the program and outputs to a file. All other instructions are treated as no-ops. The feeds obtained are given as input to the processors in verilog. We are Currently working on getting feeds per thread. The main important work here is that we need to store information like thread creation time.

Simple TM – Design Stage:

The Version management policy of transactional memory will be implemented in verilog HDL. The Conflict detection mechanism will extend the MSI cache coherence protocol to detect conflict. Conflict resolving mechanism will be coded in 'e' verification language. Our Simple TM will not handle overflows beyond L2 level. It is currently not supported with context switch or system calls or I/O features. But Simple TM design can be extended easily to handle overflows and context switches at a later point. Process management and memory management will be done using 'e' verification language.

4. CONCLUSION

In this paper we have discussed the problems related to traditional locking mechanism and the suitable solutions for the earlier concurrent control mechanisms. The comprehensive review of several Transaction Memory concepts are discussed extensively and the merits and demerits of every type of Transactional Memory systems are analyzed in detail. Apart from the review of the transactional memory concepts the major two designs of the system with respect to cache coherence protocol and software design has also been proposed which will be an efficient mechanism for concurrency control that can handle I/O processing, context switching and child process execution.

REFERENCES

[1] J. Zeng, S. Issa, P. Romano, L. Rodrigues and S. Haridi, "Investigating the semantics of futures in transactional memory systems", Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 16-30, 2021.

[2] R. Filipe, S. Issa, P. Romano and J. a. Barreto, "Stretching the capacity of hardware transactional memory in ibm power architectures", Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, pp. 107-119, 2019.

[3] Z. Shang, J. X. Yu and Z. Zhang, "TuFast: A lightweight parallelization library for graph analytics", IEEE 35th International Con. on Data Engineering, pp. 710-721, 2019.

[4] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In ISCA 20, pp. 289–300, May 1993.

[5] R. Rajwar and J. R. Goodman Transactional lock-free execution of lock-based programs. In ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, pages 5–17, New York, NY, USA, October 2002. ACM Press.

[6] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In Proceedings of the 34th International Symposium on Microarchitecture, December 2001.

[7] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In Proceedings of the Eleventh International Symposium on High Performance Computer Architecture, February 2005.

[8] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. In Proc. of the 32nd Annual Intl. Symp. On Computer Architecture, Jun. 2005.

[9] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D.Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In Proceedings of the 31st Annual International Symposium on Computer Architecture, June 2004.

[10] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In Proceedings of the 12th Symposium on High-Performance Computer Architecture, Feb. 2006

[11] C. Blundell, E. C. Lewis, and M. M. Martin. Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions. Technical Report TR-CIS-06-09, University of Pennsylvania, June 2006.

[12] Peter S. Magnusson et al. Simics: A Full System Simulation Platform. IEEE Computer, 35(2):50–58, Feb. 2002.