Configurable Version Management Hardware Transactional Memory for Multi-processor Platform

Jeevan Sirkunan[1], Chia Yee Ooi[2], N. Shaikh-Husin[3], Yuan Wen Hau[4], Trias Andromeda[5], M. N. Marsono[6]
[1,3,6] Faculty of Electrical Engineering, Universiti Teknologi Malaysia, 81310 Skudai, Johor, Malaysia
[2]Malaysia-Japan International Institute of Technology, Universiti Teknologi Malaysia Kuala Lumpur, 54100 KL, Malaysia
[4] IJN-UTM Cardiovascular Engineering Center, Faculty of Biosciences and Medical Engineering,
Universiti Teknologi Malaysia, 81310 Skudai, Johor, Malaysia
[5] Department of Electrical Engineering, Diponegoro University, Semarang, Indonesia, 50275.
Email: [1]jeevan2@live.utm.my, [2]ooichiayee@ic.utm.my, [3]nasirsh@fke.utm.my,
[4]hauyuanwen@biomedical.utm.my, [5]triasandromeda@undip.ac.id, [6]nadzir@fke.utm.my

***Abstract*— Programming on a shared memory multi-processor platforms in an efficient way is difficult as locked based synchronization limits the efficiency. Transactional memory (TM) is a promising approach in creating an abstraction layer for multi-threaded programming. However, the performance of TM is application-specific. In general, the configuration of a TM is divided into version management and conflict management. Each scheme has its strengths and weaknesses depending on executing application. Previous TM implementations for embedded system were built on fixed version management configuration which results in significant performance loss when transaction behaviour changes. In this paper, we propose a hardware transactional memory (HTM) with interchangeable version management. Random requests at different contention levels are used to verify the performance of the proposed TM. The proposed architecture is targeted for embedded applications and is area-efficient compared to current implementations that apply cache coherence protocols.**

Keywords— Hardware transactional memor, Embedded system, Multi-processor

I. INTRODUCTION

Parallel programming model partitions a singular task to be executed into several smaller tasks. Message passing and shared memory are the most common parallel programming models. Message passing needs explicit communication, in which programmers are required to synchronize memory access. On the other hand, shared memory requires blocking synchronization or lock which are usually done implicitly by hardware [1]. Fine-grained lock yields better performance but requires expert programmers to tap to its full potential. Meanwhile, coarse-grained lock is simpler to implement but performs poorly since it limits parallelism.

Transactional Memory (TM) provides non-blocking wait-free synchronization among memory sections. Each transaction is atomic, isolated, and consistent. It is aimed to simplify multi-threaded programming while making full use of multi-processor hardware capacity. The magnitude of simplification was quantified by Rossbach et al. [2] on a multi-player game programming assignment. In TM, changes made by conflicting transactions are undone and the transactions are either aborted or restarted. On the other hand, changes from successful transaction become permanent.

Several hardware transactional memory (HTM) architectures have been proposed [3-7]. However, most are aimed for high performance system with cache coherence protocols [8]. Nonetheless, there are many embedded applications such as network processing that use multiple light-weight Reduced Instruction Set Computer (RISC) or micro-engines. References [8-10] have proposed HTM for embedded systems. The performance of the HTM is dependent on both the configuration and its application. In general, HTM configuration is divided into version and conflict management, and the application is categorized based on its contention level. Previous implementations on embedded system, e.g. [8] focused only on conflict management.

In this work, we propose a light-weight HTM architecture for embedded system with both version management schemes to give maximum performance depending on the application. We propose a similar approach to Configurable Transactional Memory (CTM) [8], but in addition, we integrate interchangeable version management to cater for different kind of applications. The fully associative cache within CTM [8] architecture allows both version managements to be deployed without much additional hardware resources. In our work, we verify our HTM with random requests at different contention levels to model various types of applications.

The rest of this paper is organized as follows. Section II presents related works in transactional memory with hardware support. Section III presents the HTM configuration overview. Section IV shows the system architecture of the proposed configurable version management HTM. Section V discusses the performance evaluation towards random request with different contention levels. This paper is concluded in Section VI together with suggestions for future work.

II. RELATED WORKS

Currently, lock-based synchronization schemes are widely used for synchronizing multi-processor. The increasing

corresponding author: M. N. Marsono, nadzir@fke.utm.my

application programming complexity has created the need for research on TM. Software implementations of transactional memory (STM), e.g. [11] give poor performance. Most of current works focus on HTM are based on cache coherent protocols. References [3, 4] are some of the earliest works that introduced additional instruction set by adding additional cache line for TM.

HTM placed at the cache level are bounded by the maximum cache size, and software programmers need to take this into consideration. References [5] and [6] proposed HTM schemes to provide better abstraction layer for programmers at the cost of performance loss in certain conditions. DynTM [7] and FlexTM [12] were proposed with an interchangeable HTM configuration in order to adapt to changes in application behaviour. ZEBRA [13] proposed a new approach by associating contention with data accessed by transactional codes rather than the code block itself, thus allowing a more efficient partitioning between eager and lazy managements. The aforementioned works focus on building HTMs for high performance cache coherent systems. These architectures were implemented and tested in simulation environment. Several other works targeted the implementation of HTM for field programmable gate array (FPGA) platform. ATLAS [14], Real Time Transactional Memory (RTTM) [15] and NetTM [9] were all hardware implementations that were built based on fixed configurations. Their performance are highly dependent on running applications.

CTM [8] was introduced with a generic approach in building HTM for embedded system. In this design, the system can be configured to lazy or eager conflict management to suit the application demand (probability of conflict). Its architecture consists of a unified cache for all processors, eliminating the need for coherence protocol. However in this work [8], version management context was not exploited since on transaction commit, the transactional memory cache inside CTM still needs to update main memory one word at a time. Another architecture that was targeted for embedded system is Embedded-TM [10]. Its focus is to reduce power consumption on HTM. However, it also uses cache coherent protocols that is usually absent in multi-processor FPGA platform [8].

III. HTM CONFIGURATION OVERVIEW

A. HTM criteria

Various architectures proposed by [5, 8, 9, 16] can be categorized into two main aspects: Version Management and Conflict Management [7]. Fig. 1 is a generic depiction of the HTM architecture, where *n* is the number of processors. *TM_buffer* and *Main_memory* are both running at similar frequency and thus, the access time of both memories are similar. This paper proposes a HTM architecture based on [8]. The CTM architecture is resource-lean and is able to work in a Multi-Processor System on Chip (MPSoC) system with or without cache coherence support. It can also work with heterogeneous core accelerators making it ideal for embedded system implementation.

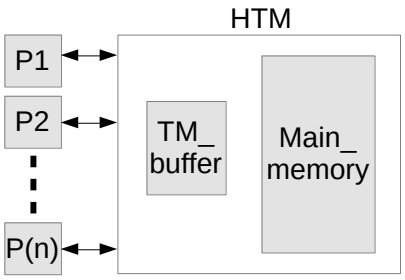

Fig. 1: System Overview of HTM in MPSoC

1) Version Management: Version management is categorized based on the location of the modified transaction [7]. For the eager version management, old data are kept in the *TM_buffer* and updates are written directly on the *Main_memory*. For the lazy version management, it is vice versa. Additional clock cycles are needed when the *TM_buffer* updates the *Main_memory* regardless with new or old data. Consequently, eager version management would have lower T_{commit} lazy version management has lower T_{abort} (3).

2) Conflict Management: Conflict management defines how conflicts are detected and managed [7]. The eager conflict management detects conflicts during read or write phases and then resolves them immediately. On the other hand, the lazy conflict management detects conflicts in the read, write or commit phases, but they will only be resolved during the commit phase. Both management schemes resolve conflicts by aborting and restarting transactions to avoid dead-lock. For eager conflict management, processors will be given random delay before it can restart in order to avoid live-lock.

B. HTM criteria

The processing time needed for HTM can be defined as follow.

$$T_{process} = T_{access} + T_{commit\_abort} \quad (1)$$

$$T_{access} = T_{read\_write} \times n \times s \quad (2)$$

From (1), $T_{process}$ represents the total clock cycles needed by HTM to handle a finite amount of transactions, whereas T_{commit_abort} is the additional processing time needed for HTM to handle commit and abort. It is similar to the hit miss penalty in a cache system. T_{access} from (2) is the total time taken for read/write request, regardless whether it is useful or otherwise. n represents the number of transaction, whereas s is the size of transaction. T_{read_write} depends on the architecture of the memory. In our design, we use a fully associative buffer to reduce the address search time to one clock cycle. The *TM_buffer* can hold either the old or new transaction values, and the access flag for each address.

In HTM, penalty occurs during commit and abort. During the update period, other processors are not allowed to access the memory.

$$T_{commit\_abort} = \sum_{i=0}^{n}((T_{commit})(P_{commit}) + (T_{abort})(P_{abort})) \quad (3)$$

$$P_{commit} = 1-(P_{abort}) \quad (4)$$

T_{commit} and T_{abort} depend on the configuration of the HTM. From (3), at higher contention (high P_{abort}), T_{abors} needs to be reduced to get a lower penalty, and vice versa for low conflict situations. Reference [8], the HTM structure uses lazy version management but it does not integrate fast abort. The *CTM_cache* and *Main_memory* update their memory one word at a time during commit and abort phases based on the address FIFOs. During an abort or commit, each entry needs to check with the *shared flag* to make sure that it does not remove flag of ongoing transactions. In order to have fast abort for lazy version management or fast commit for eager version management, each entry would require its own *shared flag*.

For low contention situation, lazy conflict management is preferred. Overheads caused by processors to check conflict status can be avoided. On the other hand, eager conflict management can minimize penalty caused by conflicting transactions accessing the memory (zombie transactions), making it suitable for high contention applications [8]. If the total random delay (of aborted transactions in the eager conflict management) in addition to the delay for processor to update its conflict status is greater then the delay of zombie transaction, lazy conflict management will perform better.

TABLE I: Contention level preference towards HTM Configuration

		Conflict Management	
		Lazy	*Eager*
Version Management	*Lazy*	High/Low	High/High
	Eager	Low/Low	Low/High

Lazy version management is suitable for high contention condition as it allows faster abort, while eager version management is more appropriate for low conflict condition. Table 1 shows the relationship of the configuration (version and conflict management) towards the contention level preference. In our implementation, we focus on version management. The architecture of CTM [8] detects conflicts eagerly by default since all memory entries are shared. However, lazy conflict management is done by notifying the processor during commit phase only. Variances in performance for different conflict management schemes may also be due to different pathology [17], processor delay, and also the random delay after a conflict. Therefore, in our proposed architecture, we fixed the conflict management to lazy in order to obtain a fair comparison between the two version management schemes.

Table II: Contention policy for attacker and defender

Transaction		**Conflict Status**	
Attacker	**Defender**	**Attacker**	**Defender**
Read	Read	✓	✓
Read	Write	×	✓
Write	Read	✓	×
Write	Write	✓	×

IV. HTM ARCHITECTURE

Fig. 4 shows the overview of the proposed HTM system architecture for four processors that have access to a shared memory. Round robin arbitration is used to give each processor equal priority. The HTM architecture is divided into four parts : *TM_buffer*, *Control unit*, *Address_FIFO*, and *Main_memory*.

The *TM_buffer* consists of several sub-parts: *Valid*, *Address*, *Read Write Set* and *Data*. Each processor is given 2 flags to keep track on its transactions. The *Valid* bit is asserted to notify that the location is already in used. A conflict is detected when the *TM_buffer* is being updated. *Read Write Set* for the current address is compared with the flags of the current access. *Write-on-write*, *read-after-write*, and *write-after-read* from two different transactions are the conditions that resulted in conflict [4]. Table 2 shows the policy of the attacker towards the defender when a conflict occurs. An attacker represents a transaction that wants to have access to a memory location that belong to different transactions (defenders). The tick in Table 2 shows that the transaction is still valid whereas the cross is vice versa. When a transaction is already in conflict, all its future transactions will become zombie transactions.

The Control Unit determines the behaviour of the proposed HTM. During the eager mode, the updated data (new) is kept in the *Main_memory*, whereas the old data (old) is kept in the TM_buffer; and vice versa for lazy mode. For the lazy version management's read and write phases, the *TM_buffer* is updated. If a miss occurs, the *TM_buffer* data will be modified and flags will be updated. The *Main_memory* will hold the old data, while the *address_FIFO* will hold the location of memory access. If it is a hit, the *TM_buffer* data and flags will be updated. During a commit request, the corresponding processor checks its conflict flag. If there is no conflict, the transaction commit is successful. The old data from *Main_memory* will be replaced with the *TM_buffer* data corresponding to the *address_FIFO* of that transaction. If the location is shared, only the corresponding processor flag is removed, else the whole entry is removed. However, if there is an abort, the modified data from *TM_buffer* will be removed. Similarly if the location is shared, only the corresponding processor flag is removed, else the whole entry is removed. An array of shared flag comparator is used to allow *TM_buffer* to update itself within one clock cycle. A similar process takes place with eager version management, but the modified data is now kept in *Main_memory* whereas the old data in *TM_buffer*.

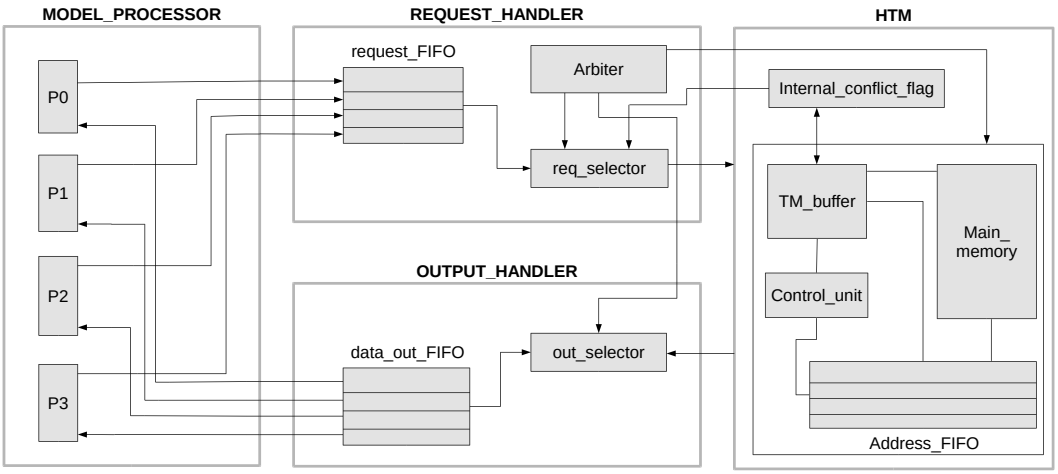

Fig. 2: Top level of proposed HTM architecture

Based on [8], the defender must always win during a conflict and the conflicted processor needs to undo all of its transactions before the next request can be processed. In our architecture, we allow either attacker or defender to win. The policy of attacker and defender can be seen in Table 2. This can be further extended by determining the winner based on the age of the transaction [4]. However, this may cause an entry to have both flags from an aborting and committing transaction. The abort will undo the commit if the commit takes place earlier since the shared flag protects the aborting transaction flags.

Therefore, we introduce internal and external abort flags for each processor. When a transaction is flagged as conflict, the internal and external abort flags are asserted. The internal abort flag will be given higher priority compared to request coming from the processor, allowing the transaction to abort first. The external abort flag remains asserted and the transaction will be considered as zombie until the processor commits. Besides, removing an early aborted transaction protects other transactions from conflicting with it. This feature also allows the version management to be changed at run time, by aborting all previous transactions and restarting with a different management scheme.

V. PERFORMANCE EVALUATION

The performance evaluation is done by comparing the proposed architecture with two different version managements. The aim is to observe the performance of different version management at different levels of contention. We alter the distribution of memory accesses to represent applications at different contention levels. The experiment has been carried out with four identical processors models pumping in random requests to the HTM. All requests are normally distributed, and the contention level is adjusted by changing the standard deviation. A smaller standard deviation would result in a higher contention rate. We used Quartus 2 v13.0 to evaluate the system. The HTM is modelled in Verilog and the RTL simulation in Quartus is done using ModelSim 6.6 to obtain cycle accurate results. We implemented the system on Cyclone 4 EP4CE115F29C7 device

TABLE II: Area and Max frequency comparison

Processors	Size	Logic element	Max Frequency (Mhz)
2	64	3,220	96.81
	128	6,096	84.67
	256	11,778	70.58
4	64	4,882	90.81
	128	9,244	83.97
	256	17,423	67.05

The resources usage and maximum frequency of the proposed HTM with different sizes is shown in Table 3. The sizes represent the maximum number of memory entries in the *TM_buffer*. In this analysis, each entry is one byte and the *Main_memory* has 256 bytes. Having a large fully-associative cache would results in a high critical path delay during memory access due to the comparator array.

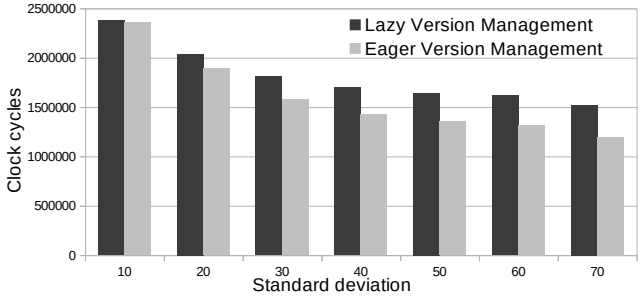

Fig. 2: Clock cycle versus standard deviation of memory access for eager and lazy version management

Fig. 2 shows the difference in performance (in the number of clock cycles) that HTM requires for lazy and eager version management schemes for the size of transaction of 8 at different contention levels. Each processor performs 500 successful transactions. Fail transactions are retried until all the

transactions are committed. Different standard deviation represents different contention levels of any application. From this experiment, we observe that eager management scheme performs better in most cases, even for high contention, and the eager performance is comparable to lazy. This is because the abort penalty is smaller than the commit penalty in our case study. If a transaction has been flagged as conflict early in its transaction, the entries that have to be undone up to this point. The penalty of zombie transactions after a conflict has been detected is similar for both version managements. On the other hand, for a successful commit to occur, all memory accesses within that transaction needs to be updated. This makes the overall penalty of abort to be insignificant and the advantage of eager version management less compelling. However, this scenario also happens for commit if the processor request locale addresses, making the improvement during commit for eager version management less significant.

VI. Conclusion and Future Work

This paper proposed a HTM architecture with configurable version management for multi-processor platform targeted for embedded application. Eager version management is for low conflict, while lazy version management is for high conflict application. On a shared memory system, the total size of data being committed and aborted determines the best HTM version management. Since the main objective of applying HTM is to create an abstraction layer for programmers to do multi-threaded programming, having the system configurable is inadequate. The hardware must be able to adapt its configuration based on the application without the need for programmer to manually specify it. An integrated decision making system is needed to determine the optimized version configuration based on the application contention. These are the topics of our future research.

Acknowledgment

This work is supported in part by Ministry of Education of Malaysia Fundamental Research Grant (UTM Vote No 4F327).

References

[1] M.L. Navazo, “Hardware approaches for transactional memory,” M.Sc.Thesis, Technical University of Catalonia, 2008.

[2] C. J. Rossbach, O. S. Hofmann, and E. Witchel, “Is transactional programming actually easier?” in Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Bangalore, India, Jan 2010, pp. 47–56.

[3] M. Herlihy and J. E. B. Moss, “Transactional memory: Architectural support for lock-free data structures,” SIGARCH Comput. Archit. News, vol. 21, no. 2, pp. 289–300, May 1993.

[4] 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,” SIGARCH Comput. Archit. News, vol. 32, no. 2, Mar 2004.

[5] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, “Unbounded transactional memory,” in Proceedings of the 11th International Symposium on High-Performance Computer Architecture, Washington, DC, USA, Feb 2005, pp. 316–327.

[6] L. Yen, “Signatures in transactional memory systems,” Ph.D. Dissertation, University of Wisconsin, 2009.

[7] M. Lupon, G. Magklis, and A. Gonz´alez, “A dynamically adaptable hardware transactional memory,” in Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Dec 2010, pp. 27–38.

[8] C. Kachris and C. Kulkarni, “Transactional memories for multiprocessor FPGA platforms,” Journal of Systems Architecture, vol. 57, no. 1, pp. 160–168, Jan 2011.

[9] M. Labrecque and J. G. Steffan, “The case for hardware transactional memory in software packet processing,” in Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, La Jolla, California, USA, Oct 2010, p. 37.

[10] C. Ferri, S. Wood, T. Moreshet, R. I. Bahar, and M. Herlihy, “Embedded-TM: Energy and complexity-effective hardware transactional memory for embedded multicore systems,” Journal of Parallel and Distributed Computing, vol. 70, no. 10, pp. 1042–1052, Oct 2010.

[11] N. Shavit and D. Touitou, “Software transactional memory,” in Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, Ottowa, Ontario, Canada, Aug 1995, pp. 204–213.

[12] A. Shriraman, S. Dwarkadas, and M. L. Scott, “Flexible decoupled transactional memory support,” in Proceedings of the 35th Annual International Symposium on Computer Architecture, Beijing, China, June 2008, pp. 139–150.

[13] R. Titos-Gil, A. Negi, M. Acacio, J. Garcia, and P. Stenstrom, “ZEBRA: Data-centric contention management in hardware transactional memory,” IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 5, pp. 1359–1369, May 2014.

[14] N. Njoroge, J. Casper, S. Wee, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun, “ATLAS: A chip-multiprocessor with transactional memory support,” in Proceedings of the Conference on Design, Automation and Test in Europe, Nice, France, Apr 2007, pp. 3–8.

[15] M. Schoeberl and P. Hilber, “Design and implementation of realtime transactional memory,” in Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL), Milan,Lombardy,Italy, Aug/Sept 2010, pp. 279–284.

[16] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III, “Software transactional memory for dynamic-sized data structures,” in Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing, Boston, MA, USA, July 2003, pp. 92–101.

[17] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, “Performance pathologies in hardware transactional memory.” in Proceedings of the 34th Annual International Symposium on Computer architecture (ISCA), New York, NY, USA, June 2007, pp. 81–91.