ISSN:0975-9646
Shweta Kulkarni et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 2 (2) , 2011, 766-775

Taxonomy for Transactional Memory Systems

Shweta Kulkarni* Prachi Kulkarni* Juie Deshpande* Priyadarshini Kakade*
Priti Ranadiveξ Madhuri Tasgaonakar* Shilpa Deshpande*

* Cummins College of Engineering for Women, Pune, India
ξ Center for Research in Engineering Sciences and Technology (CREST),
KPIT Cummins Infosystems Ltd., Pune, India

Abstract-The emergence of multi-core systems has given rise to the need of developing multi-threaded applications. To ensure synchronization between concurrent operations lock based mechanisms are used. Nevertheless, conventional lock based synchronization techniques lead to problems such as deadlocks, priority inversion and convoying. Moreover, lock based synchronization mechanisms are not scalable. Transactional memory (TM) is being considered as an effective alternative to conventional lock based synchronization mechanisms. In the past decade, different methods to implement TM systems have been proposed. These approaches are either software-only, hardware-only or hybrid approaches. In this paper, we present a review of the different design approaches to implement TM systems. We also present a taxonomy that classifies these design approaches and discuss the common issues that need to be considered while implementing a TM system. In addition we present taxonomy for TM in embedded systems.

Keywords: Transactional memory, Lock-free synchronization, TM Taxonomy

I. INTRODUCTION

Advances in semiconductor technology were initially responsible for improved processor performance. However, practical limits on power dissipation restrict the increase in clock speed. The current decade marks the transition from sequential to parallel computation. With the advent of multi-core systems multi-threaded applications are being developed. To ensure that the concurrent operations do not interfere, synchronization mechanisms such as locking are essential. For each data structure, a lock indicates whether the structure is in use or not. Threads cooperate by acquiring the lock before accessing the corresponding data.

Nevertheless, conventional synchronization techniques based on locks have substantial limitations. Coarse grained locks do not scale while fine-grained locks increases lock overhead. In particular, they introduce problems such as deadlocks, priority inversion and convoying. Programs written using other synchronization constructs such as semaphores and monitors are difficult to design, construct, maintain and often do not perform well.

Transactional memory [23] is a new programming construct that provides a high-level abstraction for writing parallel programs. Transactional memory tries to reduce the difficulty of writing concurrent programs by providing atomic and isolated execution of code. TM shifts the burden of correct synchronization from the programmer to TM system. Transactional Memory borrows concepts from the domain of database systems. Similar to database transactions, TM has Atomicity, Consistency, and Isolation (ACI) properties: Atomicity guarantees that transactions execute as an indivisible unit and either commit or abort as a whole, Consistency guarantees that transactions follow the same order during the whole process, and Isolation guarantees that each transaction's operations are isolated to other transactions.

The proposed Transactional Memory approaches can be broadly classified as Hardware (HTM) [18][19][12][20][21][23][25][26][28] and Software (STM)[10][11][13][15][16][29][30][31][33][34][35][36]. Hardware approach exhibits high performance and strong atomicity but has shortcomings such as lack of support for unbounded transactions, architectural limitations, less flexibility and design complexity. In order to overcome the problem of limited hardware resources and to support unbounded transactions most implementations employ some virtualization techniques. Software implementations are cost-effective and flexible than HTMs but are slower as compared to HTMs. Moreover, poor performance and weak atomicity are two serious concerns while implementing TM totally in software. Thus, each of these approaches has its own advantages as well as limitations. In order to avail the benefits of the two, hybrid implementations have been proposed [1][2][3][4][5][6][7][8].The most notable proposal Hybrid Transactional memory (HyTM) by Damron et al. [2] exploits HTM support to achieve high performance and scalability while using software to support transactions that exceed hardware limits.

Recently TM implementations for embedded multi-core systems are being proposed considering energy consumption and complexity as a major design aspects [36][37][38][39][40]. These implementations are mostly hardware-only implementations.

In this paper we present a review of the different implementations and also provide taxonomy for TM systems. Chen Fu et al. [46] have proposed a taxonomy but limited to HTM. We extend this to include all types of TM along with Embedded-TM. We also give a brief introduction to other synchronization apart from TM and locks. The rest of the paper is organized as follows: section II mentions the proposed taxonomy, section III mentions other synchronization approaches apart from locks and TM such as TLRW [41]. In this section we also discuss an investigation of interaction of TM and locks based codes [43]. Section IV presents the observations and conclusions.

II. TAXONOMY

We take a top-down approach to classify TM systems that deal with issues of transaction conflicts, support for virtualization, isolation and nesting. We also classify the systems on basis of whether modifications are performed at the processor level, in the operating system or only in software.

A. Taxonomy based on conflicts

Conflicts occur when two or more threads access the same resource. We classify first and foremost on basis of conflict detection.

Conflict detection method can be lazy or eager. In eager conflict detection the conflicts are resolved as soon as thread seeks data that conflicts with one or more other transactions. On the other hand, lazy mechanism executes a transaction optimistically assuming no conflicts. Conflicts are resolved when a transaction conflicting with other transaction seeks to commit. Conflict detection is usually combined with version management parameter.

Version management refers to how to store new and old data. The existing proposals can be classified on basis of version management as eager and lazy. Lazy version management leaves old values in the memory while new values are stored elsewhere and written back after a transaction executes successfully. Although this makes aborts faster, the more common case of transaction commit shows degradation in performance. On the contrary, in eager version management updates are carried out “in place” while old values are stored elsewhere (in a log). Naturally, commits are faster and aborts involve considerable overhead of writing data back to the memory. On combining conflict detection and version management, we have four categories viz. Eager-eager, eager-lazy, lazy-eager, lazy-lazy. No proposals attempted to combine eager version management and lazy conflict detection possibly due to the semantic problem of eagerly updating data while postponing conflict detection until commit. Recently, Anurag Negi et.al. [47][48] have proposed a HTM with lazy versioning and eager/lazy conflict resolution method.

The next question arises as to what to do when a conflict is detected. The implementation may either stall a transaction (risk of deadlock) or abort it (risking a live lock) or leave the decision to a software contention manager. Further the TM system has to take a decision regarding which transaction to abort. The victim can be chosen based on various conflict resolution policies briefly discussed as follows.

i) Time-stamp: Transactions are assigned a timestamp using the real time clock on begin. When a conflict is detected, the time-stamps of the conflicting transactions are compared. Logically later transactions are forced to either stall or abort. Using this scheme, the oldest transaction gets the highest priority.

ii) Write-set size: Write-set size is the number of blocks (depending on granularity) modified by a transaction. This scheme suggests aborting a transaction that has modified comparatively less number of locks involves is cost-effective.

iii) Polite: It uses an exponential back-off strategy to resolve the conflict. The transaction is aborted after a specific number of unsuccessful attempts to commit.

iv) Polka: It uses a back-off strategy for conflict resolution. The back-off interval is proportional to the difference in priorities between the conflicting transactions.

Most implementations choose the victim depending on its age (time-stamp) or its write-set size. Therefore we include only these two policies in our tree structure, Figure 4. However, it may be noted that proposals may use others mentioned above as well. Figure 1 outlines a taxonomy based on parameters described above.

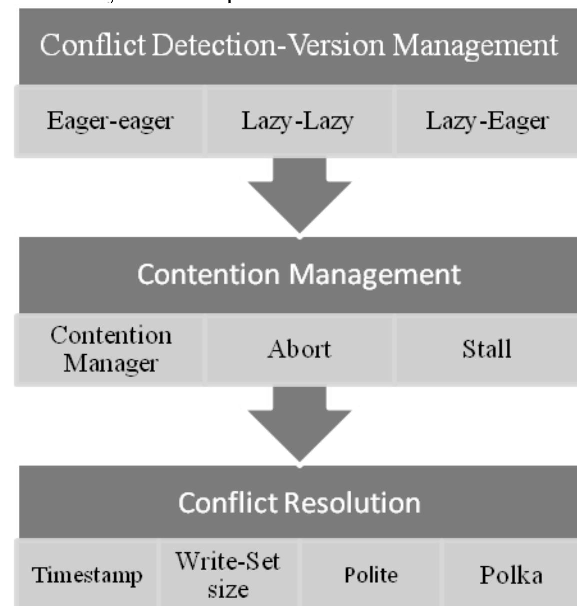

Figure 1: Taxonomy based on conflicts

We now present case studies each representing one of the paths in the tree structure in Figure 4 constructed on basis

of the taxonomy. In addition, we also represent some other proposals in the tree structure. Some proposals specify only some parameters described above and leave the rest flexible for system design.

Log-TM [20]: Eager-Eager: It advocates use of the eager-eager conflict detection-version management scheme. Log-TM stores old data values in a per-thread log in virtual memory while new values are directly written into the memory locations. To abort, Log-TM has to write back old values to their addresses by referring the log making the process slower. As with the eager versioning scheme, commits are faster and aborts slower. It enables eager conflict detection by using directory based MOESI cache coherence protocol. The coherence protocol is extended to handle even the blocks that are evicted from the cache. On detection of a conflict, Log-TM either stalls or aborts one of the transactions. Log-TM makes aborts less common by using stalls to resolve conflicting transactions when deadlock is not possible. In cases where a transaction could lead to a deadlock, it traps to a software conflict handler. Transactions are ordered using the time-stamp method where logically earlier transactions are forced to abort or wait.

HyTM [2]: Lazy-Lazy: HyTM uses best effort hardware transactional memory. It first tries to execute transaction in hardware, if hardware resources exhausted then it executes the transaction in STM. This approach uses Lazy-Lazy conflict detection and version management scheme. A contention manager is used to resolve conflicts. The decision regarding whether to stall or abort transaction, which transaction to abort is left to the manager. The victim transaction is chosen on basis of the timestamp method described above. It supports nested transactions with flattening, gives weak isolation for transactional blocks.

Dynamic Software Transactional memory (DSTM) [9]: Eager-Lazy: Dynamic Software TM was proposed to support dynamic-sized data structures which create transactions dynamically. DSTM a low-level (API) application programming interface uses C++ and Java API‘s to program dynamic data structures. DSTM exploits obstruction free mechanism for synchronizing shared memory. It is an example which employs lazy version management in combination with eager conflict management policy. DSTM uses an explicit contention manager to resolve conflicts. The policy to choose the victim also depends on the contention manager. It supports flattened nested transactions and also provides weak isolation with object-based granularity.

EazyHTM [27]: Eager-Lazy: EazyHTM employs lazy version management in combination with eager conflict detection policy. It enables eager conflict detection by using directory based MOESI cache coherence protocol. By making small hardware modifications in the protocol the EazyHTM detects conflicts eagerly and resolves them lazily by either aborting or stalling the conflicting transaction to avoid cascading waits. EazyHTM proposal leads to faster commits and aborts and allows non-conflicting transactions to commit in parallel. It provides strong isolation for the transactional blocks. Thus it provides remarkable performance improvements compared to the prior HTM designs.

B. Taxonomy based on modifications

While implementing a TM system, changes may be made to one or both of the processor local hierarchy and the operating system. Many implementations also use some software support along with TM modified processor. There also exist some proposals implementing TM entirely in software without any support from the underlying system. The proposals can be sorted as per the regions defined by a Venn diagram as shown in Figure 2.

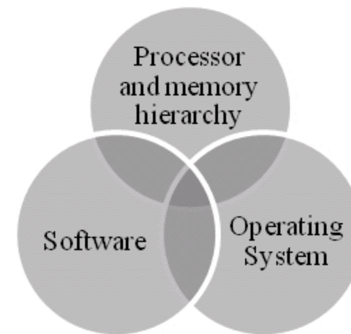

Figure 2: Classification based on modifications

Proposals lying in each region of the Venn diagram have some advantages and drawbacks. Table 1 highlights the advantages and drawbacks of the modifications. We now present case studies each describing modifications at different levels.

Unbounded transactional memory [18]: This approach requires modifications to both the processor chip and the memory subsystem. The processor is modified to support unbounded transactions. New instructions are added to the instruction set architecture. Modification in form of ‘S’ (saved) bit is required for handling rollback. This proposal uses snapshot for register renaming table instead of physical registers. This ensures that before transaction commits, data in the physical registers is not reused. ‘S’ bit vector in the snapshot tracks the physical structure, maintains register reserved list to avoid overwriting of transactions. For committed transaction value of ‘S’ bit vector is cleared and data form register reserved list is copied to register free list. For aborted transaction contents of the renaming table are restored.

McRT-STM [14]: McRT is basically a software transactional memory build within a multi-core run time (McRT) to support C/C++ applications. It is the first algorithm for C/C++ and other applications that use explicit memory management. In McRT synchronization

is achieved by using a two-phased locking protocol that permits multiple concurrent transactions to read and only one transaction to modify. In order to reduce conflicts, McRT employs a scheduler that performs pre-emption to prevent inactive transactions from blocking other active transactions. The former STM were based on non-blocking design that used complex memory management schemes like the hazardous pointers. In contrast McRT uses a shared memory allocator. By employing this method the numbers of aborts have been reduced and memory management has been simplified.

TABLE 1: Advantages and drawbacks

Modification in	Advantages	Drawbacks
Processor local hierarchy only [18][19][12] [20][21][23] [24][26][28]	- high performance - high atomicity	- high implementation and verification cost - no support for context switches - limit on transaction size - impose constraints on programmer
Software only [10][11][13] [15][16][29] [30][31][33] [34][35]	- no hardware support needed - flexibility - no hardware cost	- high overheads - slow - lower execution speed as compared to hardware only proposals
Processor hierarchy and Software [1][2][3][4] [5][6][7][8]	- combines benefits of hardware and software support - can exploit best effort hardware support to boost performance	- extends execution time of large transactions - most implementations rely completely on software exception handling.
Processor and OS [25]	- Supports Unbounded transaction sizes - flexible - high performance and strong atomicity	- memory overhead as compared to hardware proposals
Software and OS [44]	- improved transaction throughput as compared to only software - kernel level scheduling support significantly reduces number of aborts	- lower execution speed as compared to only hardware proposals
All [32]	- Shrinks the functionality gap between hardware TM systems and software ones - does not rely completely on user mode exception handling as in hybrid systems	- lower execution speed as compared to only hardware proposals

Hybrid transactional memory [1]: This scheme lies in between only hardware and only software approaches. It uses software mechanism only if the transaction exceeds hardware resource limitations. Implementing HyTM requires some modifications at the processor level. Hardware for transactional memory includes a transactional state table and a transactional buffer. The transactional state table provides an entry for each hardware context on the processor to track the mode of transactional execution of that hardware context. Each entry in the transactional buffer holds both (old and new) values, bit vectors to indicate which hardware contexts have speculatively read or written the line, and the conventional tag and state information. Apart from that, additional bits are provided for each line for conflict detection. The software scheme for handling transactions that exceed hardware limits is based on DSTM [9]. This approach combines the performance benefits of a pure hardware scheme with the flexibility of a pure software scheme.

HTMOS (Hardware Transactional Memory with Operating System Support) [25]: HTMOS is a modification to the conventional Hardware Transactional Memory (HTM). This implementation suggests changes in OS virtual memory mechanism and architecture instead of the cache subsystem to support allocation and manipulation of memory space for bookkeeping. Performance improvement is achieved by introducing some new data structures and modifying the existing ones at the OS level. For example the Page Table in the classic OS implementation is extended to be Virtual Page Table (VPT), which holds addresses of the secondary copy of the pages. The various changes made result into making the HTMOS fast, simple like HTMs and flexible like STMs. It supports unbounded transactions, reduces read-write overhead of transactions, is fast like other HTMs and provides strong atomicity.The drawback of this implementation is that it requires a lot of memory for maintaining page information.

Scheduling Support for Transactional Memory Contention Management [44]: TM implementations have traditionally used user-level software contention managers to resolve conflicts. However, these contention managers do not ensure reasonable performance under high workloads. This algorithm uses a shared memory segment to provide lightweight communication between the user-level STM library and the kernel-level scheduler. The shared memory region contains a table of consisting of elements equal in number to the maximum number of threads. Each element is a structure that stores STM information for a given thread. The OS thread structure is augmented with a pointer to the respective entry in the

table. Child threads are subsequently linked to the next available entry as part of their STM initialization process. Thus the interaction with kernel is simplified as the application simply fills in data in the shared structure. There is no direct user-kernel interaction. This approach has been implemented Linux and Solaris and has proved to be effective in reducing the number of aborts and retaining throughput even under high contention.

Extending Hardware Transactional Memory to Support, Non-bus, Waiting and Non-transactional Actions [32]: This proposal demonstrates software ideas adapted to work in hardware system with some support from the OS. This paper attempts to shrink the functionality gap between software transactional memory systems and hardware ones. The paper focuses on the Virtual Transactional Memory (VTM) [19], which is a combined hardware-software implementation. Much of the software stack associated with VTM is implemented as part of the Linux kernel in this proposal. The VTM hardware/software interface contains two main data structures - The global transaction state segment (GTSS) that holds the overflow count and a pointer to the XADT structure discussed in [19]. The kernel allocates one GTSS per address space and local transaction state segment (LTSS) per thread. Pointers to these data structures are written into separate registers on a context switch. Most kernel modifications are encountered only by the transacting instructions and thus the impact on other instructions is minimal.

C. Taxonomy based on other parameters

Several other parameters can be taken into consideration while designing a TM system. These are described as follows.

Nesting: Transactional nesting allows a transaction to start inside another. There are three basic mechanisms that support nesting viz. Flattening, open and closed nesting.

i) Flatten: The flattening model includes all nested transactions in the outmost transaction. All transactions share a common read and write-set. On completion of the inner transaction, the outer transaction resumes execution. However, conflict with an inner transaction forces the outer ones to abort as well.

ii) Closed nesting: Closed nesting allows partial abort i.e. only the conflicting inner transaction is aborted and re-executed. The nested transactions have their own read and write sets which merge with the sets of the outer level on commit. On abort, the innermost conflicting transaction rolls back to its original states but not to the top level.

iii) Open nesting: When an open nested transaction commits, its read-write sets are visible to all other transactions. The new values of data can be accessed without having to wait for the outer transaction to commit. Rollback and commit is thus independent of the outer parent transactions.

Isolation or atomicity: Isolation can be strong or weak. When non-committed updates cannot be read from the outside of a transaction, isolation is said to be strong. Strong isolation is easier to implement in hardware using cache coherence protocols to track reads and writes. Most HTM proposals provide strong isolation. When non-transactional code can read non-committed updates, isolation is said to be weak. Shared data may be accessed from outside a transaction that was supposed to be executed atomically. Weak isolation model is easier to implement than strong isolation one but provides a less intuitive model to the programmer.

Memory model: TM system designers have chosen either the shared memory or the message passing model.

i) Shared memory: In shared memory systems communication takes place implicitly through load and store instructions to a global address space. Synchronization and communication are distinct in this model. Shared memory is a simple programming model but it has a complex hardware configurations.

ii) Message passing: Message passing system is like an interrupt driven system where communication takes place explicitly through messages between processors. Synchronization is achieved through sending and receiving of messages between processors. Message passing makes software design difficult.

As seen, both these model have their drawbacks. An ideal model would be the one which combines the benefits of both. It should present a shared model to programmer while at the same time take advantage of inherent synchronization and latency-tolerance of message passing protocols. Such a model has been proposed by TCC [17]. With immense inter-processor bandwidth available in new systems, it is now possible to exploit it and implement such a model.

Granularity: It indicates the size of the read/write sets that are tracked for conflict detection. Conflict detection is usually done at the word granularity, cache line granularity, or object granularity.

i) Word granularity: Using word granularity prevents false sharing but introduces higher space and time overhead.

ii) Cache line granularity*:* It is the preferred granularity for hybrid and hardware systems. However it poses a risk of false sharing.

iii) Object granularity: It is commonly seen in software implementations as it is convenient for the programmer. However, object granularity may also lead to false sharing.

Table 2 shows proposals classified as per above parameters.

D. Classification of Transactional memory based on energy consumption

Transitional memory proposals discussed so far have chiefly considered the aspects of throughput efficiency and ease of programming for evaluating the system. Energy efficiency for evaluating TM on embedded

systems was first considered by Ferri [40] .Unlike general purpose systems, energy consumption parameter is of utmost importance in embedded systems. We review different approaches for implementing TM on embedded systems in this paper. We consider aspects of energy efficient TM design [40] such as memory hierarchy, contention management and shutdown mode. Figure 3 shows embedded-TM classification.

***Memory hierarchy*:**

There are three ways in which different cache structures can be used in the memory hierarchy.

i) L1 with Transactional Cache (TC): As the basic architecture we consider Embedded-TM [36] which stores non-transactional data in a comparatively larger L1 cache and transactional data is stored in smaller, fully associative TC. However, the transactional cache consumes a lot of energy.

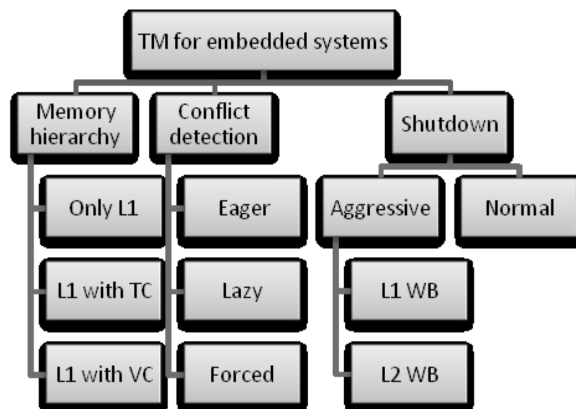

Figure 3: Taxonomy for embedded TM system

Another drawback of this architecture is that for larger transaction which cannot be accommodated, it continues with a much less efficient serial mode execution.

ii) Only L1: To address the limitations associated with TC model there is another model proposed in which both transactional and non-transactional data is kept in the L1 cache. This design eliminates the need of maintaining coherency between the caches on the same level. As L1 is much larger as compared to TC the possibility of transactional overflow is considerably reduced. However, this design is still limited by resource constraints.

iii) L1 with Victim Cache (VC): Use of victim cache between L1 and main memory overcomes drawbacks of the other two architectures. The data primarily resides in the L1 cache. The victim cache is used only when a transactional entry is evicted from the L1 cache. In this case as well, the caches are accessed sequentially, first the L1 cache and then the VC after L1 lookup fails. Here VC can be designed to be smaller than the TC used in the base Embedded-TM architecture [36]. Since L1 is backed up with VC there is enough space for transactions making overflows less common. As VC is used only when L1 cache cannot support the transactions, it is favorable to keep VC powered down until needed. The L1+VC scheme is better than L1+TC scheme. [36]

***Conflict detection and resolution*:**

When a transaction detects conflict with another, one of them needs to abort. The general Back-off strategy used in other TM proposals is inefficient in terms of energy. For embedded systems, we consider following conflict resolution schemes-

i) Eager: This method suggests that a data conflict is detected as soon as a transaction tries to access the modified line in shared memory. This approach is advantageous because it does not require any radical changes in the original cache coherence protocol. It performs well when the data conflict rate is low but fails to do so when the data conflicts occur at a higher rate.

ii) Lazy: This is a more complex alternative for conflict resolution, useful in the higher data conflict rate environment. The conflicts are detected as before and instead of resolving them at the time they are detected, they are left unresolved until the commit time. Lazy resolution implies substantial changes to the platform and the architecture and sometimes might penalize a low conflict transaction but is well suited for high-conflict transactions. It improves both performance and energy efficiency as compared to the eager scheme.

iii) Forced serial: It is another approach that is feasible for higher data conflict rates. It can be run on top of eager or lazy contention management. The system reverts to serialized execution if a transaction has been aborted more than once. Once the transaction completes, the system reverts back to its original conflict resolution policy (i.e., eager or lazy).The brute-force approach is attractive for its simplicity and wide range of effectiveness, and it works moderately well most of the time.

Shutdown: The shutdown mode can be normal or aggressive.

Aggressive shutdown: In this mode, the modified lines are written back to the memory hierarchy on transaction commit. This allows the transactional cache which is not in use to be powered down leading to efficient energy usage. It can be implemented in two ways.

i) L1 WB: Here the modified lines are written back only to L1 cache before TC is powered down.

ii) L2 WB: Here the modified lines are written back to L1 and L2 cache before TC is powered down.

III. OTHER APPROACHES

Though the transactional memory approach is fetching a lot of attention in the parallel computing domain, there are few drawbacks found, especially for the Software Transactional Memory approach. In the following section we discuss a few approaches that talk about alternatives to pure Transactional Memory.

Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. This is an approach which considers the combination of both transactions and

locks in order to achieve the best performance in terms of synchronization. The programmer specifies the critical section. Using an adaptive locking technique, the decision whether to execute the critical section using transactions or with holding mutex locks is taken. According to the experiments performed on various benchmarks by Usui et al. [45], adaptive locks consistently outperform either of its two components (locks & transactions) used individually. It is observed that this implementation provides speedups and simplifies the programming model.

TLRW: The new algorithm TLRW [41] is based on byte locks for single-chip multi-core systems. The design of TLRW is simple and streamlined as compared to the lock free STM systems and delivers scalable performance. Read-write lock-based STMs are a viable approach for single chip multi-core systems with strong progress properties and support for irrevocable transactions but for two chip multi-core systems a lock-free STM is a better approach.

Haris Volos et. al. [42] discusses behaviour of locks and TM when used together in a code based on five pathologies viz. blocking, deadlocks, livelocks, early release and invisible locking. The proposed method uses a modified lock implementation and extension of conflict detection policy of a HTM. The approach is a transaction-safe locks approach where a lock is accessed both within and outside a transaction. The paper concludes that the pathologies occur because of any of the following:

(1) Transaction conflict resolution is un-aware of locks

(2) Lock variables may be locked both at the memory level (by TM) and logical level (by locks)

(3) System does not respect lock semantics during abort of commit operations.

Polina Dudnik et al. [43] discuss how transactional memory implementations do not consider conditional variables mechanism in their design. They propose alternative methods to implement conditional variable so as to suit TM.

IV. CONCLUSIONS

In this paper we present taxonomy for TM systems based on parameters like conflict detection, version management, energy, memory model and isolation. We also give a brief introduction to other approaches apart from locks and TM.

We observe that hardware implementations adopt eager conflict detection-eager version management approach. On the other hand, software approaches prefer eager conflict detection and lazy or eager version management.

Secondly, most hardware proposals choose the victim based on timestamp or write-set resolution policy. An equally large number resort to a software contention manager, most of them being software proposals.

As seen in the Venn diagram, comparatively few proposals have implemented TM with support from Operating System. Future implementations are likely to use Operating System support considering the advantages it offers.

From Table 2, we see that Hardware proposals prefer cache-line granularity while software proposals prefer object granularity. Most proposals prefer a shared memory model. This is probably due to insufficient intra-processor bandwidth. However as newer systems promise higher bandwidth, proposals which use a combination of both such as TCC [17], LogTM [20], EazyHTM [27], TTM [30] can be used in future.

We conclude that for implementing TM in embedded system we can have a number of different approaches. L1 with victim cache is beneficial in the scenario where there are memory constraints. Eager and Lazy conflict detection policies are used in different data conflict rates. In most of the cases aggressive shutdown mode is used for energy efficiency [36].

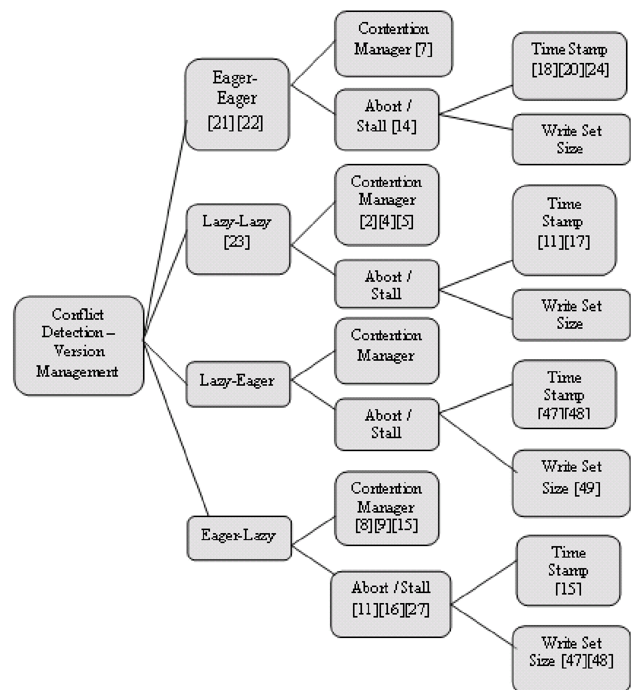

Figure 4: Classification of proposals based on conflict related parameters

Table 2: Classification based on other approaches

Scheme	***Isolation***	***Nesting***	***Granularity***	***Memory model***
DSTM [9]	Weak	Flattened	Object	Shared
ASTM [10]	Weak	Not supported	Object	Shared
McRT STM [14]	Weak	Closed	Cache-line or object	Shared
TL2 [11]	Weak	Not supported	Word, object	Shared
DracoSTM [16]	Weak	Closed	Object	Shared
Strongly Atomic STM [12]	Strong	Closed	Object	Shared
Elastic Transactions [15]	Weak	Supported	Word	Shared
Swiss TM [13]	Weak	Not supported	Word	Shared
HyTM [2]	Weak	Flattened	Word, Cacheline	Shared
Hybrid TM [1]	Weak	Flattened	Object, Cacheline	Shared
PhTM [6]	Weak &Strong	Sometimes supports	Word, Cacheline	Shared
NZTM [5]	Weak	Not supported	Object, Cacheline	Shared
SigTM [4]	Strong	Supported	Word, Cacheline	Message passing
UFO hybrid TM [7]	Strong	Flattened	Cacheline	Shared
SpHT [8]	Strong	Open & Closed	Cacheline	Shared
LogTM_SE [22]	Strong	Open & Closed	Block, Page	Shared
OneTM [21]	Strong	Flattened	Cacheline	Shared
TCC [17]	Strong	Flattened	Object, Cacheline	Shared and message passing
TokenTM [28]	Strong	-	Block	Shared
LTM [18]	Strong	Flattened	Cacheline	Shared
UTM [18]	Storng	Flattened	Cacheline	Shared
VTM [19]	Strong	Flattened	Cacheline	Shared
LogTM [20]	Strong	Flattened	Word, Cacheline	Shared and message passing
HMTM [23]	Strong	-	Cacheline	Shared
HTMOS [25]	Weak	flattened	Cacheline	Shared
EazyHTM [27]	Strong	Not specified	Cacheline	Shared and message passing
TTM [30]	Strong	flattened	Word, Cacheline	Shared and Message passing

V. REFERENCES:

[1] S.Kumar, M.Chu, C. J. Hughes, P. Kundu, and A.Nguyen, “Hybrid Transactional Memory”, 11th ACM Symposium on Principles and Practice of Parallel Programming, March 2006.

[2] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, D. Nussbaum, “Hybrid transactional memory”, 12th International Conference on Architectural Support for Programming Languages and Operating Systems, 2006, pp. 336–346.

[3] Shriraman, M. F. Spear, H. Hossain, V. Marathe,S. Dwarkadas, M. L. Scott, “An integrated hardware-software approach to flexible transactional memory”, 34th International Symposium on Computer Architecture, 2007, pp. 104–115.

[4] Cao Minh, M. Trautmann, J. Chung, A. McDonald,N. Bronson, J. Casper, C. Kozyrakis, K. Olukotun, “An effective hybrid transactional memory system with strong isolation guarantees”, 34th International Symposium on Computer Architecture, 2007, pp. 69–80.

[5] F. Tabba, M. Moir, J. R. Goodman, A. W. Hay, C. Wang, “NZTM: Nonblocking zero-indirection transactional memory”,21st Symposium on Parallelism in Algorithms and Architectures, 2009, pp. 204–213.

[6] Y. Lev, M. Moir, D. Nussbaum, “PhTM: Phased transactional memory” Workshop on Transactional Computing (TRANSACT), 2007.

[7] L. Baugh, N. Neelakantam, C. Zilles, “Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory“, 35th International Symposium on Computer Architecture, 2008, pp. 115–126.

[8] Y. Lev, J.W. Maessen, “Split hardware transactions: true nesting of transactions using best-effort hardware transactional memory”, 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2008, pp. 197–206.

[9] M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, “Software transactional memory for dynamic-sized data structures”, 22nd Symposium on Principles of Distributed Computing, 2003, pp. 92–101.

[10] V. J. Marathe, W. N. Scherer III, M. L. Scott, “Adaptive software transactional memory”, 19th International Symposium on Distributed Computing, Vol. 3724 of Lecture Notes in Computer Science, 2005, pp. 354–368.

[11] Dice, O. Shalev, N. Shavit, “Transactional locking II”, Proc. of the 20th Int’l Symp. Distributed Computing, Vol. 4167 of Lecture Notes in Computer Science, 2006, pp. 194–208.

[12] M. Abadi, T. Harris, M. Mehrara, “Transactional memory with strong atomicity using off-the-shelf memory protection hardware”, 14th ACM Symposium on Principles and Practice of Parallel Programming, 2009, pp. 185–196.

[13] Dragojevi´c, R. Guerraoui, M. Kapalka, “Stretching transactional memory”, ACM SIGPLAN Conference on Programming Language Design and Implementation, 2009, pp. 155–165.

[14] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, B. Hertzberg, “McRT-STM: a high performance software transactional memory system for a multi-core runtime”, 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2006, pp. 187–197.

[15] P.Felber, V. Gramoli, R. Guerraoui, “Elastic transactions”, 23rd International Symposium on Distributed Computing, Vol. 5805, 2009, pp. 93–107.

[16] J.E. Gottschlich, D. A. Connors, “DracoSTM: a practical C++ approach to software transactional memory”, Symposium on Library-Centric Software Design, 2007, pp. 52–66.

[17] 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”, 31st Annual International Symposium on Computer Architecture, June 2004.

[18] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, S. Lie, “Unbounded transactional memory”, 11th International Symposium on High-Performance Computer Architecture, 2005, pp. 316–327.

[19] R. Rajwar, M. Herlihy, K. Lai, “Virtualizing transactional memory”, 32nd International Symposium on Computer Architecture, 2005, pp. 494–505.

[20] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, D. A. Wood, “LogTM: Log-based transactional memory”, 12th International Symposium on High-Performance Computer Architecture, 2006, pp. 254–265

[21] C. Blundell, J. Devietti, E. C. Lewis, M. M. K. Martin, “Making the fast case common and the uncommon case simple in unbounded transactional memory”, 34th International Symposium on Computer Architecture, 2007, pp. 24–34.

[22] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, D. A. Wood, “LogTM-SE: Decoupling hardware transactional memory from caches”, 13th International Symposium on High-Performance Computer Architecture, 2007, pp. 261–272.

[23] M. Herlihy, J.E.B. Moss, “Transactional memory: architectural support for lock-free data structures”, 20th Annual International Symposium on Computer Architecture, 1993, pp. 289–300.

[24] K.E. Moore, M.D. Hill, and D.A. Wood, “Thread-level transactional memory”, Technical Report: CS-TR-2005-1524, Dept. of Computer Sciences, University of Wisconsin, Mar. 2005.

[25] Sasa Tomic , Adrian Cristal , Osman Unsal , Mateo Valero, “Hardware transactional memory with operating system support”, Conference on Parallel processing, 2007

[26] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood, “Performance Pathologies in hardware

transactional memory", 34th International Symposium on Computer Architecture, June 2007.

[27] S. Tomi´c, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, M. Valero, "EazyHTM: Eager-lazy hardware transactional memory", 42nd International Symposium on Microarchitecture, 2009, pp. 145–155.

[28] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, D. A. Wood, "TokenTM: Efficient execution of large transactions with hardware transactional memory", 35th International Symposium on Computer Architecture, 2008, pp. 127–138.

[29] N.Shavit, D.Touitou, "Software transactional memory", 14th ACM Symposium on Principles of Distributed Computing, 1995, pp. 204–213.

[30] T.Riegel, C.Fetzer, P. Felber, "Time-based transactional memory with scalable time bases", 19th ACM Symposium on Parallelism in Algorithms and Architectures, 2007, pp. 221–228.

[31] M. Herlihy, V. Luchangco, M. Moir, "A flexible framework for implementing software transactional memory", 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, 2006, pp. 253–262.

[32] Craig Zilles, Lee Baugh, "Extending Hardware Transactional Memory to Support,Nonbusy Waiting and Nontransactional Actions", Computer Science Department,University of Illinois at UrbanaChampaign

[33] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman, "Compiler and runtime support for efficient software transactional memory", ACM SIGPLAN Conference on Programming Language Design and Implementation, 2006, pp. 26–37.

[34] T. Harris and K. Fraser, "Language support for lightweight transactions", 18th Conference on Object-Oriented Programming, Systems, Languages, and Applications, Oct. 2003, pp. 388–402

[35] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi, "Optimizing memory transactions", ACM SIGPLAN Conference on Programming Language Design and Implementation, 2006, pp. 14–25

[36] Cesare Ferri, Samantha Wood, Tali Moreshet, Iris Bahar and Maurice Herlihy, "Embedded-TM: Energy and Throughput Efficient Transactional Memory for Embedded Multicore Systems", Journal of Parallel and Distributed Computing, Vol 70, Issue 10, Oct 2010,pp,1042-1052

[37] Tali Moreshet, R. Iris Bahar and Maurice Herlihy, "Energy-Aware Microprocessor Synchronization: Transactional Memory vs. Locks", Workshop on Memory Performance Issues, held in conjunction with the International Symposium on High-Performance Computer Architecture, February 2006.

[38] Cesare Ferri, R. Iris Bahar, Tali Moreshet, Amber Viescas and Maurice Herlihy, "Energy Efficient Synchronization Techniques for Embedded Architectures", ACM Great Lakes Symposium on VLSI, May 2008.

[39] Tali Moreshet, R. Iris Bahar and Maurice Herlihy, "Energy Reduction in Multiprocessor Systems Using Transactional Memory", International Symposium on Low Power Electronics and Design, August 2005.

[40] Cesare Ferri, Amber Viesdcas, Tali Moreshet, R. Iris Bahar and Maurice Herlihy, "Energy Implications of Transactional Memory for Embedded Architectures", Workshop on Exploiting Transactional Memory and Other Hardware-Assisted methods, April 2008.

[41] Dave Dice, Nir Shavit, "TLRW: Return of Read-Write Lock", 22nd ACM Symposium on Parallelism in Algorithms and Architectures, 2010, pp. 284-293

[42] Haris Volos, Neelam Goyal and Michael M. Swift, "Pathological Interaction of Locks with Transactional Memory", 3rd ACM SIGPLAN Workshop on Transactional Memory (TRANSACT), February 2008.

[43] Polina Dudnik and Michael M. Swift, "Condition Variables and Transactional Memory - Problem or opportunity", 4th ACM SIGPLAN Workshop on Transactional Computing, TRANSACT 2009.

[44] Walther Maldonado, Patrick Marlier,Pascal Feber, Danny handler, Adi Suissa, Alexndra Fedorova,Julia L. lawall,, Gillies Muller ,Danny handler," Scheduling Support for Transactional Memory Contention Management",Principals and practice of parallel programming ,Procedings of the 15th,ACM SIGPLAN symposium on Principals and practice of parallel programming,2010,banglore,India

[45] Takayuki Usui and Yannis Smaragdakis and Reimer Behrends,"Adaptive Locks: combining Transactions and Locks for Efficient Concurrency", in: TRANSACT~'09: 4th Workshop on Transactional Computing.

[46] Chen Fu, Dongxin Wen, Xiaoqun Wang and Xiaozong Yang, "Hardware Transactional Memory: A high performance parallel programming model", The EUROMICRO Journal of Systems Architecture, Volume 56 , Issue 8, Aug 2010, Pages: 384-391

[47] Anurag Negi, MM Waliullah and P. Stenstrom, "LV*: A low complexity Lazy Versioning HTM Infrastructure" ", In proceedings of 10th IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS), Greece, July 2010.

[48] Anurag Negi, MM Waliullah and P. Stenstrom, "LV*: A Class of Lazy Versioning HTMs for Low-Cost Integration of Transactional Memory Systems" 2nd ACM International Forum on Next Generation Multicore/Manycore Technology, in conjunction with ISCA 2010, Saint Malo, France, June 2010.

[49] Gramoli Vincent, Guerraoui Rachid, Letia Mihai, "The Many Faces of Transactional Software Composition", Technical Report, EPFL-REPORT-150654, Aug 2010