International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Software Transactional Memory: A Comprehensive Review of Design, Challenges, Applications, and Future Prospects

Meenu

Submitted:13/03/2024 **Revised**: 28/04/2024 **Accepted**: 05/05/2024

Abstract: This paper provides a comprehensive review of Software Transactional Memory (STM) systems, emphasizing their evolution, design, challenges, and applications. STM has emerged as a key solution for managing concurrency in modern software, offering a flexible alternative to traditional synchronization methods. The study traces the evolution of Transactional Memory models, including Hardware (HTM), Software (STM), and Hybrid (HyTM), comparing their features, benefits, and limitations. It explores crucial design elements affecting STM's performance, such as contention management, concurrency control, and memory overhead, while addressing the complexities of nested transactions and ensuring global consistency. The paper highlights STM's versatility, showcasing applications in diverse domains that benefit from its ability to enable scalable and high-performance parallel programming. It also examines challenges such as scalability, optimization, and integration with existing systems, presenting opportunities for future research. Proposed directions include improving STM's efficiency, scalability, and adoption in real-world scenarios. By summarizing the advancements and limitations of STM, this study underscores its role as a powerful tool for enhancing concurrency control in parallel computing. It serves as a valuable resource for researchers and practitioners aiming to optimize software systems through improved concurrency mechanisms.

Index Terms: *Concurrency Control, Nested Transactions, Parallel Programming, Software Transactional Memory (STM), Transactional Memory Models.*

I.INTRODUCTION

This section highlights the challenges of concurrency in multicore systems and introduces Transactional Memory (TM) as a solution [1]. It explains the core principles of TM—atomicity and isolation—and discusses three TM models: HTM, STM, and HyTM. The section also outlines the paper's structure.

The advent of multicore and multiprocessor systems has revolutionized parallel programming, enabling significant performance gains in modern computing [2]. However, these advancements bring inherent challenges, particularly in managing concurrent access to shared memory. Efficiently coordinating multiple threads accessing shared resources is vital to harness the full potential of multicore systems. Traditional synchronization mechanisms, such as locks, semaphores, and monitors, are widely used to handle concurrency. Yet, they suffer from well-documented drawbacks, including deadlocks, convoying, priority inversion, and complexity in fault tolerance [3] [4]. These issues often result in reduced system efficiency, making it difficult for developers to achieve optimal performance in parallel computing environments. To address these challenges, Transactional Memory (TM) has emerged as a promising paradigm for simplifying concurrent programming [5]. TM systems allow multiple threads to perform operations on shared memory through transactions—a sequence of operations executed atomically and in isolation. This abstraction eliminates the need for manual lock management, reducing complexity and minimizing common synchronization problems. TM provides an intuitive and robust approach to managing concurrency, making it easier for developers to design and implement high-performance parallel applications.

Department of CSE, M. M. M. U. T., Gorakhpur, India
**myself_meenu@yahoo.co.in*

Transactional Memory operates by ensuring two core properties for transactions:

A. ATOMICITY: Ensures that a transaction's operations are completed entirely or not at all, preventing partial updates to shared memory.

B. ISOLATION: Guarantees that intermediate states of a transaction are not visible to other concurrent transactions, preserving consistency.

These properties enhance reliability and predictability in parallel applications, especially in scenarios where multiple threads interact frequently with shared resources. Over the years, researchers have developed various models of TM to address specific needs and limitations in different environments. These models include:

A. HARDWARE TRANSACTIONAL MEMORY (HTM): Relies on specialized hardware to manage transactions [6].

B. SOFTWARE TRANSACTIONAL MEMORY (STM): Implements TM purely through software mechanisms [7].

C. HYBRID TRANSACTIONAL MEMORY (HYTM): Combines hardware and software approaches to leverage their respective strengths [8].

Each model has unique advantages and limitations, making them suitable for different types of applications and system configurations.

This paper focuses primarily on Software Transactional Memory (STM), exploring its evolution, design, challenges, and applications. It reviews the development of TM models, with particular emphasis on STM's role in optimizing parallel programming. The paper aims to guide researchers and developers toward more efficient and scalable STM implementations.

The structure of the paper is organized as follows: Section 2 explores the evolution of Transactional Memory (TM), focusing on the three primary models—Hardware Transactional Memory (HTM), Software Transactional Memory (STM), and Hybrid Transactional Memory (HyTM)—and provides a comparative analysis of their features, advantages, and challenges. Section 3 delves into the critical design parameters that influence the performance and behaviour of STM systems. Section 4 examines the challenges associated with implementing nested transactions in TM systems and discusses potential solutions. Section 5 highlights various applications of STM, showcasing its relevance across different domains. Section 6 outlines future directions for STM, emphasizing innovative approaches to address current limitations and improve its scalability and efficiency. Finally, Section 7 concludes the paper by summarizing the key insights and contributions.

II.EVOLUTION of TRANSACTIONAL MEMORY

This section covers the evolution of Transactional Memory (TM), focusing on three models: HTM, STM, and HyTM. It compares their features, advantages, and challenges, summarized in Table I, helping to understand their practical applications and trade-offs.

Transactional Memory (TM) has evolved into three distinct models, each with unique structures, advantages, limitations, and challenges. These models—Hardware Transactional Memory (HTM), Software Transactional Memory (STM), and Hybrid Transactional Memory (HyTM)—address different application needs and system constraints. Their comparative analysis is outlined in Table I, offering a comprehensive view of their core characteristics and trade-offs.

TABLE I

COMPARATIVE EVALUATION OF VARIOUS TRANSACTIONAL MEMORY MODELS

S.No.	Transaction Model	Transaction Structure	Merits	Demerits	Challenges
1.	Hardware Transactional Memory (HTM)	Hardware-based	High efficiency with minimal overhead	Dependent on hardware, limited scalability	Complexity in implementation
2.	Software Transactional Memory (STM)	Software-based	Simple to program, offers flexibility	Higher overhead costs, conflicts in access, metadata management	Durability is unnecessary but adds overhead
3.	Hybrid Transactional Memory (HyTM)	Combination of hardware and software	Combines advantages of both HTM and STM	Increased complexity in managing the hybrid system	Need for dynamic adaptation between HTM and STM

In conclusion, Transactional Memory (TM) models include Hardware Transactional Memory (HTM), Software Transactional Memory (STM), and Hybrid Transactional Memory (HyTM), each with distinct features and challenges. HTM utilizes hardware components for transaction management, offering high efficiency and low overhead but faces scalability limitations and implementation complexities due to hardware dependency. STM operates at the software level, providing flexibility and ease of programming, yet incurs higher overhead from metadata maintenance and access conflicts, which can impact performance despite not requiring durability. HyTM combines hardware and software advantages, enhancing adaptability and performance, but the integration introduces significant complexity, particularly in dynamically adapting between HTM and STM systems. The comparative analysis in Table I provides a clear and concise overview of the TM models, aiding in understanding their practical applications, strengths, and areas where further improvements are necessary. Each model's adoption depends on the specific needs of the application, balancing trade-offs between performance, flexibility, and implementation challenges.

In conclusion, the evolution of Transactional Memory (TM) has resulted in the development of three distinct models—HTM, STM, and HyTM—each tailored to different system requirements and application needs. While HTM offers high efficiency and low overhead through hardware-based transaction management, it is limited by scalability and complexity. STM, on the other hand, provides flexibility and ease of programming but incurs performance overhead due to metadata maintenance and access conflicts. HyTM combines the strengths of both HTM and STM, enhancing adaptability and performance but adding complexity in managing the hybrid system. The comparative analysis of these models highlights their unique trade-offs, and the choice of model depends on the specific requirements of the application, balancing performance, scalability, and implementation complexity. Further advancements in TM will likely focus on optimizing these trade-offs to achieve more efficient and scalable concurrency solutions.

III. ARCHITECTURAL ASPECTS of SOFTWARE TRANSACTIONAL MEMORY

This section examines the key design aspects that significantly affect the performance and behaviour of Software Transactional Memory (STM) systems. It emphasizes the importance of understanding these parameters to fine-tune STM implementations for specific application needs, ultimately enhancing concurrency management, efficiency, and reliability.

STM systems are shaped by various interconnected factors that influence transaction handling, conflict detection, memory management, and contention resolution. A thorough understanding of these design parameters is crucial for developers seeking to optimize STM systems for different application scenarios, such as real-time systems, high-performance computing, or distributed environment [6] [9] [10]. s. These parameters directly impact how STM systems manage transactions, detect conflicts, handle memory, and resolve contention. The way these factors interact contributes to the overall efficiency and stability of STM systems, particularly in high-concurrency environments. By carefully selecting and adjusting these design parameters, developers can tailor STM systems to meet the specific needs of diverse applications, ensuring optimal performance and reliability. The following Table II summarizes these key design parameters and their associated examples and considerations.

TABLE II

STM DESIGN ASPECTS

S.No.	Category	Description	Examples/Notes
I.	Transaction Granularity	The basic unit over which STM detects conflicts.	Word-based STM: Detects conflicts at the word level (high accuracy but high cost). Object-based STM: Uses object-level granularity (easier to implement, lower cost). e.g. STM Haskell uses object based.
II.	Update Policy	Defines how a transaction updates an object.	Direct Update: Direct modification of the object. Deferred Update: Updates made to a private copy, applied at commit time. STM Haskell uses deferred update.
III.	Read Policy [11].	Defines how transactions read shared resources.	Invisible Reads: No conflict detection until commit. Visible Reads: Locks and reader lists used. STM Haskell uses visible reads.
IV.	Acquire Policy [11].	Defines how transactions acquire shared resources.	Eager Acquire: Transaction acquires and modifies resources immediately. Lazy Acquire: Modifies memory at commit time (better for buffered writes).
V.	Write Policy	Defines how transactions write changes to memory.	Write-through or Undo: Direct writes to shared memory, but more costly on abort.

			Buffered Write: Writes occur only on successful commit.
VI.	Conflict Detection	Identifies conflicts when multiple transactions try to operate on the same object.	Early Conflict Detection: Detects conflicts before commit. Late Conflict Detection: Detects conflicts at commit time. e.g. STM Haskell uses lazy conflict detection.
VII.	Concurrency Control [12]	Manages simultaneous transactions accessing shared resources.	Pessimistic Concurrency Control: All events (conflict occurrence, detection, and resolution) happen during execution. A transaction claims exclusive access to a resource and prevents others from accessing it. Two-Phase Locking (2PL): Transactions acquire a lock before accessing resources. Optimistic Concurrency Control: Allows concurrent access to resources. Conflicts are detected and resolved only before a transaction commits. Blocking Synchronization (Lock-based): Transactions are blocked until they acquire a lock on a resource, ensuring exclusive access. Does not guarantee forward progress for all threads. Lock-based STM: Transactions are blocked until a lock is acquired for accessing shared resources. Non-blocking Synchronization: Guarantees that threads can make progress without blocking each other. Includes wait-free, lock-free, and obstruction-free techniques. Wait-free STM: Guarantees that all threads make progress without waiting. Lock-free STM: At least one thread progresses even if others are stalled. Obstruction-free STM: Progress is made when there is no contention between threads.
VIII.	Memory Management [11]	Manages allocation and deallocation of memory used in transactions.	Proper handling of memory allocation and deallocation to prevent memory leaks and ensure recovery on transaction failure.
IX.	Contention Management	Resolves conflicts when transactions compete for resources.	e.g. Timid: Always aborts a transaction on conflict. [13] Polka: Backs off based on priority difference. [14] Greedy: Guarantees commits within bounded time. [15] Serializer: Like greedy but with priority adjustment. [16] [11]
X.	Isolation	Ensures one transaction does not interfere with another.	e.g. STM Haskell uses weak isolation, allowing some transactions to access shared resources outside the atomic block.
XI.	Nesting Model [9] [17]	Supports composability and nested transactions.	Flattening: Transactions are flattened into the outermost level, and sub-transactions are managed by the outer transaction. e.g. DSTM (Dynamic STM): Synchronizes dynamic data structures like lists and trees without locks [18] . RSTM: Provides flattened transactions to support nesting [19]. Linear Nesting: Hierarchical structure with one nested

			transaction active at a time. Both closed) [4], and open nested transactions [20] [21] are supported. Closed Nested Transactions (CNTs): e.g. Haskell STM: Uses type systems and supports retry and recovery mechanisms [22] [23] [24] [25] [26] [27] [28]. McRT-STM: Implemented in C++ and Java with closed nesting [29] . NOrec: Minimal overhead with closed nested transactions [30]. Nested LogTM: Supports both open and closed nested transactions [31] [32] . Open Nested Transactions (ONTs): e.g. ATOMOS: Java extension supporting open nested transactions with atomicity [33]. Parallel Nesting: Allows for multiple nested transactions to run in parallel, enabling more complex and independent tasks. e.g. NeSTM: Based on McRT-STM, supports parallel nesting [34]. HParSTM: Hierarchical STM with opacity and progressiveness [35]. NePalTM: Combines parallelism with atomic blocks using OpenMP and Intel STM [36]. CWSTM: Based on Cilk for multithreaded parallel programming [37] . PNSTM: Based on CWSTM with a simpler work-stealing approach [38] . SSTM: Based on .NET CLR, uses xfork API for managing sibling transactions [39].

In conclusion, the design parameters outlined in the Table II are crucial for shaping the behaviour and performance of Software Transactional Memory (STM) systems. Each parameter plays a vital role in determining how transactions are managed, conflicts are detected, and resources are allocated. The interplay between these factors must be carefully considered when optimizing STM systems for specific applications, ensuring that the system supports high concurrency, minimizes contention, and operates efficiently across various environments. By understanding the trade-offs and selecting the appropriate parameters, developers can achieve the desired balance between performance, reliability, and resource utilization. The careful consideration of STM's design choices enables the development of robust and scalable systems, particularly in scenarios that require complex transaction handling and high-performance computing. Thus, the successful implementation and optimization of STM systems depend on the nuanced understanding and application of these key design parameters.

IV.CHALLENGES AND ISSUES OF TRANSACTIONAL MEMORY SYSTEM

This section addresses the challenges encountered when implementing nested transactions in Transactional Memory (TM) systems, emphasizing key difficulties and their respective solutions [40]. The solutions are detailed in Table III to enhance system performance and ensure effective operation in complex transactional environments.

The Nested Transaction Model brings about specific challenges that can significantly hinder performance if not properly managed [41]. Therefore, implementing robust solutions is crucial for optimizing system efficiency and ensuring stability. Table III below outlines these challenges along with the proposed solutions, based on existing research [10] [34].aimed at improving the execution of nested transactions and minimizing the risk of system failures.

TABLE III

TRANSACTIONAL MEMORY (TM) CHALLENGES AND RESOLUTIONS

S.No	Issues in Transactional Memory (TM) systems	Description	Issues Resolution
1.	Transformation of Transactional Code	Non-transactional code may run as a transaction in STM.	Develop methods to separate or dynamically classify transactional and non-transactional code.
2.	Conflict Detection Scheme	Tracking dependencies hierarchically in nested parallel transactions is challenging.	Create a scheme that tracks dependencies hierarchically and manages conflicts without aborting the parent transaction.
3.	Memory Overhead	Minimizing memory overhead for tracking nested transactions.	Implement efficient memory management techniques to reduce tracking overhead.
4.	Single Level of Parallelism	Managing overhead in single-level parallelism applications.	Optimize STM to efficiently handle single-level parallelism and streamline resource allocation.

In conclusion, the solutions presented in Table III provide a comprehensive overview of how to address the challenges posed by the Nested Transaction Model in Transactional Memory (TM) systems. These solutions focus on enhancing system efficiency, reducing memory overhead, and ensuring smooth execution of nested transactions. By implementing these strategies, TM systems can overcome the complexities introduced by nested transactions and improve overall performance. The proposed resolutions, derived from existing research offer valuable insights for optimizing the management of nested transactions, minimizing the risk of failures, and maintaining system stability in complex transactional environments.

V.APPLICATIONS of SOFTWARE TRANSACTIONAL MEMORY

This section highlights the broad applications of Software Transactional Memory (STM) as a concurrency control mechanism, emphasizing its effectiveness in ensuring atomicity, consistency, and scalability across various computational domains.

Software Transactional Memory (STM) is a powerful concurrency control mechanism designed to simplify synchronization in multi-threaded environments. Unlike traditional locking mechanisms, which can introduce complexities such as deadlocks and performance bottlenecks, STM ensures that memory updates are atomic and consistent by executing transactions in isolation and committing them only when they are conflict-free. This approach not only improves scalability and performance but also provides a more straightforward programming model for managing shared memory.STM has proven its versatility across a variety of domains. In concurrent data structures, it enables safe and efficient access to shared resources, such as linked lists and hash maps, without relying on locks [7]. Similarly, in parallel computing, STM facilitates synchronization among tasks, allowing parallel algorithms to execute seamlessly without the overhead of traditional locking mechanisms [42] . Database systems also benefit from STM's capabilities, as it simplifies transaction management by ensuring atomic operations and enabling safe concurrent access to data structures in multi-threaded environments [27]. Functional programming languages, such as Haskell, leverage STM to maintain stateful computations while preserving immutability, thus providing a safe and predictable framework for multi-threaded programming [43]**.** High-performance computing (HPC) systems utilize STM to manage synchronization in large-scale scientific simulations, ensuring efficient memory updates across massive computational tasks [42] .STM is equally valuable in embedded systems, where its ability to handle atomic updates in resource-constrained environments makes it an efficient alternative to traditional synchronization methods [6] . Moreover, STM plays a crucial role in multicore and manycore systems by enabling atomic memory updates across multiple cores, simplifying synchronization, and enhancing parallel performance in shared memory environments [7].

In conclusion, Software Transactional Memory (STM) has revolutionized the management of concurrency in multi-threaded and parallel systems, offering an abstraction that ensures atomic, consistent, and deadlock-free synchronization. Its broad applications, from concurrent data structures and HPC to embedded systems and functional programming, underscore its critical role in modern computing. By abstracting traditional complexities and enhancing performance, STM represents a cornerstone technology for scalable and efficient system design. Its continued evolution promises to unlock new opportunities in concurrent programming, making it an indispensable tool for the future of computing.

VI. FUTURE RESEARCH DIRECTIONS

Software Transactional Memory (STM) has established itself as a transformative paradigm for managing concurrency in parallel computing. Its ability to simplify synchronization and eliminate challenges like deadlocks and race conditions positions it as a promising alternative to traditional lock-based methods. However, to unlock its full potential, several critical research areas demand attention. Scalability remains a pressing concern, especially in large-scale systems where performance bottlenecks arise due to contention in many-core processors and distributed environments. Developing hierarchical models, adaptive contention management strategies, and optimization techniques for distributed STM systems could address these challenges and enable efficient scaling across thousands of processors or nodes. Integrating STM with Hardware Transactional Memory (HTM) offers another promising avenue. While STM provides flexibility, it incurs overhead from software-based conflict detection, whereas HTM ensures faster transaction execution but lacks STM’s adaptability. Hybrid STM-HTM models that dynamically switch between the two based on workload characteristics can leverage the strengths of both, achieving optimal flexibility and performance. Furthermore, STM’s adaptation to real-time systems is crucial, particularly for applications with stringent timing constraints, such as embedded systems and robotics. Deadline-aware scheduling, real-time prioritization mechanisms, and integration with real-time operating systems are essential to ensure transaction completion within predefined time bounds. Fault tolerance is another critical challenge, especially for distributed STM systems operating in unreliable environments. Techniques like checkpointing, transactional snapshots, and undo logs can enhance reliability, allowing systems to recover seamlessly from failures. Similarly, energy efficiency is an area of growing importance, particularly in energy-constrained environments such as IoT and mobile systems. Lightweight STM designs and optimizations in conflict detection and retries are necessary to minimize power consumption while maintaining performance. High contention in STM systems often leads to frequent transaction rollbacks, degrading performance. Advanced conflict resolution algorithms, including transaction prioritization and reordering, can mitigate this issue and ensure smooth execution. Additionally, STM’s application-specific optimizations hold significant promise for domains like machine learning, high-frequency trading, and scientific computing, where tailored STM frameworks can address unique requirements for latency, throughput, and scalability. Security is increasingly vital as STM systems are deployed in distributed and cloud environments. Transaction-level encryption, secure conflict resolution protocols, and access control mechanisms must be seamlessly integrated to ensure data integrity and confidentiality. Another barrier to STM adoption is the complexity of its programming models. Developing high-level abstractions, intuitive libraries, and debugging tools can simplify its implementation, making STM more accessible to developers. Moreover, embedding STM capabilities into modern programming languages like Java, Python, and Rust, as well as frameworks like TensorFlow and Spark, will facilitate its integration into contemporary software ecosystems. Emerging technologies such as quantum computing, neuromorphic computing, and robotics also present exciting opportunities for STM research. Adapting STM principles to manage concurrency in these advanced domains can address unique challenges, such as quantum bit state management or neural architecture synchronization.

By addressing these research challenges, STM can overcome its current limitations, broaden its applicability, and evolve into a cornerstone technology for managing concurrency in modern and future computing systems. Its ongoing refinement and adaptation will ensure its continued relevance in the dynamic landscape of parallel computing.

VII. CONCLUSION

This paper provides a thorough examination of Software Transactional Memory (STM) and its significant role in enhancing concurrency control in multi-threaded systems. Introduction highlighted the fundamental aspects of STM, emphasizing its ability to simplify synchronization and provide an efficient alternative to traditional locking mechanisms. The concept of STM as a tool for managing memory transactions atomically and consistently was introduced, showcasing its potential for improving parallel computing environments. In the Evolution of Transactional Memory, the evolution of memory management techniques from traditional locking mechanisms to STM was explored. The development process was outlined, demonstrating how STM has addressed many of the limitations of earlier approaches, offering better scalability and flexibility for handling concurrent operations in multi-core systems. The Design Parameters of Software Transactional Memory were then discussed, focusing on key factors that influence the performance and efficiency of STM. These parameters, including transaction size, conflict detection, and rollback strategies, are essential to ensuring that STM systems perform optimally in various environments. The importance of these design choices in balancing performance and reliability was emphasized. The Issues and Challenges of TM Systems section examined the primary obstacles that STM faces when implementing nested transactions, such as conflict detection, memory overhead, and handling parallelism in nested transactions. Solutions such as dynamic conflict detection, efficient memory management, and optimizing single-level parallelism were proposed to mitigate these challenges, improving STM's overall performance. In the Applications of STM, the paper highlighted STM's versatility in various domains, including parallel computing, high-performance computing, functional programming, and embedded systems. Its ability to simplify synchronization and manage atomic updates in concurrent data structures made it invaluable across different fields. The wide range of applications demonstrates STM's flexibility and utility in modern computing. Finally, Future Research Directions addressed key challenges STM still faces, such as scalability, real-time system integration, fault tolerance, and energy efficiency. The importance of hybrid STM-HTM models, as well as research in conflict resolution algorithms and security mechanisms, was discussed. The section emphasized the need for future research to enhance STM's scalability, flexibility, and robustness in an increasingly diverse range of computing environments.

In conclusion, this paper has provided a comprehensive analysis of STM's evolution, its design considerations, current challenges, and its applications. It has also highlighted key areas for future research, offering valuable insights into how STM can continue to evolve and meet the growing demands of modern and future computing systems. These insights offer a solid foundation for ongoing research and development in Software Transactional Memory**,** ensuring its continued relevance in the field of concurrency control.

References

[1] H. Grahn, "Transactional memory," Journal of Parallel and Distributed Computing, vol. 70, no. 10, pp. 993-1008, 2010.

[2] J. B. K. C. L. K. R. a. Y. Z. J. R. Blumofe, "Cilk : An efficient multithreaded runtime system," Journal of Parallel and Distributed Computing, vol. 37, no. 1, pp. 55-69, August 1996.

[3] P. B. a. N. Goodman, "Concurrency Control in Distributed Database Systems," ACM Computing Surveys, vol. 13, no. 2, p. 185 – 221, 1981.

[4] R. a. M. M. S. Alexandru Turcu, " On closed nesting in distributed transactional memory," in Seventh ACM SIGPLAN workshop on Transactional Computing, 2012.

[5] J. R. L. a. R. R. T. Harris, Transactional Memory, 2 ed., Synthesis Lectures on Computer Architecture Morgan & Claypool Publishers, 2010, pp. 1-247.

[6] M. H. a. J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in Proceedings of the 20th annual international symposium on Computer architecture (ISCA '93)., May 1993.

[7] N. &. T. D. Shavit, "Software transactional memory," in Proceedings of the 14th Annual ACM Symposium on Principles of DistributedComputing, Ottawa, Can, 1995.

[8] F. Y. L. V. L. M. M. D. N. Peter Damron, "Hybrid transactional memory," in Proceedings of the 12th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006.

[9] N. C. J. Diegues, "Review of nesting in transactional memory," Tech. rep., Technical Report RT/1/2012, Instituto Superior Técnico/INESC-ID , 2012.

[10] G. A. Asi, "Performance Tradeoffs in Software Transactional Memory," Master Thesis Computer Science, School of Computing Blekinge Institute of Technology, No:MCS-2010-28, Sweden, May 2010.

[11] S. Classen, "LibSTM: A fast and flexible STM Library," Master's Thesis, Laboratory for Software Technology, Swiss Federal Institute of Technology, ETH Zurich, Feb, 2008.

[12] I. a. M. Raynal, "A Lock-Based STM Protocol That Satisfies Opacity and Progressiveness," in Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS'08, 2008.

[13] W. N. S. I. a. M. L. Scott, "Contention Management in Dynamic Software Transactional Memory," in Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs, Canada, July 2004.

[14] N. S. a. M. L. S. y, “Advanced contention management for dynamic software transactional memory,” in Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing, Las Vegas, NV, USA, 2005.

[15] e. a. R. Guerraoui, “Toward a theory of transactional contention managers,” in Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing, Las Vegas, NV, USA, 2005.

[16] M. L. Scott, “Applications Included with RSTM WebPage,” [Online]. Available: http://www.cs.rochester.edu/research/synchronization/rstm/applications.shtml.

[17] T. H. a. S. Stipic, “Abstract nested transactions,” in Second ACM SIGPLAN Workshop on Transactional Computing, 2007.

[18] M. &. L. V. &. M. M. &. S. W. Herlihy, “ Software Transactional Memory for Dynamic-Sized Data Structures,” in Proceedings of the Annual ACM Symposium on Principles of Distributed Computing, 2003.

[19] M. S. C. H. A. A. D. E. W. S. I. a. M. S. V. Marathe, “Lowering the overhead of Software Transactional Memory,” in 1st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT '06), 2006 .

[20] T. a. B. Ravindran, “ On open nesting in distributed transactional memory,” in 5th Annual International Systems and Storage Conference (SYSTOR) ’12, 2012.

[21] S. M. A.-R. A.-T. A. L. H. R. L. H. J. E. B. M. S. a. T. S. Y. Ni, “Open nesting in software transactional memory,” in PPoPP ’07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming ,ACM Press, New York, NY, USA, 2007.

[22] R. C. Ammlan Ghosh and Haskell, Implementing Software Transactional Memory using STM, vol. 2, Advanced Computing and Systems for Security ,Springer AISC, 2016, pp. 235-248.

[23] M. R. Y. a. M. F. Le, “Revisiting software transactional memory in Haskell,” ACM SIGPLAN Notices, vol. 51, no. 12, pp. 105-113, 2016.

[24] Du Bois, “An Implementation of Composable Memory Transactions in Haskell,” in Software Composition, SC 2011,Lecture Notes in Computer Science,Springer, Berlin, Heidelberg., 2011.

[25] H. T. M. S. J. S. S. S. Discolo, “Lock Free Data Structures Using STM in Haskell,” in Functional and Logic Programming, FLOPS , 2006.

[26] M. L. V. &. M. M. Herlihy, “A flexible framework for implementing software transactional memory,” ACM SIGPLAN Notices, vol. 41, no. 10, pp. 253-262, 2006.

[27] S. M. S. P. J. a. M. H. T. Harris, “Composable memory transactions,” in Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’05, Chicago, IL, USA, 2005.

[28] G. a. S. F. S. Peyton Jones, “Concurrent Haskell,” in 23rd ACM Symposium on Principles of Programming Languages (POPL’96), 1996.

[29] A.-R. A.-T. R. H. C. C. M. a. B. H. B. Saha, “McRT-STM: a high-performance Software Transactional Memory system for a multi-core runtime,” in SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06), 2006.

[30] M. S. a. M. S. L. Dalessandro, “NOrec: Streamlining STM by abolishing ownership records,” in Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10), 2010.

[31] B. M. M. M. H. a. D. W. K. Moore, “LogTM: log-based transactional memory,” in Proceedings of the 12th High-Performance Computer Architecture International Symposium (HPCA '06), 2006.

[32] B. K. E. M. L. Y. M. D. H. B. L. M. M. S. a. D. M. J. Moravan, “Supporting Nested Transactional Memory in LogTM,” in 12th International Conference on Architectural Support for Programming Languages and Operating Systems in SIGPLAN Notices (Proceedings of the 2006 ASPLOS Conference), 2006.

[33] M. H. C. J. C. C. M. C. K. a. K. O. B. Carlstrom, “The ATOMOS Transactional Programming Language,” in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06), 2006.

[34] N. B. C. K. a. K. O. W. Baek, “Implementing and evaluating nested parallel transactions in software transactional memory,” in Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA ’10,, Thira, Santorini, Greece, 2010.

[35] R. K. a. K. Vidyasankar, “HParSTM: A Hierarchy-based STM Protocol for Supporting Nested Parallelism,” in 6th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT '11), 2011.

[36] W. A.-R. A.-T. T. S. X. T. a. R. N. H. Volos, “NePaLTM: Design and Implementation of Nested Parallelism for Transactional Memory Systems,” in Proceedings of the 23rd European Conference on Object-Oriented Programming(ECOOP '09), 2009.

[37] J. T. F. a. J. S. K. Agrawal, “Nested parallelism in transactional memory,” in Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08), 2008.

[38] D. P. F. R. G. a. M. K. J. Barreto, “ Leveraging parallel nesting in transactional memory,” in Proceedings of the 15th ACM SIGPLAN

Symposium on Principles and Practice of Parallel Programming (PPoPP '10).

[39] H. R. a. E. Witchel, “The xfork in the road to coordinated sibling transactions,” in 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT '09), 2009.

[40] L. H. J. Eliot B. Moss, “Nested transactional memory: Model and architecture sketches,” Science of Computer Programming, vol. 63, no. 2, pp. 186-201, 2006.

[41] J. E. B. Moss, “Nested Transactions: An Approach to Reliable Distributed Computing,” Ph.D. Thesis, Technical Report MIT/LCS/TR-260,MIT Laboratory for Computer Science, Cambridge, MA, April 1981.

[42] T. L. V. M. B. R. M. B. S. a. T. S. Ali-Reza Adl-Tabatabai, “Compiler and runtime support for efficient software transactional memory,” in Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada , 2006.

[43] S. Peyton-Jones, Beautiful concurrency, A. O. a. G. Wilson, Ed., O'Reilly, 2007.

Mrs. Meenu is an Associate Professor in the department of Computer Science & Engineering at the Madan Mohan Malaviya University of Technology, Gorakhpur where she has been a faculty member since 2003. She is Chairperson of Women Cell as well as Women Welfare and AntiHarassment Cell. She completed her M.Tech. at Madan Mohan Malaviya University of Technology. She has served as the Session Chair for UPCON-2018 (5th IEEE Uttar Pradesh Section International Conference). She is the author of 64 research papers, which have been published in various National & International Journals/Conferences. She is a reviewer of many International Journals/ Conferences and Editorial Board member of International Journals. She is also member of many Professional Societies. Her research interest lies in the area of Distributed Real Time Database Systems. She has collaborated actively with researchers in several other disciplines of computer science, particularly machine learning.