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

Transactional Memory: A Comprehensive Review of Implementation, Applications, Performance, Challenges, Framework Comparisons, and Future Prospects

Meenu

Submitted:11/03/2024 **Revised**: 26/04/2024 **Accepted**: 03/05/2024

Abstract Transactional Memory (TM) offers a high-level synchronization abstraction for parallel programming, improving scalability, reliability, and productivity. It addresses challenges in multicore and distributed systems, surpassing traditional methods like locks and semaphores. TM implementation strategies—Software Transactional Memory (STM), Hardware Transactional Memory (HTM), and Hybrid Transactional Memory (HyTM)—present trade-offs in performance, scalability, and adaptability, catering to diverse workloads. Advanced features, including Nested Transactions, enhance fault tolerance and minimize rollback costs through modular transaction management.TM's lock-free synchronization finds applications in concurrent data structures, graph algorithms, scalable systems, and real-time computing, boosting reliability and system performance. Performance analyses of STM, HTM, and HyTM highlight their strengths and limitations in handling varying workloads. However, challenges persist, such as programming model integration, contention management, and efficiently managing large or nested transactions. Innovations like Dynamic STM, Adaptive Conflict Resolution, and extended HTM support tackle these issues, advancing TM capabilities. Frameworks such as TCC and LogTM, along with STM and HTM implementations, illustrate TM's evolution. Future research aims to overcome current limitations, ensuring TM's role in high-performance computing, real-time systems, and large-scale data processing. TM simplifies synchronization, empowering parallel programming to meet modern and future system requirements efficiently.

Index Terms: *Concurrency Management, Nested Transactions, Parallel Programming, Software Transactional Memory (STM), Transactional Memory (TM)*

I.INTRODUCTION

This section introduces the challenges of synchronization in multicore systems and presents Transactional Memory (TM) as a scalable solution, highlighting its principles, motivations, and potential to simplify parallel programming [1].

The shift from single core to multicore processors has fundamentally transformed the landscape of computing, enabled the parallel execution of tasks and delivered substantial performance improvements [2]. However, this transition introduces new challenges in parallel programming, particularly in managing access to shared resources. Effective synchronization among concurrent threads is critical, but ensuring correctness and efficiency in this context remains a complex task. Traditional synchronization mechanisms, such as locks and semaphores, have been widely used but often come with significant drawbacks. These include issues like deadlocks, livelocks, priority inversion, and poor composability, which can severely hinder scalability and complicate the development of reliable parallel programs [3] [4]. To overcome these challenges, more advanced techniques have been introduced, with Transactional Memory (TM) emerging as a leading solution [5]. TM provides a novel abstraction that simplifies synchronization in parallel programming, making it especially suited for the needs of multicore systems. This survey examines the core concepts, motivations, and practical applications of TM, highlighting its potential to revolutionize parallel programming by offering a more efficient and manageable synchronization model.

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

A. TRANSACTIONAL MEMORY

This section introduces Transactional Memory (TM) as a solution to overcome the limitations of traditional synchronization methods in parallel programming.

As multicore processors become the norm in modern computing, parallel programming has become a necessity. However, traditional synchronization methods like locks and semaphores often fall short when faced with challenges such as deadlocks, priority inversion, and reduced composability. These limitations create significant bottlenecks, impacting scalability and the reliability of parallel programs. Transactional Memory (TM) presents an innovative alternative to these conventional synchronization methods by replacing locks with transactional execution. TM ensures that critical sections of code are executed atomically and in isolation, simplifying synchronization, reducing contention, and

improving overall reliability. This survey provides an in-depth exploration of the foundational principles and practical implications of TM, synthesizing insights from key research to offer a comprehensive analysis of its capabilities.

B. CORE PRINCIPLES AND MOTIVATION

This section highlights the core principles of Transactional Memory (TM), focusing on its ability to simplify synchronization and improve scalability in multicore systems.

The core idea behind Transactional Memory is to treat a series of operations on shared data as a single atomic transaction. In this model, transactions either commit (complete fully) or abort (revert all changes), ensuring atomicity, consistency, and isolation of operations. Unlike traditional lock-based approaches, TM abstracts away the complexities of manual synchronization, providing a model that is more composable and scalable. This abstraction is especially important as the adoption of multicore systems continues to rise. Traditional synchronization models struggle to meet the demands of these systems, leading to errors, performance bottlenecks, and challenges in maintaining modularity. TM provides an elegant solution to these issues by ensuring safe concurrency and improved performance, making it a highly relevant tool in modern parallel programming.

In conclusion, Transactional Memory represents a fundamental shift in how parallel programming challenges are addressed in multicore systems. By simplifying synchronization and offering a more scalable and composable approach, TM enhances both the reliability and performance of concurrent applications. Its ability to abstract synchronization complexities allows developers to focus on higher-level program logic instead of dealing with low-level implementation details, ultimately reducing development time and minimizing errors. As multicore architectures continue to dominate computing, the relevance and importance of TM will only increase. However, challenges related to hardware implementation, performance overhead, and integration with existing programming models remain areas of active research. This survey lays the groundwork for understanding TM's principles, motivations, and potential, providing a foundation for exploring its practical applications and future developments in the realm of parallel computing.

II. IMPLEMENTATION APPROACHES

This section explores the implementation approaches for Transactional Memory (TM): Hardware (HTM), Software (STM), and Hybrid (HyTM). It highlights their trade-offs, suitability for different workloads, and the importance of selecting the right approach based on application needs.

The successful implementation of Transactional Memory (TM) is crucial for realizing its potential to improve performance, scalability, and applicability in parallel computing systems. There are several ways to implement TM, each with its own advantages, limitations, and suitability for different use cases. Understanding these implementation strategies is essential for selecting the most appropriate solution based on the workload, system architecture, and specific requirements of the application. Before examining the individual approaches, it's important to consider the fundamental trade-offs between hardware-centric and software-centric solutions. Hardware-based implementations prioritize performance, offering low latency and high throughput. In contrast, software-based solutions emphasize flexibility and portability, as they can be deployed on various hardware platforms. A hybrid approach combines the strengths of both, offering a balance of performance and adaptability for a range of workloads.

There are three primary TM implementation approaches—Hardware Transactional Memory (HTM), Software Transactional Memory (STM), and Hybrid Transactional Memory (HyTM). Each approach provides unique solutions to address the challenges of synchronization in multicore systems, with distinct considerations for efficiency, scalability, and ease of use.

A. HARDWARE TRANSACTIONAL MEMORY (HTM)

This section highlights Hardware Transactional Memory (HTM's) use of specialized hardware for efficient transaction management.

HTM uses specialized hardware to manage transactions efficiently. First introduced by Herlihy and Moss [6], HTM offers several key features:

1) TRANSACTIONAL CACHE

Temporary changes made during a transaction are stored in a dedicated cache until the transaction commits, helping to minimize memory traffic.

2) SPECIALIZED INSTRUCTIONS

Hardware-level instructions, such as Load-Transactional and Commit, help manage transactional execution.

3) CONFLICT DETECTION

Hardware mechanisms dynamically detect and resolve conflicts between transactions, ensuring consistency and isolation.

HTM delivers high performance for short, low-contention transactions by reducing memory access overhead compared to traditional lock-based synchronization methods. However, its reliance on specialized hardware means that it can struggle with larger transactions or complex workloads, as hardware limitations such as cache size and transaction complexity can hinder scalability. Benchmarks involving tasks like counting operations and linked lists demonstrate HTM's strengths in low-contention scenarios.

B. SOFTWARE TRANSACTIONAL MEMORY (STM)

This section discusses Software Transactional Memory (STM), focusing on its software-based execution, portability, and suitability for high-contention environments.

STM simulates transactional execution entirely in software, removing the need for specialized hardware. Proposed by Shavit and Touitou [7], STM operates based on the following principles:

1) PORTABILITY

STM can be deployed on any hardware, providing broad applicability across different systems.

2) NON-BLOCKING EXECUTION

STM allows progress even under contention by using software-based conflict resolution mechanisms.

3) LOGGING AND METADATA

To maintain consistency and atomicity, STM tracks read and write operations through metadata and logs, simulating the behaviour of atomic transactions.

While STM offers flexibility and portability, it incurs higher overhead due to the need for managing metadata and logging. Despite this, STM excels in high-contention scenarios, enabling lock-free implementations of complex data structures. Experimental results suggest that STM achieves higher throughput and fewer retries when handling concurrent tasks in such environments [8] [9].

C. HYBRID TRANSACTIONAL MEMORY (HYTM)

This section outlines Hybrid Transactional Memory (HyTM), which blends HTM and STM to balance performance and flexibility for different workloads.

HyTM [10] combines elements of both HTM and STM to leverage the strengths of each approach. Key features of HyTM include:

1) DUAL EXECUTION PATHS

Transactions are first attempted in hardware. If hardware limitations are exceeded, the system falls back to software-based execution, ensuring that transactions are still processed correctly.

2) CONFLICT DETECTION

HyTM maintains consistency between the hardware and software transactional executions, resolving conflicts in both paths.

3) SCALABILITY

HyTM offers a balance between the efficiency of HTM and the flexibility of STM, making it scalable across diverse workloads.

HyTM adapts to the specific demands of a workload, offering high performance for short transactions while maintaining flexibility for larger or more complex tasks. It has been shown to perform well in high-contention scenarios, and benchmarks like SPLASH-2 highlight its scalability, making it a promising approach for practical implementations of TM.

In conclusion, the choice of TM implementation approach depends heavily on the specific requirements of the application and the constraints of the underlying hardware. HTM is ideal for low-contention environments with short transactions, leveraging hardware-level optimizations for maximum performance. STM offers broad applicability across hardware platforms, excelling in high-contention scenarios but incurring higher overhead due to its software-based nature. HyTM, by combining HTM and STM, offers an adaptable solution that dynamically adjusts to the workload, providing the best of both worlds in terms of performance, portability, and scalability. Understanding the strengths and weaknesses of each approach allows developers to make informed decisions about integrating TM into their systems, ensuring that TM remains a viable solution for efficient synchronization across a diverse array of applications. As both hardware and software continue to evolve, future research and development in TM implementations promise to further refine these solutions, improving their performance, scalability, and broader applicability in emerging computing environments.

III.NESTED TRANSACTIONS

This section covers Nested Transactions, which improve efficiency and fault tolerance through different nesting models and architectural optimizations.

In complex transactional systems, handling large transactions can be challenging, particularly when it comes to the high costs associated with rollbacks. While rollback operations are essential for maintaining atomicity and consistency, they become increasingly costly as transactions grow in size and complexity. This is particularly problematic when large numbers of

operations must be undone due to a failure or inconsistency. To address these challenges, Nested Transactions were introduced as a solution to facilitate more efficient error handling and improve the overall management of transaction execution [11]. By enabling partial rollbacks without requiring the entire transaction to be undone, Nested Transactions offer a modular approach that significantly reduces rollback costs, enhances concurrency, and improves fault tolerance. Breaking a large transaction into smaller, more manageable subtransactions allows for better failure management. If an error occurs within a subtransaction, only the changes in that subtransaction need to be rolled back, leaving other operations unaffected. This approach ensures that unrelated operations can continue, making it possible to maintain system progress even in the presence of failures.

A. NESTING MODELS

This section discusses three primary nesting models—Closed, Open, and Linear Nesting—and examines architectural innovations aimed at enhancing the performance and scalability of Nested Transactions.

Nesting transactions come in different models, each with unique strengths and trade-offs, influencing how subtransactions are executed, committed, and rolled back. These models significantly impact the performance and scalability of the overall system [12] [13].

1) CLOSED NESTING

In this model, subtransactions are committed to their parent transaction. When a subtransaction completes successfully, it becomes part of the larger parent transaction, and any partial rollbacks are contained within that subtransaction. This model provides strong control and isolation, ensuring that the overall transaction's integrity is maintained. However, it may limit concurrency as subtransactions must be executed sequentially within their parent.

2) OPEN NESTING

Open Nesting allows subtransactions to commit independently and make intermediate changes visible to the broader system. This approach increases concurrency, as different subtransactions can progress in parallel without waiting for one another. However, it introduces additional complexity in managing rollbacks, as intermediate changes must be carefully reverted without affecting system consistency. Open Nesting is best suited for highly parallel applications but requires compensatory mechanisms to handle failures effectively.

3) LINEAR NESTING

Linear Nesting restricts concurrency by allowing subtransactions to execute sequentially, one after another, within a single transactional branch. This model simplifies implementation and rollback management, as there is no need to coordinate multiple concurrent subtransactions. It is a straightforward approach, making it ideal for simpler systems where parallelism is not a priority.

To improve the performance of Nested Transactions, architectural innovations such as transactional data caches and hierarchical tracking mechanisms are critical. These enhancements allow efficient tracking of subtransactions, ensuring that rollbacks can be performed swiftly without consuming excessive resources. By supporting the effective execution and rollback of nested transactions, these optimizations contribute to the robustness and scalability of transactional memory systems, even in complex environments.

In conclusion, Nested Transactions provide an effective mechanism for managing large, complex transactions by offering the flexibility to perform partial rollbacks and preserving system consistency. They help decompose large transactions into smaller, manageable subtransactions, which can improve fault tolerance and reduce the overhead associated with rollbacks. Each of the three primary nesting models—Closed Nesting, Open Nesting, and Linear Nesting—offers distinct trade-offs, enabling the model to be chosen based on application requirements. Closed Nesting provides strong isolation and control, Open Nesting boosts concurrency but adds complexity in rollback management, and Linear Nesting simplifies implementation at the cost of limiting concurrency. Furthermore, architectural innovations like transactional data caches and hierarchical tracking mechanisms play a crucial role in optimizing the performance and scalability of nested transactions in high-performance systems. Ultimately, Nested Transactions enhance the flexibility and efficiency of transactional memory systems, enabling more modular, robust, and scalable programming. [14] [15] [16].As the complexity and parallelism of modern systems continue to grow, the adoption of Nested Transactions will remain an essential strategy for managing transaction execution and ensuring system reliability.

IV.APPLICATIONS OF TRANSACTIONAL MEMORY

This section highlights the benefits and applications of Transactional Memory (TM), focusing on its role in improving concurrency, scalability, and performance in various domains.

Transactional Memory (TM) has gained widespread recognition for its ability to simplify synchronization and enhance concurrency in parallel programming. By abstracting the complexities of managing concurrent operations, TM allows developers to focus on higher-level program logic instead of low-level synchronization details. This makes TM a powerful tool for addressing the

challenges posed by parallelism, offering scalability and adaptability across various domains—from data structures to large-scale, high-performance systems. A primary advantage of TM is its support for efficient, lock-free operations in shared memory environments, which eliminates the overhead associated with traditional synchronization mechanisms like locks. This leads to better performance and improved reliability in systems where concurrency is essential. As such, TM is being increasingly explored and applied in a wide range of fields, providing effective solutions to some of the most persistent challenges in parallel computing.

A. *KEY APPLICATIONS*

Key Applications of Transactional Memory (TM) include

1) CONCURRENT DATA STRUCTURES

TM enables efficient, lock-free operations on shared structures such as linked lists, queues, and hash tables. This enhances scalability and minimizes synchronization errors, making it ideal for dynamic, concurrent data environments.

2) GRAPH ALGORITHMS

TM supports parallel execution of complex operations like graph traversal and updates. This significantly improves the performance of graph-based computations in data-intensive workloads, such as those in social networks, web crawlers, or computational biology.

3) SCALABLE SYSTEMS

TM helps in the efficient management of concurrent operations in large-scale systems such as game servers. By abstracting synchronization, it simplifies system design and enhances the reliability of these systems under high loads.

In conclusion, Transactional Memory represents a transformative shift in parallel programming, simplifying the challenges of concurrency and synchronization while improving system performance and scalability. Its ability to support lock-free operations on shared resources brings about significant gains in efficiency, reliability, and scalability across a wide range of applications. From concurrent data structures to complex graph algorithms and scalable systems like game servers, TM offers a powerful solution to longstanding parallel programming problems. As TM continues to evolve, it holds the promise of unlocking even greater potential in multicore and distributed systems. The continued development of TM technologies will drive advancements in high-performance computing, real-time systems, and large-scale data processing, helping to shape the future of parallel programming. With its flexibility, efficiency, and robustness, TM will be at the heart of next-generation programming models designed to meet the demands of increasingly complex applications.

V.PERFORMANCE INSIGHTS

This section compares HTM, STM, and HyTM based on their strengths, limitations, and suitability for different applications, as detailed in Table I.

Transactional Memory (TM) systems can be broadly categorized into three types: Hardware Transactional Memory (HTM), Software Transactional Memory (STM), and Hybrid Transactional Memory (HyTM). Each approach leverages unique methodologies for handling transactions, offering distinct trade-offs in terms of performance, resource management, conflict detection, and ease of programming. The Table I below provides a detailed comparison of HTM, STM, and HyTM, highlighting their features, strengths, and limitations to help understand their suitability for different use cases.

TABLE I

COMPARISON OF HTM, STM, AND HYTM

S.No.	Feature	HTM	STM	HyTM
1.	Implementation	Hardware-based: TM logic is integrated into hardware (e.g., caches and registers), making it fast and efficient but hardware-dependent.	Software-based: Uses data structures and runtime libraries, offering flexibility but adding software overhead.	Hybrid: Combines HTM for efficient small transactions and STM for larger ones, balancing performance and flexibility.
2.	Performance	High:	Moderate to Low:	High to Moderate:

		Low overhead for small transactions due to direct hardware execution, ideal for common cases.	Slower due to runtime checks and metadata management but supports more complex cases.	Matches HTM performance for small transactions; STM fallback adds overhead for large or complex transactions.
3.	Resource Limitations	Hardware-limited: Constrained by physical resources like cache size and associativity, leading to potential transaction aborts for large memory footprints.	Unbounded: Can handle arbitrarily large transactions, limited only by system memory, but at the cost of higher runtime complexity.	Mixed: Uses HTM for hardware-limited cases and switches to STM for transactions exceeding hardware capabilities, providing flexibility but introducing transition overhead.
4.	Conflict Detection	Eager: Detects conflicts during transaction execution using hardware mechanisms like MESI protocols, ensuring early resolution but adding some latency.	Eager or Lazy: Detection occurs either during execution (eager) or at commit time (lazy), offering flexibility but varying in efficiency.	Hybrid: Eager detection in hardware mode for speed; lazy detection in STM fallback to optimize resource usage.
5.	Rollback Mechanism	Fast: Relies on cache invalidation or other hardware mechanisms for efficient rollbacks, minimizing wasted computation.	Software-based: Uses undo logs or private buffers to revert changes, making rollbacks slower but more flexible.	Hybrid: HTM rollbacks are quick; STM rollbacks rely on undo logs or other software mechanisms, slowing down the process.
6.	Scalability	Limited: Scalability is constrained by hardware shared resources like cache and interconnect bandwidth, affecting performance in large systems.	Flexible:	Improved: Benefits from STM's scalability

			Better scalability with software but incurs higher runtime overhead, particularly under high contention.	for large transactions while leveraging HTM's efficiency for small transactions.
7.	Ease of Programming	Transparent: Requires minimal code changes; programmers benefit from hardware-level optimizations automatically.	Annotation Required: Programmers must annotate transactions in code and use runtime libraries, increasing complexity.	Hybrid: Transparent in HTM mode; STM fallback may require annotations or runtime integration, adding some complexity.
8.	Examples	TCC, LogTM, Azul Vega, Sun Rock, IBM BG/Q, Intel Haswell. : Focus on hardware optimizations for small, efficient transactions.	RSTM [17], TL2 [48], TinySTM [49],SwissTM [50],DSTM [18], McRT-STM [19], NORec [20], Nested LogTM [21] [22], Haskell STM [23] [24] [25] [26] [27] [28] [29], ATOMOS [30], NeSTM [31], HParSTM [32], NePalTM [33], CWSTM [34], PNSTM [35], SSTM [36]. : Software-centric solutions for diverse and complex transactional needs.	Combines both approaches to balance performance, scalability, and flexibility.

In conclusion, this comparison highlights the strengths and limitations of HTM, STM, and HyTM. HTM systems excel in speed and simplicity for small transactions but are constrained by hardware limitations. STM offers flexibility and scalability, making it suitable for complex and unbounded transactions, albeit at the cost of performance. HyTM provides a middle ground, leveraging HTM's efficiency for small transactions while falling back to STM for larger or more complex scenarios. Selecting the right TM approach depends on the specific application requirements, such as transaction size, contention levels, and hardware capabilities. As TM technology continues to evolve, hybrid solutions are expected to play a pivotal role in achieving a balance between performance and scalability in multicore environments.

VI.CHALLENGES AND LIMITATIONS

This section outlines the key challenges faced by HTM and STM systems, as well as common issues shared by both, including resource limitations, performance overhead, and conflict resolution, highlighting the need for advancements to improve TM efficiency and scalability [37].

Transactional Memory (TM) systems, while offering a promising approach to parallel programming, face several challenges that stem from their specific architectures and implementations. These challenges vary based on whether

the system is Hardware Transactional Memory (HTM) or Software Transactional Memory (STM), and some are shared across both. The Table II below categorizes these challenges, providing a description of each issue and its potential impact on TM performance and scalability.

Category	Challenge	Description	Impact
HTM-Specific Challenges	Limited On-Chip Resources	Constrained by limited buffer size, restricting transaction size.	Large transactions may not fit, causing overflows and requiring re-execution.
	Bounded Transactions	Transactions are limited by hardware buffer size.	Large transactions can overflow, causing degradation in performance.
	Unbounded Transactions	Supporting large transactions adds complexity, especially in hybrid systems.	Performance cliffs occur when switching from HTM to STM for larger transactions.
	Instruction Set Architecture (ISA) Support	HTM requires specific ISA extensions, and levels of support vary widely.	Poor ISA support limits flexibility; excessive support complicates hardware design.
STM-Specific Challenges	Runtime Overhead	Managing transactional state and conflict resolution incurs runtime overhead.	Affects performance, especially in high-contention scenarios.
	Atomicity and Code Interaction	Weak atomicity allows errors when mixing transactional and non-transactional accesses.	Causes synchronization issues and data races.
	Inconsistent Reads	Transactions may read inconsistent data due to conflicts not being detected early.	Leads to incorrect results, infinite loops, or program failures.
	Zombie Transactions	Transactions doomed to abort but still execute until detected.	Leads to inconsistent data access, infinite loops, and runtime failures.
Common Challenges	I/O Operations	Handling I/O in transactions is problematic, especially undoing or deferring operations.	Impacts real-time and interactive systems where I/O must be processed consistently.
	Nesting Transactions	Closed Nesting: Commit or abort as a unit. - Open Nesting: Independent commits for inner transactions.	Closed nesting limits concurrency; open nesting increases programmer complexity.
	Programming Model Integration	Integrating TM with models like OpenMP or MPI, which were not designed for TM.	Requires significant changes to programming models, affecting ease of use and performance.
	Conflict Detection and Resolution	Detecting and resolving conflicts effectively, especially in STM.	Adds complexity and overhead, especially with per-thread views of memory in STM.

TABLE II

CHALLENGES IN TRANSACTIONAL MEMORY SYSTEMS

In conclusion, the Table II highlights the multifaceted challenges faced by TM systems, emphasizing the trade-offs between hardware- and software-based implementations. HTM systems excel in speed but are limited by physical constraints and scalability issues, while STM systems offer flexibility at the cost of performance and complexity. Common challenges such as I/O handling, nesting, and programming model integration underscore the need for innovative solutions to make TM more robust and user-friendly. By addressing

these challenges, TM technology can unlock its full potential, enabling efficient and scalable parallel programming for a wide range of applications.

VII. Comparative Study of Transactional Memory (TM) Frameworks

This section compares various Transactional Memory (TM) frameworks, including key HTM and STM systems. It highlights their approaches, strengths, and trade-offs, offering insights into their suitability for different applications and workloads.

A. COMPARISON OF TCC AND LOGTM

This section compares TCC and LogTM, focusing on their distinct approaches to transaction management and the trade-offs in performance, scalability, and complexity [37].

Transactional Memory (TM) systems are designed to handle parallel execution efficiently by enabling atomic and isolated memory transactions. Two prominent implementations, Transactional Coherence and Consistency (TCC) [38] and Log-Based Transactional Memory (LogTM) [21], adopt different approaches to manage commits, aborts, and conflicts. These systems showcase the diversity in TM design philosophies, each with unique trade-offs in terms of performance, scalability, and conflict management. The Table III compares the features of TCC and LogTM, highlighting their respective strengths and limitations.

TABLE III

COMPARISON OF TCC AND LOGTM

S.No.	Feature	TCC	LogTM
1.	Commit	Slower: TCC requires broadcasting the transaction's write set across the bus to maintain consistency, which increases commit latency.	Faster: LogTM commits by updating values in place without broadcasting, making commits quicker.
2.	Abort	Faster: TCC uses speculative rollback to handle aborts efficiently, discarding changes quickly without complex recovery steps.	Slower: LogTM requires traversing a log to undo changes, which is more time-consuming during an abort.
3.	Coherence Mechanism	Bus-based: TCC relies on a bus architecture for communication, simplifying coherence but limiting scalability in systems with more processors.	Directory-based: LogTM uses a directory to track memory states across processors, enabling better scalability in larger systems.
4.	Conflict Detection	Lazy: Conflicts are detected only at commit time, reducing overhead during transaction execution but increasing rollback likelihood.	Eager: Conflicts are detected during each read or write, enabling earlier resolution but with higher runtime checking overhead.
5.	Conflict Resolution	Abort Self: TCC resolves conflicts by aborting the conflicting transaction itself, simplifying resolution.	Oldest Timestamp Wins: LogTM prioritizes older transactions, aborting newer ones to preserve progress and fairness.
6.	Write Visibility	At Commit: TCC makes write changes visible to other processors only after a transaction successfully commits, ensuring atomicity.	Immediate: LogTM updates shared memory during execution, improving concurrency but requiring more robust conflict management.
7.	Always in Transaction	Yes: TCC treats all operations as part of transactions, providing uniformity but adding overhead for non-critical code.	No: LogTM only uses transactional mechanisms when needed, reducing overhead for non-transactional operations.
8.	Nesting Support	Yes:	Yes:

		TCC supports nesting of transactions, enabling more complex workflows.	LogTM also supports nested transactions but with different conflict detection and resolution strategies.

In conclusion, the comparison between TCC and LogTM underscores their distinct approaches to handling transactional memory. TCC excels in simplicity and speculative execution but suffers from scalability challenges due to its bus-based architecture. In contrast, LogTM offers better scalability and concurrency through its directory-based coherence mechanism and immediate write visibility but incurs higher overheads during aborts and conflict detection. Understanding these trade-offs is crucial for selecting the appropriate TM system based on the application's requirements, such as scalability, transaction complexity, and contention levels. As TM technology evolves, hybrid approaches that combine the best features of TCC and LogTM may address their respective limitations.

B. COMPARATIVE OVERVIEW OF HTM SYSTEMS

This section compares prominent HTM systems, highlighting their features, architectures, limitations, and trade-offs in performance and scalability.

Hardware Transactional Memory (HTM) systems leverage hardware-level mechanisms to manage transactional operations efficiently. By integrating speculative execution, conflict detection, and rollback mechanisms directly into the processor architecture, HTM systems aim to improve performance and simplify programming for parallel workloads. Various HTM implementations have been developed, each with unique features and architectural designs, but they also face specific limitations. The Table IV provides a comparative overview of some prominent HTM systems, focusing on their key features, architecture, limitations, and implementation status [39].

TABLE IV COMPARATIVE OVERVIEW OF HTM SYSTEMS

S.No.	HTM System	Key Features	Architecture	Limitations	Status
1.	Azul Vega [40] [41]	TM integrated with Java Virtual Machine. Speculative execution with SPECULATE, ABORT, and COMMIT instructions.	64-bit RISC; up to 16 processors (54 cores each, total 864 cores). L1 Cache: 16KB private per core. L2 Cache: 2MB shared among 9 processors.	- Memory conflicts limit speedup to ~1.1×. Capacity overflow is rare but impacts runtime.	Commercially implemented.
2.	Sun Rock [42] [43]	Checkpoint-based speculative execution with the ability to revert to a safe state. SPECULATE and COMMIT instructions; conflicts abort transactions.	High-performance SPARC processor.	Supported for only 32 L2 cache lines.	Cancelled before release.
3.	IBM BG/Q [44] [45]	Multi-versioned 16-way L2 cache for speculative state storage. Supports short- and long-running transactional modes.	Blue Gene/Q supercomputer architecture.	L1 cache cannot store speculative state. Requires evictions or aliasing for long-running transactions.	Used in Sequoia supercomputer.
4.	Intel Haswell [46]	Transactional Synchronization Extensions (TSX):	x86 architecture.	Detailed architectural implementation not disclosed.	

		XBEGIN, XEND, and XABORT instructions.			

In conclusion, the Table IV illustrates the diversity in HTM designs, highlighting the trade-offs and challenges faced by different systems. While Azul Vega and Intel Haswell have achieved commercial success, systems like Sun Rock faced technical limitations that led to their cancellation. IBM BG/Q demonstrates the potential of HTM in high-performance computing environments, albeit with architectural constraints. These implementations underscore the importance of balancing performance, scalability, and reliability in HTM systems. Continued advancements in HTM technology will be crucial for addressing these limitations and expanding its applicability in parallel computing.

C. COMPARATIVE ANALYSIS OF STM IMPLEMENTATIONS

This section compares four STM implementations, highlighting their features, strengths, and trade-offs for different application needs [47].

Transactional Memory (TM) systems offer a flexible framework for simplifying concurrent programming by eliminating many of the challenges associated with traditional lock-based synchronization. To cater to diverse workloads and system requirements, several Software Transactional Memory (STM) implementations have been developed, each with distinct features and approaches. The Table V provides a comparative analysis of four popular STM implementations: RSTM [17], TL2 [48], TinySTM [49], and SwissTM [50]**.** These implementations are evaluated across various dimensions, including granularity, update policy, write policy, and concurrency control [6] [14] [47] [51] [52] [53] [54] [55] [56]. The comparison highlights their strengths, trade-offs, and suitability for different scenarios.

TABLE V COMPARATIVE ANALYSIS OF STM IMPLEMENTATIONS

S.No.	Feature	RSTM	TL2	TinySTM	SwissTM
1.	Granularity	Object-based: RSTM operates at the object level (e.g., arrays, lists), meaning it treats data structures as atomic units. This is useful for high-level abstractions and large data structures.	Both: TL2 supports both object-based and word-based granularity. The flexibility allows it to work with both larger objects or finer memory locations, offering more control based on the workload.	Word-based: TinySTM operates at a finer level, managing individual memory locations (e.g., words or cache lines), providing better control for more granular transactions.	Word-based: Similar to TinySTM, SwissTM operates at the word level. This helps improve memory efficiency and allows finer control over transaction granularity.

2.	Update Policy	Deferred: Updates are applied to memory only at commit time. This helps in reducing conflicts and ensures that partial, speculative updates do not affect other transactions.	Deferred: TL2 uses deferred updates, meaning changes are buffered and only applied when the transaction commits. This ensures consistency and minimizes premature side effects.	Both: TinySTM offers both deferred updates (written at commit time) and immediate updates (applied immediately), providing flexibility based on the workload.	Deferred: SwissTM defers updates until commit to ensure that any changes are atomic and consistent. It uses buffered memory writes to avoid conflicts during execution.
3.	Write Policy	Buffered: Writes are buffered and stored in a private transaction memory space. They are only visible to other transactions once the transaction commits.	Buffered: TL2 uses a similar approach by buffering writes in a private memory area until the transaction is successfully committed, ensuring isolation and consistency.	Both: TinySTM supports both buffered writes, where changes are stored in a temporary buffer until commit, and immediate writes, which are applied during execution.	Buffered: SwissTM uses buffered writes, ensuring that memory updates only happen when the transaction commits. This provides isolation and consistency during execution.
4.	Acquire Policy	Both: RSTM supports both eager and lazy acquisition of locks/resources. In eager acquisition, resources are locked immediately, whereas in lazy acquisition, locks are taken only when needed.	Lazy: TL2 uses lazy acquisition, meaning locks are acquired only when necessary, typically during execution if conflicts are about to occur. This reduces unnecessary overhead.	Both: TinySTM can either acquire locks eagerly (immediately) or lazily (on-demand), providing flexibility depending on the context and workload.	Both: SwissTM supports both eager and lazy lock/resource acquisition. The system can adapt based on the transaction's needs or configuration.
5.	Read Policy	Both (Visible and Invisible): RSTM supports both visible and invisible reads. Visible reads allow other transactions to see the data immediately, whereas	Invisible: TL2 primarily uses invisible reads. Reads are not visible to other transactions until the	Invisible: TinySTM uses invisible reads, meaning that the data read by a transaction is not visible to other transactions	Invisible: SwissTM also uses invisible reads to ensure consistency and isolation, making sure that data changes are not exposed until the

		invisible reads keep data private until commit.	transaction commits, ensuring isolation and preventing conflicts during execution.	until the transaction commits.	transaction commits.
6.	Conflict Detection	Both (Eager and Lazy): RSTM allows for both eager and lazy conflict detection, depending on the transaction configuration. Eager detection checks conflicts immediately, while lazy detection checks at commit time.	Both (Eager and Lazy): TL2 supports both eager and lazy conflict detection strategies. Eager detection checks conflicts as operations happen, while lazy detection waits until commit.	Early: TinySTM employs early conflict detection, identifying conflicts as soon as they occur during the transaction execution, reducing retry rates and improving performance.	Mixed Invalidation: SwissTM uses a mixed invalidation method, detecting write-write conflicts early and read-write conflicts lazily, offering a balance between performance and correctness.
7.	Concurrency Control	Optimistic: RSTM uses optimistic concurrency control, assuming that conflicts are rare and allowing transactions to execute concurrently. Conflicts are resolved when they are detected, typically at commit time.	Optimistic: TL2 uses optimistic concurrency control, meaning it allows transactions to execute concurrently with the assumption that conflicts will be rare. Conflicts are detected and resolved at commit time.	Optimistic: TinySTM uses optimistic concurrency control, allowing transactions to execute concurrently and resolving conflicts when they are detected. This leads to higher throughput in low contention.	Both (Optimistic & Lock-based): SwissTM uses optimistic concurrency control for low-contention scenarios and switches to lock-based control when high contention is detected. This provides flexibility and better handling of diverse workloads.
8.	Progress Guarantee	Obstruction-free: RSTM guarantees obstruction-free progress, meaning transactions will eventually complete even if other transactions are delayed or blocked.	Lock-based: TL2 uses lock-based control, meaning progress depends on acquiring and releasing locks. In high-contention	Lock-based: TinySTM uses lock-based control in certain cases, guaranteeing progress as long as locks are properly acquired and released, but potentially	Lock-based: SwissTM uses lock-based concurrency control, ensuring transaction isolation. It may face delays in high contention but can switch between

			situations, it may face delays or deadlocks.	causing delays in highly concurrent environments.	optimistic and lock-based methods based on workload characteristics.

In conclusion, this comparative analysis highlights the diverse approaches adopted by STM implementations to balance performance, scalability, and correctness. RSTM and TL2 offer simplicity and consistency with their deferred updates and optimistic concurrency control, while TinySTM and SwissTM provide more flexibility, catering to workloads with varying contention levels. By understanding these differences, developers can choose the STM implementation best suited to their application's needs, ensuring efficient and reliable transactional memory operations in multicore environments. As research progresses, future STM systems are likely to incorporate hybrid techniques that further optimize performance and usability across a wider range of scenarios.

VIII. FUTURE DIRECTIONS

This section explores the future advancements and directions for Transactional Memory (TM), focusing on overcoming current limitations and broadening its applicability in modern computing environments.

As modern computing systems become increasingly complex, TM is evolving to meet the growing demands of parallel programming. Researchers are working to refine TM's design and implementation, with a goal of enhancing its scalability, flexibility, and reliability across a wide range of workloads and system architectures. While TM holds the potential to simplify concurrency management, unlocking this potential requires addressing existing challenges and integrating TM more effectively into the broader computing ecosystem.

Key advancements and future directions include:

A. DYNAMIC STM

Extends STM to support dynamic memory access patterns, enhancing flexibility.

B. ADAPTIVE CONFLICT RESOLUTION

Dynamically adjusts backoff mechanisms to minimize transaction abort rates.

C. EXPANDED HTM SUPPORT

Developments in processor design aim to handle larger and more complex transactions.

D. *INTEGRATION WITH WEAK MEMORY MODELS*

Enables TM to function effectively in systems with relaxed consistency constraints.

In conclusion, these innovations are set to enhance TM's robustness, making it an essential tool in modern parallel programming. As TM continues to evolve, its expanding applicability will enable it to address a wide variety of workloads and emerging applications. The combination of improvements in Dynamic STM, Adaptive Conflict Resolution, and expanded hardware support will refine TM's performance and facilitate its integration with advanced computing paradigms, such as weak memory models. As TM adapts alongside developments in hardware, software, and system architectures, it will remain central to advancing parallel computing, empowering developers to harness the full potential of multicore and distributed systems.

IX.CONCLUSION

Transactional Memory (TM) has emerged as a transformative paradigm for managing synchronization in parallel programming, offering an efficient, high-level abstraction that overcomes the limitations of traditional synchronization methods like locks and semaphores. This paper has provided a comprehensive exploration of TM by examining its foundational principles, implementation strategies, applications, challenges, and future directions.TM was introduced as a response to the increasing complexity of synchronization in multicore and distributed systems. It provides atomicity, consistency, and isolation, simplifying concurrency management and enabling developers to focus on scalable program design. The core implementation strategies of TM—Software Transactional Memory (STM), Hardware Transactional Memory (HTM), and Hybrid Transactional Memory (HyTM)—offer flexibility, low-latency synchronization, and dynamic adaptability, respectively. Each approach has specific strengths, limitations, and applicability, depending on the workload and system requirements. Nested Transactions were highlighted as an effective way to improve fault tolerance, concurrency, and rollback efficiency. By modularly breaking down transactions into smaller subtransactions, TM reduces overhead and enhances scalability, particularly in complex systems. TM's diverse applications, including concurrent data structures, graph algorithms, scalable systems, and real-time computing, demonstrate its effectiveness across

domains requiring high concurrency and reliability. Its lock-free synchronization capabilities significantly enhance performance and reliability. Insights into TM's performance showed the unique advantages and trade-offs of STM, HTM, and HyTM, emphasizing the importance of aligning implementation strategies with specific workload characteristics to maximize scalability and efficiency. The challenges and limitations of TM, such as integration complexity, large transaction handling, contention management, and nesting complexity, were discussed. These obstacles highlight the need for continued innovation to improve TM's practicality and facilitate broader adoption. A Comparative Study of Transactional Memory (TM) Frameworks was conducted, providing valuable insights into different TM systems, including the comparison of TCC and LogTM, an overview of HTM systems, and an in-depth analysis of STM implementations. This comparison highlighted the distinct approaches, trade-offs, and challenges faced by different TM frameworks, offering a clearer understanding of their suitability for various applications and workloads. Looking ahead, future directions for TM include advancements like Dynamic STM, Adaptive Conflict Resolution, expanded HTM support, and integration with weak memory models. These innovations are crucial for overcoming current limitations and broadening TM's applicability in modern computing environments. In conclusion, TM represents a significant advancement in parallel programming, providing a scalable, efficient, and reliable synchronization model. As multicore and distributed systems grow in complexity, TM's adaptability and robustness position it as a cornerstone of modern computing. By addressing challenges and leveraging emerging advancements, TM is set to drive innovation in high-performance computing, real-time systems, and large-scale data processing, shaping the future of parallel programming. This exploration will help researchers by offering valuable insights into TM's principles, challenges, and future directions, guiding the development of more efficient and scalable systems in various computational domains.

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] 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.

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

[10] 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.

[11] 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.

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

[13] 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.

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

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

[16] 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.

[17] 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 .

[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] 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.

[20] 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.

[21] J. 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.

[22] J. 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.

[23] 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.

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

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

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

[27] 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.

[28] 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.

[29] G. a. S. F. S. Peyton Jones, "Concurrent Haskell," in 23rd ACM Symposium on Principles of Programming Languages (POPL'96), 1996.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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).

[36] 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.

[37] C. O. S. U. E. A. F. G. B. S. M. V. Tim Harris, " Transactional Memory: An Overview," IEEE Micro , vol. 27, no. 3, pp. 8-29, 2007.

[38] L. H. a. V. W. a. M. K. C. a. B. D. C. a. J. D. D. a. B. H. a. M. K. P. a. H. W. a. C. K. a. K. Olukotun, "Transactional Memory Coherence and Consistency," in 31st Annual International Symposium ,Computer Architecture(ISCA04), 2004.

[39] N. M. a. J. N. A. S. R. Cordeiro, A Review of Hardware Transactional Memory, 10th Workshop on Parallel and Distributed Processing (WSPPD), 2012.

[40] J. CHOQUETTE, G. TENE and K. NORMOYLE, "Speculative multiaddress atomicity". US Patent 7,376,800.

[41] Click., Azul's experiences with hardware transactional memory, 2009.

[42] R. C. M. E. M. K. A. L. S. Y. H. Z. a. M. T. Shailender Chaudhry, "Rock: A High-Performance Sparc CMT Processor," IEEE Micro, vol. 29, no. 2, pp. 6-16, 2009.

[43] Y. L. M. M. a. D. N. D. Dice, "Early experience with a commercial hardware transactional memory implementation," in 14th international conference on Architectural support for programming languages and operating systems,ASPLOS, New York, USA, 2009.
[44] P. Bright, "IBMs new transactional memory: Make-or-break time for multithreaded revolution," ARS Technica, 2011.
[45] M. G. P. W. M. O. J. N. A. C. B. R. S. a. M. M. M. A. Wang, "Evaluation of blue gene/q hardware support for transactional memories," in 21st International Conference on Parallel Architectures and CompilationTechniques, PACT '12, Minneapolis, MN, USA, 2012.
[46] J. Reinders, "Transactional synchronization in Haswell," Intel Software Network , 2012.
[47] 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.
[48] O. S. N. S. Dave Dice, " Transactional Locking II," in 20th International Symposium on Distributed Computing, Stockholm,Sweden, 2006.
[49] F. a. T. R. Pascal Felber, "Dynamic performance tuning of word-based software transactional memory," in 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP '08), New York, USA, 2008.
[50] R. G. a. M. K. A. Dragojevic, "Stretching transactional memory," in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Dublin, Ireland, 2009.
[51] 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.
[52] 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.
[53] 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.
[54] 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.
[55] 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.
[56] M. L. Scott, "Applications Included with RSTM WebPage," [Online]. Available: http://www.cs.rochester.edu/research/synchronization/rstm/applications.shtml.

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.