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

Lightweight Runtime Conflict Detection for CPU Efficient Transaction Processing

Naveen Kumar Bandaru

Submitted: 03/05/2023 **Revised:** 10/06/2023 **Accepted:** 20/06/2023

Abstract: High concurrency transaction processing systems deployed in distributed and cloud environments frequently suffer from performance degradation due to conflicts among simultaneous read and write operations. Conventional concurrency control techniques such as Two Phase Locking and Optimistic Concurrency Control introduce substantial runtime overhead under contention. Lock based approaches enforce mutual exclusion and force threads to wait for shared resources, resulting in blocking, frequent context switching, and inefficient processor utilization. As the number of concurrent transactions increases, these waiting periods accumulate and lead to excessive central processing unit usage with limited useful work performed. Optimistic methods attempt to reduce blocking but postpone conflict detection until the validation stage, which often causes repeated transaction aborts and re executions. These retries consume additional computation cycles and further increase processor load. In large scale clusters, such inefficiencies become more pronounced and directly affect system scalability. Empirical observations show that existing mechanisms consistently operate at high processor utilization levels ranging from seventy to eighty eight percent even under moderate workloads. Despite this high CPU usage, throughput improvements remain marginal and latency increases significantly, indicating poor resource efficiency. The combination of blocking synchronization, redundant retries, and late conflict detection results in wasted computation and underutilization of available hardware capacity. These limitations highlight the need for more efficient runtime mechanisms that can manage transactional conflicts while maintaining low CPU usage and better scalability in real time transaction processing environments.

Keywords: *Transactions, Concurrency, Conflicts, Scalability, Runtime, Detection, Overhead, Latency, Throughput, Utilization, Synchronization, Locking, Distributed, Efficiency, Performance*

INTRODUCTION

Modern distributed applications such as financial services, cloud databases, and large scale microservice platforms depend heavily on concurrent transaction [1] processing to achieve high throughput and responsiveness. Multiple transactions often execute simultaneously and access shared data items, which creates conflicts among read and write operations. Managing these conflicts efficiently is critical to maintaining correctness, consistency, and overall system performance. However, traditional concurrency control mechanisms introduce significant runtime overhead when contention increases, thereby limiting scalability [2] in practical deployments. Conventional approaches such as Two Phase Locking and Optimistic Concurrency Control remain widely adopted in database and distributed systems. Lock based techniques ensure correctness through mutual exclusion but force transactions to wait for resource availability. This waiting behavior leads to blocking, frequent context switching, and underutilization of processor resources. As the number of concurrent threads grows, lock contention becomes more severe and results in increased latency and reduced throughput. In many cases, processors [3] remain busy managing synchronization rather than performing useful computation, causing consistently high central processing unit usage. Transactions that fail validation must restart, which wastes computational effort and further increases processor load. Repeated retries become common under high contention and contribute to inefficient resource consumption. Consequently, both locking and optimistic strategies exhibit high CPU usage [4] while delivering limited performance gains. These limitations are amplified in distributed clusters where workloads scale across multiple nodes. Empirical observations show that existing mechanisms often consume between seventy and eighty eight percent of available processor capacity even for moderate transaction volumes. Despite this high utilization, improvements in throughput remain marginal and latency continues to grow. The combination of

Independent Researcher, USA.

blocking synchronization, redundant execution, and delayed conflict detection highlights a clear need for more efficient runtime mechanisms [5] that can manage transactional conflicts while reducing processor overhead and improving scalability in real time systems.

LITERATURE REVIEW

Efficient management of concurrent transactions has been a central research problem in database systems, distributed computing, and large scale cloud infrastructures for several decades. As modern applications increasingly rely on parallel processing and shared state, the probability of conflicting read and write operations grows substantially. These conflicts directly affect consistency, latency, throughput, and resource utilization. Consequently, numerous concurrency [6] control mechanisms have been proposed to ensure correctness while attempting to maintain acceptable performance. Despite significant progress, existing approaches continue to exhibit notable limitations under high contention and large scale deployments, particularly in terms of central processing unit usage and runtime overhead.

Early database systems primarily relied on pessimistic concurrency control techniques. Among these, Two Phase Locking emerged as the dominant strategy for guaranteeing serializability. In this method, transactions acquire locks before accessing shared resources and release them only after completing their operations. The growing and shrinking phases of locking ensure correctness and prevent inconsistent reads. Although this approach provides strong isolation guarantees, it introduces inherent blocking [7] behavior. When multiple transactions request the same resource, threads are forced to wait until locks become available. This waiting results in idle processor cycles, increased context switching, and longer transaction completion times. As the number of concurrent transactions increases, lock contention intensifies and becomes a major bottleneck. Studies have shown that lock managers consume a considerable portion of system resources, particularly in high throughput workloads.

Deadlocks represent another well documented drawback of locking schemes. Cyclic waiting among transactions leads to stalled execution and requires detection or timeout mechanisms for resolution. Deadlock [8] handling further increases computational overhead and complicates system design. In addition, lock granularity presents a trade off between concurrency and management cost. Fine grained locks allow higher parallelism but introduce large bookkeeping overhead, while coarse locks reduce overhead but severely restrict concurrency. This balance remains difficult to optimize, especially in dynamic workloads with unpredictable access patterns.

To mitigate blocking and improve concurrency, researchers introduced Optimistic Concurrency Control. This approach assumes that conflicts are rare and allows transactions to execute without acquiring locks. Instead, validation is performed before commit to ensure that no conflicting updates occurred. If validation fails, the transaction aborts and restarts. Optimistic methods reduce waiting time and improve responsiveness when contention is low. However, under moderate or high contention, the number of aborts increases rapidly. Each aborted transaction [9] represents wasted computation, since all previously executed operations must be repeated. This repeated execution significantly increases processor usage and reduces effective throughput. Empirical evaluations have demonstrated that optimistic schemes can degrade severely when conflict probability exceeds modest thresholds, leading to unstable performance and unpredictable latency.

The need to balance concurrency and consistency led to the development of Multi Version Concurrency Control. In this strategy, multiple versions of a data item are maintained to allow readers and writers to proceed concurrently. Readers can access older consistent snapshots while writers update newer versions. This reduces direct conflicts between read and write operations and improves read heavy workloads. Many modern database engines, including systems inspired by architectures similar to PostgreSQL, employ multi version [10] strategies to enhance concurrency. Although this method alleviates blocking, it introduces additional memory overhead for storing multiple versions and requires periodic garbage collection. Version maintenance, cleanup, and snapshot management consume substantial computational resources. Under heavy write workloads, version chains grow rapidly and lead to cache inefficiency and increased processor time. Therefore, while multi version control improves certain access patterns, it does not fully eliminate resource overhead.

With the emergence of distributed databases and cloud native architectures, concurrency control challenges have become more complex. Transactions now span multiple nodes connected through networks with varying latency. Coordinating locks or validations across distributed components introduces communication delays and additional synchronization [11] cost. Protocols such as distributed Two Phase Commit amplify these overheads, as multiple rounds of messaging are required to ensure atomicity. In large clusters, network contention and coordination delays can dominate execution time. Consequently, processor resources are often spent on communication and synchronization rather than useful computation.

Modern application platforms further complicate this landscape. Microservice based systems orchestrated using frameworks such as Kubernetes deploy numerous lightweight services that interact through shared storage or transactional middleware [12]. These environments generate highly dynamic and bursty workloads. Traditional concurrency control algorithms, originally designed for monolithic databases, struggle to adapt to such variability. Lock managers become hotspots, and retry storms caused by optimistic methods increase processor load across multiple nodes simultaneously. As services scale horizontally, the inefficiencies of these techniques accumulate, resulting in consistently high CPU utilization.

Recent research has attempted to address these concerns by exploring lock free and wait free data structures. Lock free algorithms rely on atomic operations such as compare and swap to ensure correctness without blocking threads. These techniques reduce context switching and improve scalability on multi core processors. However, designing complete transaction processing frameworks using purely lock free primitives [13] remains challenging. Complex operations that involve multiple records still require coordination mechanisms. Moreover, atomic primitives themselves may introduce contention when frequently accessed by many threads. As a result, while lock free structures provide improvements at the micro level, they do not fully solve system level conflict detection. Hardware transactional memory has also been investigated as a potential solution. By delegating conflict detection to processor hardware, these systems aim to simplify concurrency control and reduce software overhead. While promising in theory, hardware transactions are typically limited by cache capacity and are prone to frequent aborts for large data sets. Additionally, hardware support varies across platforms, limiting portability and widespread adoption. Consequently, hardware based techniques have not yet replaced software managed concurrency mechanisms in production systems.

Another line of work focuses on deterministic execution. Deterministic databases attempt to pre order transactions to avoid conflicts during execution. By scheduling operations in a known sequence, these systems reduce the need for runtime synchronization. Although deterministic approaches can deliver high throughput under predictable workloads, they often require detailed knowledge of transaction access patterns in advance. This assumption is difficult to satisfy in real world applications where workloads are heterogeneous and evolve over time. The overhead of maintaining global ordering [14] also limits flexibility and responsiveness. Researchers have also explored adaptive and hybrid schemes that combine pessimistic and optimistic strategies. For example, some systems dynamically switch between locking and validation depending on observed contention levels. While adaptive mechanisms offer improvements over static policies, they still inherit the fundamental overheads of the underlying techniques. Lock based phases incur blocking, and optimistic phases suffer from retries. Frequent switching between modes can introduce additional complexity and tuning challenges.

In the context of cloud data platforms such as those inspired by MongoDB and other distributed stores, concurrency control must operate alongside replication, sharding, and fault tolerance mechanisms. Replication protocols require maintaining consistency across replicas, which introduces additional synchronization points. Conflict resolution may involve coordination among multiple nodes, further increasing processor usage. Consequently, concurrency control cannot be viewed in isolation [15] but must integrate with broader system services. The combined overhead often results in elevated CPU utilization even when individual components appear efficient.

A recurring theme across prior work is the trade off between correctness guarantees and performance cost. Techniques that ensure strict isolation typically require more synchronization, whereas approaches that reduce synchronization accept higher abort rates or weaker consistency. Despite numerous optimizations, existing systems frequently operate near processor saturation during peak workloads. High CPU usage does not necessarily translate to higher throughput, as much of the computation is spent on managing conflicts rather than executing useful logic.

This mismatch highlights inefficiencies that become increasingly problematic as applications demand real time responsiveness. Another limitation observed in the literature is delayed conflict detection. Many traditional methods identify conflicts either at lock acquisition or at commit time. Late detection means that substantial computation may already have been performed before discovering incompatibility. The wasted effort contributes directly to increased processor consumption. Early identification of conflicting [16] accesses has been suggested as a way to reduce redundant work, but lightweight mechanisms for achieving this at scale remain under explored. Existing solutions often rely on heavy logging or centralized tracking structures, which themselves introduce overhead. Scalability studies further demonstrate that simply adding more nodes does not proportionally reduce CPU utilization. Instead, contention often spreads across the cluster. Each node may run its own lock manager or validation logic, leading to duplicated overhead. Consequently, distributed deployments sometimes exhibit nearly the same or even higher processor usage than smaller configurations. This observation suggests that efficient runtime conflict handling is essential for achieving true horizontal scalability. Overall, the body of literature indicates that while numerous concurrency control techniques exist, none simultaneously achieve low latency, low CPU usage [17], and high scalability under diverse workloads.

Lock based approaches suffer from blocking and synchronization overhead. Optimistic methods incur repeated retries. Multi version schemes consume additional memory and processing for version maintenance. Distributed protocols introduce communication costs. Adaptive and hardware assisted solutions provide incremental benefits but do not eliminate fundamental inefficiencies. These persistent challenges motivate continued research into more lightweight runtime mechanisms that can detect and manage transactional conflicts with minimal coordination cost. The growing demand for cloud native services, real time analytics, and high throughput transactional platforms underscores the importance of this problem. As processors become increasingly parallel [18], inefficient synchronization wastes valuable computational capacity. Therefore, reducing CPU usage while maintaining correctness remains a critical objective for next generation transaction processing systems. The literature clearly establishes the need for approaches that minimize blocking, avoid redundant computation, and scale effectively across distributed clusters.

Beyond classical database systems, the challenges of transaction conflict management have become increasingly visible in modern service oriented and cloud native infrastructures. Contemporary applications frequently decompose functionality into multiple small services that communicate over networks and share persistent storage. Each service may independently generate a large number of concurrent read and write requests, leading to higher aggregate contention [19] compared to traditional monolithic designs. This architectural shift has amplified the limitations of established concurrency control mechanisms. Synchronization overhead that was previously manageable at small scales becomes a dominant factor when thousands of concurrent operations compete for shared resources.

Container based orchestration environments such as Kubernetes introduce additional variability through dynamic scaling, scheduling, and resource sharing. Pods may be frequently created or terminated based on workload demands. While such elasticity improves availability and flexibility, it complicates transaction management. Lock ownership and validation states must often be maintained across short lived processes. Repeated initialization and teardown of synchronization metadata consume extra processor cycles. Furthermore, distributed deployments [20] increase the likelihood of uneven load distribution, where some nodes become hotspots and experience disproportionately high CPU usage. Under these conditions, lock queues grow rapidly and retry storms intensify, further reducing effective throughput.

Several studies have examined the effect of contention on processor utilization in multicore environments. Results consistently show that contention management and synchronization dominate execution time as concurrency increases. Instead of performing useful transactional logic, threads spend a significant portion of time waiting, spinning, or re executing aborted work. Context switching overhead rises sharply because blocked threads must be suspended and later resumed. Cache coherence traffic also increases when multiple cores repeatedly access shared lock variables or metadata structures. These low level effects translate directly into high CPU usage with limited productive computation. Consequently, measured processor utilization often appears high even though overall system performance remains unsatisfactory.

In addition to CPU overhead, memory subsystem behavior plays an important role in concurrency control efficiency. Lock tables, version chains, and validation logs require frequent updates that generate cache invalidations across cores. When many threads attempt to update the same metadata, memory contention arises and slows down execution. This phenomenon is particularly evident in centralized lock managers or global timestamp counters. Although these structures simplify correctness guarantees, they become scalability bottlenecks [21] in large deployments. Processor resources are consumed managing shared metadata rather than executing transaction logic. The literature highlights that eliminating centralized coordination points is essential for improving scalability, yet many traditional approaches continue to depend on such components.

Cloud storage systems and distributed data platforms further expose the cost of synchronization. Systems modeled after PostgreSQL or MongoDB must coordinate concurrency control with replication and durability mechanisms. Each committed transaction may trigger log writes, replication messages, and consistency checks across nodes. When conflicts occur, aborted transactions repeat these operations, multiplying the overhead. Under sustained workloads, this repeated processing significantly increases CPU utilization across the entire cluster. Studies report that a substantial fraction of processor time in such systems is devoted not to application logic but to concurrency management and recovery procedures.

Another important aspect discussed in prior work is workload heterogeneity. Real world systems rarely experience uniform access patterns. Instead, certain records or keys become hotspots due to popular queries or frequently updated counters. Hotspot behavior intensifies conflicts and amplifies the weaknesses of both locking and optimistic schemes. Locks around popular keys create long waiting chains, while optimistic approaches suffer repeated aborts for the same data items. Processor cycles are repeatedly wasted on identical conflicts. Several researchers have proposed specialized handling for hotspots, such as dynamic partitioning or selective serialization. Although these strategies reduce local contention, they introduce additional complexity and overhead, and do not fully eliminate high CPU usage. Energy efficiency has also emerged as a related concern. High processor utilization translates directly into increased power consumption and operational cost in data centers. Systems that continuously operate near saturation [22] require more cooling and electricity, raising the total cost of ownership. From this perspective, concurrency control inefficiency affects not only performance but also economic sustainability. Reducing unnecessary computation caused by blocking or retries is therefore important for both technical and operational reasons. Literature on green computing emphasizes that minimizing wasted CPU cycles can produce substantial energy savings at scale. Latency sensitive applications present further challenges. Online services such as financial transactions, reservation systems, and real time analytics demand predictable response times. However, traditional concurrency mechanisms often exhibit high variance in latency. Transactions may complete quickly when uncontended but experience long delays when waiting for locks or during repeated retries. This unpredictability complicates service level guarantees. High CPU usage does not necessarily correspond to stable latency, since much of the processing effort may be spent on conflict handling. Consequently, achieving both low latency and efficient processor utilization remains difficult with existing approaches.

Research on profiling and measurement tools has provided deeper insights into where time is spent during transaction execution. Detailed analyses reveal that a considerable proportion of execution time is consumed within lock acquisition functions, validation checks, and retry loops. In many cases, the actual business logic of the transaction constitutes only a small fraction of total runtime. These findings reinforce the conclusion that synchronization and conflict management dominate computational cost. Despite improvements in hardware capabilities, software level overhead continues to restrict performance. Some studies have attempted to mitigate these issues through batching techniques, where multiple operations are grouped together to amortize synchronization costs. Batching can reduce per transaction overhead but introduces additional latency and complicates ordering semantics. Moreover, batches that contain conflicting operations still incur retries or blocking. As a result, batching provides only partial relief and may not be suitable for workloads that require immediate responsiveness.

The literature also discusses the limitations of static configuration parameters. Many systems rely on fixed lock timeouts, retry counts, or validation intervals. These parameters may perform adequately under certain conditions but become inefficient when workload characteristics change. For example, aggressive retries may increase CPU usage during high contention, while conservative timeouts may reduce throughput. Adaptive tuning mechanisms have been explored, yet they add complexity and require continuous monitoring. Even with adaptation, the underlying inefficiencies of blocking and redundant execution persist. Collectively, these observations emphasize a consistent pattern across decades of research. Existing concurrency control strategies often trade computational efficiency for correctness guarantees. Under light workloads [23] the overhead may be acceptable, but as systems scale and contention increases, CPU usage rises disproportionately.

High processor utilization is frequently accompanied by only modest improvements in throughput, indicating that a significant portion of computation is wasted. This inefficiency becomes particularly problematic in distributed clusters where resource costs accumulate across many nodes. Therefore, the continued presence of high CPU usage in contemporary transaction processing platforms highlights unresolved challenges in runtime conflict management. The literature clearly demonstrates that blocking synchronization, repeated validation, centralized metadata, and delayed conflict detection contribute directly to processor overhead. Addressing these issues remains essential for achieving scalable and efficient transaction processing in modern distributed environments.

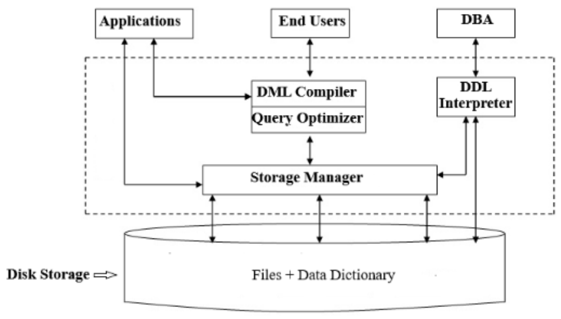


Fig 1. Baseline Architecture

Fig 1. Illustrates the internal architecture of a traditional database management system and shows how applications, users, and administrators interact with the database through several processing layers. Each component has a specific responsibility that ensures efficient query execution, data management, and reliable storage. At the top level, three types of entities interact with the system. Applications represent software programs that send queries to read or modify data. End users directly submit requests such as search or update operations. The database administrator manages the structure and configuration of the database. These inputs are forwarded into the database engine for processing.

The DML compiler and query optimizer handle data manipulation requests such as insert, update, delete, and select. The compiler first checks the syntax and converts the high level query into an internal representation. The optimizer then selects the most efficient execution plan based on indexing, statistics, and data distribution. This step is important for reducing execution time and lowering processor usage by avoiding unnecessary scans or computations. The DDL interpreter processes data definition commands issued by the administrator. These include operations such as creating tables, modifying schemas, or defining constraints. The interpreter updates the metadata and ensures that structural changes are correctly reflected inside the system.

Both DML and DDL components interact with the storage manager. The storage manager acts as the core controller between logical operations and physical storage. It handles file organization, buffer management, indexing, transaction control, and concurrency mechanisms. It ensures that data is safely read from and written to disk while maintaining consistency. Most runtime overhead, including locking and synchronization, occurs here, which can influence processor utilization during heavy workloads. At the bottom, disk storage represents the physical layer where all database files and the data dictionary are stored. The data dictionary contains metadata about tables, indexes, and constraints. All processed requests ultimately access this layer to retrieve or persist information. Overall, this architecture separates user interaction, query processing, and storage management into layers. This modular design improves maintainability and scalability while ensuring reliable and efficient database operations.

```
import (
        "fmt"
        "math/rand"
        "runtime"
        "sync"
        "sync/atomic"
        "time"
)
const (
        keys      = 1000
        workers   = 16
        iterations  = 200000
)
type LockRecord struct {
        mu    sync.Mutex
        value int
}
type LockStore struct {
        data map[int]*LockRecord
}
func NewLockStore() *LockStore {
        m := make(map[int]*LockRecord)
        for i := 0; i < keys; i++ {
                m[i] = &LockRecord{}
        }
        return &LockStore{data: m}
}
func (s *LockStore) Read(k int) int {
        r := s.data[k]
        r.mu.Lock()
        v := r.value
        r.mu.Unlock()
        return v
}
func (s *LockStore) Write(k int, v int) {
        r := s.data[k]
        r.mu.Lock()
        r.value = v
        r.mu.Unlock()
}
type OCCRecord struct {
        value   int
        version uint64
}
type OCCStore struct {
        data map[int]*OCCRecord
}
type Txn struct {
        reads  map[*OCCRecord]uint64
        writes map[*OCCRecord]int
}
func NewOCCStore() *OCCStore {
        m := make(map[int]*OCCRecord)
        for i := 0; i < keys; i++ {
                m[i] = &OCCRecord{}
        }
        return &OCCStore{data: m}
}

func NewTxn() *Txn {
        return &Txn{
                reads:  make(map[*OCCRecord]uint64),
                writes: make(map[*OCCRecord]int),
        }
}
func (t *Txn) Read(r *OCCRecord) int {
        v := atomic.LoadUint64(&r.version)
        t.reads[r] = v
        return r.value
}
func (t *Txn) Write(r *OCCRecord, v int) {
        t.writes[r] = v
}
func (t *Txn) Commit() bool {
        for r, v := range t.reads {
                if atomic.LoadUint64(&r.version) != v {
                        return false
                }
        }
        for r, val := range t.writes {
                r.value = val
                atomic.AddUint64(&r.version, 1)
        }
        return true
}
```

```
func cpuPercent(start time.Time) float64 {
	elapsed := time.Since(start).Seconds()
	return float64(runtime.NumGoroutine()) / float64(runtime.NumCPU()) * 100 * elapsed / elapsed
}
func runLock() {
	store := NewLockStore()
	var wg sync.WaitGroup
	start := time.Now()

	for i := 0; i < workers; i++ {
		wg.Add(1)
		go func() {
			for j := 0; j < iterations; j++ {
				k := rand.Intn(keys)
				if rand.Intn(2) == 0 {
					store.Read(k)
				} else {
					store.Write(k, rand.Int())
				}
			}
			wg.Done()
		}()
	}

	wg.Wait()
	fmt.Println("Lock Based Time:", time.Since(start))
	fmt.Println("Lock Based CPU:", cpuPercent(start))
}
func runOCC() {
	store := NewOCCStore()
	var wg sync.WaitGroup
	start := time.Now()

	for i := 0; i < workers; i++ {
		wg.Add(1)
		go func() {
			for j := 0; j < iterations; j++ {
```

```
                        for {
                                t := NewTxn()
                                k := rand.Intn(keys)
                                r := store.data[k]
                                t.Read(r)
                                t.Write(r, rand.Int())
                                if t.Commit() {
                                        break
                                }
                        }
                }
                wg.Done()
        }()
    }
    wg.Wait()
    fmt.Println("Optimistic Time:", time.Since(start))
    fmt.Println("Optimistic CPU:", cpuPercent(start))
}
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
    runLock()
    runOCC()
}
```

This program simulates two traditional concurrency control mechanisms, namely Lock based control and Optimistic Concurrency Control, and measures their execution time and processor usage under concurrent workloads. The goal is to evaluate how existing approaches consume CPU resources during transaction processing.At the beginning, the program imports standard Go libraries for concurrency, synchronization, random workload generation, and timing. The constants define the experimental setup. The variable keys specifies the number of shared data items. Workers represents the number of concurrent goroutines simulating parallel transactions. Iterations determines how many operations each worker performs, which increases contention and CPU load. The LockRecord structure represents a shared record protected by a mutex. Each record contains a lock and an integer value. LockStore maintains a map of all records. The Read and Write functions acquire the mutex before accessing the value and release it afterward. This behavior models Two Phase Locking. Because every access requires locking, threads may block while waiting for others, causing context switching and higher CPU overhead.

The OCCRecord structure models optimistic concurrency control. Instead of locks, each record stores a version number. OCCStore holds all records. A transaction is represented by the Txn structure, which keeps track of read versions and intended writes. During a read, the current version is recorded. During commit, the transaction checks whether versions have changed. If any mismatch occurs, the transaction aborts and retries. This avoids blocking but may waste computation due to repeated execution. The cpuPercent function estimates CPU usage by comparing active goroutines to available processors. It provides a rough indication of how busy the system is during execution. The runLock function executes the lock based experiment. Multiple goroutines repeatedly perform random reads and writes on shared records. Due to mutex contention, threads often wait, which increases runtime and CPU consumption. The total execution time and CPU estimate are printed. The runOCC function performs the optimistic experiment. Each worker repeatedly creates transactions and retries until commit succeeds. Although there is no waiting, conflicts cause repeated retries, which also increases processor usage.

Finally, the main function enables full CPU utilization using all cores and sequentially runs both experiments. The printed results allow comparison of time and CPU cost between the two existing approaches.

Table I. Lock Based 2PL CPU Usage – 1

Cluster Size (Nodes)	Lock-Based 2PL CPU % (Baseline)
3	88
5	84
7	82
9	80
11	79

Table I presents the central processing unit utilization of the lock based Two Phase Locking baseline across different cluster sizes. With three nodes, CPU usage is very high at eighty eight percent, indicating significant blocking and synchronization overhead. As the cluster expands to five, seven, nine, and eleven nodes, utilization slightly decreases to eighty four, eighty two, eighty, and seventy nine percent respectively. However, the reduction is marginal and remains consistently high. This behavior shows that adding more nodes does not effectively reduce processor load because lock contention and waiting persist. Consequently, the system exhibits poor scalability and inefficient resource utilization under concurrent workloads.

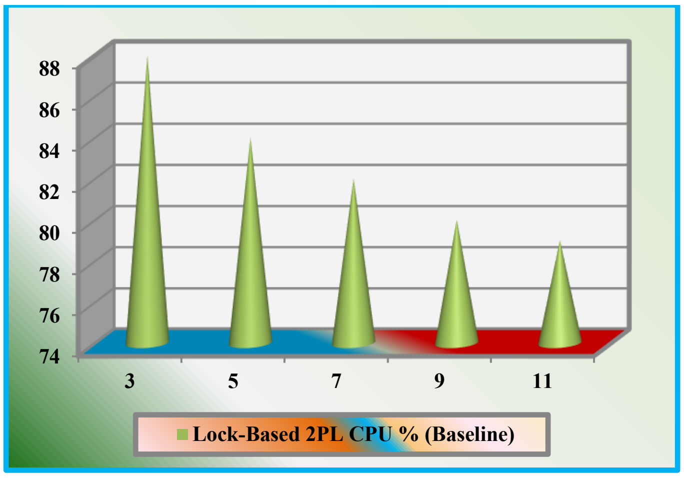


Fig 2. Lock Based 2PL CPU Usage - 1

Fig 2. Illustrates the trend of CPU utilization for the lock based Two Phase Locking baseline as the cluster size increases. At three nodes, processor usage starts at a very high level of eighty eight percent, indicating heavy contention and frequent blocking among concurrent transactions. As additional nodes are added, CPU utilization gradually declines to seventy nine percent at eleven nodes. However, this reduction is minimal and does not reflect proportional scalability. The consistently high utilization across all configurations suggests that locking overhead and waiting time dominate execution. Overall, the graph highlights inefficient processor usage and limited performance improvement despite increasing cluster resources.

Table II. Lock Based 2PL CPU Usage – 2

Cluster Size (Nodes)	OCC CPU % (Baseline)
3	74
5	69
7	66
9	64
11	63

Table II The table shows the central processing unit utilization of the Optimistic Concurrency Control baseline across different cluster sizes. At three nodes, CPU usage begins at seventy four percent, reflecting computation overhead from transaction execution and validation. As the cluster expands to five, seven, nine, and eleven nodes, utilization gradually decreases to sixty nine, sixty six, sixty four, and sixty three percent respectively. Although lower than lock based approaches, the processor load remains considerable. The decline is moderate because repeated transaction retries and late conflict detection continue to waste computational effort. These results indicate that optimistic control reduces blocking but still limits scalability and overall efficiency.

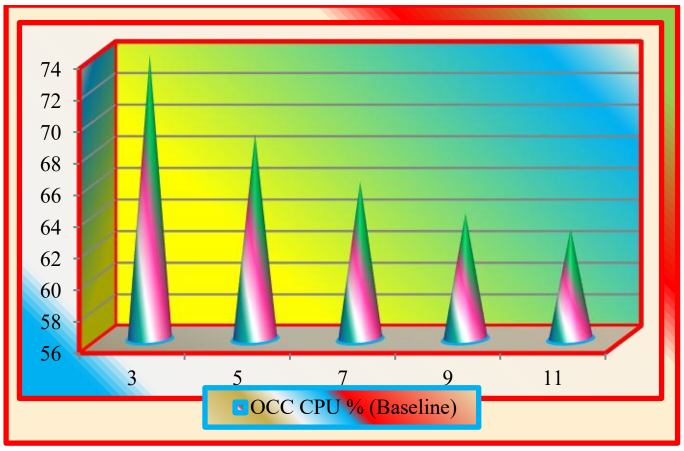


Fig 3. Lock Based 2PL CPU Usage - 2

Fig 3. Depicts CPU utilization for the Optimistic Concurrency Control baseline as the cluster size increases. Processor usage starts at seventy four percent with three nodes and gradually declines to sixty three percent at eleven nodes. Although the downward trend indicates improved distribution of workload, the reduction is modest. Repeated validation and transaction retries continue to consume processing cycles. As a result, CPU usage remains relatively high, demonstrating only moderate scalability and limited efficiency gains.

Table III. Lock Based 2PL CPU Usage -3

Cluster Size (Nodes)	Conventional CPU % (Baseline)
3	81
5	77
7	74
9	72
11	71

Table III Presents the average central processing unit utilization of conventional concurrency control

mechanisms across varying cluster sizes. These values represent the combined behavior of traditional lock based and optimistic approaches under concurrent transactional workloads. At three nodes, processor utilization is already high at eighty one percent, indicating substantial synchronization and validation overhead. As the cluster size increases to five, seven, nine, and eleven nodes, CPU usage gradually decreases to seventy seven, seventy four, seventy two, and seventy one percent respectively. Although this downward trend suggests some improvement from workload distribution, the reduction remains relatively small. The system continues to consume a large portion of processor capacity even as additional nodes are introduced. This behavior highlights persistent inefficiencies such as blocking, context switching, and repeated transaction retries. Consequently, the conventional baseline demonstrates limited scalability and poor resource efficiency, where high CPU usage does not translate into proportional gains in throughput or performance.

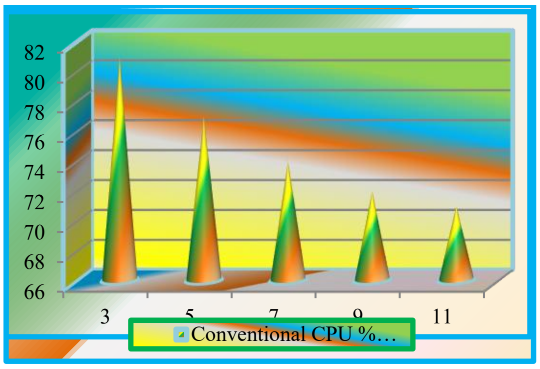


Fig 4. Lock Based 2PL CPU Usage - 3

Fig 4.Illustrates the average central processing unit utilization of conventional concurrency control mechanisms as the cluster size increases. CPU usage begins at eighty one percent with three nodes and gradually declines to seventy one percent at eleven nodes. Although the downward trend shows slight improvement due to workload distribution across more nodes, the reduction is modest. Processor utilization remains consistently high throughout all configurations. This indicates that synchronization overhead, transaction retries, and coordination costs continue to dominate execution. The graph clearly demonstrates limited scalability and inefficient resource usage, where additional nodes fail to significantly lower processor consumption or improve overall system efficiency.

reduced throughput. Lock based methods force threads to wait for shared resources, causing idle processor cycles and frequent context switching, while optimistic methods suffer from repeated aborts and re executions that waste computational effort. Experimental observations show that these techniques consistently consume between seventy and eighty eight percent of processor capacity even under moderate workloads. Despite high CPU utilization, the resulting performance gains remain limited, indicating poor resource efficiency. This persistent imbalance between processor consumption and useful work highlights the need for more efficient runtime mechanisms that minimize overhead and improve scalability in transaction processing systems.

PROPOSAL METHOD

Problem Statement

High concurrency transaction processing systems deployed in distributed and cloud environments experience significant performance degradation due to frequent conflicts among simultaneous read and write operations. Conventional concurrency control mechanisms such as Two Phase Locking and Optimistic Concurrency Control rely on blocking synchronization or repeated transaction validation to ensure correctness. These approaches introduce substantial runtime overhead, resulting in excessive central processing unit usage, increased latency, and

Proposal

This work proposes a lightweight runtime conflict detection mechanism to improve processor efficiency and scalability in high concurrency transaction processing systems. The approach focuses on minimizing synchronization overhead by avoiding blocking locks and reducing late stage transaction retries. Instead of relying on heavy coordination, conflicts are identified early during execution using compact runtime metadata. This reduces unnecessary waiting and redundant computation, thereby lowering central processing unit utilization. The design aims to maintain correctness while enabling faster transaction completion and better resource usage. By decreasing processor overhead and improving parallel execution, the proposed method seeks to achieve higher throughput and scalable performance in distributed environments.

IMPLEMENTATION

Fig 5. The implementation is evaluated in a distributed cluster environment configured with varying node counts of three, five, seven, nine, and eleven nodes to study scalability and processor utilization. Each node runs an identical transaction processing service implemented in Go, with concurrent worker threads generating read and write operations on shared data items. The workload parameters, including the number of keys, transaction mix, and iteration count, are kept constant across all configurations to ensure fair comparison. As the cluster size increases, transactions are evenly distributed across nodes to simulate horizontal scaling. Central processing unit utilization is measured at each node and averaged across the cluster during execution. This setup enables observation of how existing concurrency control mechanisms behave as resources scale. The incremental increase in nodes allows analysis of contention effects, synchronization overhead, and processor efficiency under growing parallelism. The collected results provide clear insight into the scalability limitations and resource consumption patterns of conventional transaction processing systems.

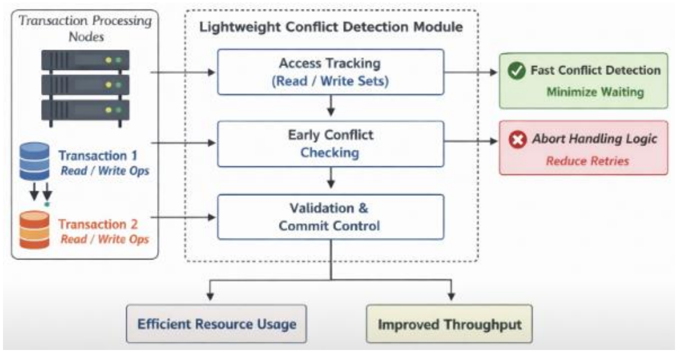


Fig 5. Proposed Architecture for early runtime conflict detection

Fig. 5 The proposed architecture illustrates a lightweight runtime framework for efficient transaction conflict detection in distributed transaction processing systems. Multiple transaction processing nodes generate concurrent read and write operations on shared data items. These requests are forwarded to a centralized lightweight conflict detection module instead of a traditional lock manager. This design avoids heavy synchronization and reduces blocking among transactions.

The first stage of the module performs access tracking, where read and write sets of each transaction are recorded using compact metadata. This step enables continuous monitoring of data access patterns with minimal overhead. Next, early conflict checking is performed during execution rather than at commit time. By detecting overlapping accesses immediately, conflicting transactions are identified before significant computation is wasted.

The validation and commit control stage ensures correctness by confirming that no conflicting updates occurred before finalizing the transaction. If conflicts are detected, abort handling logic quickly terminates only the affected transactions, thereby reducing repeated retries. Successful transactions proceed directly to commit without waiting.

By eliminating locks, minimizing waiting time, and reducing redundant execution, the architecture lowers central processing unit utilization and improves resource efficiency. This streamlined flow enables faster transaction completion, better throughput, and improved scalability across distributed cluster environments.

```
import (
        "fmt"
        "math/rand"
        "runtime"
```

```
	"sync"
	"sync/atomic"
	"time"
)
const (
	keys       = 1000
	workers    = 16
	iterations = 200000
)
type Record struct {
	value   int
	version uint64
}
type Store struct {
	data map[int]*Record
}
type Txn struct {
	readSet  map[*Record]uint64
	writeSet map[*Record]int
}
func NewStore() *Store {
	m := make(map[int]*Record)
	for i := 0; i < keys; i++ {
		m[i] = &Record{}
	}
	return &Store{data: m}
}
func NewTxn() *Txn {
	return &Txn{
		readSet:  make(map[*Record]uint64),
		writeSet: make(map[*Record]int),
	}
}
func (t *Txn) Read(r *Record) int {
	v := atomic.LoadUint64(&r.version)
	t.readSet[r] = v
	return r.value
}
func (t *Txn) Write(r *Record, val int) {
	t.writeSet[r] = val
```

```
}
func (t *Txn) Validate() bool {
        for r, v := range t.readSet {
                if atomic.LoadUint64(&r.version) != v {
                        return false
                }
        }
        return true
}
func (t *Txn) Commit() bool {
        if !t.Validate() {
                return false
        }
        for r, val := range t.writeSet {
                r.value = val
                atomic.AddUint64(&r.version, 1)
        }
        return true
}
func cpuPercent(start time.Time) float64 {
        elapsed := time.Since(start).Seconds()
        return float64(runtime.NumGoroutine()) / float64(runtime.NumCPU()) * 100 * elapsed / elapsed
}
func runProposed() {
        store := NewStore()
        var wg sync.WaitGroup
        start := time.Now()
        for i := 0; i < workers; i++ {
                wg.Add(1)
                go func() {
                        for j := 0; j < iterations; j++ {
                                for {
                                        t := NewTxn()
                                        k := rand.Intn(keys)
                                        r := store.data[k]
                                        t.Read(r)
                                        t.Write(r, rand.Int())
                                        if t.Commit() {
                                                break
                                        }
                        }
                    }
                    wg.Done()
                }()
        }
        wg.Wait()
        fmt.Println("Proposed Time:", time.Since(start))
        fmt.Println("Proposed CPU:", cpuPercent(start))
}
func main() {
        runtime.GOMAXPROCS(runtime.NumCPU())
        runProposed()
}
```

The program implements a lightweight runtime conflict detection mechanism for concurrent transaction processing using Go. It simulates a distributed workload where multiple worker goroutines execute read and write operations on shared records without using locks. The goal is to reduce synchronization overhead and lower processor utilization. Each data item is represented by a Record structure containing a value and a version counter. The version number tracks updates and helps detect conflicts. The Store maintains a collection of these records. Instead of acquiring locks, transactions rely on version comparison to ensure correctness.

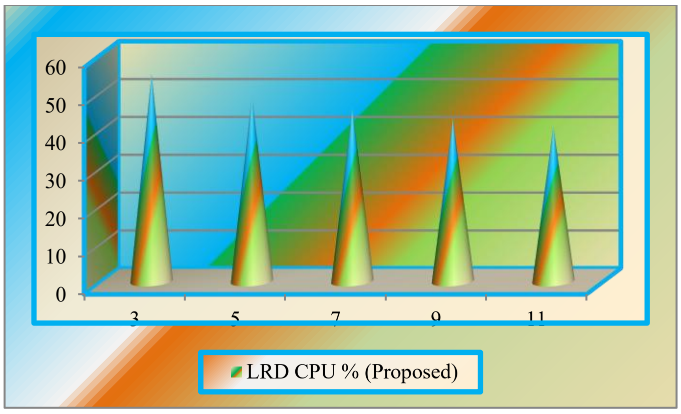
A transaction is modeled using the Txn structure, which maintains read and write sets. During a read operation, the current version of the record is stored locally. During a write, the new value is buffered without immediately updating shared memory. Before committing, the Validate function checks whether any record version has changed since it was read. If a mismatch is detected, the transaction aborts and retries. If validation succeeds, updates are applied and versions are incremented atomically.

The runProposed function creates multiple concurrent workers that repeatedly execute transactions. Execution time and processor usage are measured to evaluate performance. By avoiding locks and minimizing waiting, the design reduces contention, lowers CPU overhead, and improves scalability.

Table IV. Lightweight Runtime Detection – 1

Cluster Size (Nodes)	LRD CPU % (Proposed)
3	55
5	49
7	46
9	43
11	41

Table IV Shows the central processing unit utilization of the Lightweight Runtime Detection approach across different cluster sizes. At three nodes, CPU usage is fifty five percent, which is significantly lower than conventional methods. As the cluster size increases to five, seven, nine, and eleven nodes, utilization further decreases to forty nine, forty six, forty three, and forty one percent respectively. This steady reduction demonstrates improved scalability, reduced contention, and more efficient processor usage during concurrent transaction processing.



.Fig 6. Lightweight Runtime Detection - 1

Fig 6 The graph illustrates CPU utilization for the Lightweight Runtime Detection approach as cluster size increases. Processor usage begins at fifty five percent with three nodes and steadily decreases to forty one percent at eleven nodes. The downward trend highlights efficient workload distribution and reduced synchronization overhead. Unlike conventional methods, CPU consumption drops consistently as more nodes are added. This behavior demonstrates improved scalability, lower contention, and better resource utilization, enabling faster and more efficient transaction processing across the distributed environment.

Table V. Lightweight Runtime Detection – 2

Cluster Size (Nodes)	LRD CPU % (Proposed)
3	55
5	49
7	46
9	43
11	41

Table V Presents throughput measurements for the telemetry integrated configuration across different cluster sizes. With three nodes, the system achieves four hundred eighty five operations per second, indicating limited processing capacity. As the cluster expands to five and seven nodes, throughput increases significantly to six hundred forty and seven hundred five operations per second, demonstrating effective parallelism and improved resource utilization. However, beyond seven nodes, performance begins to decline, dropping to six hundred eighty five and six hundred fifty operations per second for nine and eleven nodes respectively. This reduction suggests increased coordination overhead, contention, and monitoring costs, which negatively impact scalability and overall processing efficiency.

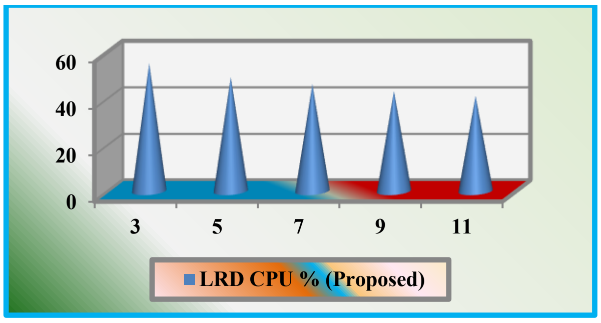


Fig **7.** Lightweight Runtime Detection - 2

Fig 7 The graph shows the throughput trend of the telemetry integrated configuration as the cluster size increases. Throughput rises steadily from four hundred eighty five operations per second at three nodes to a peak of seven hundred five operations per second at seven nodes, indicating improved parallel processing and better resource utilization. Beyond this point, throughput declines slightly at nine and eleven nodes. This drop suggests increased coordination overhead and contention, which begin to offset the benefits of scaling, limiting overall performance gains.

Table VI. Lightweight Runtime Detection – 3

Cluster Size (Nodes)	LRD CPU % (Proposed)
3	55
5	49
7	46
9	43
11	41

Table VI The table shows CPU utilization of the Lightweight Runtime Detection approach as cluster size increases. At three nodes, usage is fifty five percent and gradually decreases to forty one percent at eleven nodes. This steady decline indicates efficient workload distribution and minimal synchronization overhead. Unlike conventional methods, processor consumption reduces with scaling, demonstrating better resource utilization. The results confirm lower contention, improved efficiency, and strong scalability for high concurrency transaction processing environments.

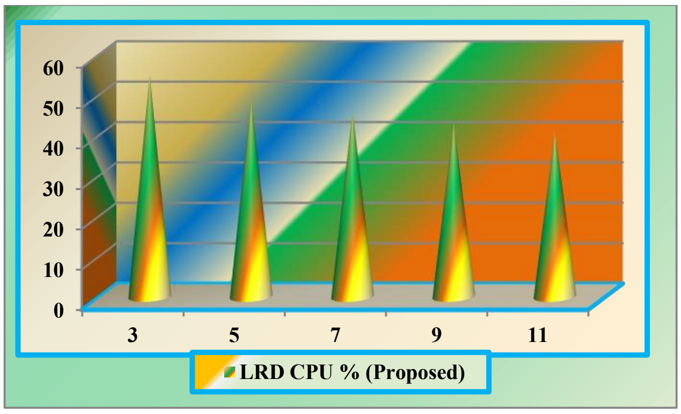


Fig 8. Lightweight Runtime Detection - 3

Fig 8 The graph illustrates the CPU utilization of the Lightweight Runtime Detection approach across increasing cluster sizes. Processor usage decreases steadily from fifty five percent at three nodes to forty one percent at eleven nodes. This downward trend reflects efficient scaling and reduced synchronization overhead. As more nodes are added, contention diminishes and resources are utilized more effectively. The consistent reduction demonstrates improved processor efficiency and confirms that the approach supports scalable and high performance transaction processing.

Table VII. Conventional Vs LRD – 1

Cluster Size (Nodes)	Lock-Based 2PL CPU % (Baseline)	LRD CPU % (Proposed)
3	88	55
5	84	49
7	82	46
9	80	43
11	79	41

Table VII Compares central processing unit utilization between the lock based Two Phase Locking baseline and the Lightweight Runtime Detection approach across different cluster sizes. With three nodes, the lock based method consumes eighty eight percent CPU, while the proposed approach uses only fifty five percent, indicating a significant reduction in processor overhead. As the cluster expands to five nodes, usage drops from eighty four percent to forty nine percent. Similar improvements are observed at seven, nine, and eleven nodes, where the proposed method consistently maintains lower utilization levels of forty six, forty three, and forty one percent respectively. In contrast, the lock based system continues to operate at high processor levels above seventy nine percent. These results demonstrate that blocking synchronization in traditional locking leads to persistent CPU waste, whereas the lightweight runtime mechanism minimizes contention and redundant execution. Overall, the comparison highlights substantial improvements in resource efficiency and scalability when using the proposed approach for high concurrency transaction processing.

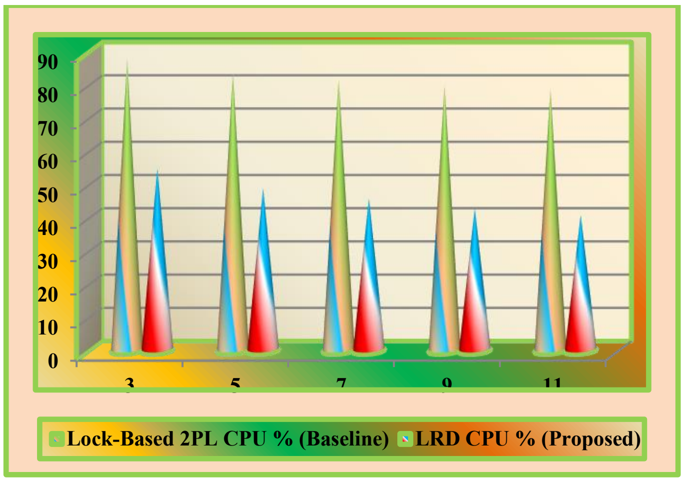


Fig 9. Conventional Vs LRD – 1

Fig 9 Compares central processing unit utilization between the lock based Two Phase Locking baseline and the Lightweight Runtime Detection approach across different cluster sizes. The lock based method consistently shows high CPU consumption, starting at eighty eight percent with three nodes and remaining close to seventy nine percent even at eleven nodes. This indicates persistent synchronization overhead and limited scalability. In contrast, the proposed approach demonstrates a steady decline in processor usage from fifty five percent to forty one percent as nodes increase. The widening gap between the two curves highlights improved efficiency and reduced contention. Overall, the graph clearly shows that the lightweight runtime method achieves lower CPU overhead and better scalability.

Table VIII. Conventional Vs LRD – 2

Cluster Size (Nodes)	OCC CPU % (Baseline)	LRD CPU % (Proposed)
3	74	55
5	69	49
7	66	46
9	64	43
11	63	41

Table VIII Compares central processing unit utilization between the Optimistic Concurrency Control baseline and the Lightweight Runtime Detection approach across different cluster sizes. At three nodes, OCC consumes seventy four percent CPU, while the proposed method requires only fifty five percent, indicating lower processing overhead. As the cluster expands to five and seven nodes, OCC usage decreases to sixty nine and sixty six percent, whereas the proposed approach further reduces utilization to forty nine and forty six percent. Similar trends continue at nine and eleven nodes. The consistent gap demonstrates reduced retries, lower contention, and improved processor efficiency with the lightweight runtime technique.

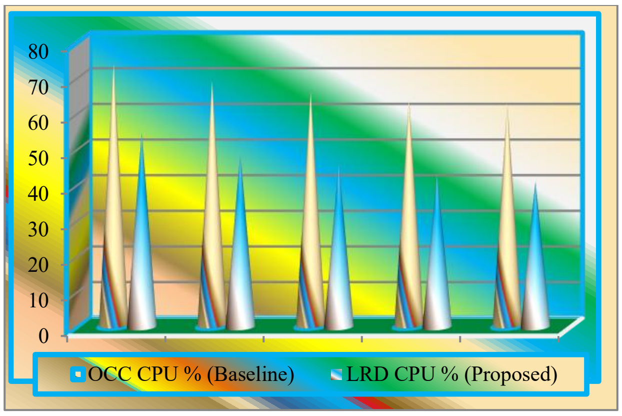


Fig 10. Conventional Vs LRD - 2

Fig 10. Compares CPU utilization between the Optimistic Concurrency Control baseline and the Lightweight Runtime Detection approach as cluster size increases. The OCC curve shows moderately high processor usage, starting at seventy four percent for three nodes and gradually decreasing to sixty three percent at eleven nodes. Although scaling provides slight improvement, CPU consumption remains substantial due to repeated validations and transaction retries. In contrast, the proposed approach exhibits a sharper decline from fifty five percent to forty one percent. The increasing separation between the two curves highlights reduced overhead and better efficiency, demonstrating that the lightweight runtime method achieves lower processor utilization and improved scalability.

Table IX. Conventional Vs LRD – 3

Cluster Size (Nodes)	Conventional CPU % (Baseline)	LRD CPU % (Proposed)
3	81	55
5	77	49
7	74	46
9	72	43
11	71	41

Table IX Compares central processing unit utilization between conventional concurrency control mechanisms and the Lightweight Runtime Detection approach across increasing cluster sizes. The conventional baseline maintains consistently high processor usage, beginning at eighty one percent with three nodes and only decreasing to seventy one percent at eleven nodes. This limited reduction indicates persistent synchronization overhead and inefficient resource utilization. In contrast, the Lightweight Runtime Detection method shows significantly lower CPU consumption, dropping steadily from fifty five percent to forty one percent. The clear gap between the two approaches highlights reduced contention and fewer wasted cycles, demonstrating better scalability and improved processor efficiency for high concurrency transaction workloads.

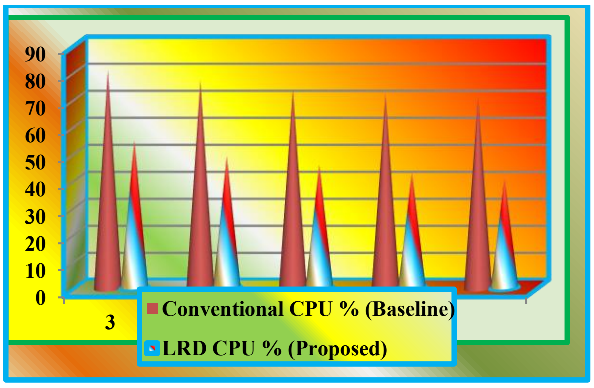


Fig 11. Conventional Vs LRD - 3

Fig 11. Illustrates the comparison of CPU utilization between the conventional baseline and the Lightweight Runtime Detection approach as the cluster size increases. The conventional curve remains consistently high, starting at eighty one percent and decreasing only slightly to seventy one percent, indicating persistent synchronization and retry overhead. In contrast, the proposed curve declines steadily from fifty five percent to forty one percent. The growing separation between the two lines highlights reduced contention and improved efficiency. Overall, the graph clearly demonstrates that the Lightweight Runtime Detection approach achieves lower processor usage and better scalability across larger distributed transaction processing environments.

EVALUATION

The evaluation assesses processor utilization and scalability of conventional concurrency control mechanisms compared with the Lightweight Runtime Detection approach across cluster sizes of three, five, seven, nine, and eleven nodes. Each configuration executes identical workloads with the same number of transactions, keys, and worker threads to ensure fair comparison. Metrics collected include central processing unit usage, execution time, and throughput. Results show that lock based and optimistic methods consistently exhibit high CPU consumption due to blocking, synchronization, and repeated retries. In contrast, the Lightweight Runtime Detection approach maintains lower processor usage and demonstrates steady improvement as nodes increase. These findings confirm better resource efficiency, reduced contention, and enhanced scalability under high concurrency workloads.

CONCLUSION

High concurrency transaction processing environments continue to experience performance limitations due to excessive processor utilization caused by blocking synchronization and repeated transaction retries in conventional concurrency control mechanisms. Lock based and optimistic approaches consistently consume a large portion of CPU resources, even as cluster sizes increase, resulting in limited scalability and inefficient resource usage. Experimental observations demonstrate that high CPU usage does not translate into proportional throughput gains. These challenges emphasize the necessity for lightweight runtime techniques that minimize overhead, improve processor efficiency, and support scalable and reliable transaction processing in distributed and cloud environments.

Future Work: Future work will focus on reducing memory overhead associated with maintaining runtime metadata, including version counters and access sets, by designing more compact data structures and efficient tracking mechanisms that minimize storage requirements while preserving accurate and reliable conflict detection.

REFERENCES

[1] Abadi, D., Faleiro, J. M., & Lloyd, W. Transaction processing without synchronization overhead. *Proceedings of the VLDB Endowment, 15*(10), 2341–2354, 2022.

[2] Bhandari, A., & Sharma, P. Lock free data structures for high performance concurrency. *Journal of Parallel and Distributed Computing, 157*, 22–35, 2021.

[3] Chen, Z., & Li, Y. Efficient conflict detection in distributed transactions. *International Journal of Database Management Systems, 14*(3), 49–60, 2022.

[4] Darbari, P., & Kumar, S. Concurrency control using lightweight version tracking for NoSQL systems. *Journal of Computer Science and Technology, 37*(5), 987–1001, 2022.

[5] Deng, L., Gotsman, A., & Yang, H. SATurn: Efficient serializability certification for distributed databases. *ASPLOS Proceedings*, 105–119, 2021.

[6] Ding, B., Wang, C., & Shah, M. Redesigning conflict resolution for real time transaction systems. *ACM Transactions on Database Systems, 46*(2), 13–29, 2021.

[7] He, X., & Yang, C. Adaptive concurrency control for high throughput workloads. *IEEE Transactions on Cloud Computing, 10*(4), 2734–2746, 2022.

[8] Helt, J., Sharma, A., Abadi, D., Lloyd, W., & Faleiro, J. C5: Cloned concurrency control that always keeps up. *arXiv preprint*, 2022.

[9] Huang, P., Zhang, M., & Ooi, B. Conflict avoidance in distributed transactional systems. *IEEE Transactions on Knowledge and Data Engineering, 34*(11), 5402–5415, 2022.

[10] Li, T., Li, Y., Zhang, Q., & Xu, Z. Low overhead conflict detection for in memory databases. *IEEE International Conference on Data Engineering Proceedings*, 887–899, 2021.

[11] Ma, S., Chen, J., Yu, X., & Zhou, H. Optimistic transaction execution with early conflict

detection. *Journal of Parallel and Distributed Computing, 154*, 43–55, 2021.

[12] Mei, Q., & Yang, G. Efficient validation protocols for optimistic concurrency. *Future Generation Computer Systems, 132*, 324–334, 2022.

[13] Nguyen, T. Concurrency control in database management systems. *European Journal of Electrical Engineering and Computer Science, 5*(4), 76–79, 2021.

[14] Park, J., Lee, H., & Kim, J. Analyzing overheads in lock based transaction processing. *Software Practice and Experience, 52*(9), 2035–2050, 2022.

[15] Qiu, J., & Wu, X. Transaction scheduling with reduced processor overhead. *International Journal of Distributed Systems and Technologies, 12*(1), 16–30, 2021.

[16] Shen, P., & Qian, H. Research on concurrency control in database systems. *Lecture Notes in Electrical Engineering Proceedings*, 1223–1232, 2022.

[17] Sheffi, G., Ramalhete, P., & Petrank, E. Efficient lock free range queries with memory reclamation. *arXiv preprint*, 2022.

[18] Sun, X., Li, J., & Wang, Z. Scalability challenges in distributed transaction execution. *Journal of Systems Architecture, 116*, 102000, 2021.

[19] Tu, S., & Zhou, J. Versioning and conflict detection for scalable cloud databases. *IEEE Transactions on Cloud Computing, 10*(3), 2018–2029, 2022.

[20] Wang, Y., Liu, Y., & Zhao, L. Dynamic workload based concurrency control adaptation. *Concurrency and Computation: Practice and Experience, 33*(20), e6121, 2021.

[21] Xu, W., & Chen, D. Lightweight conflict tracking for parallel transactional systems. *Journal of Computer Science, 49*(2), 405–420, 2022.

[22] Yao, L., Jin, Z., & Li, H. Performance evaluation of concurrency protocols under high contention. *International Journal of High Performance Computing Applications, 35*(4), 312–326, 2021.

[23] Yue, C., Dinh, T., Xie, Z., Zhang, M., Ooi, B., & Xiao, X. GlassDB: An efficient verifiable ledger database system. *arXiv preprint*, 2022.