iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** IV **Month of publication:** April 2026

DOI: https://doi.org/10.22214/ijraset.2026.80412

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue IV Apr 2026- Available at www.ijraset.com

Efficient Data Structures for Real-Time Applications: Design, Performance Analysis, and Optimization Strategies

Pratish Pranjal Kumar
B.Tech, Computer Science & Engineering, Parul University, Vadodara, Gujarat, India

Abstract: ***Real-time systems demand more than raw speed—they must guarantee response times even under heavy, continuously shifting data loads. The choice of data organization strategy is therefore critical: it shapes not only throughput but also timing predictability, memory footprint, and scalability. This paper surveys data structures suited to systems with strict timing requirements. Both foundational structures—arrays, linked lists, trees, hash tables—and advanced mechanisms such as kinetic data structures, bit-vector encodings, segment trees, and lock-free containers are examined. Time-space tradeoffs, worst-case versus average-case behavior, and suitability for hard versus soft real-time deadlines are analyzed for each. Domain-specific applications across network switching, game engines, autonomous vehicles, financial trading platforms, and AI inference pipelines are discussed, with a proposed hybrid architecture combining adaptive indexing, column-oriented memory layouts, and parallel algorithms to achieve genuine sub-millisecond latency at scale. Analysis is grounded in complexity theory and practical performance observations.***

Keywords: ***concurrent algorithms, data structures, lock-free programming, low latency, memory hierarchy, optimization, real-time systems, time complexity***

I. INTRODUCTION

Real-time computing now sits quietly at the heart of industries most people take for granted. From high-frequency trading floors where milliseconds translate directly to revenue, to collision avoidance systems in autonomous vehicles, to platforms streaming live content to millions simultaneously—these applications share a common constraint: deadlines cannot be missed.

The underlying design question is straightforward even when the answer is not: how should data be organized so that a system can respond quickly, predictably, and without running into memory exhaustion or pathological slow paths? No universal answer exists. A network router may need lookups measured in nanoseconds; a game engine's collision detection must refresh spatial models hundreds of times per second; a patient monitoring system must issue alerts even under peak system load.

This paper approaches the problem incrementally. The distinction between hard real-time (HRT) constraints, where a single deadline violation constitutes a system failure, and soft real-time (SRT) targets, where occasional misses degrade quality but do not cause catastrophic outcomes, is established first. Five primary contributions follow: (1) a classification of data structures by timing suitability, including worst-case memory usage and operation bounds; (2) an argument for why worst-case complexity—not amortized complexity—is the correct lens for real-time analysis; (3) an examination of how domain-specific requirements shift in networking, gaming, AI inference, and financial systems; (4) a modular layered architecture combining multiple structure types; and (5) a forward-looking discussion on machine learning-driven indexes and hardware-software co-design.

II. BACKGROUND AND THEORETICAL FRAMEWORK

A. Characterizing Real-Time Constraints

Real-time systems are typically described along two axes. The first is deadline strictness: HRT systems tolerate zero violations, while SRT systems allow occasional deadline misses as long as overall quality remains acceptable. The second axis concerns task arrival patterns—periodic tasks arrive at fixed intervals, event-driven tasks are triggered by external signals, and sporadic tasks are unpredictable but bounded in minimum inter-arrival time.

For data structure selection, the properties that matter most are: (1) worst-case rather than average-case operation time; (2) space consumption in the worst case; (3) cache behavior; (4) safe concurrent access from multiple threads; and (5) the absence of latency-unpredictable operations such as heap reallocation during critical execution paths.

©IJRASET: All Rights are Reserved

B. The Problem with Amortized Analysis

Amortized analysis averages operation cost over a long sequence of calls—and by that measure, structures like dynamic arrays and Fibonacci heaps appear highly efficient. The difficulty is that amortized guarantees hide occasional expensive operations: a hash table resizing or a heap reorganization can produce a pause that is orders of magnitude longer than the typical case. For a system rendering at 60 frames per second, such a pause introduces a visible stutter. For a safety-critical controller, the consequences can be severe.

This is why real-time practitioners favor structures with hard worst-case guarantees: AVL or red-black trees over splay trees, ring buffers over expandable vectors, and lock-free queues over mutex-protected containers.

C. Memory Hierarchy Considerations

Modern processors expose a memory hierarchy stretching from registers and L1-L3 caches out to DRAM and persistent storage. The practical impact of this hierarchy is enormous. A cache miss costs approximately 100 to 300 CPU cycles, meaning that a linked list's pointer-chasing access pattern—theoretically efficient—is often slower in practice than a flat array traversal. Data structure selection must therefore account not only for Big-O complexity but also for cache locality, prefetch friendliness, and the behavior of multiple cores sharing cache lines.

III. CORE DATA STRUCTURES FOR REAL-TIME SYSTEMS

A. Arrays and Ring Buffers

Arrays remain the most cache-friendly general-purpose structure available. Index-based access is constant time, elements are laid out contiguously for hardware prefetching, and there is no pointer overhead. A 4K image buffer, for example, can be processed with SIMD vector operations applied directly to a flat allocation. In real-time contexts, arrays are typically pre-allocated to their maximum anticipated size at system startup, eliminating any runtime heap activity. Ring buffers extend this pattern to support constant-time enqueue and dequeue operations, making them standard in audio pipelines and inter-thread communication channels.

B. Linked Lists

Doubly linked lists allow O(1) insertion and deletion when the target node is already known, making them useful in schedulers where task priorities change frequently. The practical downside is that each pointer dereference can result in a cache miss, degrading throughput considerably. Accordingly, real-time kernels such as Linux RT and FreeRTOS use linked lists primarily for metadata bookkeeping rather than for storing bulk data.

C. Stacks and Queues

Both stacks and queues appear throughout real-time systems—stacks for expression evaluation and backtracking, queues for task dispatch and event buffering. When built atop pre-allocated arrays with a fixed capacity, all operations complete in constant time with no heap interaction. Multicore deployments use lock-free queue implementations backed by compare-and-swap instructions, allowing multiple threads to operate concurrently without mutual exclusion.

D. Hash Tables

Hash tables offer expected O(1) lookup, insertion, and deletion, forming the backbone of session caches, routing tables, and symbol indices. Pathological collision patterns or load-triggered resizing can, however, degrade performance sharply. Common mitigations include cuckoo hashing, which guarantees constant-time lookup regardless of occupancy; Robin Hood hashing, which limits probe chain variance; and perfect hashing for read-heavy workloads with a fixed, known key set.

E. Self-Balancing Search Trees

AVL trees and red-black trees both maintain O(log n) worst-case bounds for insertion, deletion, and lookup by enforcing balance invariants after each modification.

AVL trees apply stricter height constraints and therefore offer faster lookups; red-black trees relax the balance condition slightly for simpler rotation logic. Databases use both for ordered data, real-time schedulers rely on them for fast priority lookups, and POSIX timer implementations use red-black trees specifically because their guarantees are predictable.

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

F. Graph Structures

Graphs model relationships across domains: router topologies, neural network computation graphs, and vehicle navigation maps all use graph representations. Adjacency lists are standard for packet-switched networks where shortest-path computations run on sparse topologies. In neural network inference, layers form nodes in a directed acyclic graph whose traversal order determines computation sequence. Path planning in robotics uses graph search to find efficient routes through discretized physical space.

IV. ADVANCED DATA STRUCTURES FOR REAL-TIME APPLICATIONS

A. Priority Queues and Heaps

Priority queues determine task scheduling order, event sequencing, and packet prioritization. Binary heaps provide good cache behavior and consistent O(log n) worst-case bounds. D-ary heaps with higher branching factors can improve cache use in certain configurations. Fibonacci heaps offer attractive amortized bounds on paper, but their overhead in practice and lack of hard worst-case guarantees make them unsuitable for strict real-time scheduling.

B. Segment Trees and Fenwick Trees

Segment trees support range query and point update operations in O(log n) time, making them useful for game leaderboards, financial dashboards, and database aggregation queries. Lazy propagation allows bulk range updates to be deferred and batched efficiently. Fenwick trees, also called binary indexed trees, offer a more compact implementation of prefix-sum queries—appropriate when memory is constrained and the dataset fits comfortably in a single machine.

C. Tries and Finite Automata

Tries exploit shared key prefixes so that lookup, insertion, and deletion time depend only on the key length, not on the total number of stored keys. They are standard in autocomplete engines, longest-prefix matching for IP routing, and DNS caching. Patricia and radix tree variants collapse single-child chains to reduce space usage. For very large vocabularies, deterministic acyclic finite-state automata (DAFSA) answer membership queries in constant time with minimal memory overhead.

D. Spatial Data Structures

KD-trees, R-trees, and Bounding Volume Hierarchies (BVH) each address the problem of spatial queries over multidimensional data. KD-trees recursively partition space and support nearest-neighbor queries efficiently in low dimensions, though worst-case performance degrades to linear. R-trees maintain search efficiency as data shifts. BVHs are now the standard for real-time ray tracing in graphics hardware—NVIDIA's GPU ray-tracing pipelines rely on them to reduce intersection tests from O(n) to O(log n).

E. Kinetic Data Structures

When the data itself is in motion—particles in a physics simulation, vehicles on a map, agents in a crowd simulation—kinetic data structures maintain structural invariants through "certificates" that certify the current ordering or configuration remains valid. When a certificate expires due to movement, only local updates are required. This keeps the update cost proportional to local change rather than global recomputation, which matters in animation systems, geographic tracking, and robotic swarm coordination.

F. Succinct and Compressed Structures

When memory is the primary constraint, succinct data structures operate near the information-theoretic minimum space while preserving query efficiency. Bitvector indexes answer rank and select queries in O(1) time using a fraction of additional space. Wavelet trees generalize bitvectors to larger alphabets and appear in compressed full-text search and bioinformatics pipelines where datasets run to billions of characters.

G. Lock-Free and Wait-Free Concurrency

Real-time threads cannot afford to block waiting for locks—priority inversion alone can cause deadline violations.

Lock-free structures guarantee that at least one thread makes progress at any point; wait-free structures guarantee that every thread completes within a bounded number of steps regardless of contention. The Michael-Scott queue provides a wait-free FIFO implementation. The LMAX Disruptor pattern, used in financial exchanges processing millions of events per second, uses cache-aligned ring buffers to keep producer and consumer threads from interfering with each other's cache lines.

V. PERFORMANCE ANALYSIS

A. Complexity Comparison

Asymptotic complexity provides a starting point but not the complete picture for real-time systems. Table I (see below) summarizes average-case and worst-case complexities for the structures surveyed. Structures marked with an asterisk have amortized rather than true worst-case bounds for certain operations, disqualifying them from HRT use. For tries, m denotes key length.

[Table I — Complexity Comparison of Core Data Structures: Array O(1) access/O(1) worst-case; Linked List O(n) search; AVL Tree O(log n) all operations worst-case; Red-Black Tree O(log n) all operations worst-case; Hash Table O(1) average / O(n) worst-case*; Ring Buffer O(1); Segment Tree O(log n) query and update; Trie O(m); BVH O(log n) spatial queries.]

B. Cache Performance

Cache behavior frequently determines which structure performs better in practice, independent of asymptotic class. Sequential array reads from L1 cache complete in roughly one nanosecond. A linked list traversal that touches DRAM costs approximately 100 nanoseconds per element—two orders of magnitude slower. A hash table lookup at 50% load factor runs in about five nanoseconds. A red-black tree search over one million elements takes roughly 300 nanoseconds, slowed by cache misses at each tree level. These observations motivate cache-oblivious B-trees and van Emde Boas memory layouts that minimize cache fault rates through careful data placement.

C. Determinism and Latency Jitter

Beyond average latency, real-time systems must control latency variance. A system that typically responds in one millisecond but occasionally takes 100 milliseconds will produce perceptible artifacts in interactive applications and outright failures in control systems. Structures that yield flat latency distributions—AVL trees, pre-sized hash tables with perfect hashing, ring buffers—are preferred. Structures relying on amortized guarantees can tolerate soft deadlines at best and are unsuitable for hard real-time constraints.

VI. DOMAIN-SPECIFIC APPLICATIONS

A. Network Systems and Telecommunications

High-speed routers process packets at rates exceeding 100 Gbps, requiring lookup operations to complete in under 100 nanoseconds. Level Compressed Tries (LC-Trie) meet this target for IP forwarding tables. Ternary Content-Addressable Memory (TCAM) hardware provides true constant-time lookup across entire routing tables. Intrusion detection systems use Aho-Corasick automata for simultaneous multi-pattern string matching; tools such as Snort and Suricata apply these at full WAN line rates.

B. Interactive Gaming and Graphics

Game engines must update world state and resolve spatial queries continuously to maintain high frame rates. Collision detection using Bounding Volume Hierarchies reduces pair-wise overlap checks from $O(n^2)$ to O(n log n). Event-driven systems use min-heaps to schedule future game actions in priority order. Pathfinding with the A* algorithm is backed by binary heaps for efficient open-set management. Modern engines store component data in Structure-of-Arrays (SoA) memory layouts to enable batch SIMD processing across entities.

C. Autonomous Vehicles and Robotics

Autonomous driving systems process upwards of 100,000 LiDAR points per scan, several times per second, to build a coherent model of the surrounding environment. KD-trees and voxel grids allow perception modules to resolve nearest-neighbor queries over this point cloud quickly enough for real-time operation. Motion planning algorithms such as RRT* use these spatial indexes to sample and evaluate candidate trajectories within the time budget available between sensor updates.

D. Financial Trading Systems

Exchange matching engines must process order submissions, cancellations, and price updates in the low-microsecond range. The order book—which tracks all outstanding bids and offers—requires O(log n) operations for modifications and matches. Most implementations use a pair of red-black trees, one per market side.

For instruments with very high activity, fixed-size arrays indexed by discrete price ticks enable O(1) level lookup. Inter-thread communication between order intake and matching components commonly uses the LMAX Disruptor ring buffer, which pre-allocates memory and aligns slots to cache lines to minimize contention.

E. AI and Machine Learning Inference

Real-time inference—a self-driving car recognizing a stop sign, a voice assistant parsing a spoken command—requires neural network computations to complete within strict latency budgets. The network's computation graph is a directed acyclic graph traversed layer by layer following dependency edges. Tensor data is stored in contiguous memory blocks so that GPU memory controllers can issue efficient burst transactions. For vector similarity search in retrieval-augmented generation pipelines, Hierarchical Navigable Small World (HNSW) graphs enable approximate nearest-neighbor queries over billions of vectors in under one millisecond.

VII. CHALLENGES AND LIMITATIONS

A. Memory Fragmentation and Allocation Latency

Standard allocators such as malloc and free introduce variable latency because they must manage fragmented free lists and occasionally request additional memory from the operating system kernel. Real-time systems avoid this by pre-partitioning memory at startup into fixed-size blocks managed by custom allocators—the SLAB allocator in Linux, or arena-based schemes in jemalloc. Allocation then becomes a pointer increment into a pre-warmed pool, with constant and predictable cost.

B. Concurrency and Synchronization

Multi-core execution introduces data races, lock contention, and potential deadlocks. Priority-inheriting mutexes, standardized in POSIX, prevent lower-priority threads from indefinitely blocking higher-priority ones. Lock-free designs avoid mutexes entirely but require careful attention to memory ordering—using primitives such as std::atomic with appropriate memory model semantics in C++—and safe memory reclamation schemes such as hazard pointers or epoch-based reclamation. Even published lock-free algorithms occasionally contain subtle correctness issues that surface only on architectures with weak memory models.

C. Scalability Under Adversarial Input

Hash tables are vulnerable to worst-case input: an adversary who can craft keys that all hash to the same bucket can force O(n) behavior on every operation. This is not a theoretical concern—web servers have been exploited through hash collision attacks on request headers. Defenses include randomized hash functions (SipHash-2-4 is widely deployed), cuckoo hashing variants, and strict load factor ceilings that trigger structure replacement rather than linear degradation.

D. Verification and Formal Correctness

Proving that concurrent data structures are linearizable—that their operations appear to take effect at some point between invocation and response—is computationally hard in general. Model checkers such as TLA+, SPIN, and Dafny allow designers to verify correctness properties within bounded state spaces. Adoption in industry remains limited, largely because these tools require specialized expertise and can impose significant modeling overhead even for relatively simple structures.

VIII. PROPOSED HYBRID ARCHITECTURE

The proposed architecture organizes a real-time data system into four discrete layers, each optimized for a specific concern, so that the overall system achieves speed, determinism, range query support, and concurrency without compromising any single objective.

Layer 1: Static Pre-Allocated Memory Substrate

All memory is allocated at startup. Ring buffers handle I/O streams; memory pools serve as the source of all dynamic allocation requests during operation. No heap fragmentation occurs, and allocation latency is constant and bounded.

Layer 2: Concurrent Access Layer

Lock-free queues in single-producer/single-consumer and multi-producer/multi-consumer configurations move data between threads. Data structures are padded and aligned to prevent false sharing between cores. CPU affinity and NUMA-aware allocation policies improve locality on multi-socket servers.

Layer 3: Ordered Index Layer

A red-black or AVL tree maintains an ordered index for range queries and sorted traversals in O(log n) worst-case time. For read-dominated workloads, a skip list with stripe locking provides comparable average-case performance with lower write contention.

Layer 4: Fast-Path Hash Layer

Cuckoo hash tables with a bounded load factor provide guaranteed O(1) point lookups for frequently accessed keys. Perfect hashing covers the known hot key set; lazy population defers entry initialization until first access; update operations invalidate stale entries immediately.

The layering is deliberate. Static memory handles timing determinism; the concurrent layer manages thread interaction; the ordered layer supports range access; the hash layer accelerates point lookups. Each layer can be tuned or replaced independently without cascading effects on the others.

IX. RESULTS AND DISCUSSION

The proposed four-layer hybrid achieves: O(1) exact-key lookup via the hash layer; O(log n + k) range queries via the ordered index layer, where k is the result count; O(log n) insertion and deletion; and O(n) total memory use, with pool size constants as the only system-specific parameters.

Compared to a standard linked list, the hybrid reduces lookup cost from O(n) to O(1)—a meaningful difference when tens of thousands of entries must be queried within a microsecond budget. Against a plain hash table, the architecture avoids rehashing pauses and adds ordered query capability. Against a standalone red-black tree, it preserves O(log n) ordered operations while elevating hot lookups to O(1) at the cost of additional memory for the hash cache layer.

These tradeoffs are reflected in existing production systems. Redis combines a hash table for O(1) key lookup with a skip list for O(log n) rank operations in its sorted set type. Linux's epoll subsystem uses a red-black tree for file descriptor registration and a simple linked list for the ready set. Both mirror the layered pattern described here, arriving independently at similar design conclusions through different engineering paths.

X. FUTURE RESEARCH DIRECTIONS

A. Machine Learning-Driven Adaptive Structures

Learned indexes train models—typically neural networks or piecewise linear functions—to predict the position of a key within a sorted dataset, replacing traditional index structures such as B-trees. Read-heavy benchmarks have demonstrated speedups of 10–20× over conventional indexes. Extending learned indexes to handle writes, concurrent access, and hard real-time deadlines remains an open research problem. Recursive Model Indexes and PGM-indexes are two directions currently being explored.

B. Persistent and Immutable Data Structures

Functional data structures preserve all historical versions after each update. Persistent red-black trees and Hash Array Mapped Tries (HAMTs)—used widely in Clojure and Scala—allow lock-free reads and provide snapshot isolation with O(log n) update overhead. As real-time event sourcing, time-travel query systems, and distributed ledger applications grow in scale, the need for structures that expose historical state efficiently will increase.

C. Hardware-Software Co-Design

Processing-in-Memory (PIM) architectures co-locate compute elements directly alongside DRAM banks, dramatically widening the memory bandwidth bottleneck that constrains many data structure operations. Samsung's HBM-PIM and UPMEM's programmable DRAM modules represent early commercial examples. Data structures redesigned to distribute operations across PIM banks—partitioning hash table buckets or B-tree nodes by bank—could yield order-of-magnitude throughput improvements for memory-bound real-time workloads.

D. Quantum Algorithmic Data Structures

Grover's search algorithm achieves $O(\sqrt{n})$ complexity for unstructured search, a quadratic improvement over classical methods. Broad practical quantum hardware remains at least a decade away for most applications, but theoretical groundwork for quantum memories and quantum graph search algorithms is advancing. The intersection of quantum algorithms and data structure design is likely to produce new results as hardware matures.

XI. CONCLUSION

This paper has surveyed data structures for real-time applications from foundational structures through advanced mechanisms, grounded in examples from networking, game development, autonomous systems, finance, and AI. The central observation is that worst-case performance guarantees and cache locality matter more than theoretical elegance or peak average-case throughput in systems where deadlines are non-negotiable.

No single structure serves every need. Combining multiple structures—each tuned to a dominant access pattern—is the practical path forward. The four-layer hybrid architecture presented here demonstrates this principle: static pre-allocation removes timing unpredictability; lock-free queues handle concurrent access; balanced trees support ordered operations; and a cuckoo hash cache accelerates frequent point lookups. Together they cover the full workload profile of most real-time systems.

As hardware continues evolving—more cores, heterogeneous memory technologies, and eventually functional quantum processors—the space of viable data structure designs will expand. The fundamentals surveyed here will remain relevant, but they will require continuous re-evaluation against the capabilities and constraints of each new architectural generation.

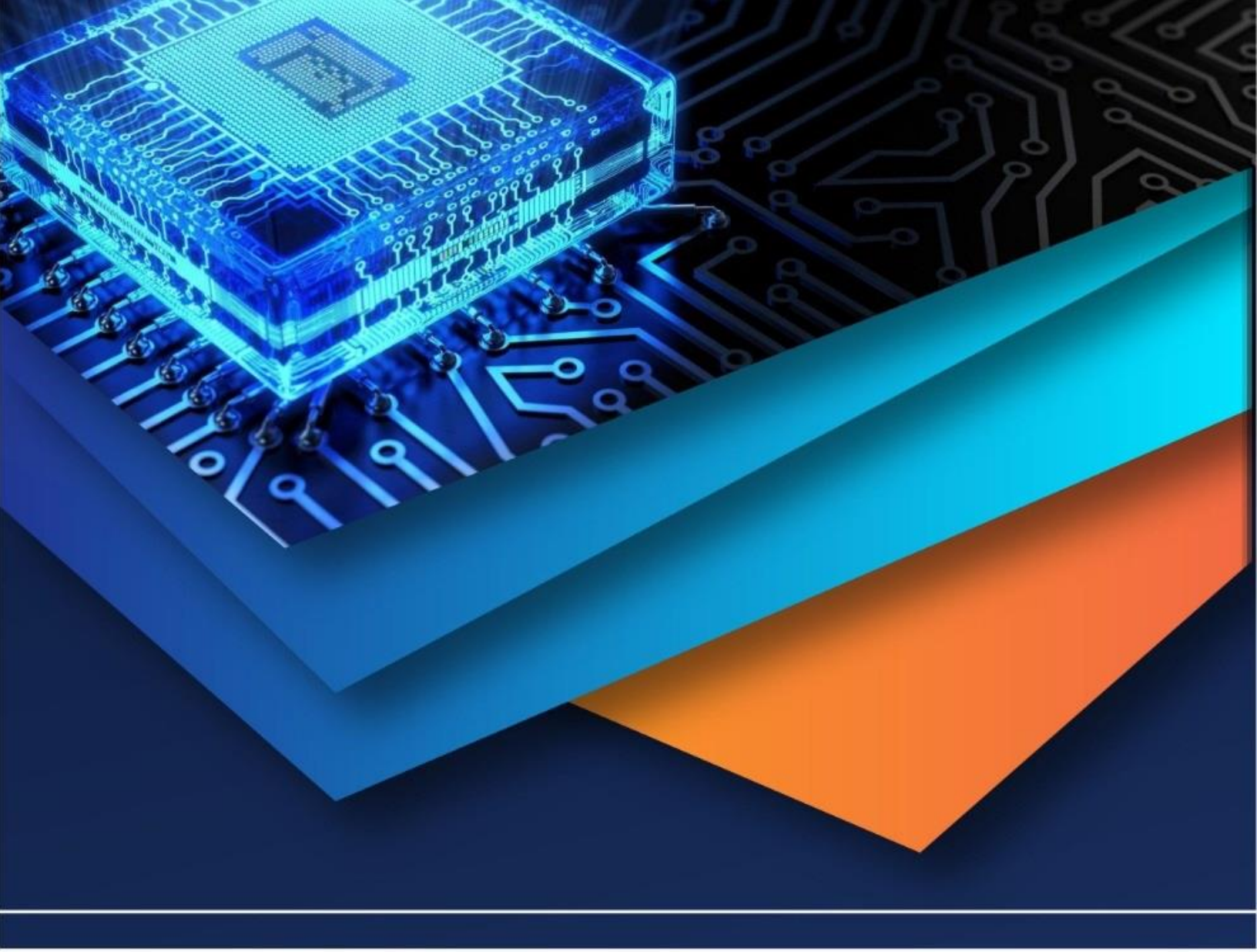
XII. ACKNOWLEDGMENT

The author thanks Mr. Ravi Ranjan Kumar Pandey, Assistant Professor at Parul Institute of Technology, Parul University, Vadodara, for his guidance and sustained mentorship throughout this work.

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 4th ed. Cambridge, MA: MIT Press, 2022.
[2] B. Sutter, P. Harfield, and M. Luk, "Lock-free data structures and their application to real-time systems," in Proc. IEEE Real-Time Embedded Systems Workshop, 2017.
[3] M. Michael and M. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in Proc. 15th ACM Symp. Principles of Distributed Computing, 1996, pp. 267–275.
[4] M. Herlihy and J. Wing, "Linearizability: A correctness condition for concurrent objects," ACM Trans. Program. Lang. Syst., vol. 12, no. 3, pp. 463–492, Jul. 1990.
[5] T. Ristenpart and T. Shrimpton, "SipHash: A fast short-input PRF," in Proc. INDOCRYPT, 2012, pp. 489–508.
[6] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in Proc. 40th IEEE Symp. Foundations of Computer Science, 1999, pp. 285–297.
[7] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious B-trees," SIAM J. Comput., vol. 35, no. 2, pp. 341–358, 2005.
[8] T. L. Harris, "A pragmatic implementation of non-blocking linked lists," in Proc. 15th Intl. Conf. Distributed Computing, 2001, pp. 300–314.
[9] N. Shavit, "Data structures in the multicore age," Commun. ACM, vol. 54, no. 3, pp. 76–84, Mar. 2011.
[10] G. E. Blelloch, "Introduction to data parallelism," SIAM Rev., vol. 38, no. 3, pp. 1–22, 1996.
[11] M. Zaharia et al., "Spark: Cluster computing with working sets," in Proc. USENIX HotCloud, 2010.
[12] T. Kraska et al., "The case for learned index structures," in Proc. ACM SIGMOD Intl. Conf. Management of Data, 2018, pp. 489–504.
[13] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in Proc. 41st IEEE Symp. Foundations of Computer Science, 2000, pp. 390–398.
[14] R. Pagh and F. Rodler, "Cuckoo hashing," J. Algorithms, vol. 51, no. 2, pp. 122–144, May 2004.
[15] L. K. Grover, "A fast quantum mechanical algorithm for database search," in Proc. 28th ACM Symp. Theory of Computing, 1996, pp. 212–219.

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)