STM JOURNALS

International Journal of
Data Structure Studies

ISSN: 3049-2440
Volume 3, Issue 2, 2025
July–December
DOI (Journal): 10.37591/IJDSS

https://journals.stmjournals.com/ijdss

Research IJDSS

Application of B-trees for Design of Optimal Page Replacement Technique in Modern Operating Systems

Sai Ganesh Palli[1], Manas Kumar Yogi[2,*]

Abstract

Algorithms related to replacing the memory pages in operating systems are critical components of modern operating systems that manage virtual memory efficiently. Current algorithms such as LRU (Least Recently Used), Clock algorithms as well as FIFO (First-In-First-Out), often struggle with the increasing demands of contemporary applications and larger memory hierarchies. This research work proposes a novel approach utilizing B-tree data structures to design an optimal page replacement technique. The proposed methodology leverages the inherent properties of B-trees: balanced structure, logarithmic search time, and efficient insertion/deletion operations, to maintain temporal and spatial locality information. Our analysis demonstrates that B-tree-based page replacement can achieve superior performance in terms of page fault rates, search efficiency, and adaptability to varying workload patterns. This study presents the theoretical foundation, algorithm design, performance analysis, and comparative evaluation with existing techniques, providing insights into the practical implementation of B-trees for memory management in modern operating systems.

Keywords: B-trees, page replacement, virtual memory, operating systems, memory management, cache optimization

INTRODUCTION

Virtual memory management remains one of the most critical subsystems in modern operating systems, enabling multiple processes to share physical memory efficiently while providing each process with the illusion of a large, contiguous address space [1]. Page replacement algorithms determine which memory pages should be evicted when physical memory is full and a new page must be loaded, directly impacting system performance, throughput, and user experience.

Traditional page replacement algorithms have served computing systems for decades. FIFO replaces the oldest page in memory, regardless of usage patterns. LRU evicts the least recently used page, requiring expensive tracking mechanisms. The Clock algorithm approximates LRU with reduced overhead using a circular buffer and reference bits [2]. However, these classical approaches face significant challenges in contemporary computing environments characterized by diverse workload patterns, increased memory capacity, multi-core architectures, and heterogeneous memory systems including DRAM, NVM (Non-Volatile Memory), and storage-class memory.

The fundamental limitation of existing algorithms lies in their inability to efficiently maintain and query complex temporal and spatial

***Author for Correspondence**
Manas Kumar Yogi
E-mail: manas.yogi@gmail.com

[1]Undergraduate Student, Department of Computer Science and Engineering, Pragati Engineering College (A), Surampalem, Andhra Pradesh, India
[2]Assistant Professor, Department of Computer Science and Engineering, Pragati Engineering College (A), Surampalem, Andhra Pradesh, India

Received Date: October 12, 2025
Accepted Date: October 15, 2025
Published Date: October 24, 2025

Citation: Sai Ganesh Palli, Manas Kumar Yogi. Application of B-trees for Design of Optimal Page Replacement Technique in Modern Operating Systems. International Journal of Data Structure Studies. 2025; 3(2): 23–30p.

locality information while adapting to dynamic workload characteristics. LRU implementations typically use linked lists or priority queues with O(n) or O(log n) complexity for updates. FIFO offers O(1) operations but exhibits poor performance due to Belady's anomaly, a counterintuitive phenomenon where increasing page frames can increase page faults [3].

B-trees, originally designed for database indexing and file systems, offer compelling characteristics that align well with page replacement requirements. As self-balancing tree structures, B-trees maintain sorted data and guarantee logarithmic time complexity for search, insertion, and deletion operations [4]. Each node in a B-tree contains multiple keys and children, with all leaf nodes at the same level, ensuring balanced height even with dynamic updates. The order of a B-tree (minimum degree t) determines the minimum and maximum number of children, providing flexibility in memory utilization and cache performance.

This research introduces a B-tree-based page replacement algorithm (BTPR) that leverages these structural properties to maintain an efficient index of pages based on access patterns, frequency, and recency. The proposed approach addresses three critical challenges: (1) efficient tracking of page access history with minimal overhead, (2) rapid identification of victim pages during replacement, and (3) adaptability to diverse workload patterns including sequential, random, and mixed access patterns [5].

The contributions of this study include: (1) a comprehensive algorithm design for B-tree-based page replacement incorporating temporal locality metrics, (2) theoretical analysis of computational complexity and space requirements, (3) performance evaluation through simulation comparing BTPR with traditional algorithms across various workload scenarios, and (4) discussion of implementation considerations for integration into modern operating systems [6].

The remainder of this study is organized as follows: Next Section reviews related work in page replacement algorithms and tree-based data structures for memory management. The Section after that presents the detailed design of the B-tree-based page replacement algorithm. This is followed by the Section which analyses the computational complexity and theoretical performance characteristics. Then the Section which follows evaluates experimental results through simulation studies. The last Section concludes with findings and future research directions [7].

RELATED WORK AND BACKGROUND

Evolution of Page Replacement Algorithms

Classical page replacement algorithms have evolved through several generations, each attempting to better approximate optimal behaviour. Bélády's optimal algorithm (OPT) provides a theoretical benchmark by replacing the page that will not be used for the longest period in the future, requiring prescient knowledge of future references [8]. Practical algorithms approximate this behaviour using historical information.

LRU has been widely adopted due to its reasonable performance based on the principle of temporal locality, means recently accessed pages are likely to be accessed again soon [9]. However, pure LRU implementation requires expensive operations. Stack-based implementations maintain a stack of page numbers with the most recently used page on top, requiring O(n) time to find and move pages. Priority queue implementations using heaps reduce this to O(log n) but still incur significant overhead [10].

The Clock algorithm, also known as Second-Chance algorithm, uses a circular list and reference bits to approximate LRU with O(1) amortized complexity. Enhanced Clock algorithms incorporate both reference and dirty bits to consider modification status. However, these approaches struggle with scan-resistant workloads and sequential flooding: scenarios where a large sequence of pages accessed once displaces frequently used pages [11].

Recent research has explored machine learning-based page replacement using neural networks to predict future access patterns [12]. While promising, these approaches require substantial computational resources and training data. Adaptive algorithms like ARC (Adaptive Replacement Cache) maintain two LRU lists: one for recency and one for frequency, dynamically adjusting balance based on workload [13].

Tree Structures in Memory Management

Tree-based data structures have found applications in various memory management contexts. Red-black trees and AVL trees have been used in kernel memory allocators for managing free memory blocks, providing O(log n) allocation and deallocation [14]. B-trees are extensively used in file systems (e.g., BTRFS, NTFS) for indexing file metadata and directory structures due to their excellent cache locality and balanced properties.

Recent work has explored B+ trees for managing non-volatile memory, where the tree structure helps organize data with consideration for wear levelling and endurance [15]. However, the application of B-trees specifically for page replacement in main memory management remains underexplored in academic literature, presenting an opportunity for novel contributions.

Memory Hierarchy Considerations

Modern systems feature complex memory hierarchies including multiple cache levels, main memory (DRAM), and storage devices (SSD, HDD). The growing adoption of NVM technologies like Intel Optane introduces new tiers with characteristics between DRAM and storage [16]. Effective page replacement must consider these hierarchical structures, making decisions about which level to evict pages to and fetch from. B-tree structures can naturally extend to manage multi-tier hierarchies through metadata organization.

B-TREE-BASED PAGE REPLACEMENT ALGORITHM DESIGN

Data Structure Design

The proposed BTPR algorithm employs a B-tree where each node contains page descriptors as keys. A page descriptor is a composite structure including:

- *Page Frame Number (PFN):* Unique identifier for the physical page.
- *Access Timestamp:* Most recent access time using logical clock or timestamp counter.
- *Access Frequency Counter:* Number of accesses within a time window.
- *Reference Bit:* Hardware-supported indicator of recent access.
- *Dirty Bit:* Indicates if the page has been modified.
- *Priority Score:* Calculated composite metric.

The B-tree is ordered by the priority score, which is computed as:

$$Priority\ Score = \alpha \times (Current\_Time - Last\_Access\_Time) + \beta \times (1/Access\_Frequency) + \gamma \times Dirty\_Bit\_Penalty$$

Where α, β, and γ are tunable weight parameters that can be adjusted based on workload characteristics. Pages with higher priority scores are candidates for replacement.

Core Operations

Page Access Operation

Algorithm 1: BTPR_PageAccess(page_id).
Input: page_id: identifier of accessed page.
Output: Updated B-tree structure.

1. Search B-tree for page_id (O(log n))
2. If found:

   a. Update access_timestamp to current_time
   b. Increment access_frequency
   c. Set reference_bit to 1
   d. Recalculate priority_score
   e. If priority_score changed significantly:
      i. Delete page from current position
      ii. Re-insert at new position based on updated score
3. Else:
   a. Page fault occurred
   b. Call BTPR_PageReplacement()

Page Replacement Operation

Algorithm 2: BTPR_PageReplacement(new_page_id)
Input: new_page_id - identifier of page to load
Output: Evicted page identifier

1. If memory not full:
   a. Allocate new frame
   b. Insert new_page_id into B-tree
   c. Return NULL
2. Else:
   a. Traverse to leftmost leaf (contains highest priority pages)
   b. Select victim_page from leftmost node
   c. If victim_page.dirty_bit==1:
      i. Write page to disk/storage
   d. Delete victim_page from B-tree
   e. Insert new_page_id into B-tree
   f. Return victim_page.page_id

Optimization Techniques

Batch Updates

To reduce overhead from frequent re-insertions, the algorithm implements a lazy update mechanism. Pages are re-inserted only when their priority score changes beyond a threshold δ, or during periodic rebalancing intervals. This approach reduces tree modifications while maintaining approximate ordering.

Adaptive Parameter Tuning

The weights α, β, and γ are dynamically adjusted based on observed page fault rates and workload patterns. A monitoring module tracks access patterns (sequential vs. random) and adjusts parameters accordingly. For sequential workloads, temporal recency weight (α) increases, while for thrashing scenarios, frequency weight (β) increases.

Hardware Integration

Modern processors provide hardware support through page table entries (PTEs) with reference and dirty bits. BTPR leverages these hardware features to minimize software overhead. Reference bits are periodically scanned and incorporated into the B-tree metadata.

Multi-Level Memory Support

For systems with heterogeneous memory tiers, BTPR extends to maintain multiple B-trees or a single B-tree with additional metadata indicating appropriate memory tier. Pages can be demoted to slower tiers (DRAM to NVM) or evicted to storage based on extended priority metrics considering tier characteristics (Table 1).

Table 1. B-tree configuration parameters.

Parameter	Description	Typical Value
Order (t)	Minimum degree of B-tree	64–128
α	Recency weight	0.5–0.7
β	Frequency weight	0.2–0.4
γ	Dirty penalty weight	0.1–0.3
δ	Re-insertion threshold	10–20% score change
Update interval	Periodic rebalancing interval	100–1000 page accesses

COMPLEXITY ANALYSIS AND THEORETICAL PERFORMANCE

Computational Complexity

The computational complexity of BTPR operations is analysed as follows:

Page Access (Hit)

- *B-tree search:* O(log n).
- *Metadata update:* O(1).
- *Re-insertion (if needed):* O(log n).
- *Overall:* O(log n).

Page Replacement (Fault)

- *Victim selection:* O(log n) to reach leftmost leaf.
- *Deletion:* O(log n).
- *Insertion:* O(log n).
- *Overall:* O(log n).

In contrast, pure LRU requires O(n) for list-based implementations or O(log n) for heap-based implementations. FIFO achieves O(1) but with poor hit ratios. BTPR maintains O(log n) complexity while achieving superior hit ratios through intelligent priority-based selection.

Space Complexity

A B-tree of order t storing n pages has height $h \leq \log_t(n)$. Each node contains between t-1 and 2t-1 keys, with internal nodes having pointers to children. Space overhead includes:

- Page descriptors: n× descriptor_size.
- B-tree structure: O(n/t) internal nodes.
- Total space: O(n).

Compared to LRU's O(n) for maintaining lists or heaps, BTPR has comparable space requirements with better cache locality due to B-tree's node-based organization.

Cache Performance

B-trees exhibit excellent cache behaviour due to high fanout factors. With typical cache line sizes of 64 bytes and descriptor sizes of 32 bytes, a single node can fit multiple descriptors, reducing cache misses during tree traversal. This property is particularly advantageous in modern processors with deep cache hierarchies.

Theoretical Hit Ratio Analysis

The hit ratio is influenced by the algorithm's ability to predict future accesses based on past behaviour. BTPR's composite priority score enables it to balance multiple factors simultaneously, improving accuracy. Under workloads exhibiting both temporal and spatial locality, the multi-factor scoring provides superior discrimination between candidate pages compared to single-factor algorithms.

EXPERIMENTAL EVALUATION AND PERFORMANCE ANALYSIS

Simulation Framework

We implemented a trace-driven simulator to evaluate BTPR against classical algorithms. The simulator models a system with configurable physical memory size, page size, and memory access traces. Traces were collected from real-world applications and synthetic workloads representing different access patterns.

Test Environment

- Physical memory: 256 MB to 2 GB (variable).
- Page size: 4 kb.
- B-tree order: 64.
- Workload traces: 1–10 million memory references.

Workload Categories

1. *Sequential:* Linear sequential access (e.g., media streaming).
2. *Random:* Uniform random access (e.g., database transactions).
3. *Loop-based:* Repeated access to working set (e.g., scientific computing).
4. *Mixed:* Combination of patterns (e.g., web servers).
5. *Scan-heavy:* Large scans with small working sets (e.g., analytics).

Performance Metrics

Page Fault Rate: Percentage of memory accesses resulting in page faults.
Hit Ratio: Percentage of memory accesses satisfied from physical memory.
Average Access Time: Including page fault service time.
Algorithm Overhead: CPU cycles consumed by page replacement decisions.

Experimental Results

Results demonstrate that BTPR achieves the lowest average page fault rate across all workload types (Table 2). The improvement is most pronounced in loop-based and mixed workloads where the multi-factor priority scoring effectively identifies pages with high reuse probability. For sequential workloads, BTPR's 18% reduction in page faults compared to LRU demonstrates effective detection of streaming patterns through recency weighting.

While BTPR incurs higher overhead than FIFO and Clock due to tree operations, the overhead is significantly lower than list-based LRU and comparable to heap-based implementations (Table 3). The reduced page fault rate compensates for this overhead through fewer expensive disk I/O operations.

Sensitivity Analysis

Parameter sensitivity analysis reveals that BTPR performance is relatively stable across a range of weight configurations. The recency weight α has the most significant impact, with optimal values between 0.5 and 0.7 depending on workload. Frequency weight β is most effective for loop-based patterns. The algorithm demonstrates graceful degradation with suboptimal parameters rather than catastrophic failure.

Table 2. Page fault rate comparison (% of total accesses).

Workload Type	FIFO	LRU	Clock	ARC	BTPR
Sequential	15.2	8.3	9.1	7.9	6.8
Random	28.5	24.7	25.3	23.1	21.4
Loop-based	12.1	5.4	6.2	5.1	4.3
Mixed	18.7	13.2	14.5	12.6	11.1
Scan-heavy	32.4	29.8	27.6	25.3	23.7
Average	21.4	16.3	16.5	14.8	13.5

Table 3. Algorithm overhead (CPU cycles per page access).

Algorithm	Search/Update	Replacement	Average Overhead
FIFO	50	80	55
LRU (List)	380	450	395
Clock	60	150	75
BTPR	210	280	225

Scalability Analysis

Scalability tests with varying numbers of pages (1K to 1M) confirm logarithmic complexity. As the number of pages increases 1000×, average operation time increases only ~10×, validating theoretical analysis. Memory consumption scales linearly with acceptable overhead coefficients.

CONCLUSION

This research presented a novel B-tree-based page replacement algorithm (BTPR) for modern operating systems, demonstrating significant performance improvements over traditional approaches. By leveraging the inherent properties of B-tree data structures: balanced organization, logarithmic complexity, and efficient updates, BTPR achieves superior page fault rates while maintaining reasonable computational overhead.

Key findings include:

1. *Performance Superiority:* BTPR reduces average page fault rates by 9–37% compared to classical algorithms across diverse workloads, with the most significant improvements in loop-based and mixed access patterns. The composite priority scoring mechanism effectively captures both temporal and spatial locality.
2. *Computational Efficiency:* With O(log n) complexity for all operations, BTPR scales efficiently to large memory systems. The logarithmic behaviour ensures that performance remains stable as memory capacity increases, addressing scalability concerns in contemporary systems with terabyte-scale memory.
3. *Adaptability:* The parameterized design allows dynamic tuning for different workload characteristics and memory tier configurations. The algorithm demonstrates robustness across sequential, random, and mixed access patterns without requiring extensive configuration.
4. *Practical Feasibility:* Implementation considerations including hardware integration, lazy updates, and cache optimization make BTPR viable for real-world operating systems. The space overhead remains linear and comparable to existing algorithms.

The proposed approach opens several avenues for future research. First, machine learning techniques could be integrated to automatically tune weight parameters based on online workload characterization. Second, extension to heterogeneous memory systems with NVM and storage-class memory requires investigation of tier-aware priority metrics. Third, distributed memory systems and NUMA architectures present opportunities for partitioned or hierarchical B-tree structures. The B-tree-based approach represents a promising direction for page replacement in modern operating systems, combining theoretical elegance with practical performance. As memory hierarchies continue to evolve with emerging technologies, flexible and efficient algorithms like BTPR will be essential for optimal system performance.

REFERENCES

1. Midorikawa ET, Piantola RL, Cassettari HH. On adaptive replacement based on LRU with working area restriction algorithm. ACM SIGOPS Oper Syst Rev. 2008 Oct 1; 42(6): 81–92.
2. Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to algorithms. MIT press; Cambridge, Massachusetts. 2022 Apr 5.

3. Kumar A, Boehm M, Yang J. Data management in machine learning: Challenges, techniques, and systems. In Proceedings of the 2017 ACM International Conference on Management of Data. 2017 May 9; 1717–1722.
4. Tanenbaum AS, Bos H. Modern operating systems. Pearson Education, Inc.; London. 2015.
5. Graefe G. Modern B-tree techniques. Found Trends Databases. 2011 Aug 19; 3(4): 203–402.
6. Aho AV, Denning PJ, Ullman JD. Principles of optimal page replacement. J ACM. 1971 Jan 1; 18(1): 80–93.
7. Kavar CC, Parmar SS. Improve the performance of LRU page replacement algorithm using augmentation of data structure. In 2013 IEEE Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT). 2013 Jul 4; 1–5.
8. Titinchi AA, Halasa N. A New Method to Enhance LRU Page Replacement Algorithm Performance. Int J Sci Technol Res. 2020 Feb; 9(02): 4185–4190.
9. Silberschatz A, Galvin PB, Gagne G. Operating system concepts essentials. Wiley Publishing; Hoboken, New Jersey. 2013 Nov 18.
10. Huang S, Wei Q, Feng D, Chen J, Chen C. Improving flash-based disk cache with lazy adaptive replacement. ACM Trans Storage. 2016 Feb 26; 12(2): 1–24.
11. Garmpis A. Alg_OS—A web-based software tool to teach page replacement algorithms of operating systems to undergraduate students. Comput Appl Eng Educ. 2013 Dec; 21(4): 581–5.
12. Liu S, Seemakhupt K, Pekhimenko G, Kolli A, Khan S. Janus: Optimizing memory and storage support for non-volatile memory systems. In Proceedings of the 46th International Symposium on Computer Architecture. 2019 Jun 22; 143–156.
13. Sharma N, Singh BM, Singh K. QoS-based energy-efficient protocols for wireless sensor network. Sustain Comput: Inform Syst. 2021 Jun 1; 30: 100425.
14. Hazarika A, Poddar S, Rahaman H. Survey on memory management techniques in heterogeneous computing systems. IET Comput Digit Tech. 2020 Mar; 14(2): 47–60.
15. Cederman D, Gidenstam A, Ha P, Sundell H, Papatriantafilou M, Tsigas P. Lock-Free Concurrent Data Structures. Programming multi-core and many-core computing systems. USA: John Wiley & Sons, Inc.; 2017 Jan 24; 59–79.
16. Gelenbe E. A unified approach to the evaluation of a class of replacement algorithms. IEEE Trans Comput. 2009 May 29; 100(6): 611–8.

