**Page Replacement Algorithm Performance Analysis Report**

**Assignment 2:** Virtual Memory Simulator - Task 2 Report

**Group members:** Hai Trung Do – a1899443, Tri Dung Nguyen – a1899360, Viet Bach Tran – a1901793.

## 1. Introduction

Virtual memory enables programs to use more memory than is physically available by storing data in pages on secondary storage and loading them into physical memory on demand. When a program needs a new page while the physical memory is full, the operating system has to evict a page. Therefore, the page replacement algorithm selection is paramount for system performance and efficiency because it impacts the rate of page faults and expensive memory disk input/output operations. This report analyzes three page replacement algorithms: random replacement, least-recently used (LRU) and clock. We use four SPEC benchmark memory traces (swim, bzip, gcc, sixpack) to answer the following questions:

How much memory does each program actually need?

- Which algorithm performs best under memory constraints?
- Does one algorithm consistently outperform others across different workloads?

The investigation will be conducted in two phases to identify transition zones and evaluate algorithm efficiency under different workloads.

## 2. Methods

### Experimental Design

We employed a comprehensive two-phase design that systematically explores performance characteristics on all four traces. This method ensures that subsequent evaluation of algorithm performance on the key regions is thorough and comprehensive.

**Phase 1 - Broad Survey:** The logarithmic sequence of frame counts (5, 10, 20, 50, 100, 200, 500 and 1000) was utilized to ascertain the general behavior and delineate regions of quick changes in fault rates.

**Phase 2 – Detailed Analysis:** The linear frame testing was performed in specific proximity zones using 15 to 25-frame increments to capture specific behavioral differences between algorithms.

### Testing Parameters

Frame counts for each trace varied from 5 to 1000. Averaged over three independent runs of the random algorithm and single run for other deterministic algorithms (LRU and clock). We established the baseline (1000 frames) to use excessive memory and determine the minimum requirement and the other (use minimal memory) to determine the maximum number of page faults.

### Performance Metrics

**Primary metric:** page fault rate (total page faults / total memory accesses), which directly indicates algorithm effectiveness at minimizing memory system overhead.

**Secondary considerations:** disk I/O operations and working set boundary identification for practical system design implications.

### Replacement Algorithms

- **Random (RAND):** Upon page faults with full frame allocation, evicts a randomly selected frame, providing an unbiased baseline for comparison.
- **LRU (Least Recently Used):** Evicts pages with the longest period since last access, exploiting temporal locality principles for theoretical optimality
- **Clock (Enhanced Second Chance):** Utilizes reference bits in a circular buffer structure to approximate LRU behavior with reduced implementation complexity

### Trace Characteristics

Our analysis employed four SPEC benchmark traces representing distinct computational domains:

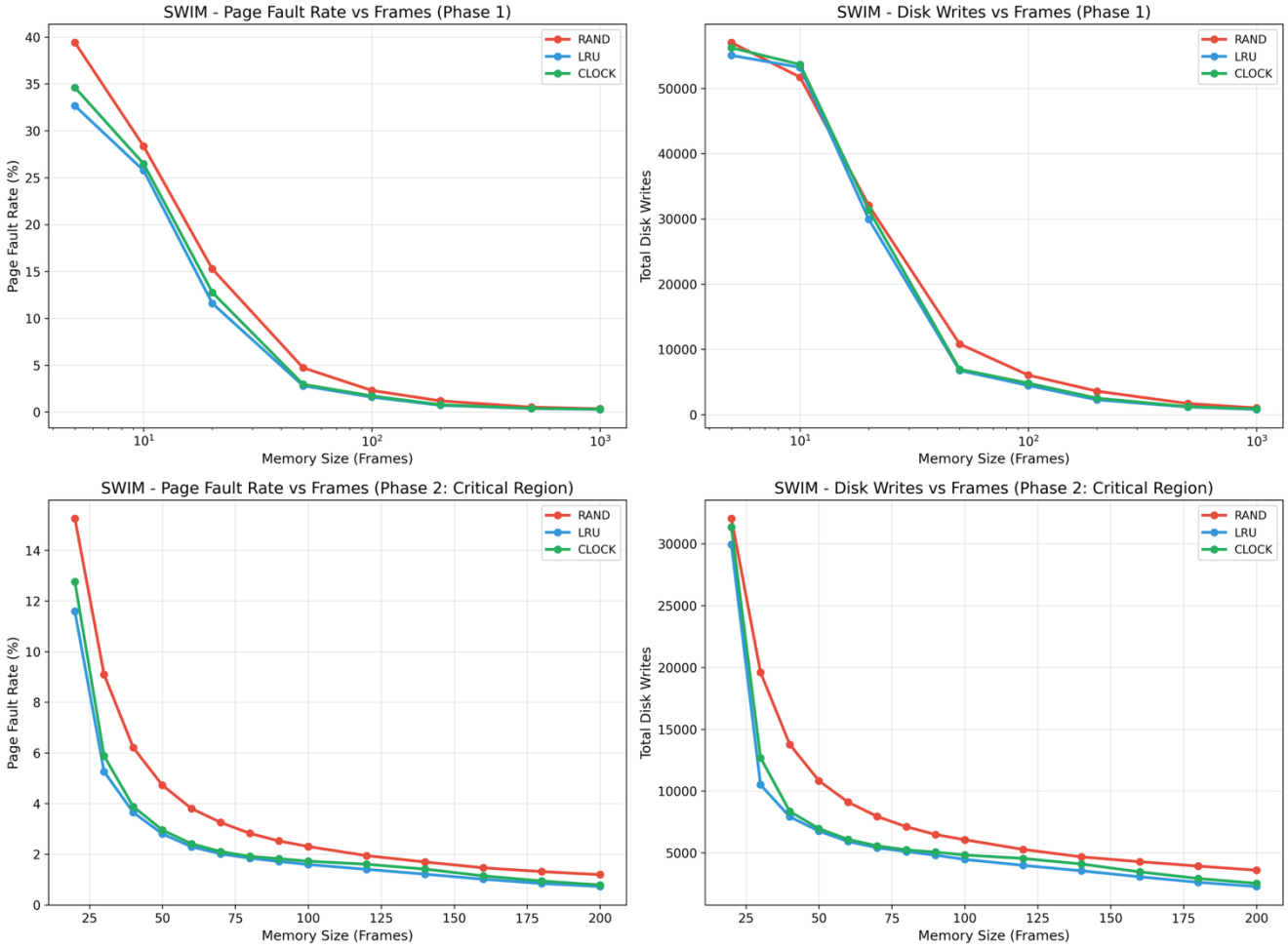
- **swim:** Computational fluid dynamics requiring substantial memory allocation with streaming characteristics

- **bzip:** Data compression utility with dictionary-based access patterns and strong temporal locality
- **gcc:** C compiler exhibiting complex symbolic processing with distributed, irregular access patterns
- **sixpack:** Scientific matrix operations featuring moderate temporal locality with well-defined working set boundaries

Each trace contains approximately one million memory references, ensuring consistent comparison across workloads.

## 3. Results

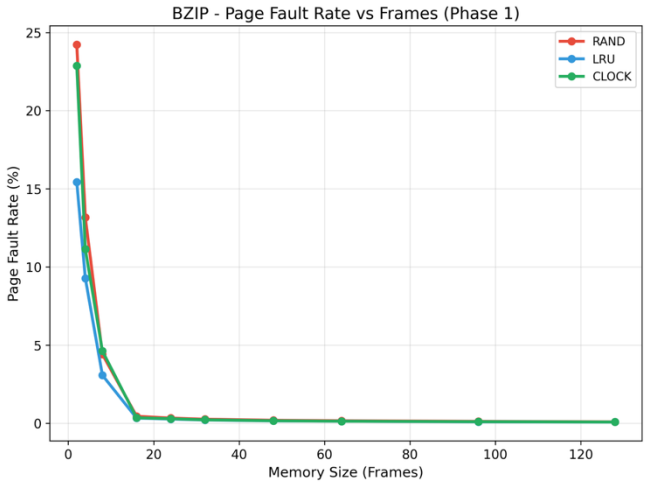
### 3.1 swim trace analysis

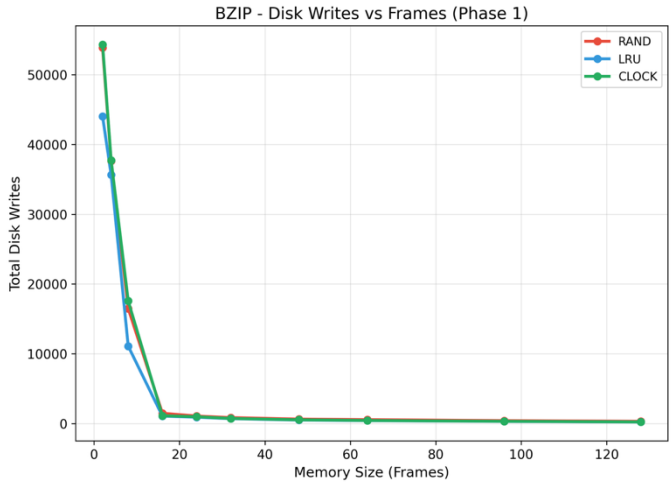


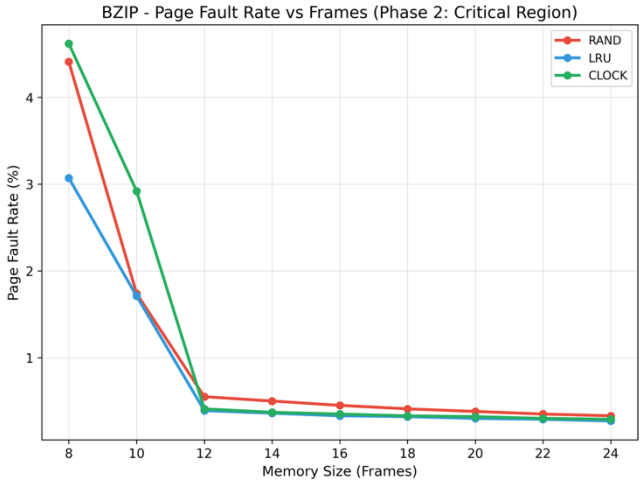

**Memory requirements:** swim trace needs substantial memory to execute efficiently. Note that LRU and Clock need about 200 frames to get less than 1% of page faults, while Random needs about 500 frames. Even at 1000 frames, the page fault rate never reaches zero (LRU remains at 0.28%), indicating streaming access patterns with ongoing compulsory misses.

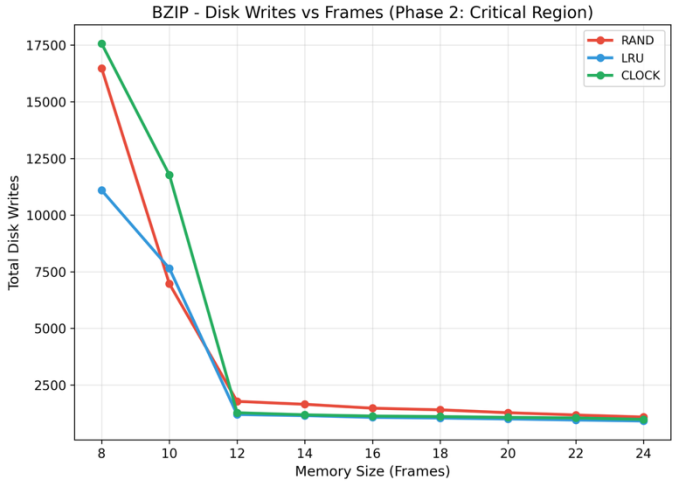
**Key findings:** LRU is consistently the best. Clock stays within 5%-10% of LRU performance for large numbers of frames (50 and up). The page fault rate decreases rapidly until the number of frames reaches around 50 in all four algorithms and then further gains become more gradual. Random replacement performs significantly worse at low memory allocation but converges with LRU and Clock as more frames are allocated.

### 3.2 bzip trace analysis



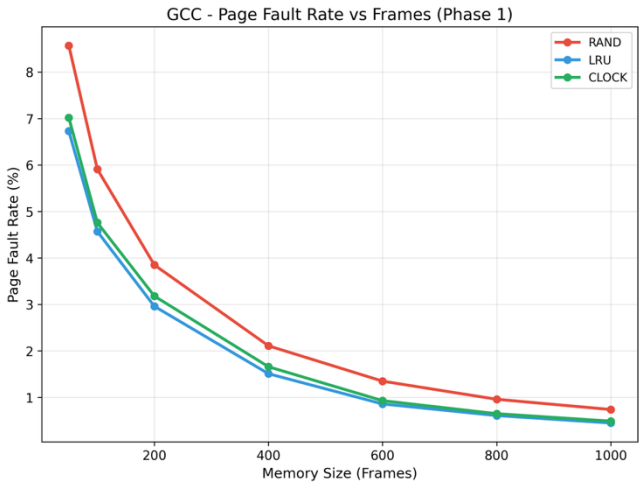



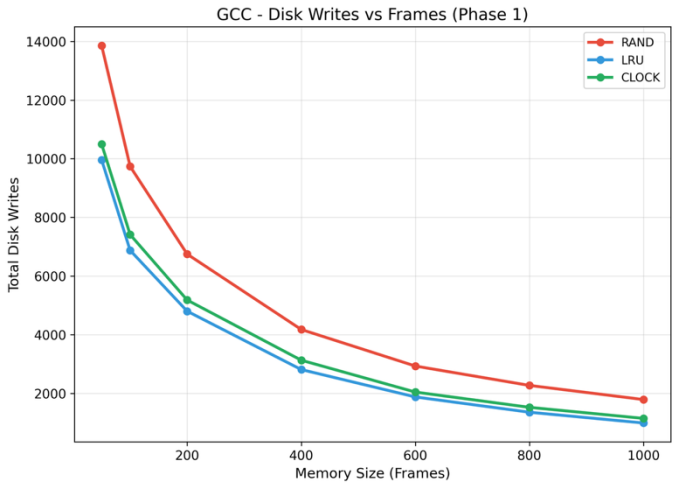



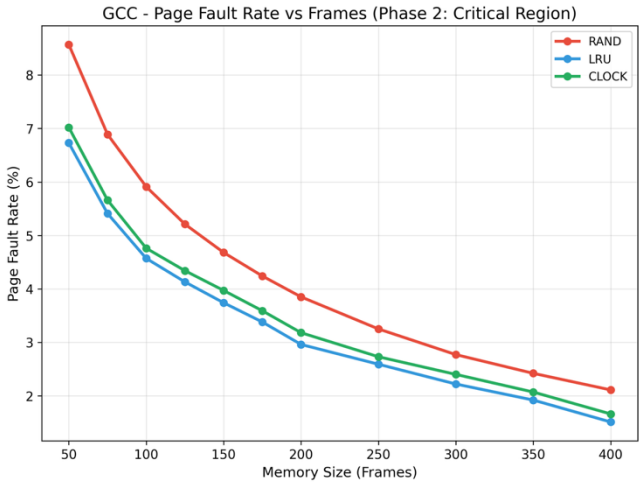

**Memory requirements:** The bzip memory trace has minimal memory requirements. All the algorithms have less than 1% page fault rate at only 12 frames. The simulator results show that only 317 distinct pages were accessed in 1,000,000 events and no page was evicted at the 500 frames since there were zero disk writes.

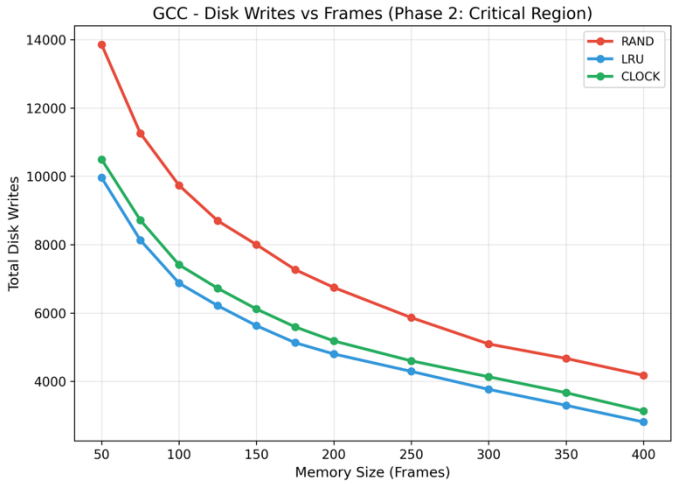
**Key findings:** bzip trace exhibits the classic 80-20 workload behavior with strong temporal locality. LRU and Clock are almost identical for memory sizes beyond 10 frames, while Random shows slightly worse results only under extreme memory pressure (below 10 frames).

### 3.3 gcc trace analysis



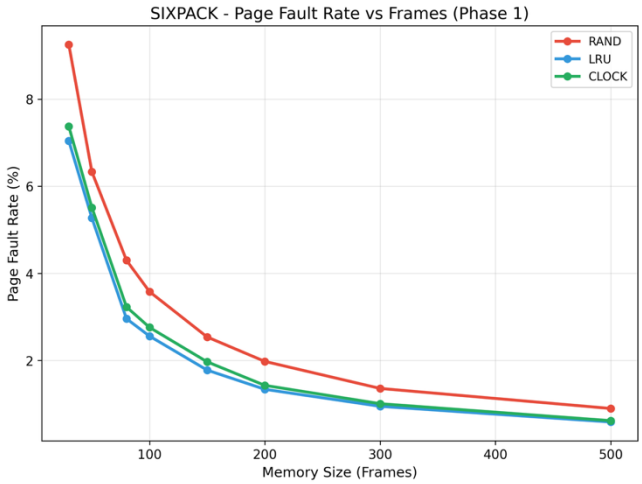


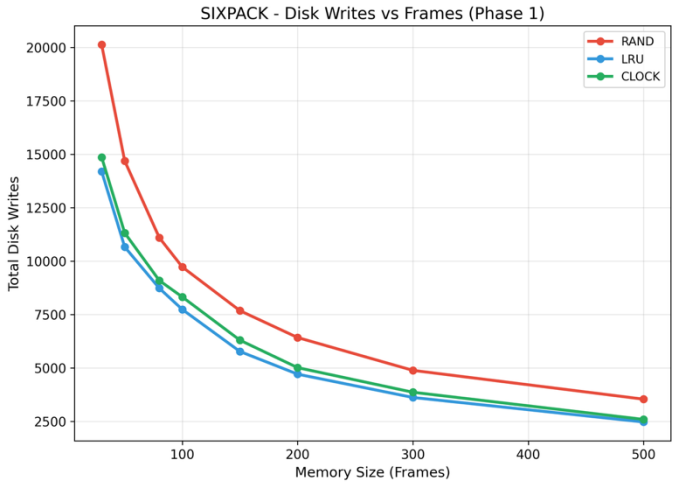



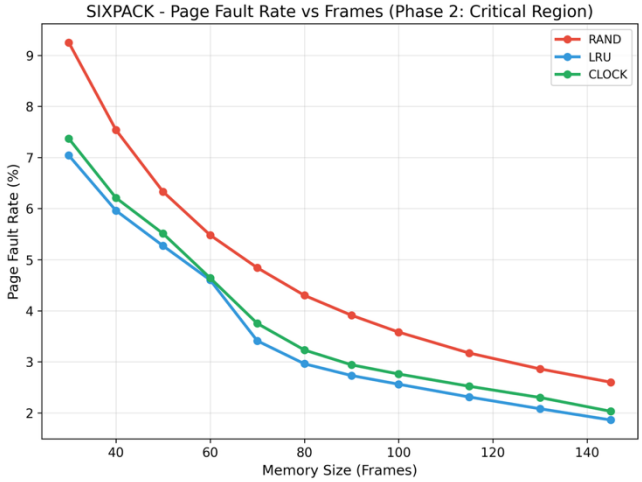

**Memory requirements:** The gcc traces workload demands a large amount of memory. The page-fault rate decreases slowly as the memory size increases from 50 to 1000 frames, rather than dropping off sharply. This suggests that the memory needs of the application are complex and working sets hard to define, at least compared to other traces.

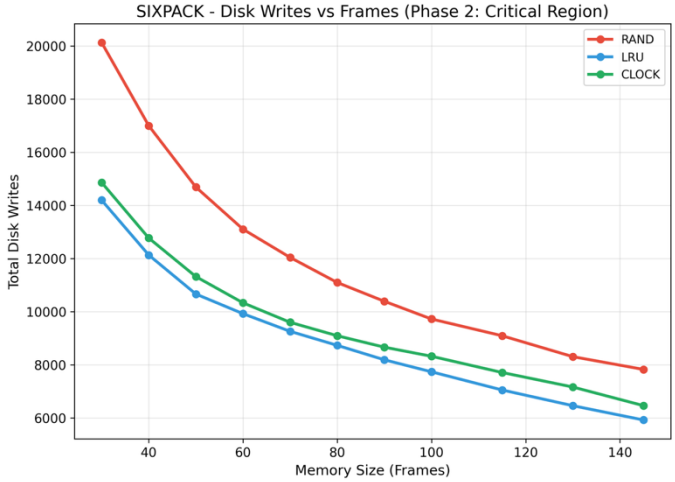
**Key findings:** LRU consistently outputs the best results, especially when the memory is low. Clock falls closely behind LRU and Random has the worst performance (when there are fewer than 200 frames).

### 3.4 sixpack trace analysis



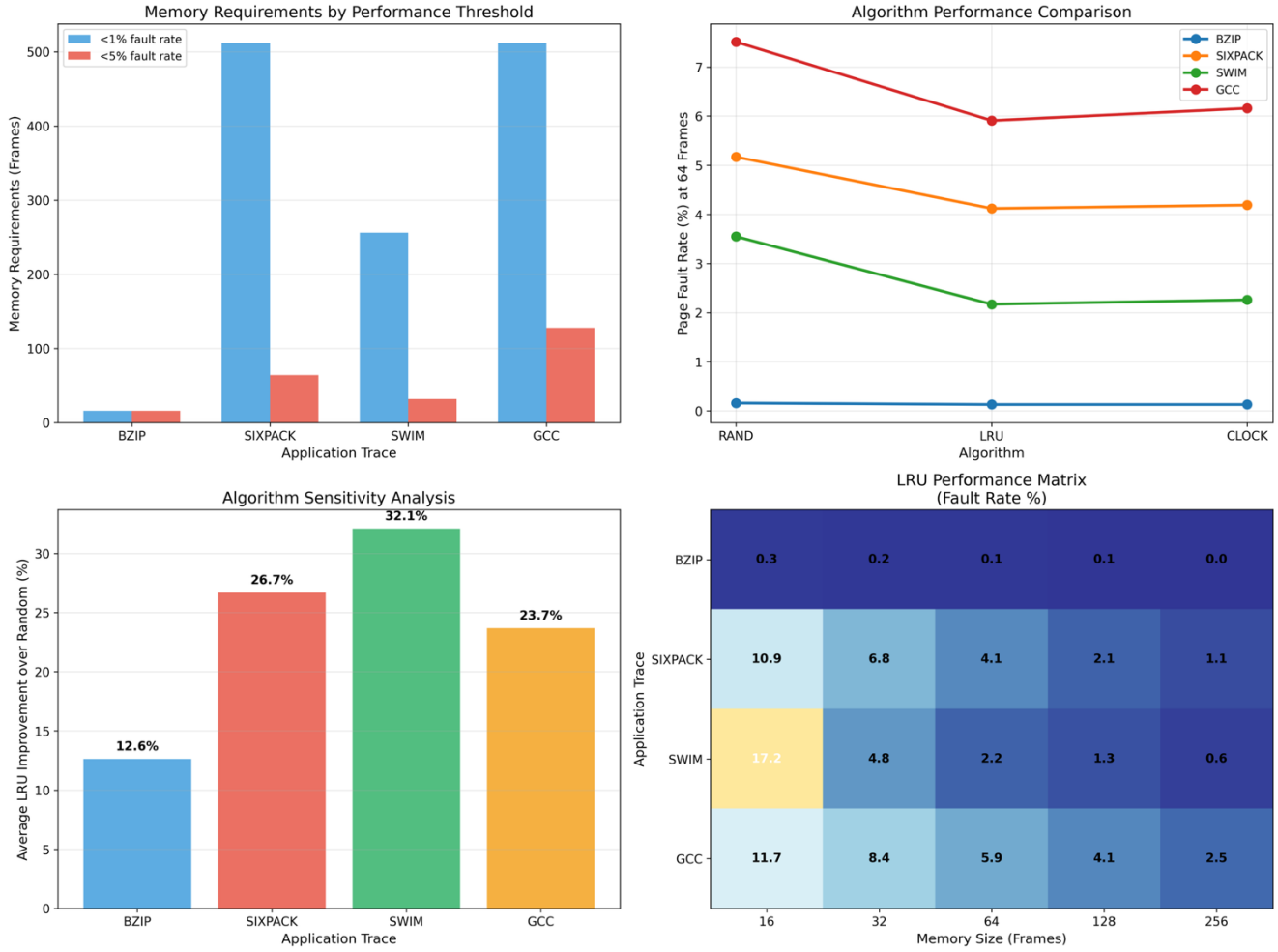



**Memory requirements:** Sixpack trace does not require as much memory as gcc, but it is also not as lightweight as bzip. A steep decline can be observed in the fault rate from 30 to 100 frames. This indicates a fairly compact working set.
**Key findings:** Performance trends are similar to the other traces, with LRU on top, Clock slightly behind it and Random trailing considerably. One notable difference is that the transition zone is more compact than GCC, which means that the program's execution has better locality behavior.

### 3.5. Cross-trace comparison




| Trace | Memory need (frames for <1% fault) | Best low memory algorithm | Dominant pattern |
|---|---|---|---|
| swim | 200 (LRU/Clock), 500 (Random) | LRU | Mixed locality + streaming |
| bzip | 12 (all algorithms) | LRU ≈ Clock | Strong temporal locality |
| gcc | ~1000 | LRU | Distributed, irregular |
| sixpack | ~100 | LRU | Moderate temporal locality |

**Universal patterns:** LRU is the best algorithm for all traces and memory sizes. Performance gaps are more pronounced under memory pressure (fewer than 50 frames). Each trace exhibits the characteristic "knee" curve where fault rates drop rapidly as memory increases but note how dramatically the position of this knee changes from trace to trace.
**Algorithm effectiveness:** The Clock algorithm is a strategy that efficiently balances performance and implementation complexity, attaining 90-95% of the LRU algorithm's effectiveness. Random

replacement is only appropriate for certain situations in which ease of implementation is more important than performance.

## 4. Conclusions

**Algorithm effectiveness:** The clock algorithm achieves 90–95% of LRU's performance while being much simpler to implement. Random replacement is appropriate only for specialized systems that prioritize implementation simplicity over performance.

**Memory requirements:** Programs need varying numbers of memory frames, from 12 (bzip) to ~1000 (gcc). And this is why operating systems need to be flexible about memory allocations rather than have fixed policies.

**Low memory performance:** LRU had the best low-memory performance and thus had the largest advantage when frame counts were severely constrained. This effect was most pronounced for applications with irregular access patterns (gcc) and least pronounced for applications with high locality (bzip). The performance benefits of pooling are especially notable on memory-constrained systems where the application is often subject to memory pressure.

**Algorithm universality:** No single algorithm is best under all workloads, but LRU was consistently among the best. However, in practice, Clock can achieve close to LRU performance on applications with high locality while being broadly adequate and simpler to implement.

**Key Insights:**

- Workload characteristics fundamentally determine algorithm effectiveness, with temporal locality patterns being the primary factor influencing replacement strategy success.
- Clock algorithm provides an excellent performance-complexity trade-off, achieving 90-95% of LRU performance with significantly lower implementation overhead.
- Memory requirements vary significantly by application type, necessitating dynamic allocation strategies rather than static resource provisioning.
- Algorithm convergence occurs at high memory levels, where remaining faults are primarily compulsory misses rather than capacity-related evictions.

**Practical implications:**

For operating systems: implement LRU when computational overhead is acceptable and strong temporal locality is expected; use Clock as a general-purpose default due to its simplicity and high performance; while Random is only recommended for specialized cases, prioritizing implementation simplicity over performance optimization.

**Limitations:**

This analysis only examined single-threaded programs using fixed 4KB pages. Real-world systems often involve multi-threaded applications and variable page sizes. Future research should also consider multi-program workloads and adaptive policies that can switch strategies based on the program's behavior patterns.

The experimental evaluation demonstrates that intelligent page replacement requires understanding both algorithm characteristics and application memory access patterns to achieve optimal system performance in modern computing environments.

## References


- Arpaci-Dusseau, R. and Arpaci-Dusseau, A. (2023). Three Easy Pieces Chapter 22: Beyond physical memory: policies. https://pages.cs.wisc.edu/~remzi/OSTEP/vm-beyondphys-policy.pdf (Accessed: September 27, 2025).