# Operating Systems
# Assignment 2
## Virtual Memory Simulator

---

Menno Brandt • Samuel Chau • Baojing Li

# Table of Contents

# Introduction

Memory management presents a constant challenge in balancing performance and resource utilisation. As programs grow in complexity, their memory needs often exceed physical limitations, necessitating efficient virtual memory systems. At the core of these systems are page replacement algorithms, each with unique approaches to managing memory allocation.

This report investigates the performance of 3 page replacement algorithms - Random (rand), Least Recently Used (LRU), and Clock, examining how each interacts with various memory constraints and program-specific demands. The analysis explores the efficiency of these algorithms under different scenarios, from memory shortages to excess availability, across diverse program types with varying memory demands and access patterns.

# Methods

### Metrics and Graphs:

The primary metric is page hit rate versus frame size. Four graphs are generated, each comparing all algorithms' performance. The graph generation process and code are detailed in *Appendix A*. Graphs follow the format used in *Appendix B*, as per Arpaci-Dusseau's Operating Systems: Three Easy Pieces[1].

### Initial Experiments:

We begin by running each memory trace (swim, bzip, gcc, sixtrack) with frame sizes ranging from 0-20 frames, in increments of 5. This provides a quick, broad overview to identify trends and pinpoint diminishing returns. Based on these results:

- If performance improvement has tapered off after a certain frame size, we will reduce the maximum frame size and rerun the trace.
- If performance continues to improve, we will increase the maximum frame size by 20 frames and rerun the tests.

Upon identifying an elbow point, we refine our analysis by shifting to 1-frame increments up to 150% of this point. Starting from 0 frames, we examine algorithm performance across scenarios of memory shortage, precise usage, and excess, revealing behaviour under varied constraints.

### Final Experiments:

Our initial testing revealed optimal frame size ranges for each trace. We identified elbow points (where performance improvement levels off) and set testing ranges accordingly.

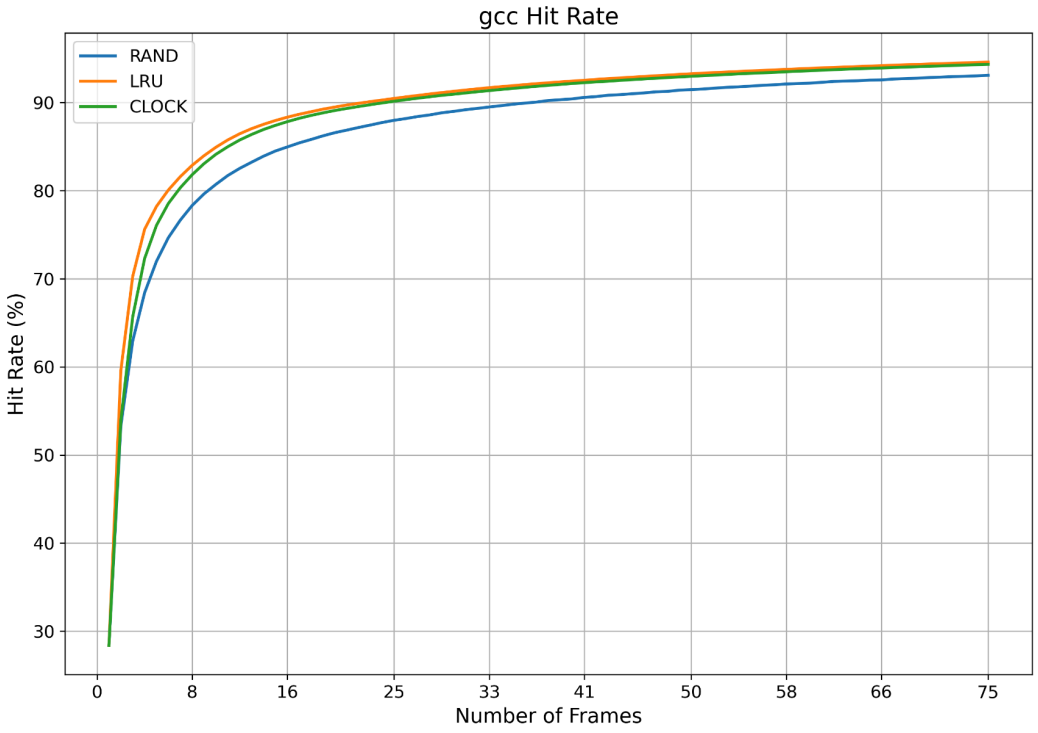
| Trace | Rough Elbow Point | Testing Range |
|---|---|---|
| gcc | 50 frames | 0-75 frames |
| bzip | 15 frames | 0-23 frames |
| swim | 40 frames | 0-60 frames |
| sixtrack | 25 frames | 0-30 frames |

## Results: gcc

The gcc Hit Rate graph shows the performance of RAND, LRU, and CLOCK page replacement algorithms across a range of 0-75 frames, based on the experimental methodology described. The trace's actual memory needs, inferred from the elbow point, can be identified at approximately 25-30 frames, where the rate of hit improvement starts to diminish despite insubstantial improvements continuing up to 75 frames.

When operating with a low number of frames, that is, below the elbow point, the LRU algorithm demonstrates superior efficacy by maintaining a higher hit rate compared to the other two, which suggests LRU's effectiveness where memory resources are constrained. This makes it the best choice with limited memory under such circumstances.

However, as the number of frames increases beyond the elbow point, the performance gap becomes gradual, since LRU and CLOCK are converging in the percentage of hit rate and marginally exceeding RAND. The mergence in performance at higher frame counts indeed elucidates no single algorithm can consistently outperform the others across all situations, especially with abundant memory resources, where distinctions between choices of algorithm become insignificant.

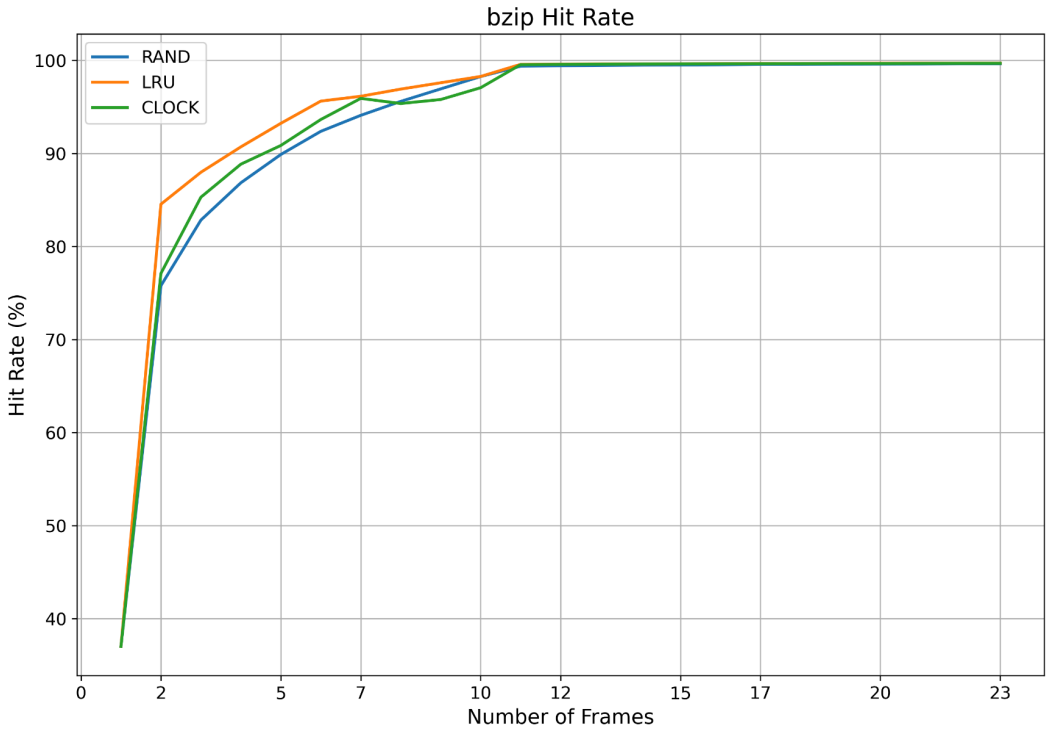


## Results: bzip

Conducted over a range of 0-23 frames, the bzip Hit Rate graph appears to require approximately 11 frames for optimal performance, indicated by the elbow point where the hit rate improvement begins to level off significantly for all algorithms. By extending the testing range to 23 frames allows us to observe behaviour beyond this optimal point, highlighting declining returns in hit rate improvements.

For low frame numbers (below 10 frames), the LRU algorithm has remarkable performance, maintaining a notably higher hit rate compared to RAND and CLOCK. This reveals that LRU is the most efficient choice for systems with limited memory resources in this case. Interestingly, it is worth mentioning that the performance of CLOCK and RAND in frames size between 7-10 varies, though CLOCK is traditionally considered as a more effective algorithm than RAND. One possible explanation can be the specific memory access patterns of bzip workload, where the performance of different algorithms heavily rely on. If the access pattern is relatively random or has a high degree of variability in temporal locality, CLOCK may suffer in certain scenarios.

Despite the fact that LRU shows clear advantages with limited frames, the performance difference becomes trivial with adequate memory allocation, underscoring the importance of considering specific workload characteristics since the optimal choice may vary given constrained memory resources relative to the program's demands.



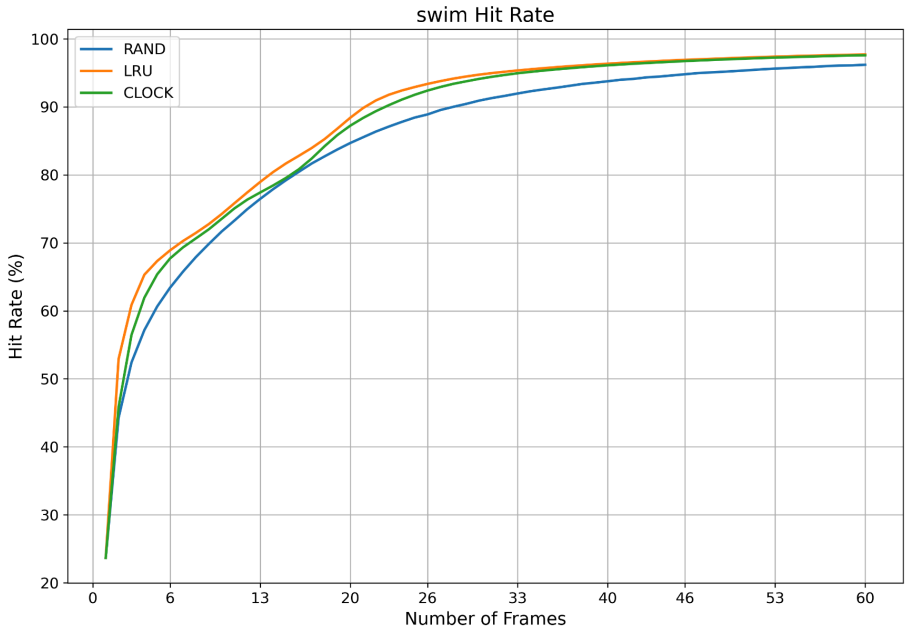
## Results: swim

The graph depicts hit rates for RAND, LRU, and CLOCK page replacement algorithms across 0-60 frames. All algorithms show rapid increase up to about 26 frames, indicating significant performance improvements with initial memory additions. Beyond 26 frames, the curves level off, suggesting this range satisfies the program's essential memory requirements, with diminishing returns thereafter.

LRU outperforms in the 0-13 frame range with CLOCK slightly behind. Notably, CLOCK shows a slight performance dip relative to RAND between 12-20 frames. This may be due to swim's memory access patterns in this range. CLOCK's use of a single "use bit" and circular scanning might be less effective for this specific pattern. However, CLOCK swiftly recovers, resuming its logarithmic trend and closely tracking LRU beyond 20 frames.

RAND underperforms compared to LRU and CLOCK until about 40 frames, where the gap narrows. As frame count approaches 60, all algorithms converge, with hit rates becoming nearly indistinguishable.

For limited memory (below 13 frames), LRU and CLOCK offer the best performance. After this point, LRU completely takes over due to clock's slight performance hitch at 12 frames. The 20-26 frame range balances performance gains and resource usage optimally. With ample memory (above 40 frames), algorithm choice has minimal impact on performance.



## Results: sixtrack

The graph illustrates hit rates for RAND, LRU, and CLOCK page replacement algorithms for the sixtrack program across a range of 0-38 frames. All three algorithms demonstrate a sharp increase in hit rate up to approximately 8-12 frames, indicating substantial performance gains with initial memory additions.

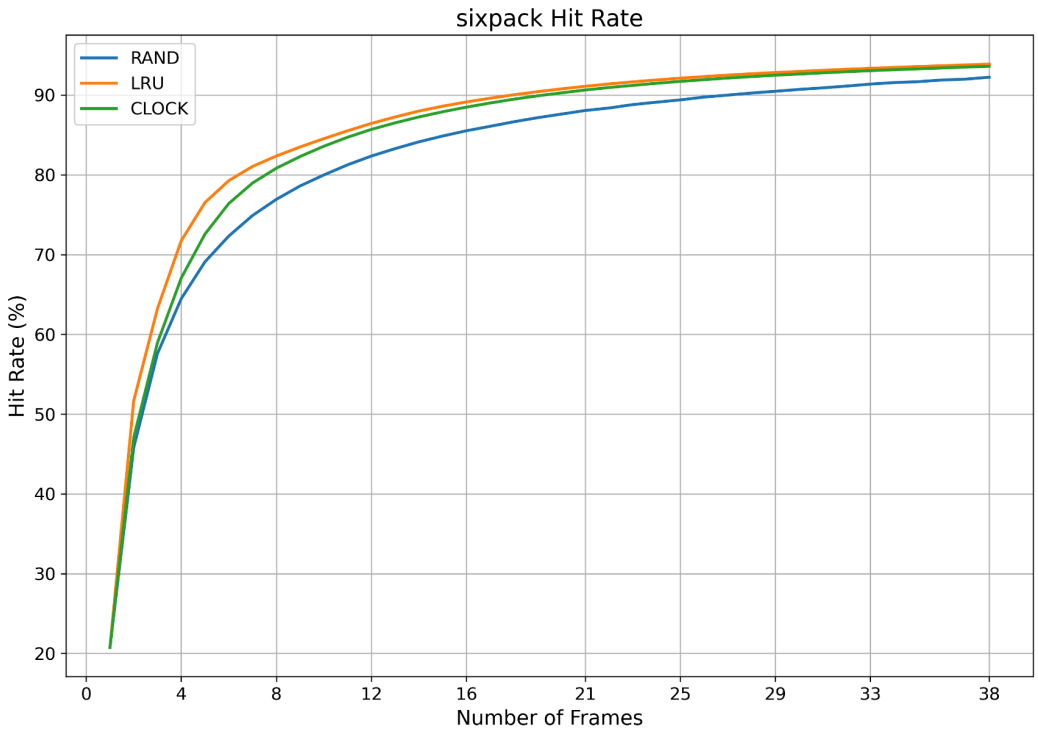
LRU exhibits superior performance in the low frame count range (0-8 frames), consistently maintaining a higher hit rate compared to CLOCK and RAND. CLOCK's performance closely tracks LRU throughout most of the range, with only a slight lag in the 4-12 frame region.

The curves begin to flatten around 12-16 frames, suggesting this range likely meets the program's core memory requirements. Beyond this point, all algorithms show diminishing returns, with smaller improvements in hit rate as frames increase.

RAND underperforms compared to LRU and CLOCK until about 25 frames, where the performance gap narrows reductively. As the frame count approaches 38, all three algorithms converge, with hit rates becoming nearly indistinguishable and approaching 93-94%.

Much like gcc, sixtrack doesn't show any notable anomalies or dips in CLOCK's performance relative to the other algorithms. However, sixtrack's optimal memory requirements are much smaller and suggest a less memory-intensive program.

The analysis implies that for systems with limited memory (below 12 frames), LRU offers the best performance, closely followed by CLOCK. There's an optimal range for memory allocation (16-21 frames) balancing performance gains and resource usage. In systems with ample memory (above 29 frames), the choice of the algorithm has minimal impact on performance for sixtrack.



## Conclusions

1. **Working Set Variability:** Programs exhibit diverse memory requirements, ranging from smaller working sets (like bzip) to larger ones (like gcc and swim)
2. **Algorithm Performance Patterns**
    a. **LRU** generally performs best at all frame counts in all programs, but especially better at lower frame counts.
    b. **CLOCK** often follows closely behind LRU's performance with some notable hitches.
    c. **RAND** typically underperforms compared to LRU and CLOCK, especially at lower frame counts.
3. **Performance Convergence:** As frames increase, the performance gap between algorithms tends to narrow, often converging at higher frame counts. In all scenarios, a 100% hit rate often acts as a performance asymptote.
4. **Diminishing Returns:** All programs show points of diminishing returns, where adding more frames yields progressively smaller improvements in hit rates. Performance can often be described as *logarithmic-like*. The larger the memory needed, the more logarithmic the graph looks.

**gcc's Larger Working Set**: As a compiler, gcc likely has a larger and more diverse working set compared to the other programs, due to its need to process and optimize complex source code[2]. This is evident in the larger number of frames required for gcc to reach its performance elbow point. The prolonged performance gap between LRU and the other algorithms at lower frame counts highlights the importance of capturing locality in gcc's memory access patterns, which LRU seems to do most effectively.

**bzip's Rapid Convergence and RAND Anomaly**: bzip's rapid hit rate convergence at a lower frame count suggests a small, stable working set due to its block-based compression algorithm[3], while RAND partially outperforming CLOCK in the 7-10 frame range indicates a specific access pattern that temporarily favors RAND's probabilistic approach over CLOCK's recency-based mechanism.

**swim's Temporal Locality:** As a shallow water modeling program using finite difference methods, swim likely exhibits high temporal locality, as it repeatedly accesses and updates the same grid points over multiple timesteps[4]. This is reflected in the strong performance of LRU, which excels at capturing temporal locality. The slight performance dip of CLOCK relative to RAND in the 12-20 frame range might indicate a specific access pattern in swim that temporarily misaligns with CLOCK's circular scanning, but CLOCK's quick recovery suggests this is a transient effect.

**sixtrack's Memory Access Patterns:** As a high energy nuclear physics accelerator design program, sixtrack likely exhibits strong spatial locality in its memory access patterns due to its use of particle tracking and beam dynamics simulations[5]. This aligns with the observed performance of LRU and CLOCK, which exploit spatial locality by keeping recently accessed pages in memory. The convergence of all algorithms at higher frame counts suggests that sixtrack's working set is relatively stable and can be accommodated effectively with sufficient memory.

## Reference List


1. Arpaci-Dusseau, R.H. & Arpaci-Dusseau, A.C. 2008–2023, *Operating Systems: Three Easy Pieces*, Arpaci-Dusseau Books, Madison, WI, viewed September 11th, 2024, https://pages.cs.wisc.edu/~remzi/OSTEP/
2. GNU Compiler Collection (2023), Wikipedia, viewed 11 September 2024, https://en.wikipedia.org/wiki/GNU_Compiler_Collection.
3. Bzip2 (2023), Wikipedia, viewed 11 September 2024, https://en.wikipedia.org/wiki/Bzip2.
4. SPEC CPU2000 1999, 171.swim SPEC CPU2000 Benchmark Description File, Standard Performance Evaluation Corporation, viewed 11 September 2024, https://www.spec.org/cpu2000/CFP2000/171.swim/docs/171.swim.html.
5. SPEC CPU2000 2000, 200.sixtrack SPEC CPU2000 Benchmark Description File, Standard Performance Evaluation Corporation, viewed 11 September 2024, https://www.spec.org/cpu2000/CFP2000/200.sixtrack/docs/200.sixtrack.html.

## Appendix A: Graph Generation Process

The following excerpt from the testing script demonstrates how memory traces are tested with different page replacement algorithms and frame sizes, with results stored in a CSV file. Afterwards, a Python script generates graphs based on these results.

1. Test Bash Script

```
# Function to run tests for a single trace with all algorithms
run_trace_tests() {
        local trace=$1
        local start_frames=$2
        local end_frames=$3
        local step_frames=$4
        for algorithm in rand lru clock; do
        for ((frames=start_frames; frames<=end_frames; frames+=step_frames)); do
                result=$($SIMULATOR "traces/${trace}.trace" $frames $algorithm quiet)
                # Extract values from result
                page_faults=$(echo "$result" | grep "total disk reads:" | awk '{print $4}')
                disk_reads=$(echo "$result" | grep "total disk reads:" | awk '{print $4}')
                disk_writes=$(echo "$result" | grep "total disk writes:" | awk '{print $4}')
                fault_rate=$(echo "$result" | grep "page fault rate:" | awk '{print $4}')
                echo
"$trace,$algorithm,$frames,$page_faults,$disk_reads,$disk_writes,$fault_rate" >> $OUTPUT
        done
        done
}
```

2. Python Script (Graph Generation)

```
import pandas as pd
import matplotlib.pyplot as plt

# Read the CSV file containing test results
df = pd.read_csv('results.csv')

# Filter data for the specific trace and calculate hit rate
data = df[df['trace'] == trace]
data['hit_rate_percentage'] = (1 - data['fault_rate']) * 100

# Plot hit rate for each algorithm
plt.figure(figsize=(12, 8))
for algorithm in data['algorithm'].unique():
        algo_data = data[data['algorithm'] == algorithm]
        plt.plot(algo_data['frames'], algo_data['hit_rate_percentage'],
label=algorithm.upper())

plt.title(f'Hit Rate vs Frame Size - {trace} Trace')
plt.xlabel('Number of Frames')
plt.ylabel('Hit Rate (%)')
plt.grid(True)
plt.legend()
plt.savefig(f'graphs/{trace}_hit_rate.png', dpi=300)
plt.close()
```

**Appendix B: Operating Systems: Three Easy Pieces - Figure 22.7**

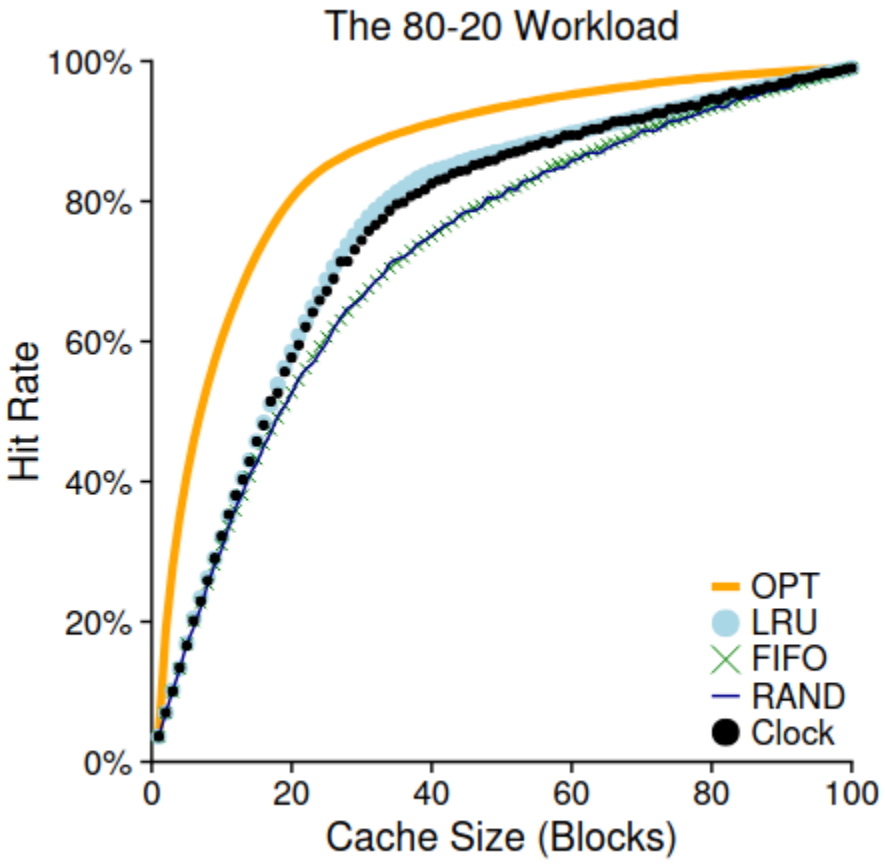



Figure 22.9: **The 80-20 Workload With Clock**