# DAS Page Replacement Algorithm

By Ramya Danappa

Supervising Professor: Dr. Song Jiang

Presented to the Faculty of the Graduate School of The
University of Texas at Arlington in Partial Fulfillment of
the Requirements
for the Degree of
MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON
May 2020

Supervising Committee:

Dr. Hao Che

Dr. Jia Rao

# Acknowledgements

I would like to express my deepest appreciation and gratitude to my advisor, Dr. Song Jiang, for giving me this opportunity to work with him and for his constant support and guidance throughout my Master thesis. I feel extremely honored to be a part of system lab, where I have had the opportunity to explore new horizons, which I never knew existed. I am also extremely grateful to Dr. Hao Che and Dr. Jia Rao for accepting to be on the thesis committee.

Finally, a huge thank you to my parents and my friends for their moral support and encouragement in the graduation days and GOD, for his grace in me.

June 2, 2020

# Abstract

There are different page replacement algorithms and yet there seems to be some drawbacks in them, making no page replacement algorithm ideal. To be one step closer to achieving the ideal algorithm it is vital to have a maximum cache hit ratio and strong consistent across different workload. In this paper we shall explore a new cache management policy called "**Dynamic and Stable page replacement algorithm**" uses frequency and recency importance dynamically". The proposed page replacement algorithm has overcome the drawbacks of the LRU (Least Recently Used) algorithm in many scenarios and also overcome the drawbacks of LFU (Least Recently Used).Like LRU can be easily polluted by a scan, that is by a sequence of one-time use only page requests leading to decrease in performance and also LFU does not pay attention to recent history.

The proposed algorithm has achieved consistent hit ratio by using the fundamentals of page replacement algorithm which are: low recency and high frequency of the blocks. To achieve this stability we have to integrate the design principles of LRU and LFU in the cache and the blocks have been placed blocks according to the hit on the block, the above mentioned is done using simple computations.

# Table of Contents

# Chapter 1: Introduction

## 1.1The LRU algorithm

For any input-output system to have performance stability it is very critical that the cache block replacement algorithm be extremely efficient. The simplest algorithm used to manage the cache data is the LRU.
The LRU algorithm ensures that when a block present in the cache is accessed the block is moved to the top of the stack. When block not present in the cache is accessed, the newly accessed block takes the place of the least recent block in the cache.

### Problems in Current LRU page replacement Algorithm

Despite the simplicity there have been many anomalies detected within this algorithm. For instance, despite having a significant increase in the cache size the hit rate only increases slightly. LRU doesn’t perform well in weak localities such as regular pattern access and access on blocks with distinct frequencies.

Here we have listed few of the LRU drawbacks:

- The LRU algorithm sequential scans can cause commonly referenced blocks in the cache to be replaced. In an ideal page replacement algorithm commonly referenced two blocks must not be replaced by blocks that haven't been used.
- The LRU algorithm for loop access does not maintain a miss rate which is close to the buffer space shortage ratio hence this algorithm always removes the blocks that will be accessed the soonest as these blocks have not been accessed recently.

1.2The LFU algorithm:

In contrast to the LRU algorithm, this algorithm computes the frequency of access for every block. When the block present in the cache is accessed the block's frequency will increase, when a block not present in the cache is accessed then the newly accessed block replaces the least frequency block in the cache.

Few problems present in LFU:

- While the LFU method may seem like an intuitive approach to memory management it is not without faults. Consider an item in memory which is referenced repeatedly for a short period of time and is not accessed again for an extended period. Due to how rapidly it was just accessed its counter has increased drastically even though it will not be used again for a decent amount of time. This leaves other blocks which may be used more frequently susceptible to purging simply because they were accessed through a different method.
- Moreover, new items that just entered the cache are subject to being removed very soon again, because they start with a low counter, even though they might be used very frequently after that. Due to major issues like these, an explicit LFU system is fairly uncommon; instead, there are hybrids that utilize LFU concepts.
- LFU does not take into consideration the recency of the blocks.

1.3 The optimal page replacement algorithm:

The OPT algorithm is a theoretical concept which evicts the block from the cache based on future access predictions of the blocks present in the cache, this algorithm is hard to achieve as future predictions on the access can not be determined.

# Chapter 2: Related work

## 2.1 Offline Optimal

For a priori known page reference stream, Belady's OPT that replaces the page that has the most considerable forward distance is known to be OPT in terms of the hit ratio. The policy OPT provides an upper bound on the achievable hit ratio by any on-line policy.

## 2.2 Recency Based Replacement:

The policy LRU always replaces the least recently used page [13]. It dates back at least to 1965, and may in fact be older. Various approximations and improvements to LRU abound, see, for example, enhanced clock algorithm [11] It is known that if the workload or the request stream is drawn from an LRU Stack Depth Distribution (SDD), then LRU is the optimal policy [2]. LRU has several advantages; for example, it is simple to implement and responds well to changes in the underlying SDD model. However, while the SDD model captures "recency," it does not capture "frequency". To quote from [12, p. 282]: "The significance of this is, in the long run, that each page is equally likely to be referenced and that therefore the model is useful for treating the clustering effect of locality but not the nonuniform page referencing."

## 2.3 Frequency Based Replacement:

The Independent Reference Model (IRM) provides a workload characterization that captures the notion of frequency. Specifically, IRM assumes that each page reference is drawn in an independent fashion from a fixed distribution over the set of all pages in the auxiliary memory. Under the IRM model, policy LFU that replaces the least frequently used page is known to be optimal. The LFU policy has several drawbacks: it requires logarithmic implementation complexity in cache size, pays almost no attention to recent history, and does not adapt well to changing access

patterns since it accumulates stale pages with high-frequency counts that may no longer be useful. A relatively recent algorithm LRU-2 approximates LFU while eliminating its lack of adaptivity to the evolving distribution of page reference frequencies. This was a significant practical step forward. The basic idea is to remember, for each page, the last two times when it was requested, and to replace the page with the least recent penultimate reference. Our Model has both frequency and recency, which cooperates eviction of any block based on the frequency and recency.

# Chapter 3: DAS page replacement algorithm

## 3.1 General Idea

In DAS (Dynamic and Stable) page replacement algorithm, we divide the cache into two partitions, one holds the LRU blocks and the other holds the LFU blocks. The LRU portion contains most recently used blocks, and their frequencies will always be less than the minimum frequency in the LFU portion frequency. LFU portion includes the most used blocks, and their frequency is always larger than any blocks in the LRU portion.

We divided the cache, where size in blocks is ‘c’, into a frequent part and recent part in terms of their size required. The recent past, with its size $c_{LRU}$, is used to store most recently used blocks, and frequent part, with its size $c_{LFU}$, is used to store high frequently used blocks, where $c_{LRU} + c_{LFU} = c$.

## 3.2 The Partial DAS Algorithm based on LRU stack

The partial DAS algorithm is effectively built on the LRU stack model, which is an implementation of the LRU structure. The LRU stack contains c entries, each of which represents a block. Usually, c is the cache size in the blocks. The DAS algorithm makes use of the LRU stack to keep track of recency, and to maintain the frequency; it estimates the frequency of all blocks. In contrast to LRU stack, it also keeps the frequency track. Still, this frequency does not participate in the eviction of the LRU blocks, but these frequencies helps to replace the least frequently used block in the LFU portion of the cache. The LRU portion of stack performs the same as LRU stack in operation, but it has a variable size. The DAS algorithm is partially inspired by the observation of improper LRU replacement behavior. If a block is evicted from the bottom often LRU stack, it means the block occupies a buffer during the period. Why do we have to afford a buffer for another long idle, when the block is loaded in the cache next time as an LRU?

### 3.3 The Partial DAS algorithm based on LFU stack

The partial DAS algorithm is effectively built using the LFU stack model, which is an implementation of the structure of LFU. The LFU stack contains l entries, each of which represents a block, and each block has a frequency count. Usually, l is the cache size in blocks. The DAS algorithm keeps track of frequency in the LFU portion. These blocks in LFU relatively have high-frequency value blocks when compared to the LRU portion blocks. The block from the LFU will be replaced by the LRU block only if the frequency of the block is less in the LFU portion. The main drawback of LFU is that it ignores the recency of the block. If the block has higher frequency then it has less chance in eviction even though it is not accessed for a longer period. These limitations inspired us to use a partial stack of LFU and partial stack as LRU.

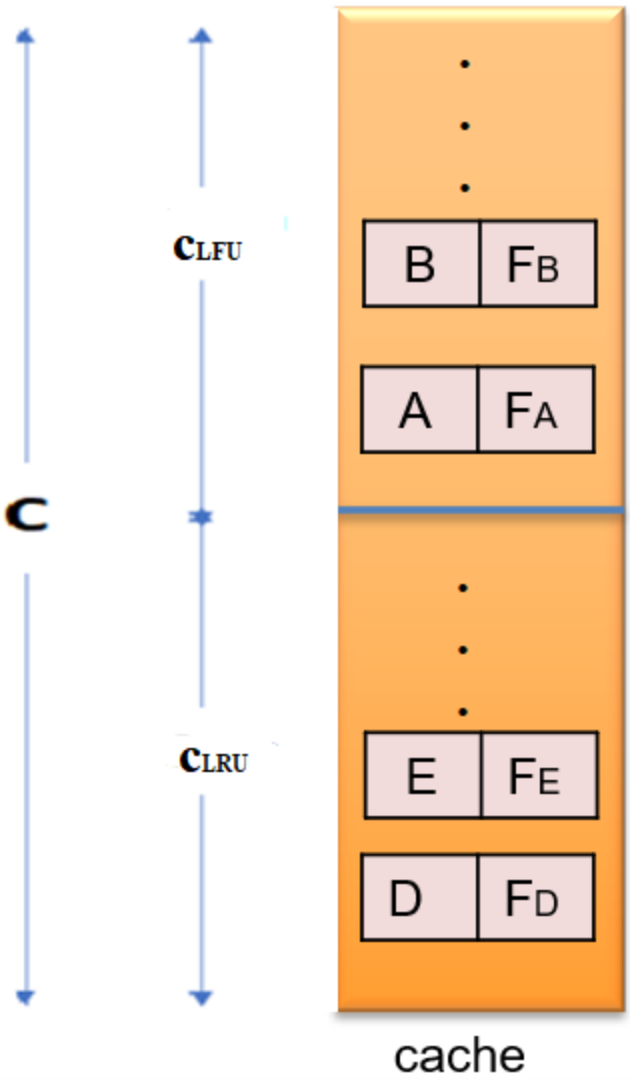


Fig 1: Architecture of DAS algorithm

Let us consider A,B,D and E are blocks and $F_A$,$F_B$,$F_D$ and $F_E$ will be there frequencies respectively. c will be the cache size $c_{LRU}$ cache of LRU portion and $c_{LFU}$ cache of LFU portion. Where c= $c_{LRU}$ + $c_{LFU}$.

### 3.4 Detailed Description

In the DAS algorithm, when a cache is set and a new block available and stack is empty, then we fill the cache. Firstly, it fills the empty portion of LRU, and then we fill the LFU portion of the stack. The block access in the cache, then one of these cases performs:

Case 1: Block X is present in LFU portion:

This access is guaranteed to be hit in the cache. We increase the frequency count of the block in LFU portion.

Case 2: Block X is present in LRU portion:

This access will be hit on the cache. Then we will update the frequency of block in cache. The block is moved to the top of the LRU portion stack. The block X has a possible chance to be part of the LFU portion in two ways 1) If LFU assigned portion has free space available then add block to LFU portion, 2)If the minimum frequency of the block in LFU portion is less than the frequency of block X. Then these blocks exchange their position.

Case 3: Block X not present:

This is miss, we need to add this new block to the free LRU portion. If LRU portion is full then we evict the block from LRU portion. And new block will be placed in top LRU portion and new frequency of that block is created.

# Chapter 4: PERFORMANCE EVALUATION:

## 4.1 Experimental Settings

To validate our DAS algorithm and to demonstrate its strength, we use trace-driven simulations with various types of workloads to evaluate and compare it with other algorithms. We have adopted many application workload traces used in previous literature, aiming at addressing the limits of LRU. We have also generated a synthetic trace. Among these traces, cpp, cs, glimpse, and Postgres are used in [6,7] (\cs" is named as \cscope" and \postgres" is named as \postgres2" there), a sprite is used in [4], multi1, multi2, multi3 are used in [5]. We briefly describe the workload traces here.

1. **2-pools** is a synthetic trace, which simulates the application behavior of example 3 in Section 1.1 with 100,000 references.

2. **cpp** is a GNU C compiler pre-processor trace. The total size of C source programs used as input is roughly 11 MB.

3. **cs** is an interactive C source program examination tool trace. The total size of the C programs used as input is roughly 9 MB.

4. **glimpse** is a text information retrieval utility trace. The total size of text files used as input is roughly 50 MB.

5. **postgres** is a trace of join queries among four relations in a relational database system from the University of California at Berkeley.

6. **sprite** is from the Sprite network le system, which contains requests to a le server from client workstations for a two-day period.

7. **multi1** is obtained by executing two workloads, cs, and cpp, together.

8. **multi2** is obtained by executing three workloads, cs, cpp, and postgres, together.

9. **multi3** is obtained by executing four workloads, cpp, Gnuplot, glimpse, and postgres, together.

Through an elaborate investigation, Choi et al. classify the le cache access patterns into four types [4]:
Sequential references: all blocks are accessed one after another, and never re-accessed;
Looping references: all blocks are accessed repeatedly with a regular interval (period);
Temporally-clustered references: blocks accessed more recently are the ones more likely to be accessed in the near future;
Probabilistic references: each block has a stationary reference probability, and all blocks are accessed independently with the associated probabilities.

The classification serves as a basis for access pattern detections and for adapting different replacement policies in their AFC scheme. For example, LRU applies to sequential and looping patterns, LRU applies to temporally-clustered patterns, and LFU applies to probabilistic patterns. Though our DAS policy does not depend on such a classification, we would like to use it to present and explain our experimental results. Because a sequential pattern is a special case of the looping pattern (with an infinite interval), we only use the last three groups: looping, temporally-clustered, and probabilistic patterns.

Policies LRU belong to the same category of replacement policies as DAS. In other words, these policies take the same technical direction | predicting the access possibility of a block through its own history access information. Thus, we focus our performance comparisons between ours and the LRU policy. We also include the results of OPT, an optimal offline replacement algorithm [2] for comparisons.

## 4.2 Performance Evaluation Results

We divide the 9 traces into 4 groups based on their access patterns. Traces cs, postgres, and glimpse belong to the looping type, traces cpp and 2-pools belong to the probabilistic type, trace sprite belongs to the temporally-clustered type and traces multi1, multi2, and multi3 belong to the mixed type. For the policies with pre-determined parameters, we used the parameters presented in their related papers. The parameter of the DAS algorithm, $c_{LRU}$ and $c_{LFU}$ assigned 10% and 90% of cache size respectively.

### *4.2.1 Replacement Performance on Looping Patterns*

Traces cs, glimpse, and postgres have looping patterns with long intervals. As expected, LRU performs poorly for these workloads with the lowest hit rates among the policies. Let us take cs as an example, which has a pure looping pattern. Each of its blocks is accessed almost with the same interval. Since all blocks with looping accesses have the same eligibility to be kept in the cache, it is desirable to keep the same set of blocks in the cache no matter what blocks are referenced currently. In the looping pattern, recency predicts the opposite of the future reference time of a block: the larger the recency of a block is, the sooner the block will be re-referenced. The hit rate of LRU for cs is almost 0% until the cache size approaches 1,400 blocks, which can hold all the blocks referenced in the loop.

Except for cs, the other two workloads have mixed looping patterns with different intervals. LRU presents stair-step curves to increase the hit rates for those workloads. LRU is not effective until all the blocks in its locality scope are brought into the cache. For example, only after the cache can hold 355 blocks does the LRU hit rate of postgres have a sharp increase from 16.3% to 48.5%.

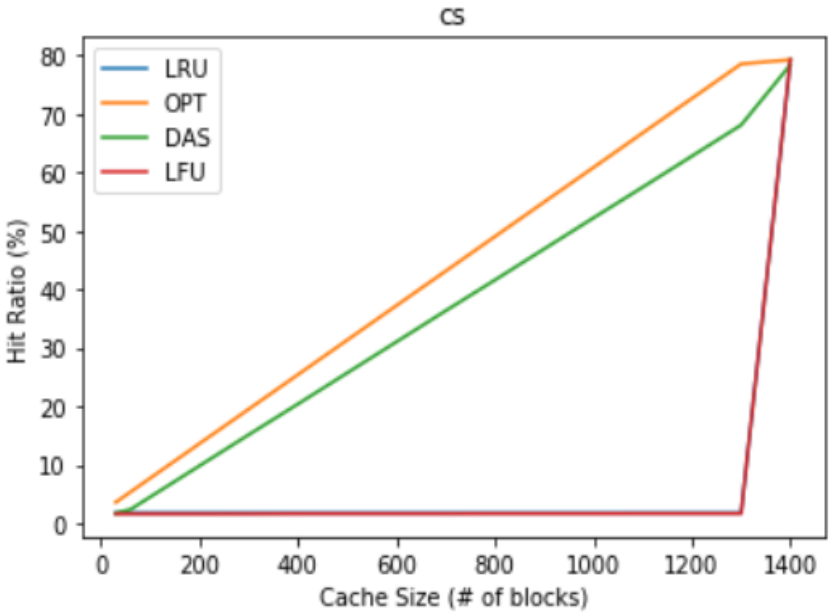


Fig 2: The hit rate curves of cs trace for the replacement algorithm

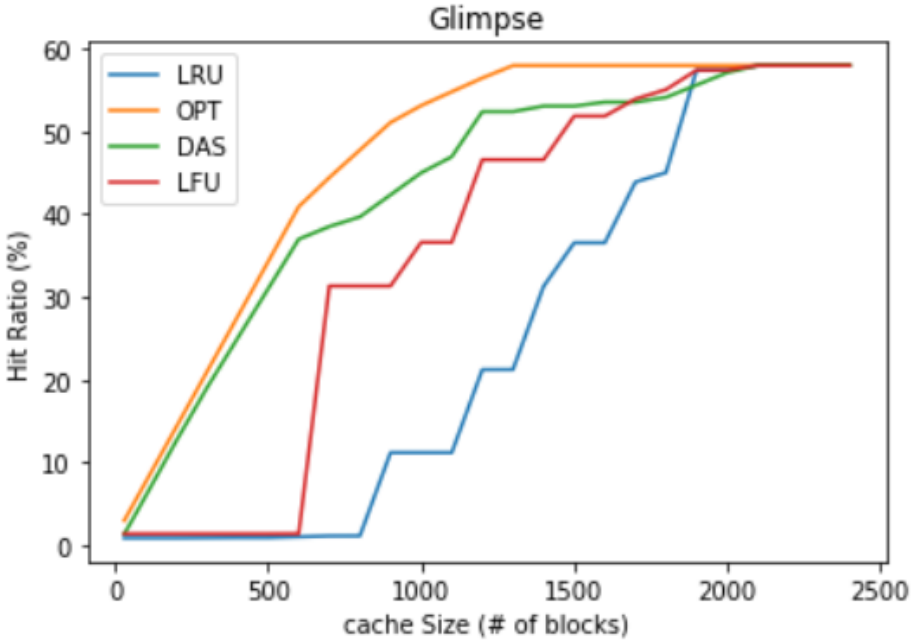


Fig 3: The hit rate curves of GLimpse trace for the replacement algorithm

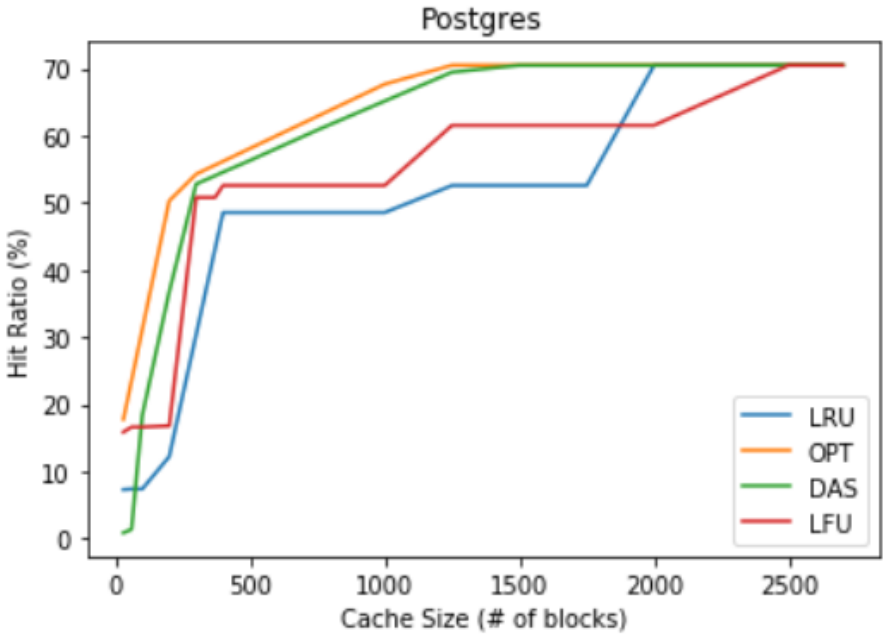

Fig 4: The hit rate curves of Postgres trace for the replacement algorithm

We simulate the results show of DAS significantly outperforms all of the other algorithms and its hit rate curves are very closely to those of OPT. Meanwhile, show that the hit rates of CS, glimpse and postgres are closer to those of OPT than the hit rates of LRU.

*4.2.2Replacement Performance on Probabilistic Patterns*

According to the detection results in workload cpp exhibits probabilistic reference pattern. The cpp hit rate in Figure 5 shows that before the cache size increases to 100 blocks, the hit rate of LRU is much lower than that of DAS for cpp. For example, when the cache size is 50 blocks, hit rate of LRU is 9.3%, while hit rate of DAS is 38.0%. This is because holding a major frequency and recency together forms needs of about100 blocks. LRU can not exploit frequency until enough cache space is available to hold all the recently referenced blocks. However, the capability for DAS to exploit recency and also frequency does not depend on the cache size. Workload 2-pools is generated to evaluate replacement policies on their abilities to recognize the long-term reference behaviors. Though the reference frequencies are largely different between record blocks and index blocks, It is hard for LRU to distinguish them when the cache size is relatively small compared with the

number of referenced blocks. This is because LRU takes only recency into consideration eviction. In workload 2-pools, the blocks with high access frequency and blocks with low access frequency are alternatively referenced, thus no sign of an early point eviction can be detected.

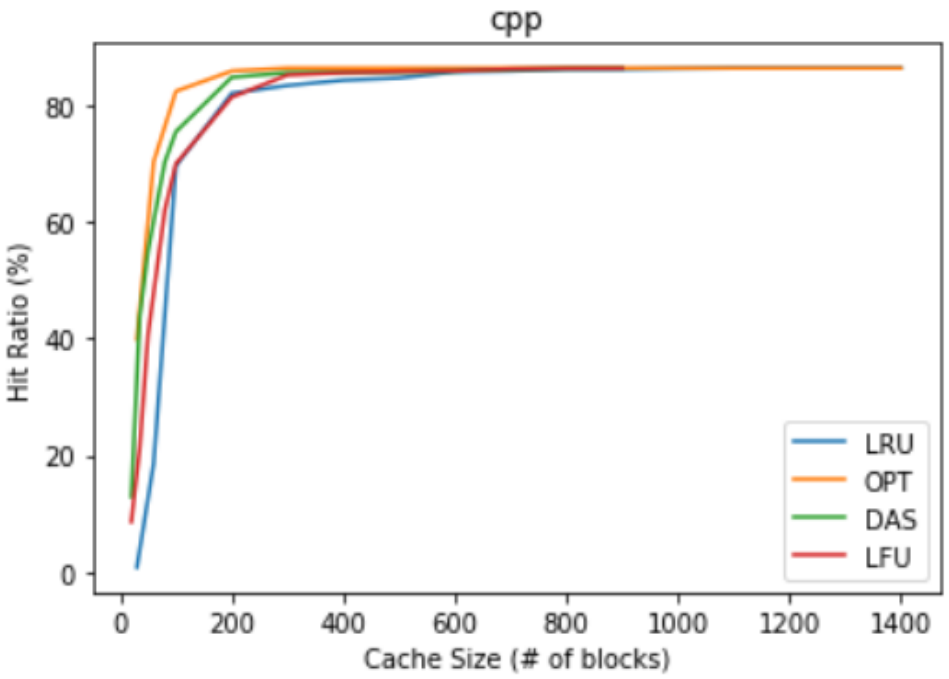


Fig 5: The hit rate curves of cpp trace for the replacement algorithm

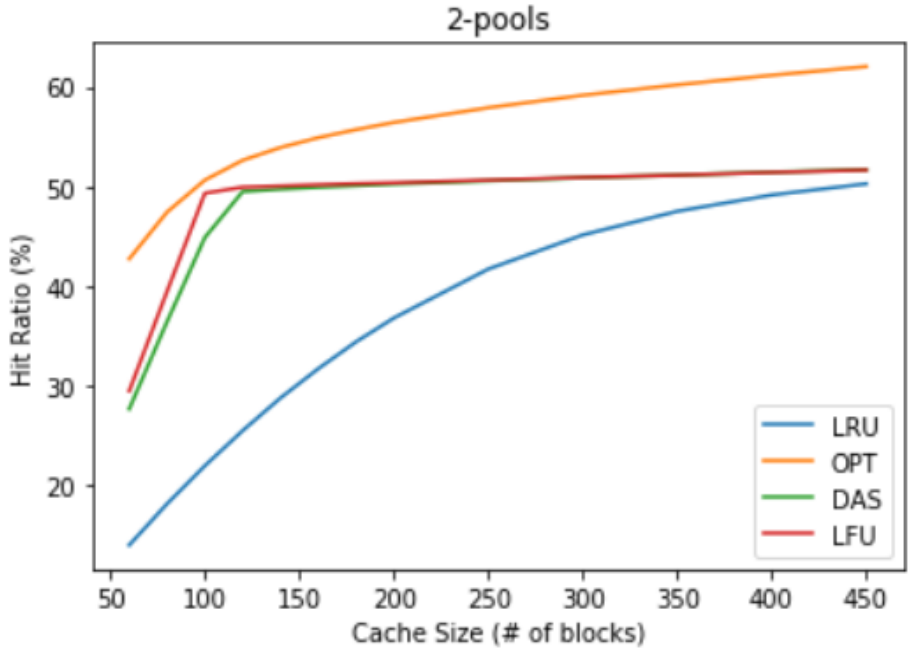


Fig 6: The hit rate curves of 2pools trace for the replacement algorithm

*4.2.3 Replacement Performance on Temporally Clustered Patterns*

Workload sprite exhibits temporally clustered reference patterns. The sprite result in Figure 7 shows that the LRU hit rate curve smoothly climbs with the increase of the cache size. Although there is still a gap between the LRU and OPT, the slope of the LRU is close to that of OPT. Sprite is a so-called LRU-friendly workload [14], which seldom accesses more blocks than the cache size over a fairly long period. For this type of workload, the behavior of all our policies should be similar to that of LRU, so that their hit rates could be close to that of LRU. Before the cache size reaches 350 blocks, the hit rates of our are slightly less than those of LRU but close to LRU. Because we have frequent locality scope changes, and the slight negative effect of the extra misses is minimal.

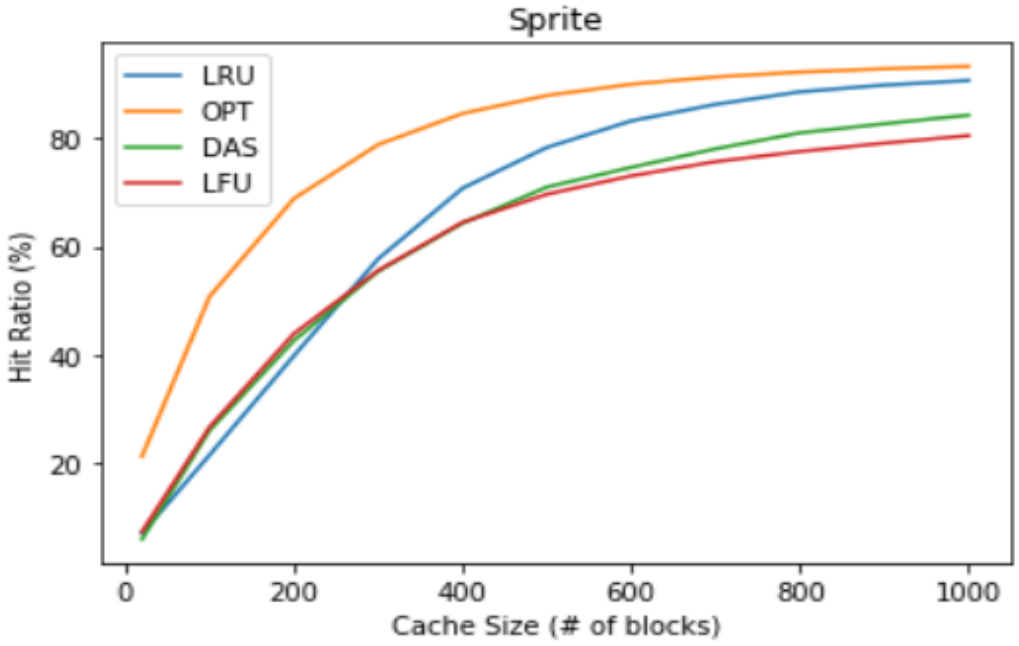


Fig 7: The hit rate curves of sprite trace for the replacement algorithm

### *4.2.4 Replacement Performance on Mixed Patterns*

Multi1, multi2, and multi3 are traces with mixed access patterns. The authors in [15] provide a detailed discussion of why their UBM shows the best performance among the polices they have considered {UBM, SEQ, 2Q, EELRU, and LRU. Here we focus on performance differences be-tween LIRS and UBM. UBM is a typical spatial regularity detection-based replacement policy that conducts exhaustive reference pattern detections. UBM tries to identify sequential and looping patterns and applies LRU to the detected patterns. UBM further measures looping intervals and conducts period-based replacements. For unidentified blocks, LRU is applied. A dynamical buffer allocation among blocks managed by different policies is employed. Without devoting specific effort to specific regularities, we can also see that DAS is stable and also performs better in all these traces.

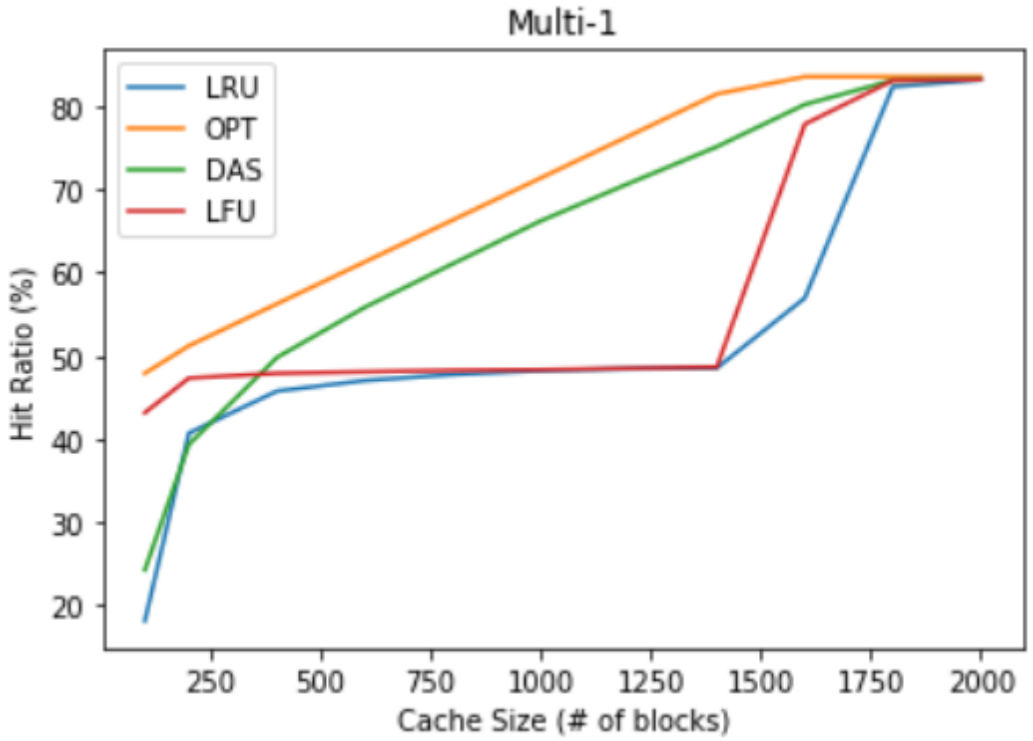


Fig 8: The hit rate curves of multi1 trace for the replacement algorithm

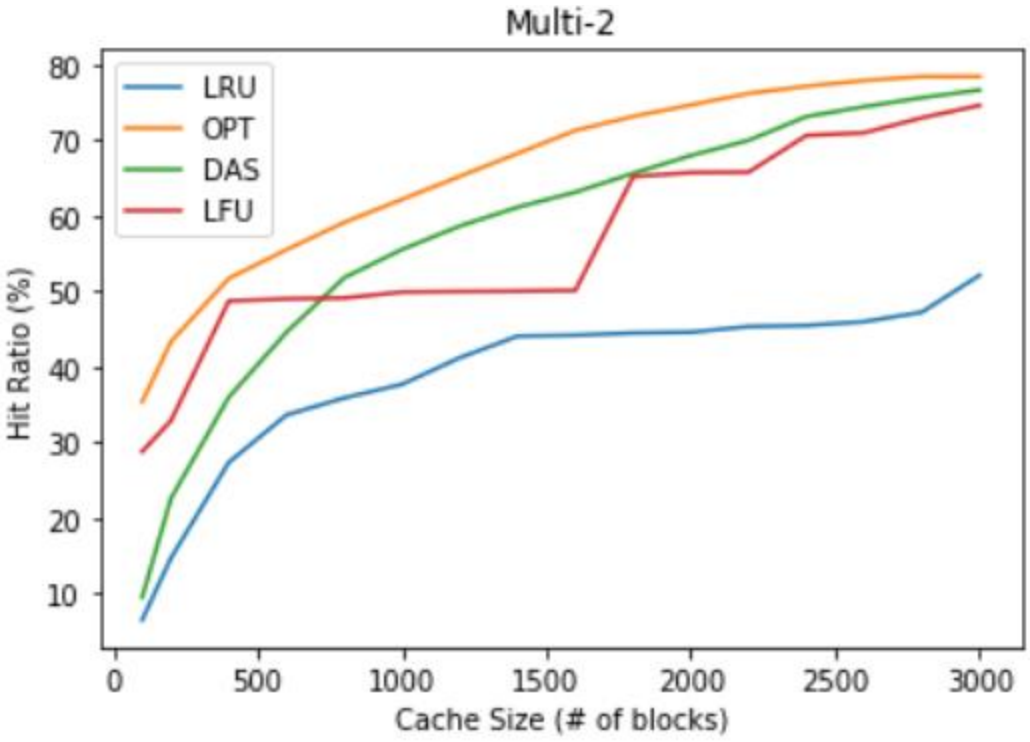


Fig 9: The hit rate curves of multi2 trace for the replacement algorithm

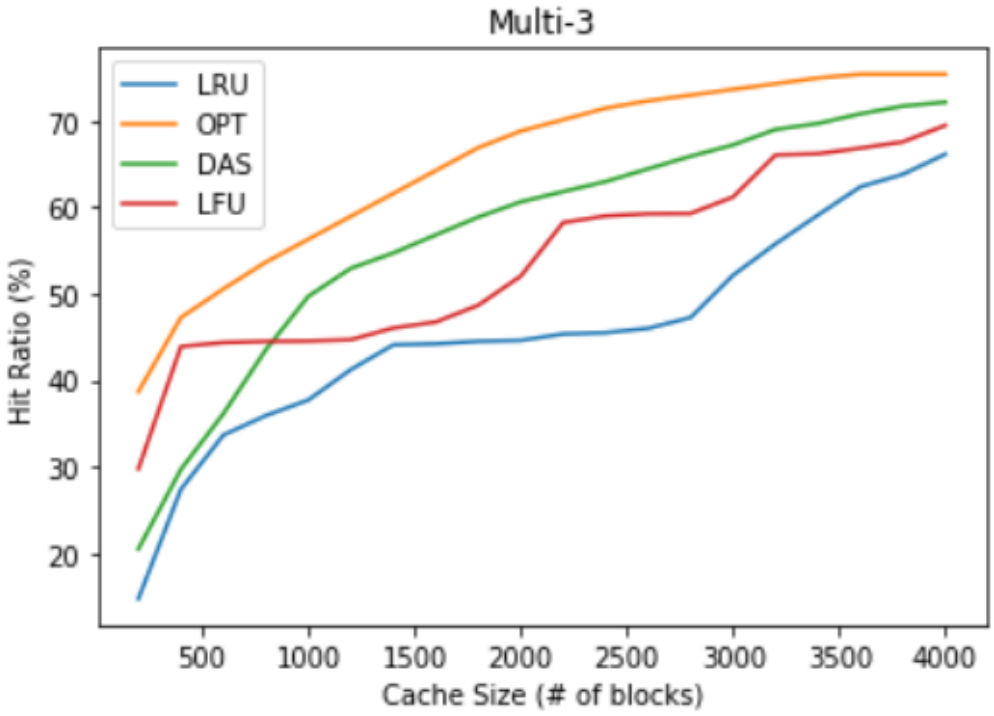


Fig 10: The hit rate curves of multi3 trace for the replacement algorithm

*4.2.4* Sensitivity Analysis on Partitioning LRU and LFU portion

DAS has one tuning parameters, i.e. partitioning the cache. By varying LRU portion and LFU portion we can observe difference in hit ratio.We use the cs (Fig11) to identify the changes in the DAS algorithm in different ratio portion. LRU and LFU both lie in the same curve, whereas in DAS we can observe implementing recency and frequency together form a stable and better-caching approach.

We can also observe special feature in DAS that is inter-relation between the frequency and recency. This mainly impact on the DAS algorithm. For example, in figure 11, where we can observe the hit ratio increase when we increase LFU portion. This unique feature is mainly because LRU portion is always controlling in LFU entry blocks.

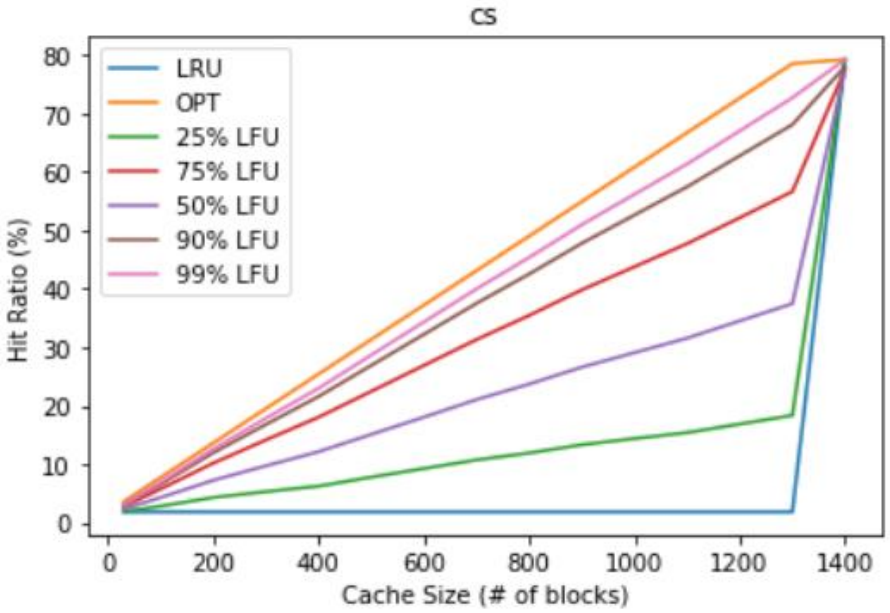


Fig 11: The hit rate curves of cs trace with different partition size of LRU and LFU for the replacement algorithm

4.2.5 Overhead Analysis:

LRU and LFU are known for their simplicity and efficiency. Comparing the time and space overhead of DAS and LFU, we show that DAS keeps the LFU

merit of low overhead. The time overhead of DAS algorithm is O(1), which is almost the same as that of LFU with a few additional operations such as the swapping the LRU block and LFU blocks.

# Chapter 5: Conclusion

We have presented a new page replacement algorithm called DAS that is dynamic. DAS is very simple to implement and has low space overhead with minimum computation. We have empirically demonstrated that DAS outperforms the LRU page replacement algorithm. The proposed algorithm is dynamic has achieved stability in hit ratio by using the fundamentals methods of page replacement algorithm which is: low recency and high frequency of the blocks.

As future work, we are planning to implement the self-tunable parameter. That helps to partition the cache into the LRU portion and LFU portion. We have also observed in sensitivity analysis tunable parameter proportion of results. Hence self-tunable parameter will help to achieve a better hit ratio.

# References


1. L.A. Belady, R.A. Nelson, and G.S. Shedler, “An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine,” Comm. ACM, vol. 12, pp. 349-353, 1969
2. E.G. Coffman and P.J. Denning, Operating Systems Theory. Prentice-Hall, 1973.
3. P. Cao, E.W. Felten, and K. Li, “Application-Controlled File Caching Policies,” Proc. USENIX Summer 1994 Technical Conf., pp. 171-182, June 1994.
4. J. Choi, S. Noh, S. Min, and Y. Cho, “Towards Application/FileLevel Characterization of Block References: A Case for FineGrained Buffer Management,” Proc. 2000 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems, pp. 286-295, June 2000
5. J. Choi, S. Noh, S. Min, and Y. Cho, “An Implementation Study of a Detection-Based Adaptive Block Replacement Scheme,” Proc. 1999 Ann. USENIX Technical Conf., pp. 239-252, June 1999.
6. C. Ding and Y. Zhong, “Predicting Whole-Program Locality through Reuse-Distance Analysis,” Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation, pp. 245-257, June 2003.
7. W. Effelsberg and T. Haerder, “Principles of Database Buffer Management,” ACM Trans. Database Systems, pp. 560-595, Dec. 1984.
8. S. Jiang and X. Zhang, “LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance,” Proc. 2002 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems, pp. 31-42, June 2002.

9. V. Phalke and B. Gopinath, “An Inter-Reference Gap Model for Temporal Locality in Program Behavior,” Proc. 1995 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems, pp. 291-300, May 1995.
10. G. Glass and P. Cao, “Adaptive Page Replacement Based on Memory Reference Behavior,” Proc. 1997 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems, pp. 115-126, May 1997.
11. W. R. Carr and J. L. Hennessy, “WSClock – a simple and effective algorithm for virtual memory management,” in Proc. Eighth Symp. Operating System Principles, pp. 87–95, 1981.
12. R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, “Evaluation techniques for storage hierarchies,” IBM Sys. J., vol. 9, no. 2, pp. 78–117, 1970.
13. T. Johnson and D. Shasha, “2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm,” Proc. 20th Int’l Conf. Very Large Data Bases, pp. 439-450, Sept. 1994.
14. C. Gniady, A.R. Butt, and Y.C. Hu, “Program Counter Based Pattern Classification in Buffer Caching,” Proc. Sixth Symp. Operating Systems Design and Implementation, pp. 395-408, Dec. 2004.