Page Replacement Policies in Specialized Workloads

Michael Lanthier[1]

Abstract— **As the total amount of memory allocated on a system exceeds the size of physical memory, decisions must be made on what data can be shifted to secondary storage. Operating systems implement page replacement policies to make these eviction decisions, attempting to reduce the rate at which they must go to disk to retrieve pages. Specialized applications like databases have traditionally implemented their own memory management techniques, managing page replacement themselves. In recent years, some databases have given this responsibility back to operating systems rather than managing it themselves. This paper analyzes performance of different page replacement policies under specialized workloads like databases. By simulating page replacement policies, we identify that specialized applications under varying workloads may not have a singular one size fits all policy. Different workloads identify different optimal page replacement policies, forcing these systems to select a best average performing policy. We also identify that policy selection can certainly decrease page fault rates for certain workloads, but at the risk of overfitting performance to that workload.**

I. Introduction

A. The Problem

Operating systems are responsible for managing how memory is allocated within a given system. Processes read and write from memory as they get their necessary work done. In some systems, there comes a time where the amount of memory that processes need exceeds the amount of memory the physical hardware provides. At this point, a decision must be made on what data should physically reside in memory and what data should be written to secondary storage, such as disk.

The decision on what data to evict from memory is a difficult one. For the sake of this paper we will consider memory being managed at the page level. Pages are chunks of data of a predefined size that can be moved in between memory and disk. When a process needs to access a page in memory, but it is not loaded into physical memory, this is called a page fault. If a page fault occurs but physical memory is full, an existing page must be evicted from memory. The page to be evicted is selected by a page replacement policy. These policies aim to reduce the page fault rate, as page faults require reads from disk to fulfill the request.

Page replacement policies are implemented at different levels of the stack. Operating systems must implement page replacement to keep running. If the total amount of memory allocated by processes exceeds that of the system, the OS must start writing pages to disk. Processes may also manage their own memory, paging data in and out of files themselves. Some processes may rely on the OS to manage that paging for them through memory mapped files.

Processes all have different needs and access patterns though. Some may access pages sequentially while others more randomly. Some processes have hotter pages that require more frequent access than others. Operating system must use a policy that provides a good enough page fault rate across all reasonably expected access patterns. These systems often do not know page access patterns apriori. An application managing its own memory may not be bound to that requirement. A specialized application knowledgeable of its expected access pattern could tune its page replacement policy specifically to its access pattern. With DRAM latency in the tens of nanoseconds [1], and NVME based SSD in the scale of hundreds of microseconds [2], lower page fault rates can certainly reduce the amount of I/O waiting that occurs.

How much can a specialized application benefit from its own page replacement policy? Building page replacement policies and complex memory management can be costly. It requires engineering effort, testing, and benchmarking, all to ensure

[1]M. Lanthier is a M.S. student within the Computer Science Engineering Department in the Baskin School of Engineering at the University of California, Santa Cruz.

the system is fast and correct. Does a reduced page fault rate bring enough performance benefit to justify the effort or is the OS good enough?

B. Project Goals

This paper analyzes the page fault rates of a selection of existing page replacement policies under different page access workloads. Page access traces were gathered via synthetic generation as well as from PostgreSQL under load from common database benchmarks. The goal was to analyze general and specialized page replacement policies under different conditions to see how their page faults differ. We aimed to see if those database focused page replacement policies provided consistently better performance under general and database specific access patterns.

The results show that there is no one size fits all page replacement policy even for more specialized workloads like databases. While they are still specialized, different workloads induce enough variation in page accesses that general policies that give best average performance must be chosen. In even more specialized applications with very well defined access patterns, certain page replacement policies can be tuned to squeeze out extra performance at the risk of overfitting to those access patterns.

C. Paper Outline

The rest of the paper is in the following structure. Section II covers background on page replacement policies. Section III details the construction of the simulator and how traces are gathered. Section IV presents the results of the simulator being run on the traces. Section V covers related work. Section VI gives insights into future work. Section VII includes concluding remarks with Section VIII detailing some challenges and learning's.

II. Background

A. Managing Page Replacement

Managing page replacement is a required task of the operating system. Once physical memory fills up, pages must be evicted from memory to make room for new pages. Swap files, also known as page files, are a common technique to manage this. Certain pages in physical memory are written out to a swap file on disk to make room for other pages. Therefore, when a process requests a page in its virtual address space it may not be in physical memory. To satisfy the read, the OS must read the page from the swap file on disk and load it back into physical memory. This is transparent to processes but will result in a slow down due to the kernel needing to read out to disk. Operating systems also provide explicit mechanisms for memory management such as memory mapped files. Using system calls, like $mmap$, processes can map a file into its own virtual address space. It can then read and write to the file as if it was fully in memory. What portions of the file are loaded into memory, flushed to disk, and evicted are all controlled by the operating system. Here the process gives the operating system explicit control over whether portions of the file reside in physical memory or not.

Applications may choose to manage page replacement themselves. Databases traditionally use buffer pools to manage memory under the assumption it can manage its memory better than the operating system [3]. Newer system have challenged the traditional buffer pool technique, relinquishing page replacement decisions to the OS via $mmap$ instead [4]. Some of these systems have since migrated off $mmap$, citing performance reasons, while others continue to use it. MongoDB's storage manager was built using $mmap$ but in recent years has transitioned away from it [4]. On the other side PostgreSQL has always used a buffer pool itself, but does use $mmap$ for other tasks within the database [5]. The traditional view was for databases to manage memory themselves, but modern system designers have challenged that notion. For some, the operating system's memory management abilities is good enough.

B. Page Replacement Policies

Over the years, many different replacement policies have been designed catering to different access patterns. These policies generally track statistics on page accesses and evict pages based on these statistics. These policies generally track how recently a page was accessed, how frequently they are accessed, or a combination of the two.

Recency based policies rely on the intuition that when pages are accessed has a bearing on when they will be used in the future. Least recently used

(LRU) for example will evict the page whose most recent reference is farthest in the past. The intuition here is pages that have not been referenced recently will likely not be referenced for a while. The opposite can also be implemented, using most recently used (MRU). MRU evicts the most recently used page, assuming that it will not be used again soon. A workload with sequential scans where a page will not be read again after it's initial access may benefit best from MRU, while a workload with commonly accessed pages will benefit better from LRU based policies.

Frequency based policies exist as well, tracking how often a page is accessed. In least frequently used (LFU), every page has an associated counter. Whenever that page is requested, the counter is increased by 1. When an eviction occurs, the page in memory with the lowest counter is removed. Frequency based policies will allow for frequently used pages to stay in memory, but struggle to quickly adapt to recently accessed or new pages.

Some policies exist that use both frequency and recency to determine which pages to evict. LRU-K is a policy that was built to approximate frequency while also keeping recently accessed pages in memory. LRU-K is a LRU based policy but it takes into account the last K references of a page. LRU only takes advantage of 1 historical reference, which makes it equivalent to LRU-1. Tracking the last 2 references (LRU-2) is the standard configuration [6]. More frequently accessed pages will benefit by having their tracked reference histories being temporally clustered. Recently selected pages will benefit as it is still an LRU based policy, just selecting with more history. To benefit recency even more, a correlated reference period parameter can be tuned as well. This parameter will default LRU-K to keep the most recently referenced pages in memory. How correlated reference periods work is best read in O'Neil et al[6], but the key concept for this paper is it is a tunable parameter for the policy along with K. A more recently developed policy called adaptive replacement cache (ARC) also takes into account frequency and recency. Unlike LRU-K, there are no parameters to tune.

These policies have been designed and used for a variety of purposes. Least recently used policies have been proposed to manage virtual memory

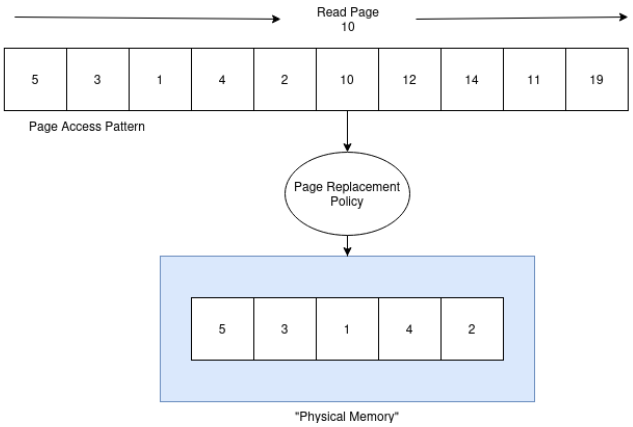

Fig. 1. Simulator architecture with a max memory size of 5

since the 1970s [7] and have been used in early databases as early as the 1980s [3]. They continue to be used today, in a variety of systems. The Linux kernel currently uses a least recently used based policy, maintaining a set of active pages and inactive pages which can be evicted [8]. The more recent policies, while possibly generally applicable, were aimed at database systems. O'Neil et al focus on the benefits of LRU-K's ability to work on a buffer pool [6] while ARC's evaluation focuses on testing a number of database traces [9]. So in practice specialized systems seem to adapt more specialized policies over more general ones that operating systems employ.

III. Simulating Page Replacement

To measure and compare page replacement policy performance, a number of tasks were undertaken. A simulator was constructed which implemented multiple different common page replacement policies. To simulate these policies, a collection of page access traces were collected. Three of them were generated synthetically, while three others were collected from PostgreSQL under different operating conditions. The simulator was run under varying conditions to analyze how the many different page replacement algorithms performed under different workloads and memory sizes.

A. Simulator

A simulator was constructed to model page replacement. Simulation was selected as it allows the page replacement policy performance to be isolated and measured without interference of other

system components. It also reduced the amount of time needed to start experimenting and reduced the total amount of time for running the experiments as no disk reads occur.

The simulator works very similar to how a memory mapped file would. The workload being inputted into the simulator is a list of integer page identifiers that are to be read in the provided order. These page ids would map to different portions of the disk. As the simulator "reads" the pages, it tracks the pages that would be read into memory via a set of bounded size. The total number of pages that can reside in memory at a given moment is configurable and defined prior to running a workload. Eventually the max size of the set is met, denoting that no more pages can fit in memory. For subsequent reads, if a page id does not reside in the set, a page replacement policy must be used to decide which page id to evict. Once a page id is evicted, the currently read page id will be added to the set. As the simulator runs, it internally tracks the number of total reads and how many times evictions occur, i.e. page faults. The page fault rate is calculated ($pagefaultcount/readcount$) at the end of the run.

The simulator expects a few inputs. This includes the total number of pages that can reside in memory, an implemented page replacement policy, and a list of pages that need to be accessed. The list of pages that need to be accessed is a list of integers. A simulator with a total memory size of 5 can be found in Figure 1.

B. Page Replacement Policies

A selection of page replacement policies were written to manage page faults in the simulator. To cover a variety of techniques, a selection of policies that weighed either page frequency, recency, or took both into account were used. Additionally a couple of low overhead page replacement policies were chosen to provide some comparison and baseline performance. There are many policies that exist and used in practice. Due to the limited time bounds of this project, policies that were relatively simple to implement took priority over more complex ones. Certain replacement policies also have multiple different implementations. Linux, MySQL, and the LRU-K paper all have differing implementations of LRU under the hood. For efficiency, simple implementations like the LRU-K implementation were selected.

1) Recency Based Policies: Both LRU and MRU were implemented as recency based policies. LRU is equivalent to LRU-1, which is what it is referenced as for the rest of this paper. A first in first out (FIFO) policy was also implemented. In this case a queue of page ids is maintained. When a page fault occurs, the new page is placed on the back of the queue. The page id at the head of the queue is removed and evicted from memory. This results in the removal of pages that have been in memory the longest. FIFO is very efficient but is described as having poor performance in practice.

2) Frequency Based Policies: LFU was implemented as the frequency based policy. A counter is kept for every page id that is encountered. Whenever a read occurs to a page, the counter for that page is incremented. When eviction occurs, the page id in memory with the lowest counter is evicted.

3) Other policies: LRU-K was the policy implemented to use both frequency and recency metrics. LRU-2 and LRU-3 were used in the simulator with varying correlated reference periods. Throughout this paper, the correlated reference period is generally omitted when referring to policies. When included, LRU-K with a $K = 2$ and the correlated reference period of 20 is denoted as LRU2-20.

To provide a baseline of performance, a random page replacement policy was built. During a page fault, every page has a uniformly random chance of getting evicted. Random replacement is efficient and reduces the need to maintain statistics about page access frequency and time.

C. Workloads

To properly stress the page replacement policies, different page access patterns must used. A set of synthetic page access patterns were created simulating scans, random access, and hot pages. Additionally page access traces were harvested from PostgreSQL under load from common database benchmarks.

1) Synthetic Workloads: Three different synthetic page access patterns were generated to simulate scans, random access, and normal distribution

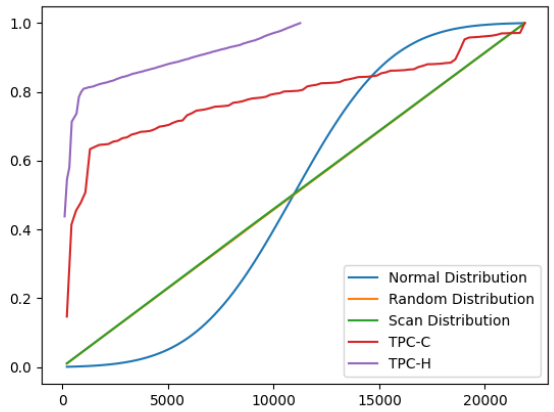

Fig. 2. Cumulative Distribution Functions of pages being accessed. Synthetic workloads are normalized to the size of the TPC-C benchmark for visualization. The x-axis are page ids and the y-axis is the probability of access.

access. The goal here was to provide a baseline for performance using access patterns that could exist on any system. All synthetic workloads generated a total of 10,000 read requests. The working set size was 1,000 pages, with the frequency of each page being accessed dependent on the synthetic workload. Scans are an in order linear scan of the full working set. Once all pages were scanned, the scan will restart at the zero'th page. The probability of a page being accessed in the random access workload was equal across all pages. The probability of accessing a page for the normal distribution access workload followed a normal distribution. The least and greatest page ids were within 3 standard deviations of the median page id. It was difficult to model hot pages as different papers used different distributions and parameters defining how likely pages would be accessed. A normal distribution was used for its simplicity and to represent some form of more likely access for certain pages.

The cumulative distribution functions of the benchmarks can be found in Figure 2. The random and scan workloads have equal distributions, but the order in which pages are accessed differs creating different behavior in the page replacement policies. Here the distribution functions were normalized to the number of reads and number of unique pages of one of the real world benchmarks for readability and comparison.

2) PostgreSQL Traces: To measure real world performance in specialized workloads, traces were extracted from PostgreSQL under different conditions. PostgreSQL includes probes which allow users to receive traces on the inner workings of the system [10]. A built in probe called the buffer-read-start probe was used to output the page ids of the buffer that PostgreSQL was accessing. A tracing software called SystemTap was used to read out these traces and output them to a file during the time PostgreSQL was under test. This outputted line delineated files with the order in which pages were accessed by PostgreSQL.

These PostgreSQL buffer access traces were gathered using two different benchmarks, TPC-C and TPC-H. TPC-C is an industry standard benchmark used to measure database OLTP performance[11]. It has a mixture of read only and update transactions modeling a business managing sales across multiple warehouses. Transactions tend to be smaller, interacting with disjoint subsets of the rows across multiple tables. TPC-H is another industry standard benchmark, but focuses on OLAP performance [12]. It runs a set of complex queries that are designed to answer business oriented questions. These are exclusively reads, accessing many rows across many tables with aggregate operators. Both benchmarks were run using BenchBase, an open source database benchmarking tool [13].

These benchmarks were chosen as they cover the two common use cases and access patterns of database systems. Systems like PostgreSQL are generally known for their OLTP performance, but still must perform well under OLAP workloads. The number of rows TPC-C accesses is relatively small at any given time and may change over time as different transactions come in. This stresses a policies ability to adapt to new data as it comes in, and lots of requests to system tables. TPC-H queries on the other hand needs to touch larger ranges of data at a given moment. This requires the page replacement policies to run evictions on recently loaded into memory data, and potentially evict commonly used tables when large queries are encountered.

PostgreSQL buffer access traces were gathered

twice using TPC-C and once using TPC-H. A low concurrency TPC-C trace was collected running 1 thread against PostgreSQL for a total of 60 seconds under the serializable isolation level. A higher concurrency TPC-C trace was also collected running 4 threads against PostgreSQL for a total of 20 seconds under the read committed isolation level. The low concurrency test logged 6.4 million page accesses while the higher concurrency test logged almost 5 million page accesses. Both tests accessed roughly 25,000 unique pages. A single trace was gathered while running TPC-H. The benchmark ran the 22 different transactions defined in the TPC-H specification. Roughly 11,000 page reads occurred to 9,000 unique pages. This benchmark was not bounded by time but happened to run under roughly 30 seconds.

The cumulative distribution functions in Figure 2 show a much different page access pattern relative to the synthetic benchmarks. Pages 1 through 10 had a vast majority of the accesses as they are system pages tracked by PostgreSQL. These are consulted when PostgreSQL needs to figure out which pages data resides on. In TPC-C you can also see certain hot pages denoted by steeper slopes in the cumulative distribution function. TPC-H does not have any clearly hot pages outside of pages backing system tables, showing a fairly smooth and steady increase.

3) Source Code: The simulator code, data, and PostgreSQL trace configurations are all available on Github.

IV. Results

The simulator was run against the 3 synthetic and 3 PostgreSQL traces. Each trace ran multiple times against differently configured simulators. They were configured with varying total memory sizes from 50 pages up to 450 pages that could fit in memory. Different page replacement policies were used for each run. LRU-K was run multiple times with different K sizes (1,2,3) and varying correlated reference periods as well (0 to 1,000). Synthetic traces ran on 10,000 page accesses while PostgreSQL traces were limited to a total of 2 million reads. The 6.4 million reads generated by TPC-C took a prohibitively long time to simulate replacement, especially when running multiple page replacement policies with varying memory sizes. Page fault rate was measured for comparison across all tests. The high concurrency and low concurrency TPC-C traces generated identitical results, so analysis only occurs on the low concurrency traces.

A. Synthetic Traces

The synthetic traces showed the cleanest and most consistent results across the page replacement policies. The random access scans test showed little differentiation between the policies under truly random access. In the smaller memory size tests the difference between the best and worst performers was a 1% page fault rate. In the least constrained tests, this difference shrunk to a half a percent. There was no clear differentiation between any policy's performance relative to one another.

The normal distribution tests is where differentiation between the page fault rates shows. The purely frequency based policy, LFU, performs the best as it is able to track the normal access distribution perfectly (Figure 3). This is to be expected as LFU tracks access frequency which is never reset. The LRU based policies perform the next best. LRU-3 performs the best as it holds onto the most history, allowing it to better index on the frequency pages arrive at. LRU-2 and LRU-1 have less history, hence performing slightly worse.

On the flip side, the synthetic scan tests shows the worst case scenario for many of these policies (Figure 4). The intuitive assumption that recently used data should stay in memory is thrown out the window. In these scans, a page will not be read again until all other pages have been read. This results in MRU being the only policy that is able to properly adapt to the pure scan access pattern. Random page replacement's random removal allows for a reduction in page faults as more recently accessed pages are just as likely to get evicted as other pages. Interestingly LRU-3 is able to provide a slight reduction of page faults, even doing better than random replacement at lower memory sizes. It is not immediately clear why the additional history LRU-3 tracks provides enough to reduce page fault rate while LRU-2 is unable to provide any benefit.

These synthetic workloads re-enforce that certain policies will do better under specific workloads but may not generalize to others. For work-

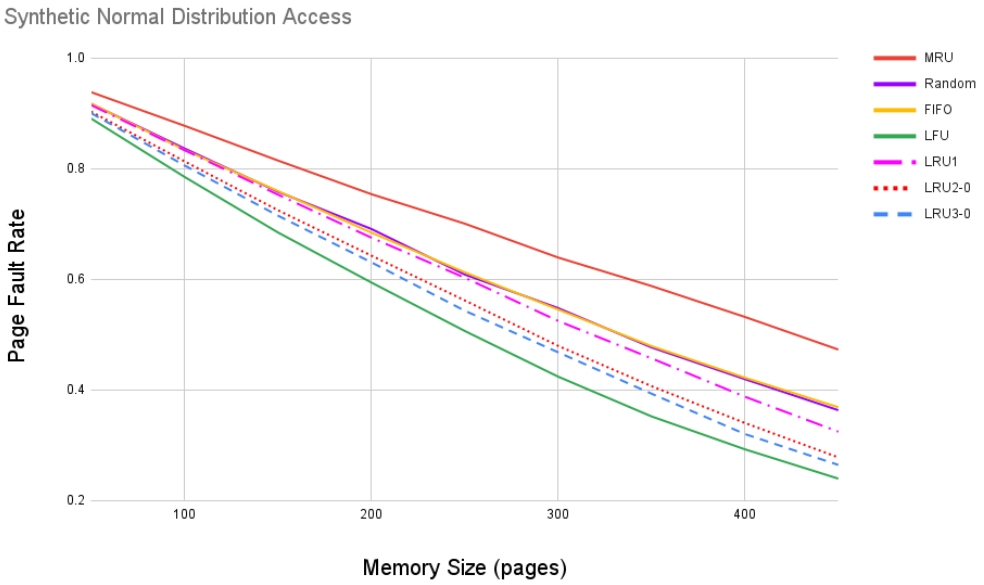

Fig. 3. Page fault rate at varying memory sizes when the probability of accessing a page follows a normal distribution.

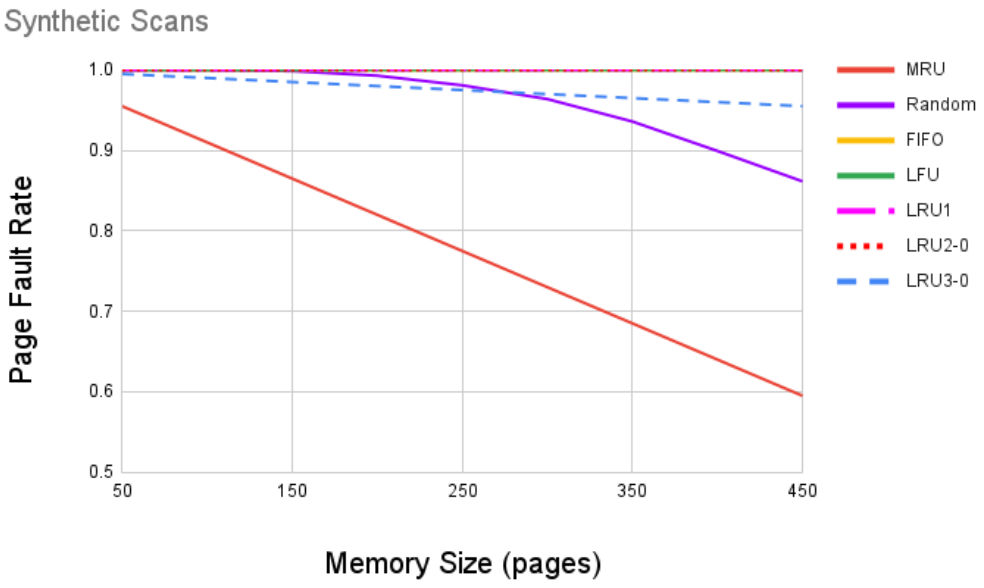

Fig. 4. Page fault rate at varying memory sizes during sequential scans.

loads that aggressively scan and do not re-read pages for a long time, if ever, MRU may provide significant benefits. But if the access pattern adapts to a different distribution, it may do much worse in the average case. Selecting LRU-3 may give you good average performance, but too many scans may defeat its good overall benefits. Selecting one giving best performance can only be done knowing the access pattern apriori. This is something that an operating system does not have the privilege of knowing, but a specialized application may.

B. *PostgreSQL Traces*

When run through the simulator, the PostgreSQL traces provided varying results. Across

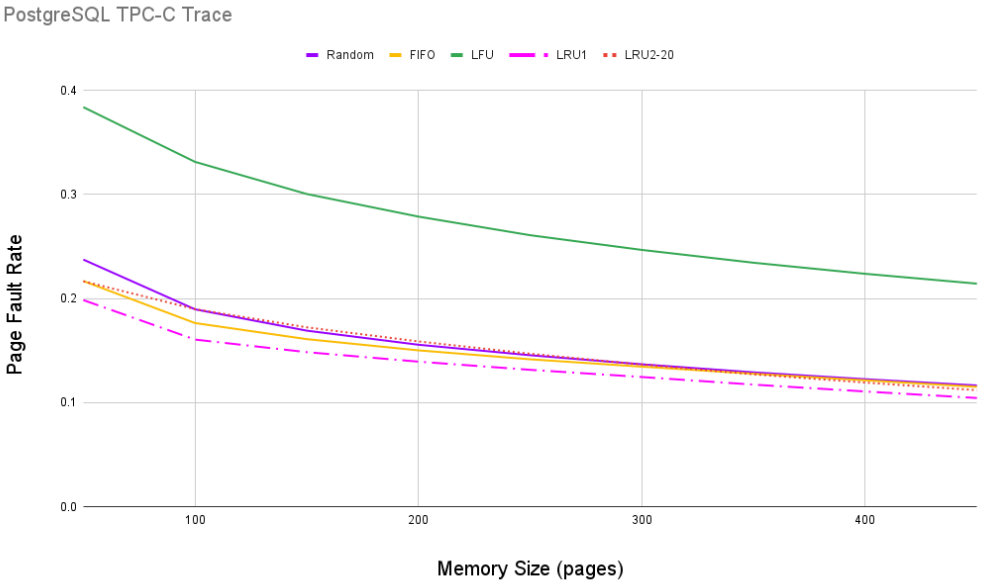

Fig. 5. Page fault rate at varying memory sizes with the low concurrency TPC-C benchmark.

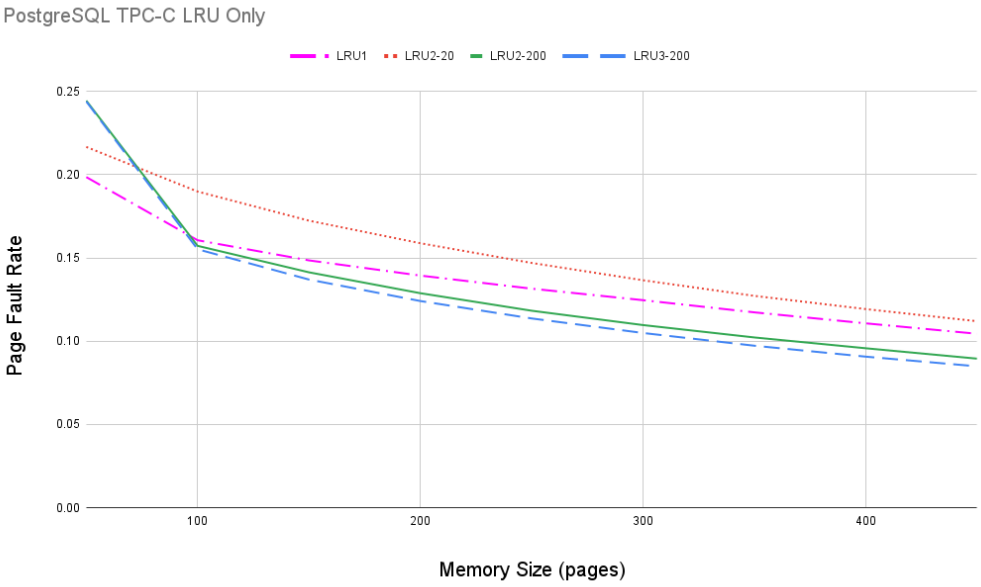

Fig. 6. LRU Page fault rate at varying memory sizes with the low concurrency TPC-C benchmark.

all PostgreSQL benchmarks, the pure frequency tracking employed by LFU fell apart. In the low concurrency TPC-C trace (Figure 5) LFU performs significantly worse than all other page replacement policies. This is due to it being unable to adapt to changing data access patterns over time. In LFU the page access frequencies are never reset. TPC-C queries tend to be on disjoint sets of records, which it fails to adapt to. It struggles as well in TPC-H (Figure 7), where each subsequent query touches different tables.

LRU-K running against the PostgreSQL traces provides results inconsistent with the synthetic benchmarks. In the low concurrency TPC-C trace (Figure 5) LRU-1 is the top performer, rather than

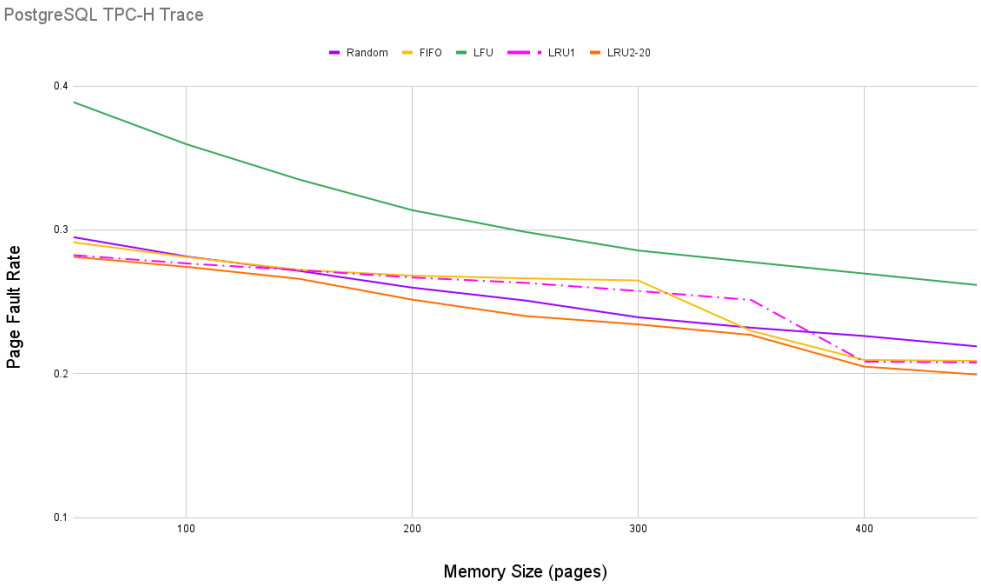

Fig. 7. Page fault rate at varying memory sizes with the TPC-H benchmark.

LRU-2 which tracks more page access history. LRU-2 even provides worse performance than random and FIFO until larger page sizes. LRU-3 is not graphed as it worse than LRU-2. The tables flip in TPC-H though (Figure 7) with LRU-2 providing better performance and LRU-1 performing similar to random and FIFO. The difference between LRU-2 and LRU-1 in both benchmarks is roughly 1%.

There was no specific LRU-K configuration that provided optimal page replacement for both TPC-C and TPC-H workloads. The tunability of LRU-K did allow for instances where tuning parameters could squeeze out lower page fault rates. Figure 6 graphs LRU-K performance under different configurations against the low concurrency TPC-C trace. LRU-1 was the top performing generally tuned policy, but by tuning the correlated reference periods for LRU-2 and LRU-3 to 200, the page fault rate was reduced by 1.5% to 2%. While these are some nice performance gains under TPC-C in this context, they are not generalizable to other workloads. LRU-2 and LRU-3 with a correlated reference period of 200 are not graphed in any of the other benchmarks because they perform worse then their LRU-K counterparts with correlated reference periods that worked well consistently across all workloads. This shows the risk of overfitting when trying to get the most performance out of these policies. As workloads change, the performance of the page replacement policy may degrade at an unacceptable rate.

While PostgreSQL may be specialized to data management, it still needs to perform well across many cases. Selecting a page replacement policy based on maximizing TPC-C performance may work great for some, but if more TPC-H like queries begin to arrive, that performance benefit may go away. Choosing a generally well performing algorithm like LRU-1 or LRU-2 with correlation periods tuned to general performance seems to be the only option. PostgreSQL may not get the best performance, but in most configurations and queries it will perform well. Adding environment settings to allow for tuning these algorithms could be one way to give more control to administrators. This does have the same overfitting risk, the responsibility is just passed to the database admin.

C. Quantifying Results

But what is the performance difference between a top performing policy here and a worse performing policy? In the TPC-C low concurrency benchmark, the average difference between LRU-1's performance and the next best policy is a 1% higher page fault rate. If the SSD latency was

50 microseconds [2], and DRAM latency is 100 nanoseconds[1], the read penalty of going out to the SSD is 49.9 microseconds. Over the 6.4 million read's, this 1% higher rate adds up to 3.2 seconds just waiting for the SSD to start sending bytes back. Over the 25,000 transactions this adds up to 0.13 milliseconds per transaction, a small blip to a client sending in a query. Will a better page replacement policy give back 3 seconds worth of work in this situation? Probably not, as applications can take advantage of asynchronous I/O operations and batching reads to reduce these read penalties. But this does illustrate the fact that these smaller percentage points do matter and can add up to large performance improvements especially if optimization techniques to mask slower read times cannot be used. $mmap$, for example, does not know the file access pattern apriori and cannot perform these optimizations. On the other hand, a database doing a table scan does know the next pages that need to be accessed and can exploit that for performance gains. Holding onto the responsibility of managing memory can therefore bring twofold performance benefits, reduced page fault rates and further I/O optimization.

V. Related Work

The original inspiration for this project comes from Crotty et al's paper covering whether mmap should be used for database system memory management rather than traditional buffer pools [4]. They conclude that mmap has performance limitations that systems will run into and result in additional complexity to provide the consistency required for these systems. The paper does not cover page replacement policies, focusing on the existence of I/O stalls, the lack of asynchronous I/O as well as performance issues relating to TLB shootdowns. This turned the focus of this paper to page replacement, as it seemed like an area that could extract more performance. The evaluation of the PostgreSQL benchmarks here highlight why page replacement was not necessarily a target of the authors in that paper. There was not a standout best performer for the different workloads that were presented, even in isolation looking at just the traces extracted from PostgreSQL. While databases are specialized, they still need to support a wide variety of page access patterns. Any extreme performance benefits gained tuning to one workload may be lost in another, so a generally good policy needs to be selected. An outstanding question remains though, do asynchronous I/O and intelligently batching reads dwarf the performance improvements of improved page replacement policies?

There have been other assessments of page replacement policies as well. Swain et al. benchmarked page replacement techniques with varying cache sizes in a 2011 paper [14]. They do not describe the traces that they run their simulations on nor details the probability of data being accessed. Their results are consistent with the performance results here. Simple policies like FIFO and Random still perform fairly well while modern techniques like LRU-K tend to do a bit better. They do show additional policies like ARC and optimal which perform better than FIFO, random, and LRU-K.

There has also been analysis around the LRU-K algorithm that differ from the findings presented here. O'Neil et al. analyzed performance in their original proposal of the LRU-K page replacement policy [6]. They run two synthetic benchmarks which show LRU-2 performs better than LRU-1, consistent with the normal distribution synthetic benchmark run here. They also run their benchmarks against a page access trace extracted from a CODASYL database under a workload representing that of a bank. They show consistently better performance in LRU-2 relative to LRU-1 across all buffer pool sizes. A set of Masters students at the University of Wisconsin repeated these findings in 2013 [15]. To reproduce the real world workload, they used PostgreSQL. Traces were extracted by modifying source files to output page ids and pg_bench, which simulates a banking workload too, was used when gathering a page access trace. These newer results were consistent with the original findings of O'Neil et al, LRU-2 performing consistently better than LRU-1. In this paper PostgreSQL traces did not show LRU-2 as the consistent better performer. It is possible that the way PostgreSQL accesses pages is no longer consistent with the access patterns of the prior results. CODASYL followed the network model rather than relational model which has seen

widespread adoption. PostgreSQL also has seen a fair amount of development in the last 12 years. Additionally both pg_bench and the original CODASYL traces use a bank style workload. TPC used to provide a bank style workload but has since argued that they are too simplistic to evaluate the realities of complex database workloads [16]. Running this analysis using pg_bench could shed further insight into these inconsistencies.

There were few if any available page replacement policy simulators that were built to test page replacement policies at the page level. Both the University of Washington [17] and University of Michigan [18] provide cache replacement simulators for CPU caches. That seems to be the focus of many existing simulators. There is a body of work covering optimal page replacement policy simulators though. Based on system traces these simulators are able to simulate the optimal page replacement policy efficiently. An optimal page replacement policy evicts the page that is to be referenced farthest in the future. This is used to measure performance of page replacement policies relative to an optimal baseline. This shows how much room for improvement can be attained by better cache management. ScaleOPT was built in 2019 as an efficient way of running the optimal page replacement policies [19] and built off of other optimal replacement policy simulators. Given more time it would have been interesting to include the optimal policy in the results.

VI. Future Work

Additional work measuring different page replacement policies under different conditions may be able to provide a clearer indication of whether specialized replacement policies thrive in systems like databases. Implementing the Linux based LRU page replacement policy would allow for a clearer baseline to compare other policies against. Also implementing other modern and more used algorithms like ARC and clock sweep may show more differentiation from the more simple algorithms like LRU, Random, and FIFO which were measured here. Real world traces from other processes also would provide another point of comparison and baseline, showing other situations that operating systems need to generalize too. Running pg_bench as well would have been nice. This would allow for these algorithms to be compared against previous research.

A future area of exploration that was outside the scope of this work is trying to quantify how much techniques like asynchronous I/O operations and batching reads can improve application performance. While page replacement is certainly a part of applications managing memory, efficiently scheduling I/O operations can certainly reduce total amount of time reading from secondary storage. Is development time better spent optimizing how to interact with I/O rather than when?

VII. Conclusion

The benchmarks run here re-enforce the difficulty that page replacement policies have to deal with when built for general and even more specialized systems. Through all experiments there was not a stand out policy that performed best. Even within the PostgreSQL trace tests, there was not a singularly configured policy that did best. Each workload showed a page replacement policy that was more optimal. While PostgreSQL is certainly a specialized application, it still needs to support a variety of access patterns. Any performance benefits gained fitting to one workload may be lost in another. So squeezing extra performance out of page replacement policies while not overfitting to specific workloads becomes a difficult task. Similar to operating systems they must find a general policy that works for a variety of page access patterns.

For very specialized workloads with known access patterns, there certainly is room to tune page replacement policies to get as small of a page fault rate as possible. For example scan based workloads exploiting a MRU based policy can yield extreme benefits. This requires extensive testing of representative workloads and understanding of how memory constrained the application will be. While this is certainly possible, it may not be an engineering effort that is worth going through. Other benefits can be gained via other techniques like asynchronous I/O and batching I/O operations. These may provide greater benefits than optimizing page replacement.

So are page replacement policies an area that specialized applications can extract performance

from? Yes, but with some risks. Static unchanging access patterns can benefit from highly configured page replacement policies. Applications with changing workloads, even if specific to certain use cases, will need to temper these expectations and use a good enough policy. This may be via the operating system, but exploiting other gains like batching and asynchronous work may be lost.

VIII. Challenges and Learning

A. Building Open Source Software

In order to extract page access traces from PostgreSQL, it had to be built from source with a flag enabling the tracing utilities. SystemTap, the tracing software, also needed to be built from source and needed access to specific kernel header files to compile. It took a full day of trial an error to get a build of PostgreSQL and SystemTap to spit out the traces. While PostgreSQL is very popular, the number of folks actually building it from source is limited. This meant there was little detail online about getting around certain failures during the build process.

B. Poorly Optimized Policies Are Problematic

The runtime of a page replacement policy can be hugely problematic. All the page replacement policies were implemented in a naive fashion. Not much time was spent to reduce the runtime of deciding which page to evict. This became an issue when running page replacement policies on 6.4 million page access requests. It very much shed the light on why estimations of LRU or LFU are often used, as properly computing an accurate answer can take a while. This made me better understand why some of these papers went to great lengths to show how fast and low memory footprint their policies were.

References

[1] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary DRAM architectures," in *Proceedings of the 26th annual international symposium on Computer architecture*, ser. ISCA '99, USA: IEEE Computer Society, May 1, 1999, pp. 222–233, ISBN: 978-0-7695-0170-3. DOI: 10.1145/300979.300998. [Online]. Available: https://dl.acm.org/doi/10.1145/300979.300998.

[2] G. Haas and V. Leis, "What modern NVMe storage can do, and how to exploit it: High-performance i/o for high-performance storage engines," *Proc. VLDB Endow.*, vol. 16, no. 9, pp. 2090–2102, May 1, 2023, ISSN: 2150-8097. DOI: 10.14778/3598581.3598584. [Online]. Available: https://dl.acm.org/doi/10.14778/3598581.3598584.

[3] I. L. Traiger, "Virtual memory management for database systems," *SIGOPS Oper. Syst. Rev.*, vol. 16, no. 4, pp. 26–48, Oct. 1, 1982, ISSN: 0163-5980. DOI: 10.1145/850726.850729. [Online]. Available: https://dl.acm.org/doi/10.1145/850726.850729.

[4] A. Crotty, A. Pavlo, and V. Leis, "Are you sure you want to use MMAP in your database management system?" In *12th Annual Conference on Innovative Data Systems Research (CIDR '22)*., 2022.

[5] "PostgreSQL resource consumption," PostgreSQL Documentation, Accessed: Nov. 19, 2025. [Online]. Available: https://www.postgresql.org/docs/18/runtime-config-resource.html.

[6] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-k page replacement algorithm for database disk buffering," *SIGMOD Rec.*, vol. 22, no. 2, pp. 297–306, Jun. 1, 1993, ISSN: 0163-5808. DOI: 10.1145/170036.170081. [Online]. Available: https://dl.acm.org/doi/10.1145/170036.170081.

[7] P. J. Denning, "Virtual memory," *ACM Comput. Surv.*, vol. 2, no. 3, pp. 153–189, Sep. 1, 1970, ISSN: 0360-0300. DOI: 10.1145/356571.356573. [Online]. Available: https://dl.acm.org/doi/10.1145/356571.356573.

[8] M. Gorman, *Understanding The Linux Virtual Memory Manager*. 2007. [Online]. Available: https://www.kernel.org/doc/gorman/html/understand/understand013.html.

[9] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proceedings of FAST '03: 2nd USENIX Conference on File and Stor-*

age Technologies, 2003. [Online]. Available: https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache.

[10] "PostgreSQL dynamic tracing," PostgreSQL Documentation, Accessed: Nov. 27, 2025. [Online]. Available: https://www.postgresql.org/docs/18/dynamic-trace.html.

[11] Transaction Processing Performance Council (TPC), *TPC-c specification*, version 5.11.0, Feb. 2010. [Online]. Available: https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[12] Transaction Processing Performance Council (TPC), *TPC-h specification*, version 3.0.1, 2022. [Online]. Available: https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf.

[13] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "OLTP-bench: An extensible testbed for benchmarking relational databases," *Proceedings of the VLDB Endowment*, vol. 7, no. 4, pp. 277–288, Dec. 2013, ISSN: 2150-8097. DOI: 10.14778/2732240.2732246. [Online]. Available: https://dl.acm.org/doi/10.14778/2732240.2732246.

[14] D. Swain, B. N. Dash, and D. Swain, "Analysis and predictability of page replacement techniques towards optimized performance," in *International Conference on Recent Trends in Information Technology and Computer Science (IRCTITCS)*, 2011.

[15] V. Bittorf, I. Canadi, L. Li, and S. Pollen, *Revisiting LRU-k page replacement algorithm performance experiments*, Apr. 21, 2013. Accessed: Nov. 19, 2025. [Online]. Available: https://github.com/igorcanadi/lru-k/blob/master/paper/paper.pdf.

[16] C. Levine, "Why TPC-a and TPC-b are obsolete," in *Digest of Papers. Compcon Spring*, San Francisco, CA, USA: IEEE Comput. Soc. Press, 1993, pp. 215–221, ISBN: 978-0-8186-3400-0. DOI: 10.1109/CMPCON.1993.289669. [Online]. Available: http://ieeexplore.ieee.org/document/289669/.

[17] "UW cache simulator," Accessed: Dec. 7, 2025. [Online]. Available: https://courses.cs.washington.edu/courses/cse351/cachesim/.

[18] "U michigan cache simulator," Accessed: Dec. 7, 2025. [Online]. Available: https://vhosts.eecs.umich.edu/370simulators/cache/simulator.html.

[19] H. Han, S. Lee, and Y. Son, "ScaleOPT: A scalable optimal page replacement policy simulator," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 8, no. 3, 44:1–44:25, Dec. 13, 2024. DOI: 10.1145/3700426. [Online]. Available: https://dl.acm.org/doi/10.1145/3700426.