Implementing a DBMS Bufferpool Manager with Varied Page Replacement Policies

Samir Farhat Dominguez
Boston University
Boston, MA
safarhat@bu.edu

Yuxin Li
Boston University
Boston, MA
yuxinli@umich.edu

Stephany Yipchoy
Boston University
Boston, MA
syipchoy@bu.edu

ABSTRACT

Within the context of Database Management Systems (DBMS), it is often impossible for a bufferpool to fit the entirety of a workload's prospective page requests, let alone be populated exactly with those pages. As such, page replacement and eviction becomes a necessary part of any DBMS systems employing a bufferpool. There are several different page replacement policies that bring their own benefits and drawbacks in terms of performance. The selection of a page eviction policy is a tremendously nuanced issue, as the degree of these benefits and drawbacks are influenced by factors such as the DBMS system being used, the workload of an application, the machine a DBMS system is running on, the physical structure of the memory hierarchy members, and many other aspects. This work will seek to focus on page replacement policies, performing a comparative analysis under a diversity of relevant parameters- focusing on different read and write ratios, and skew percentage.

1 INTRODUCTION

One of the central paradigms in computer architecture is the notion of memory hierarchies with decreasing capacities and increasing access times. Within this paradigm, a critical subset of computing problems arise. These are: the selection of data structures maintained in smaller but faster memory, the algorithms maintaining these data structures, and the policies deciding how to discard data from this higher value memory. One such proposition to address the former two, is a bufferpool and its respective insert, delete and read algorithms. Bufferpools are constituted by a series of frames, that can be either occupied by a page from the disk, or not.[1]

1.1 Motivation

A bufferpool is often used in DBMS systems, and occupies a portion of main memory. The bufferpool acts as a cache for table and index data on the disk, which is quite large but tremendously slow in terms of read/write times.[10]

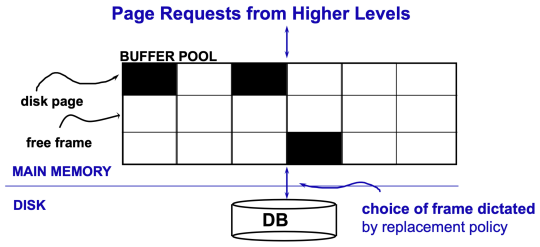

Figure 1: Depiction of a bufferpool

Therefore, optimal performance with a bufferpool can be achieved such that when an application requests a page from the disk, the bufferpool contains this page, a page hit takes place, and the page is read with the faster main memory access time. If a page miss does occur, then the DBMS is forced to go down in the hierarchy, and find the requested page from the disk, which is far slower to access. This page is then written to the bufferpool, either by populating an empty frame, or replacing a page contained in a populated frame. If a replacement does occur, bufferpools maintain a 'dirty' bit, indicating if the page has been modified since it was brought in from disk. If it was, the disk must be written to with the updated value, requiring another very slow access and write procedure. If it is not dirty, then it is simply discarded from the bufferpool.[5]

When working with a DBMS, there exists a persistent goal to make your workload faster and more efficient. Examining the bufferpool and its page replacement policy is an essential place to begin on this journey because it determines if you will be able to immediately access a page or have to begin the slow journey down the memory hierarchy.

1.2 Problem Statement

Analyzing the performance of a general workload under four different bufferpool page replacement policies (FIFO, LRU, CFLRU, and LRU-WSR) will give us an idea of which to choose when speedily trying to increase the efficiency of a DBMS, and give a bouncing off point for more research on how changing the bufferpool page replacement policy leads to performance gains. We have implemented LRU, CFLRU,

LRU-WSE and FIFO replacement polices. Our program simulates disk management (read / write pages in disk) along with the bufferpool in order to accurately test and compare the performances of implemented replacement policies.

1.3 Contributions

The work has been divided into two sections: implementation of the bufferpool and four replacement policies (LRU, CFLRU, LRU-WSR and FIFO), and testing and comparison of the performance of the DBMS bufferpool manager with different parameters.

We began by implementing a basic bufferpool, which holds a queue of vectors; the vectors very simply just hold a page id and a dirty bit. We proceeded to set up the basic functionality of the bufferpool: creating functions to search for a page id, processing read requests, and processing write requests. Next, we implemented the actual page replacement policies that will be used to measure performance. These are elaborated on in the next two sections of the work, but just to enumerate these we have LRU, FIFO, CFLRU, and LRU-WSR. With these page replacement policies we include tracking of relevant statistics, most notably timing and I/O procedures. We also implement a simulation for disk behavior, by performing reads and writes on a file throughout the program.

Having implemented the above, we sought to consolidate our testing to perform a comparative analysis of these page replacement policies. As such, we developed a testing suite, where we are able to conduct a full barrage of tests within parameters of interest. We also add functionality to isolate page replacement policies and test them under the other permutations of parameters, furthering our understanding of the strengths and weaknesses of these. Finally, in our testing suite, we implement a graphs and plotting engine to eventually present relevant results in Section 5.

2 BACKGROUND

The physical structure and logic of the bufferpool is quite straightforward. This work focuses on how the choice of page replacement policy drives the performance of the bufferpool, which is why implementing different policies is a major component of this project. We will be implementing the LRU, FIFO, CFLRU, and LRU-WSR policies.

LRU is the most frequently used replacement policy- it is based on the assumption that a page that was more frequently referenced is more likely to be referenced again than a page that was referenced longer ago. Ergo, temporal locality is its principal ethos. For the implementation of this page replacement policy, a list is maintained which is sorted by how recently a page was accessed, and chooses the page that was accessed the longest time ago to be removed from the cache to make room for another page.[11]

FIFO, or First In First Out, is known as a more naive example of bufferpool page replacement policies, where a list is maintained of pages accessed, and each time a new page is accessed it is simply appended to the back of the list. When it is time to remove a page from the bufferpool, the first page in the list is removed.[4]

CFLRU is a page replacement policy that has been optimized for flash based storage, which is employed for storage devices like SSDs that use nonvolatile memory to read and write data. It is an extension of the LRU policy in that it divides the list maintained in LRU into two parts: working and clean first. The working part contains pages that have been recently accessed, while the clean first part contains clean, or unaltered, pages. CFLRU works by first removing pages from the clean first part, and once all those pages have been removed, it begins using the basic LRU algorithm to remove dirty, or updated pages. The purpose of this order is to minimize the number of writes that will be done to the pages.[12]

LRU-WSR, or LRU with Write Sequence Reordering, is similar to CFLRU in that it is also optimized for flash based storage, and is also an extension of the LRU page replacement policy. It works by prioritizing replacing all dirty pages last, meaning that an altered page that has not been accessed recently will remain in the bufferpool for longer than a clean page that was recently accessed. This works by maintaining a cold flag on the list of pages, which is reset when a page is accessed. If a dirty page's cold flag is off, it is not a candidate for page replacement, while all clean pages are evicted.[9]

3 ARCHITECTURE

We implement a bufferpool simulation system consisting of the following header and corresponding `.cc` files.

(1) `buffmanager.cc`: This is where our main is found. We implement the parsing of arguments, progression of the simulation, building the workload, and running the bufferpool simulation while tracking statistics.
(2) `parameter.cc and parameter.h`: This defines and implements the `Simulation_Environment` class. Where the parameters are managed and shared across the program.
(3) `workload_generator.cc and workload_generator.h`: The workload generator files define and implement the workload generation function, where pageIDs and operation types(Read or Write) are defined.
(4) `executor.cc and executor.h`: The executor files represent the bulk of the implementation. Defining both the `Buffer` and `WorkloadExecutor` classes, which

handle everything related to instruction handling for the disk and Bufferpool.

All of the above are elaborated upon in this section.

3.1 Program Flow and Externals

There are separate execution scripts that we have written in `Python`, nominally to run tests, collect data, and process it. However, when it comes to the core bufferpool simulation this has been fully implemented in `C++`. We begin with the initial command with non-specific parameters to correctly run the program. This is `./buffermanager -b -n -x -e -a -s -d -k -w`. `buffermanager` is the resulting executable from the `make` command defined in the `Makefile`. The parameters are defined in the `parameter.h` and defaults set in `parameter.cc` as part of the `Simulation_Environment` class. These parameters are as follows:

(1) `-b`: The size of the bufferpool in regards to page count.
(2) `-n`: The size of the disk in terms of pages, which will always be larger than the size of the bufferpool.
(3) `-x`: The total number of operations, which will be either reads or writes.
(4) `-e`: The read and complement of write percentage distribution of operations.
(5) `-a`: The page replacement replacement algorithm for the simulation run.
(6) `-s`: The skewed distribution of operations on data.
(7) `-d`: The skewed data percentage which defines the distribution of operations on data, which defines the concentration of page collisions.
(8) `-k`: This is our own implemented parameter relevant to disk handling, and is the number of bytes per page. This parameter alongside the `-n` define the size of the disk.
(9) `-w`: This is an optional parameter entirely for our implementation of the CFLRU window size. It sets the percentage of the buffer which is part of the CFLRU window, which we elaborate in 3.7.3.

Once we issue the command we can elaborate on program flow by discussing `buffermanager.cc`, where our main function is found. We begin by instantiating the environment with the relevant parameters, and perform some print statements as a sanity check. We then load our `WorkloadGenerator` object and generate the respective workload, which is a series of instruction types and page IDs. Finally we run the workload, where we read the workload generated. We open and populate the disks, and then run the read and write operations until its terminated. Throughout the logic implemented in the executor files is running to do the appropriate sequence of actions. Finally we write statistics to the appropriate files and standard out as a sanity check and overview of the run.

3.2 Bufferpool Structure

A buffer pool is an area of main memory that has been allocated by the database manager for the purpose of caching data as it is read from disk. Therefore, the database bufferpool holds blocks of data retrieved from the disk. The bufferpool structure is a vector. Each member of the vector represents a page. Pages in the bufferpool can be either in-use or not, and they can be dirty or clean. While the data structure might be different based on different replacement policies, the essential data stored for each page in the bufferpool contains a page id; and a dirty bit, a value to denote if the page is dirty or not. Bufferpool has a fixed size and therefore needs different replacement policies to maintain fewer I/O operations result, which means faster access to stored data.

3.3 Search

The search function uses the page id to perform a linear search in the bufferpool. Since every page id is unique, the function will return the position index of this page in the bufferpool if it found a match and increase buffer hit index by 1. Otherwise it returns -1 and increases buffer miss index by 1.

3.4 Bufferpool update: Read

If the instruction is "R", The read function will be called with the page id and the chosen replacement function. We will first search if the page id is in the bufferpool and return it's position index. If the bufferpool is empty or no match id can be found return -1.

(1) If the page id is found in the bufferpool, the data in that page will be directly accessed from the bufferpool. And therefore the function will only be updating the LRU list by using an iterator to search the page id in the LRU list, delete the matching element from the LRU list and append a new one with the same page id at the front.

(2) If the page id is not found and the bufferpool is not full yet, we read the page from disk into the bufferpool, mark the page clean and increase the read index by 1. We then update the LRU list by simply append the new page id the front of the list which is the position of MRU.

(3) If the page id is not found and the bufferpool is full, we proceed to call replacement functions based on chosen policies (LRU, CFLRU, LRU-WSE and FIFO) to get the position of eviction target in the bufferpool. After we found the target page to be evicted, we also need to check the dirtyBit of the target page. If the target page has been marked dirty, we write this page into the disk and increase write index by 1. We then replace its page id with new page id we read in from the disk, mark the page clean and increase the read index by

1. Finally we update the LRU list by simply append the new page id the front of the list.

3.5 Bufferpool update: Write

If the instruction is "W", The write function will be called with the page id and the chosen replacement function.

First, we search for the specified page id that will be updated in the bufferpool. If it is in the bufferpool, we return its position, and then we use an iterator to find the page in the LRU list, delete it from the list, and them append a new one with the same page id and a dirty bit to the front of the LRU list.

If the page id cannot be found in the bufferpool, we first check to see if the bufferpool is full or not. If the bufferpool is not yet full, we first read the page from the disk, and turn on the page's dirty bit to indicate that it has been updated. We then append the page to the front of the LRU list, and increment the read index.

If the page id is not in the bufferpool, and the bufferpool is full, we first use the chosen replacement function to get the position of the least recently accessed page in the LRU list. If the dirty bit of the page has been turned on, indicating that it has been updated, we write this page back to the list and increment the write index. We then remove this page from the bufferpool, and read in the new page, and increment the read index. We finally add this page to the front of the LRU list, and turn on its dirty bit to indicate that it has been updated.

3.6 Disk Management

The disk management aspect of the codebase is implemented in the `WorkloadExecutor` class.[8] In the run workload we create our disk object as an `fstream` pointing to `disk.txt`. The stream itself allows input and output and is part of the instance of the buffer in the workload. The disk is worked through in 2 different functions.

3.6.1 `writeDisk`. The `writeDisk` function takes an instance of the buffer and is utilized as an initial population of the Disk. We simply use a pseudo-random generator with a seed generated from the timestamp in order to populate a string with a certain amount of characters. In `C++` the standard `char` type is 1-byte. As such we populate the string with the amount of characters set by the parameter, exception being the last character which is simply and endline character. As such, each page is divided by the rows in the disk text file.

3.6.2 `diskOp`. The `diskOp` function takes an instance of a buffer, an integer indicating if its a read or a write, and a page ID. The page ID and `pageSize` taken from the buffer instance are used to calculate the amount of bytes prior to the starting point of the page of interest. The `seekg` function is used in both cases to move the pointer to this computed byte. If the operation is a read we simply retrieve byte by byte into a temporary string. Since we aren't actually interested in what is contained in that string that terminates the read, as we are only simulating the action of loading a page byte-by-byte into memory. If the operation is a write, we select a character from a sequence, and replace they bytes of the page byte-by-byte. This simulates writing into disk and thus will have a more realistic impact on the runtime of our program, thereby facilitating our comparative analysis.

3.7 Replacement Policies Implemented

In this section, we will briefly talk about the structure of bufferpool and LRU list including how we managed to correctly update and maintain them. And we will mainly focus on our design and implementation details about implemented replacement policies. [6]

3.7.1 LRU. LRU (Least Recently Used) function will only be called when the bufferpool is full but needs to read in fresh data. The LRU function returns the evict position in bufferpool. When a new page needs to be stored and the bufferpool is full, the LRU function will go to the last element in the LRU list and get the least recently used page id. Delete this element in the LRU list as this page will be replaced. Using the target page id to perform a linear search in the bufferpool and return the position of that page id in the bufferpool.

We also have the LRU list, which is a list of pointers. There is a most recently used end, and a least recently used end. Each pointer in the list refers to a page in the database buffer cache. The way we implement is maintaining a deque storing page ids as the pointer to that page in the bufferpool since every page id is unique. This is because deque is faster for adding/removing elements to the end or beginning. The front of the deque represents the most used end and the back of the deque represents the least used. Generally, every time a block in the bufferpool is read or written as the result of a query, the pointer to that block (here is page id) is moved to the most end of the LRU list (MRU). Therefore, over time, this natural sorting mechanism results in the pointers to the least recently used blocks ending up at the least end of the list. These are the least likely to be required by any subsequent queries, so they are always the first ones to be replaced when the bufferpool needs to read in fresh data.

3.7.2 FIFO. FIFO, which stands for "First In First Out", is a straightforward page replacement algorithm that is applicable for small systems. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

The difference between LRU and FIFO is that when a page hits, LRU will move this page to the MRU position to update the LRU list while FIFO will make no changes to the queue. The main advantage of the FIFO page replacement algorithm is its simplicity. However, when the number of incoming pages is large, it might not provide excellent performance because every frame needs to be taken account off.

3.7.3 CFLRU. CFLRU, also known as Clean First LRU, has a similar algorithm to LRU's eviction policy, and also returns the evict position in the bufferpool. CFLRU divides the bufferpool into two sections: the working section, and the clean first region. For CFLRU we implemented a parameter which dictates the percentage of the bufferpool that will compose the clean first region- the default value is .3, which is the industry standard for CFLRU. When looking for a page to evict in the bufferpool, the algorithm first looks in the clean first region. If there is a clean page in the clean first region, then the first one will be the page that is evicted. If there is no clean page in the bufferpool's clean first region, then the least recently accessed page will be the one evicted.

3.7.4 LRU-WSR. LRU-WSR, also called Write Sequence Reordering, is another replacement function that is similar to LRU policy. The only difference between those two policies is that LRU-WSR assigns each page with a bit flag called "cold-flag". And this cold flag is the key to make the LRU-WSR algorithm a "second chance algorithm". The LRU-WSR function also returns the targeted evict page position in bufferpool. When the bufferpool is full and a new is coming, we check the last page in the LRU list (LRU position). If this page is clean, this page is the target page will be evicted and its position in the bufferpool will be returned. Otherwise, if the page is dirty, we check its cold flag. If the cold flag is set, this means the page has been the evicted target before, then this page is the target page to be evicted and its position in the bufferpool will be returned. If the cold flag is not set, the function will move this page to the front (MRU position) and set its cold flag. Then the function will point at the current LRU position page and do the above process again until it finds a page that is qualified to be evicted.

4 EXPERIMENTATION

4.1 Automation and Batch Testing

For our testing, we implement a separate set of python modules in the `workload_suite` directory; which implements testing automation, execution, data processing, and plotting.[6]

In the `full_barrage.py` script, we enable testing for all permutations of parameters as well as policy focused test.[3] We do so by taking in a `-algo` argument at the command line. Here, we use some logic to enter the right sequence of tests. So to run the full workload of tests we utilize the `all` flag, but to only run a barrage of runs for a specific algorithm we use its pertinent flag, e.g `lru`, `lruwsr`, `fifo` and `cflru` respectively. Moreove, we can use the `skew` flag to run the barrage of tests under varying skews as discussed in 4.2.

From there, we use the helper functions in `testing_helpers` to build the commands with the intended parameters and run them as bash terminal commands. Results are saved in the `raw_results` and `runs` directory. The `raw_results` director contains a more verbose summary of the run, as an example the below:

```
b  n  x  e  e'  a  s  d  p  k
10  1000  1000  10  90  1  90  10  0  1024
generation succeed
Page: 0 PageID: 72 Dirty-Bit: 1
Page: 1 PageID: 42 Dirty-Bit: 1
Page: 2 PageID: 31 Dirty-Bit: 1
Page: 3 PageID: 3 Dirty-Bit: 1
Page: 4 PageID: 58 Dirty-Bit: 1
Page: 5 PageID: 55 Dirty-Bit: 1
Page: 6 PageID: 87 Dirty-Bit: 1
Page: 7 PageID: 35 Dirty-Bit: 1
Page: 8 PageID: 15 Dirty-Bit: 1
Page: 9 PageID: 99 Dirty-Bit: 1
********************************************
Printing Stats...
Number of operations: 1000
Buffer Hit: 73
Buffer Miss: 926
Read IO: 927
Write IO: 842
Global Clock: 17.3494ms
********************************************
```

On the other hand, the data contained in the `runs` directory is specifically scant such that our `data_processing.py` can retrieve relevant data for processing and plotting. Again, as seen below the corresponding output of the verbose above:

```
1000
73
926
927
842
17.3494
```

4.2 Parameter Selection

As far as the different parameters we utilize to simulate the bufferpool behavior under different constraints, we wanted to tackle what would be feasible within our scope but also characteristic to real-life DBMS tasks.[13] Below we include the parameters we were interested in testing.

(1) `-b`: The size of the bufferpool was such that we attempted to reflect real-life conditions in that it should be balanced with the page size while still being significantly smaller than the disk. To this end the parameters we chose were `[10, 100, 250, 500]` in pages.
(2) `-n`: We kept the disk size much larger than the bufferpool across all tests and always in the realm of megabytes at least(the page size plays a role in this). The selection was `[1000, 1500, 3000, 10000, 50000]`
(3) `-x`: We kept the total number of operations proportional to the disk and such that we were basically guaranteeing bufferpool evictions. `[1000, 5000, 10000]`
(4) `-e`: For the read-write ratio we wanted to characterize 3 different types of workloads; read-intensive, balanced, and write-intensive. `[10, 50, 100]`
(5) `-a`: Obviously we used the 4 ones we implemented. `["LRU", "LRUWSR", "FIFO", "CFLRU"]`
(6) `-s`: The skewed distribution we kept at 90 since we wanted to keep this consistent for the initial algorithm comparison. However, for our skew-specific tests we varied across the spectrum. `[10, 30, 50, 70, 100]`
(7) `-d`: We kept this consistent as we wanted to focus on the first skew parameter as our independent variable. `[90]`
(8) `-k`:For the page size we wanted to mimic real-life scenarios. So we kept it as a power of 2 and we considered only page-sizes greater than 4096 to have substantive timing differences. `[128, 512, 1024, 4096, 8192]`
(9) `-n`: For the disk size in pages, we simply chose to make it significantly larger than the buffer, such that multiplied by the page size it would be in the realm of MB for the larger page sizes. `[1500, 3000, 5000, 10000]`
(10) `-w`: We kept this parameter at its default of 30% to avoid having an unbalanced dataset(more tests for CFLRU then the others).

4.3 Processing and Plotting

Now that we have discussed the collection of data we can move onto the processing and visualizing the data. We implement several new functions in the `testing_helpers.py` which help us discriminate filenames for parameters of interest and grouping. The first of these is the `get_params` function which parses a filename and returns the desired parameter. So, for instance, if we want to know the number of operations in the command that produced a certain file, we run `get_params(filename, "x")`, since x is the parameter designated to that. Next we implement the functions `discriminate` and `discriminate_skew` which essentially filter out files produced by our full workload script depending on certain aspects, for instance if the buffer is too large with respect to the disk, or if the page size is too small. As the name implies, `discriminate_skew` does the same but filters out files that aren't relevant to our skew analysis..

Finally, we wrote the `data_plotting.py` script which implements the two plotting functions `plot_algos`, and `plot_skews`.[2] The script takes an argument "type" (`-t`) to select what sort of test we are doing; these are either skew test(`skew`) or algorithm performance test (`algo`). For both cases, we parse through the files in the `runs` directory, using the discriminate functions to filter out and append relevant data to our results, we then average out the statistics in the hundreds of files deemed relevant to examine and plot the result.

4.4 Testing System

We ran our tests on a Mass Open Cloud virtual machine. It's specifications are as follows, and retrieved[7] with standard unix commands to uncover:

- Ubuntu 20.04 focal
- 5.13.0-40-generic Linux Kernel
- 8 Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
- 16130160 kB DDR4 RAM
- 512 GB SSD

5 RESULTS

5.1 Read-intensive Workloads

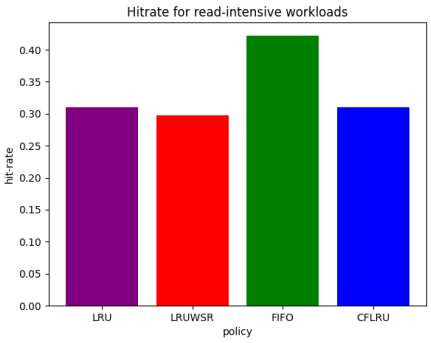

Figure 2: Bufferpool page hitrate for 90% read runs

Performance for Read Intensive Workloads can be found in Figure 2 and Figure 3. Figure 2 shows the hitrates for the 4 eviction policies, and displays the same trend as with balanced and write skewed workloads: LRU, CFLRU, and LRU-WSR have a hit rate of .3, and FIFO has a hit rate of .4. Furthermore, the execution time of the policies is in Figure 3, and you can see that the execution time for LRU, CFLRU, and LRU-WSR is about .01ms, and for FIFO it is about .008ms. Read intensive workloads have the shortest execution times

across the board because there are fewer updates to write back to the disk, which takes extra time.

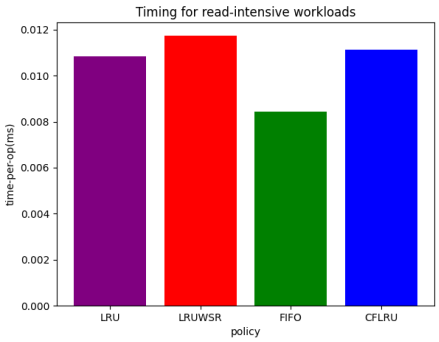

Figure 3: Time per operation for 90% read simulations

5.2 Balanced Workloads

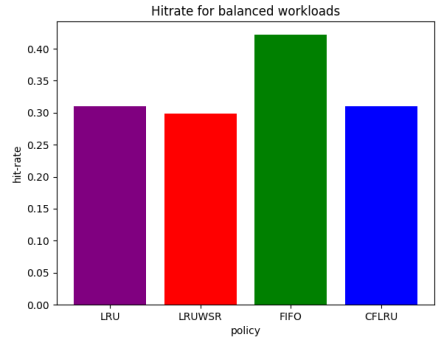

Figure 4: Bufferpool page hitrate for 50% read runs

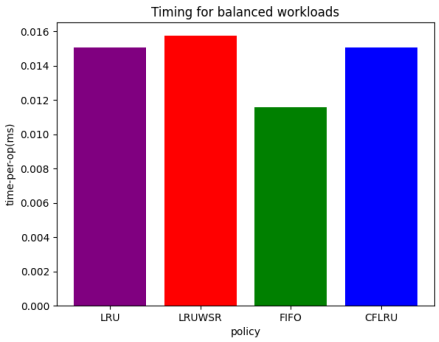

Figure 5: Time per operation for 50% read simulations

Analysis on balanced workloads across the 4 eviction polices(LRU, CFLRU, LRU-WSR, and FIFO) can be seen in Figures 4 and 5, with Figure 4 detailing the hitrate, and Figure 5 showing a timing breakdown. As you can see in these figures, LRU, LRU-WSR, and CFLRU perform similarly with all of them having about .015 ms of execution time, and a hitrate of about .3, but FIFO outperforms these metrics with an execution time of .01ms and a hitrate of .4. Since FIFO is the least computationally complex eviction strategy of the 4 policies, it stands to reason that it would have the quickest execution time. Furthermore, since the workloads generated for this experiment are random, there is no reason for the data that will be accessed again to have also been recently accessed. This performance is likely to change in real data applications where "live" data is also that which is more frequently accessed, and would give the edge to any of the other 3 policies.

5.3 Write-Intensive Workloads

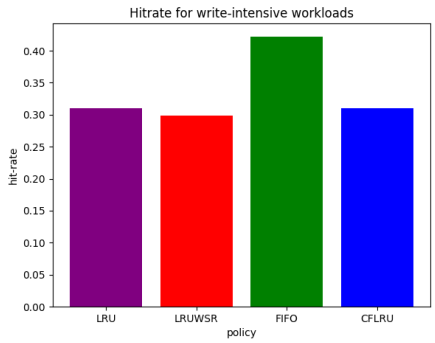

Figure 6: Bufferpool page hitrate for 10% read runs

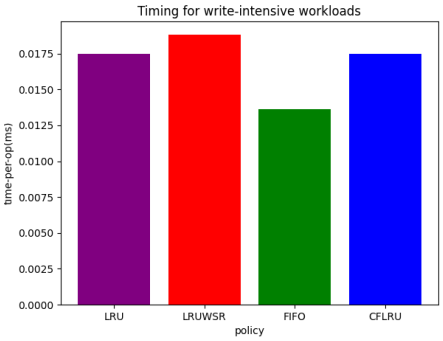

Figure 7: Time per operation for 10% read simulations

The breakdown of performance for write intensive workloads can be seen in Figures 6 and 7, where Figure 6 depicts the hitrate for each eviction policy, and Figure 7 shows the timing for each eviction policy. Write Intensive workloads have a similar hitrate and execution breakdown as balanced

workloads, where LRU, CFLRU, and LRU-WSR have a hitrate of about .3 and FIFO outperforms with a hitrate of .4. Execution time is longer for write intensive workloads, likely because frequent writes comes with writing that information back to disk. For LRU, CFLRU, and LRU-WSR, the execution time is about .0175ms, and .0125ms for FIFO.

5.4 Analyzing Effect of Skew

Percentage of Skew is a parameter for us to control the rate of page collisions in the workload generation script. As such we would predict evictions to increase drastically, resulting in hit-rates reducing linearly with increased skew. This is observed in the relevant figures below but is emphasized in different ways across policies.

5.4.1 LRU. Performance for LRU policy with variation of skew percentages is shown in Figure 8. Since we kept data parameter consist with 90 percent, by varying the percentage of skew we could control the rate of page collisions in the workload. It is noticeable that when the skew is 10 percent, the hit rate is the highest in the graph with 0.34 hit rate. And from 10 percent to 30 percent to 50 percent, every step caused a drastic decrease on hit rate. This is because with 10 percent of operations on 90 percent of data, the rest of 90 percent of operations would be on only 10 percent of data. That means the operation targets would always be concentrated on a few pages. Since that situation is perfectly fitting the idea behind LRU (pages that were heavily used before will most likely be heavily used in the future) and therefore the hit rate is high. However, as we increase the percent of skew, the operation target pages become scattered and therefore the hit rate decreases as the bufferpool need to access more and more new pages. If we are using 100 percent of skew on 100 percent data, that basically means the bufferpool might need to access most of the pages if not every single one of them.

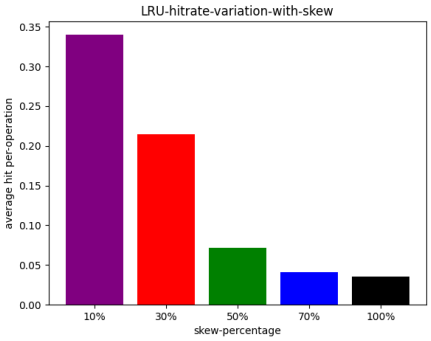

Figure 8: Hitrate variation with skew for LRU

5.4.2 LRU-WSR. Figure 9 breaksdown the performance of LRU-WSR in terms of hit rate as skew is modified. In the figure we see hitrate decreasing as skew percentage increases. With 10 percent skew, LRU-WSR experiences a hit rate of greater than .4. This is especially interesting because it outperforms the other algorithms when they also have 10 percent skew, as can be seen in Figures 8, 10, and 11. While some of this performance can be contributed to random workload variability, we believe that the functionality of the cold flag is at the heart of this performance improvement with low skew. Since it works as a "second chance" algorithm, it keeps dirty pages in the bufferpool for longer, and only evicts those which have previously been a target for eviction, meaning that they have not been recently accessed in a longer period. This tracks because LRU-WSR continues to out perform the other eviction policies at 30 percent skew (.28 hitrate), and then begins to perform on par with FIFO, which was the best performing algorithm, at 50 percent skew(.15 hitrate). LRU-WSR hitrate continues to decrease as skew increases, with a hitrate of .1 at 70 percent skew and about .07 at 100 percent skew.

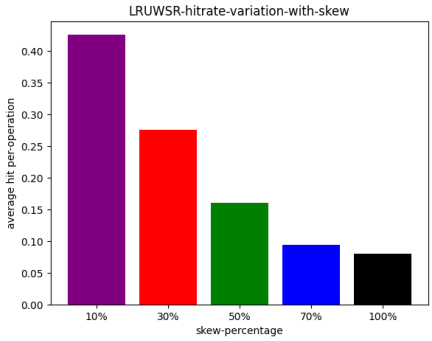

Figure 9: Hitrate variation with skew for LRU-WSR

5.4.3 FIFO. The FIFO policy was our best performing policy, and its behavior under increasing amounts of skew doesn't seem to change that. We see that at the 100% and 70% skew we observe a good tolerance in that the drop off is not as drastic as with say LRU or CFLRU. In fact its drop off seems more linear then the logarithmic characteristic exhibited in those. In addition to the points discussed in the previous subsection, we can see that this improved performance and improved trend relate to the wielding of temporal locality inimical to the FIFO policy

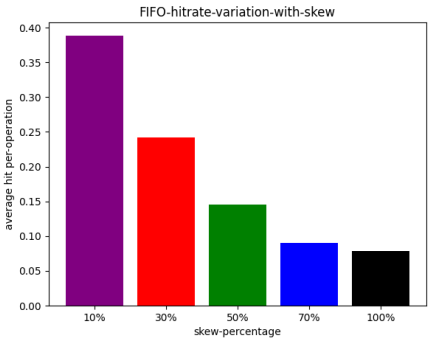

Figure 10: Hitrate variation with skew for FIFO

5.4.4 CFLRU. CFLRU algorithm keeps dirty pages in the buffer without consideration of the access frequencies of these pages. As a result of keeping dirty pages in the buffer, the overall performance may decrease because it lowers the hit ratio. Therefore, as we can see in Figure 11, even with 10 percent of skew, the hit rate for CFLRU is still lower than LRU. Also, if we compare the hit rate of 30 percent skew with 50 percent skew, there is a drastic decrease from 0.2 to 0.06. This maybe cause by the workload – when the workload contains a lot of READ instructions but the buffer is mostly filled up with dirty pages. Since this algorithm tends to keep dirty pages in the buffer to reduce the frequency of writing to memory, pages with read instruction are more likely to be evict sooner than LRU. Thus, in a situation that the workload firstly has a few write instruction and a lot of READ instructions after that, CFLRU may cause more page miss than LRU.

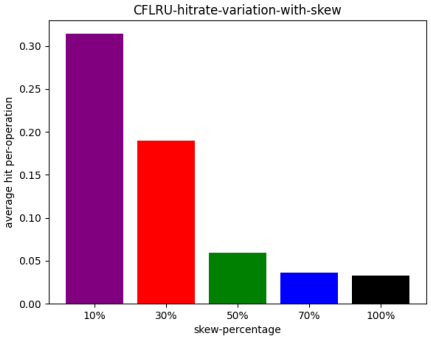

Figure 11: Hitrate variation with skew for CFLRU

6 CONCLUSION

In this project, we have implemented 4 types of page replacement algorithms and tested their performance by using provided workload generator and a self-implemented simulator for disk operations. In order to further explore the connections between different polices as well as their strengths and weaknesses, we developed comprehensive test suites. We run test suites on self wrote scripts to do extensive testing and multiple dimension plotting, and have performed a comparative analysis based on experiment results.

Across the board we found that FIFO outperformed the other algorithms(LRU, CFLRU, and LRU-WSR) in terms of hitrate and execution time. We concluded that it makes sense that FIFO would have the fastest execution time since it has the least computation of any of the algorithms since the end of the deque is selected for eviction with no further reordering or complexity. We found FIFO's higher hitrate performance to be very peculiar because in other papers and work it has been found to have a lower hitrate than LRU and extensions of LRU. We believe that this unusual performance is due to the nature of the random workload generator, since it doesn't contain the patterns we see in real data applications where "live" data tends to be more recently accessed, which is why bufferpool eviction policies optimize with this in mind. Since the workload is random, it makes the effects of using LRU and its extensions useless, and in this even detrimental to the performance of the bufferpool. Our last finding experimented with changing the skew parameter, and found that as skew increases, hitrate decreases for all of the algorithms as a result of more page collisions.

6.1 Future Work and Extension

To further improve this project, here are a few tasks for potential future work:

6.1.1 Implement LFU. LFU, stands for Least Frequently Used cache, is a caching algorithm in which the least frequently used cache block is removed whenever the cache is overflowed. In LFU we check the old page as well as the frequency of that page. Since currently, most of our implemented replacement algorithms are based on LRU, adding LFU may give us more insights on periodically repeated data. This approach to cache eviction is very efficient in cases where the access patterns of the cached objects do not change often. While LRU caches will evict the assets that would not be accessed recently, the LFU eviction approach would evict the assets that are not needed any more after the hype has settled.

6.1.2 Optimize Implementation For Lower Latency. Currently, our implementation for bufferpool and LRU list was designed as a combination of a vector(bufferpool) and a deque(LRU list). To reduce the access latency, we can replace the vector to a hashmap(unordered map). This is because, when we search a page in bufferpool, lookup in the vector would take O(n) time while lookup in the hashmap takes O(1). Since we need to lookup pages from the bufferpool both in the

search function and when returning the evicting page position, when a bufferpool is large this new implementation may significantly reduce the runtime latency. However, because the elements in hashmap in c++ are not stored in an ordered sequence, so if you implement the LRU cache only using hashmap, you may find that the order of inserted/deleted items is a mess. Thus, to fix this order issue we will use a list because it is flexible enough to support inserting items into any position and erasing an arbitrary item in constant time. Therefore, to find an item in list in constant time, the best way is constructing a hashmap to record the key and its corresponding address in list. The design map is shown in Fig 12.

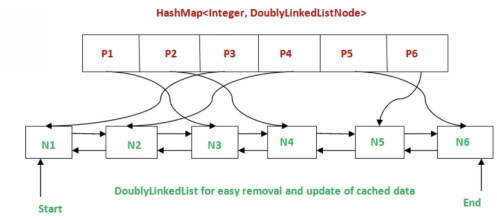

Figure 12: Implementation design for bufferpool and LRU

REFERENCES

[1] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. 1994. The uniform memory hierarchy model of computation. *Algorithmica* 12, 2 (1994), 72–109.

[2] Niyazi Ari and Makhamadsulton Ustazhanov. 2014. Matplotlib in python. In *2014 11th International Conference on Electronics, Computer and Computation (ICECCO)*. IEEE, 1–6.

[3] Yadu N Babuji, Kyle Chard, Ian T Foster, Daniel S Katz, Mike Wilde, Anna Woodard, and Justin M Wozniak. 2018. Parsl: Scalable Parallel Scripting in Python.. In *IWSG*.

[4] Clifford E Cummings. 2002. Simulation and synthesis techniques for asynchronous FIFO design. In *SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers*.

[5] Jaeyoung Do, Donghui Zhang, Jignesh M Patel, David J DeWitt, Jeffrey F Naughton, and Alan Halverson. 2011. Turbocharging DBMS buffer pool using SSDs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 1113–1124.

[6] Samir Farhat Dominguez, Yuxin Li, and Stephany Yipchoy. 2022. *Bufferpool Manager, https://github.com/SamirFarhat17/bufferpool-manager*.

[7] Jeffrey Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. 2012. Engage: a deployment management system. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 263–274.

[8] Mark S Gockenbach, Matthew J Petro, and William W Symes. 1999. C++ classes for linking optimization with complex simulations. *ACM Transactions on Mathematical Software (TOMS)* 25, 2 (1999), 191–212.

[9] Hoyoung Jung, Hyoki Shim, Sungmin Park, Sooyong Kang, and Jaehyuk Cha. 2008. LRU-WSR: integration of LRU and writes sequence reordering for flash memory. *IEEE Transactions on Consumer Electronics* 54, 3 (2008), 1215–1223.

[10] Papon Manos Athanassoulis. 2022. The uniform memory hierarchy model of computation. *Data-intensive Systems and Computing Lab Department of Computer Science College of Arts and Sciences, Boston University* 12, 2 (2022), https://bu–disc.github.io/CS561/projects/CS561–S22–SysProj–Bufferpool.pdf.

[11] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record* 22, 2 (1993), 297–306.

[12] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. 2006. CFLRU: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. 234–241.

[13] Ilsa M Schiefelbein. 2002. *Fault segmentation, fault linkage, and hazards along the Sevier fault, southwestern Utah*. Ph.D. Dissertation. University of Nevada, Las Vegas.