Beefing Up the Buffer Cache

Júlía Oddsdóttir
Sandra Ósk Sigríðardóttir Bender

Final Report of 12 ECTS credits submitted to the School of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of
Bachelor of Science (B.Sc.) in Software Engineering

May 2018

Supervisors:

Ýmir Vigfússon, Assistant Professor

Gylfi Þór Guðmundsson, Adjunct Professor

Examiner:

Marcel Kyas, Assistant Professor

Beefing Up the Buffer Cache

Júlía Oddsdóttir
Sandra Ósk Sigríðardóttir Bender

May 2018

Abstract

Modern life demands fast computations. Even the slightest latencies can have severe consequences and cost companies a fortune in lost revenue every hour. A vital component for reducing latency in computer systems is the cache. A cache's main purpose is to store data that is frequently re-accessed, and thereby reducing the time it takes to fetch popular items, instead of constantly fetching from slower devices.
In this report, we examine the page cache in the Linux kernel. We investigate how the page cache can be monitored and profiled, as well as discuss how it could be improved upon to reduce latencies in computations. The page cache is complex so we will seek to explain some of its relevant features. We introduce a method of tracing the page cache to obtain information about its current implementation, and seek to explain its behaviour. We will also feature ideas on how the page cache's replacement policy could possibly be improved.

Acknowledgements

Ýmir Vigfússon, our supervisor and teacher. We are grateful for the opportunity to work on this project with him, even though we have run into so many walls that our noses now resemble that of a pug. He allowed us to enter the world of research in computer science and held our hand along the way, despite him having more than enough on his plate.
Gylfi Þór Guðmundsson, our advisor and master report reviewer. Every time we had fallen into a deep pit of despair, he was there to pick us up. His advice and happy-go-lucky nature has been a saving grace these past weeks.
Freysteinn Alfreðsson, for helping us be able to navigate the Linux kernel. He was under no obligation to assist us, but out of the kindness of his heart, he spent time teaching two newbies the basics of kernel programming. Had it not been for him, we'd probably still be searching for the location of the page cache.
Caroline Butchek, for taking the time out of her busy schedule to share her experiences with the kernel and to help us understand how to create proper traces.
Jason Yang, the creator of Mimircache. He was always ready to help us with the usage of Mimircache and answer our questions, no matter how stupid they were.
For proofreading this report (and putting up with our insanity for the past years), we'd like to thank our significant others, Vigfús Ægir Vigfússon and Sölvi Hjaltason. Your support has been invaluable.
Last but not least, we would like to thank David Thue and the rest of the CADIA team for allowing us to use their offices to work on the project, as well as granting us access to endless amounts of coffee and biscuits.

Contents

1 Introduction

An ever-growing number of companies rely on online systems to operate. High latencies can cost a fortune in lost revenue every hour [1]. Sensitivity to latencies leads to a need for increased efficiency in all operations being run on online systems. A vital part of decreasing latency is the cache. A cache is a small and fast storage, whose purpose is to decrease delay that occurs while fetching data from a larger and slower secondary storage, such as the hard disk drive [2]. The cache does that by keeping previously retrieved data available, such that subsequent accesses can be served faster. Several different kinds of caches can be found in the kernel, which is the core of every operating system. The kernel facilitates communication between applications and the computer's hardware, such as the CPU, memory, and various peripheral devices, like keyboards, monitors and speakers.

Linux is an operating system, a member of the large family of Unix-like operating systems, and has been in development since 1991. Linux is an open source collaborative project, written by thousands of developers, managed by Linus Torvalds. Perhaps the most appealing quality is that Linux is fully customisable in all its components, which is why people started writing code based upon the original Linux operating system to suit their own needs. These customized versions of the Linux operating system are called Linux distributions, and include Ubuntu, Fedora and Arch. Companies have also utilized the fact that Linux is open source, and have built their operating systems on top of the Linux kernel, notably Google creating Android. The success of Linux is largely based on the Linux kernel, which is a monolithic [3], preemptive [4] kernel that allows multithreaded applications and kernel threading, among many other qualities. The Linux kernel is run on millions of devices worldwide, on anything from light bulbs to supercomputers.

The Linux kernel is very large, and contains millions of lines of code [5]. Each subsystem can take a programmer years to master. Figure 1.1 breaks the kernel down into its main components and shows how vastly complicated the kernel is, as many intricate components are required to properly work so that the whole is functional. The main focus of this report will be the page cache, which is a bridge between the virtual and logical file systems, and is circled in red in the figure mentioned above.

The page cache is the main cache for the hard disk drive, and is necessary because fetching data from the disk is extremely slow. The page cache is located in the otherwise unused portions of the physical RAM, and the pages in the page cache correspond to physical blocks on the disk. Because the page cache is only allowed to use free memory in RAM, it is dynamic so it grows and shrinks in size as the memory is either freed or occupied [6].

The page cache has seen multiple changes over the years, and has been optimized quite a lot. It currently uses a cache replacement algorithm based on the Least Recently Used (LRU) algorithm that will be discussed in section 2.1.2, with some additions in order to make it more efficient. However, given that the page cache replacement algorithms have not been changed in a long time, but there have been many recent advancements in the field of cache replacement algorithms, it is natural to wonder whether the performance of the page cache in the Linux kernel could be improved. This question is especially interesting now, since the popularisation of the Solid State Drives (SSD) altered the behaviour of disk drives drastically.

The goal of this project is to find a way to improve the current replacement policy of the page cache. There are multiple ways to achieve this goal. It is possible to find weaknesses of the current replacement policy and alter the source code to better the algorithm's performance. A more compelling idea is to explore the question *"can we guess what will be put into the cache and when?"* Implementing this would require a machine learning algorithm to be added to the kernel, which's purpose would be to predict what data will be referenced in the near future, based on previous experiences. This algorithm would have to be carefully implemented, as it would need to exceed the current algorithm's performance, while not taking too much memory from the cache itself.

To be able to measure whether or not an alteration improves the current replacement algorithm's performance, a method of tracing the page cache will be implemented. Cache performance depends on what kind of a workload is performed, so a method for generating a workload is required. Each trace can then be generated by performing a different workload, so a general overview of how the page cache behaves will be obtained. By performing analysis on the traces, it is possible to see whether an alteration to the cache replacement algorithm is or is not for the better.

In this report we will discuss the current implementation of the Linux kernel's page cache and how it currently behaves. We will examine the strengths and weaknesses of the current replacement policy in section 3. A tracing method we devised to understand the cache's behaviour and measuring performance will be introduced in section 4. Finally, related work will be discussed in section 5, along with a discussion on future work in section 9.

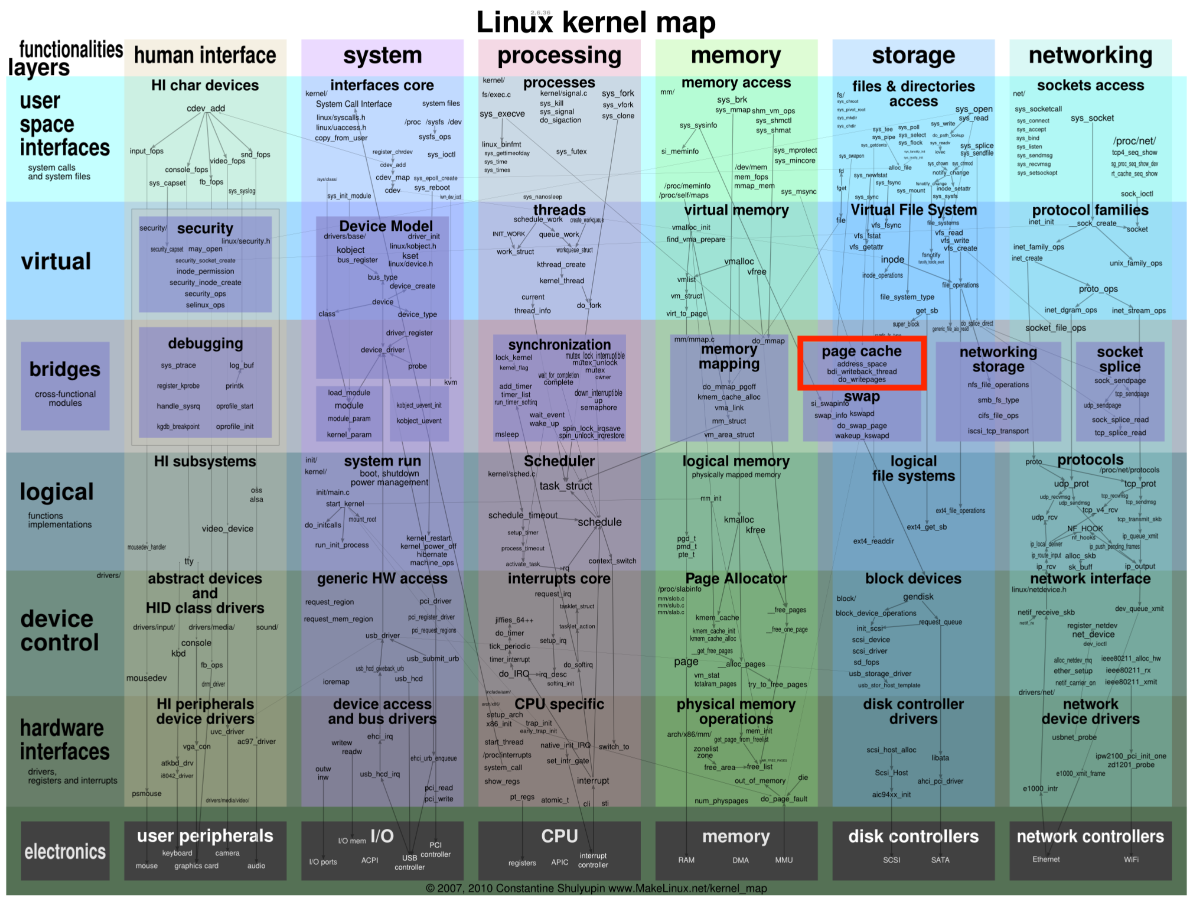


Figure 1.1: Map of the Linux Kernel [7], the page cache is circled in red.

2 Background

Recall that a cache is a small and fast storage, whose purpose is to decrease delay of a look-up operation in the slower, secondary storage. When accessing a block, which is a contiguous chunk of memory, a search algorithm is used to perform a fast look-up in the cache. Search algorithms are in most cases rather fast, the search algorithm is even physically baked onto the CPU chip. If the requested block is already in the cache, a *cache hit* occurs. If the block is not in the cache, a *cache miss* occurs, and the block has to be fetched from a disk and placed into the cache. A cache is finite, so eventually it will fill up and a block has to be removed so a new block can be accepted into the cache. The desired functionality is that cache hits occur more often than cache misses, so a replacement algorithm is needed to decide which block would be most efficient to remove. In this section, replacement algorithms and strategies related to caching will be introduced.

2.1 Replacement Algorithms

In an ideal world, there would be one cache replacement algorithm that could determine which blocks would be ideal to keep in the cache, while removing the blocks that will not be used again in the near future. A theoretical solution to this problem was was proposed by Bélády, and is called the Optimal algorithm. He stated that the most efficient algorithm was one that will always evict the block that will be used farthest in the future [8]. This is not implementable in practice, since such an algorithm would require perfect knowledge of all future access patterns. However, multiple different cache replacement algorithms have been implemented. Some of these algorithms are implemented to be applicable in multiple or all caches, while others attempt to find ways to get as close to the Optimal policy for one application as possible. In this section, some of the more prominent replacement algorithms will be featured.

2.1.1 First In First Out Algorithm (FIFO)

The First In First Out (FIFO) algorithm proposes an idea where the blocks in the cache are stored in an ordered list. When a block is referenced and is not in the cache already, it is placed at the back of the list. When the cache is full and a new block needs to be added, the block that is at the front of the list will be evicted. This algorithm always evicts the oldest block without taking advantage of how often each block has been accessed. FIFO is a very fast algorithm, but the reason for why the algorithm is rarely used is because of its inability to utilize locality of reference [9]. As a result, FIFO's hit ratio is very low. Andrew Tanenbaum and Herbert Bos [10] explain this algorithm in further detail in *Modern Operating Systems*, chapter 3.4.3.

2.1.2 Least Recently Used Algorithm (LRU)

When the Least Recently Used (LRU) algorithm decides which block to remove from the cache, it finds the block that has been unreferenced for the longest time. To ensure that the right block is evicted, LRU monitors when blocks are referenced. This is done by keeping a linked list that stores the order in which blocks were referenced. When a new block needs to be added and the cache is full, the block at the back of the linked list is evicted, and the new block gets added to the front.

LRU has a few appealing advantages, such as that it is amenable to full statistical analysis, can be easy to implement, and its running time per request is essentially independent of the cache's size. It effectively exploits temporal locality: a block that has been accessed recently is likely to be accessed again in the near future. However, LRU is well known for its inability to cope with access patterns with weak locality [11]. In addition to that, LRU requires locking when blocks are being added and removed from the linked list, to make sure no data is lost. Further information can be found in *Modern Operating Systems* [10], chapter 3.4.6 about the LRU algorithm, written by Andrew Tanenbaum and Herbert Bos.

2.1.3 Least Frequently Used Algorithm (LFU)

The Least Frequently Used algorithm (LFU) [12] works similarly to LRU, but instead of keeping track of when which block was used, the LFU algorithm keeps track of how often each block is referenced while it is in the cache. The block that is least frequently referenced will be the one removed from the cache when a new block needs to be added. LFU has been proven to work well with specific workloads, such as web caching, where LFU outperforms LRU [13]. However, LFU's main disadvantage is that its time complexity is logarithmic, because the data structure needs to maintain sorted order of frequency, and therefore it is rarely used in practice.

2.1.4 Clock Algorithm

The Clock algorithm [10] keeps the blocks in the cache in a circular list. Each block in the list contains a binary reference bit, which is set to one when the block is referenced. The Clock algorithm keeps a pointer, called *hand*, that points to the oldest block in the list. When a block has to be evicted, the block that the *hand* points to is the first one to be inspected. If the reference bit is zero, that block will be evicted and the new block is added in its place. If the reference bit is not zero, it will be set to zero and the *hand* is moved clockwise until a block with its reference bit set to zero is found. The hit ratio of the Clock algorithm has been proven to be worse than in LRU, but LRU uses locks while Clock can be updated in parallel and thus is useful in concurrent operating systems like multi-core (SMP) enabled Linux. Lastly, since the Clock algorithm approximates LRU, disadvantages of LRU also apply to Clock [14].

2.1.5 Segmented Least Recently Used Algorithm (SLRU)

The Segmented Least Recently Used (SLRU) [15] algorithm is implemented on top of LRU. SLRU adds a reference bit into each cache line and divides the cache set into two segments, a probationary segment and a protected segment, which are ordered using LRU. Data added to the cache will go to the end of the probationary segment. If a cache hit occurs, the block will be promoted to the protected segment. When a block has to be removed from the cache it will be evicted from the probationary segment. However, if the probationary segment is empty, the block will be removed from the protected segment. Since blocks in the protected segment are more likely to be referenced again in the near future, they stay longer in the cache when using SLRU than they would in LRU. A downside to this method is that it becomes possible for blocks to remain in the cache long after they were last referenced.

2.1.6 Adaptive Replacement Cache Algorithm (ARC)

The Adaptive Replacement Cache algorithm (ARC) [16] was originally implemented to improve upon SLRU. ARC is scan-resistant, so it allows one-time-only sequential read requests to pass through the cache without flushing pages that have temporal locality [17]. ARC contains two dynamically sized lists, named L_1, which contains blocks that have each been referenced once recently, and L_2, which contains blocks that have each been referenced more than once recently. LRU is used to order both lists. When a block that is not in the cache is requested, the block goes to the head of L_1. If the block is referenced again while it is still in L_1, it will be promoted to the head of L_2, and L_1 consequently shrinks by one. The two lists are roughly the same size, so if L_2 is growing too fast, blocks at the end of L_2 have to be demoted to L_1. The block that will be evicted is always selected from the end of L_1.

The main difference between the ARC and SLRU is that ARC will only remove blocks from a single list, while SLRU can remove from either one, provided one of them is empty. In SLRU, blocks are never demoted, which allows the protected segment to grow larger than the probationary segment. Blocks might therefore stay in the protected segment long after they were last referenced. This is a fault of SLRU that ARC improves upon by keeping both lists of roughly the same size.

2.2 Ghost List

A ghost list [18] is a mechanism used alongside ARC. The ARC lists L_1 and L_2 have extension lists, M_1 and M_2, named ghost lists. These lists are used to track blocks that have recently been evicted from the cache. When a block that was at some point upgraded to L_2 gets evicted, information about the block, called metadata, will be stored in M_2. If the evicted block gets referenced again while its metadata is still in M_2, then it is assumed that L_2 is not big enough, and the number of slots in L_2 will be increased by one. The same will happen if a block's metadata was stored in M_1, the number of slots in L_1 will be increased by one. The ghost lists have finite memory, so eventually all evicted blocks' metadata will be removed, in accordance with LRU.

A block's metadata is much smaller in size than the block itself, so storing a list of metadata would not take up too much space. By maintaining a ghost list, the algorithm could "regret" having recently kicked something out by virtue of seeing the ghost list metadata still around, and take that into consideration by changing the size of the lists.

2.3 Search Algorithms

As the main requirement of caching is to decrease latency, every component of a cache must be able to execute as quickly as possible. Since the number of blocks in a cache can quickly grow large, performing look-up operations must not slow down the execution significantly. Therefore, there is a need for search algorithms that are able to quickly find the relevant block. In this section, search algorithms used in caching will be featured.

2.3.1 Trie

A trie [19] is a search tree that stores a value in a node that is associated with a part of a key, henceforth referred to as a key fragment. The edges in a trie contain a part of a key, and a cumulation of all key fragments in a path from the root to a leaf form the whole key. The key fragments of edges are ordered in an alphabetical order or bit wise lexicographic order, which allows for faster look-up. The trie contains two types of nodes, inner nodes and outer nodes. All inner nodes contain the value of the key fragment leading to that node, concatenated to the value of its predecessor. Outer nodes are either the root node, which's value is always empty, or leaf nodes, which contain a value corresponding to the key. Tries are space saving, as prefixes are shared among nodes.

A look-up operation is performed by starting at the root node, and following a path corresponding to the search key until a leaf node is reached. The value contained in the leaf node is then returned. The look-up operation for tries is efficient, as it has a worst-case time complexity of $O(n)$, where n is the length of the key, which corresponds to the depth of the tree. More information about tries can be found in *Swift Data Structure and Algorithms* by Erik Azar and Mario Alebicto, in the *Trie Trees* chapter.

2.3.2 Radix Tree

A Radix tree [20] is a compressed implementation of the trie algorithm. While values corresponding to each edge in a trie may only contain a single character or digit, values in a Radix tree may contain multiple. This allows Radix trees to have fewer edges and nodes in the tree, and thereby reducing the amount of memory required to store the tree. Due to the fact that fewer edges are required, the distance from root to leaf can be shorter than in a corresponding trie, allowing for faster look-up. Therefore, Radix trees are more efficient than tries. Figure 2.1 shows a trie and a Radix tree for the same set of data. The figure makes it apparent that fewer nodes and edges are needed in a Radix tree than in a trie.

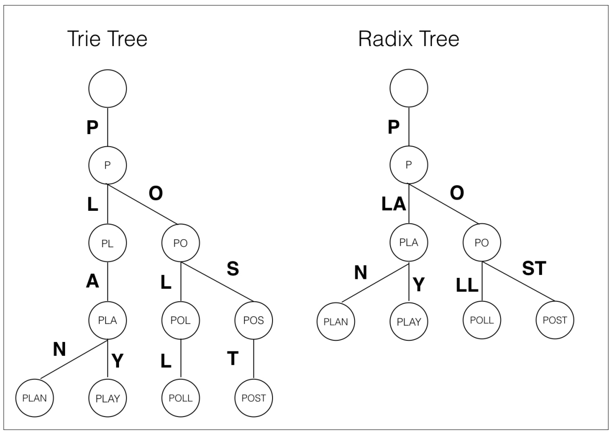


Figure 2.1: Difference between trie and Radix tree [20].

3 The Buffer Cache & the Page Cache in the Linux Kernel

The buffer cache and the page cache were two essential components in the Linux kernel. Like most parts of the kernel, these caches are vastly complicated and have undergone many changes over the last thirty years. In this section, the history of the two caches will be summarized, and different components will be explained in order to show how the buffer cache and the page cache work in today's kernel.

3.1 History of the Buffer Cache & the Page Cache

The buffer cache was originally a portion of physical memory allocated during system startup and was dedicated to caching filesystem data. It was implemented because reading and writing from the disk is a slow operation, and the waiting period can get large once multiple system operations are performed. The memory is organized into buffers, which are chunks of data. Memory accesses are fast, which is why keeping disk blocks that are frequently accessed available in memory, reduces the number of disk accesses and improves the performance of the system. Later, the page cache was implemented, and was a portion of available system memory used for caching file data, managed by the virtual memory system [21].

Common system calls, such as read and write operations, were simple to perform, as these were I/O operations and had direct access to the buffer cache. However, the coexistence of the two caches caused problems for other system calls, such as mmap, which creates a new mapping in the virtual address space of the calling process [22]. When the mmap command was used, the system had to use the page cache to store its data, since the buffer cache was not a part of the virtual memory and could not be mapped into an application address space. This caused a page fault, and to resolve it the file data was then copied from the buffer cache to the page cache, where it was modified and copied back to the buffer cache. There, the altered version could finally be updated on the disk. This is called double-caching, which is inefficient since twice as much memory is used and it leads to more cache misses. Copying the data between caches also wastes the CPU's time and leads to more computations, slowing down the system. There is also a risk of the two copies of the file becoming inconsistent, which could lead to application errors.

In the year 2000, the Unified Buffer Cache (UBC) [21] subsystem was proposed by Chuck Silver. UBC implemented a solution where the file data for mmap accesses was only stored in the page cache, without going through the buffer cache. The same change was made to other system operations. This system allowed physical memory to have more flexibility, which improved performance of applications that use caching to decrease latency.

Up to version 2.4 of the Linux kernel, the page cache and the buffer cache coexisted. The unification of these caches began in version 2.2, based on the idea of UBC, and was centered around read operations. The entire subsystem was not completely implemented until the release of version 2.4. After that, the buffer cache was maintained separately for filesystem metadata and caching of raw block devices, but other operations formerly performed by the buffer cache were from then on handled by the page cache [23].

Hereafter, there will be no references to the buffer cache in this report, as the main functionality of the buffer cache was overtaken by the page cache after the unification. The buffer cache has become irrelevant to this project, and only the page cache will be referenced.

3.2 Contents of the Page Cache

Today, the page cache is the main disk cache used by the kernel, and contains pages, which are fixed-size blocks of memory [24]. Currently, the size of a page is 4096 bytes. Since almost all read and write operations rely on the page cache, applications can request a new page to be added to the cache if it is not already present. While read operations are performed as soon as a process requests a read, write operations are deferred. The cache delays writing new changes to the disk, in case more write operations are performed on the same page quickly after the first write, this is called a write-back policy. Further information can be found in *Understanding the Linux kernel* [25] by Daniel Bovet and Marco Cesati in the *Page Cache* chapter.

3.3 Searching the Page Cache

When a large file is accessed, the page cache may become filled with so many pages that searching through them all would be too time consuming. In order to perform an efficient look-up, the Linux kernel makes use of the Radix tree algorithm, described in section 2.3.2. The Linux Radix tree algorithm [26] proposes a solution where each node contains up to 64 slots, and all slots store a pointer. This would be comparable to a node in a typical Radix tree split into 64 parts. Each of these pointers point to one of the following: the next level in the tree, a slot in the same node (called a sibling entry), an indicator that the entry in the slot has been moved to another slot in the tree, or a *NULL* pointer. The last two bits in a slot determine which location the pointer references. A slot in a node is indexed with a portion of the search key.

The Radix tree is sparse, meaning that if a page cannot be found following a path corresponding to the key, the page is not in the cache, and has to be fetched from the disk. This means that for each page, only one look-up has to be performed. The tree is also not allowed to have height larger than six, so search operations will never have a larger time complexity than $O(6)$ [25]. These additions to an already fast algorithm, make it so that look-up operations are very quick, which is essential in the page cache.

3.4 The Replacement Algorithm in the Page Cache

The page cache in the Linux kernel uses a replacement algorithm similar to ARC, described in section 2.1.6. It also contains shadow entries [27] which is a similar mechanism to the ghost list described in section 2.2.

The ARC algorithm inspired Linux developers to add lists to the page cache in a similar fashion to those described in the ARC protocol. These lists are called the inactive and the active lists, and they perform a similar function to the L_1 and the L_2 lists in ARC. The only difference between the lists in these two algorithms is the internal order. While the two lists in the ARC algorithm use LRU for file pages, the inactive and the active lists use the Clock algorithm [27]. The Clock algorithm is described in section 2.1.4. In all other aspects, the inactive list behaves like L_1 and the active list behaves like L_2.

The concept of shadow entries is similar that of the ghost list in the way that both keep track of pages that have been evicted from the cache. A shadow entry contains information about a page, taking the place of that page in its radix tree slot after eviction. Like the ghost list, shadow entries also affect the size of the lists. All that is known about the active list is that it contains pages that have been referenced more than once. At some point, pages in the active list might not be referenced as frequently anymore, so they have to be demoted to the inactive list, and eventually they are evicted from the cache. When that happens, a shadow entry is created to replace the evicted page. After the page's eviction, the shadow entry maintains a counter that is used to calculate how long the page has been outside of the cache, called refault distance. If the refault distance is smaller than the number of evicted pages since the refaulting page got evicted, the number of slots in the active lists slots will be decreased. This will improve the chances of the refaulting page being promoted to the active list if it is referenced again. If the number of slots in the active list is decreased, the number of slots in the inactive list is subsequently increased.

4 Performance Analysis

To be able to understand cache behaviour, analysis needs to be performed. One method for performing cache analysis is to create traces that can show which operation is performed when. These traces can then be further analyzed to get statistical information about the cache's behaviour. Using this technique, it is possible to compare and contrast the performance of different algorithms used by a cache. In this section, tools for analysis of cache performance will be introduced.

4.1 Traces

Tracing is a specialized logging method to record information about a program's execution. This information can then be used to analyze performance of a program. It can also be used for debugging purposes. By using traces to observe the functionality of a cache it becomes possible to see which operations are performed on the cache, when they are performed, how long a page is in the cache, how a page is moved around, and how many pages there are in the cache, etc. When working with traces in the Linux kernel, writing a trace without it influencing execution is no trivial task, as will be discussed in section 6.

4.2 Flexible I/O Tester

Flexible I/O Tester (FIO) [28] facilitates testing of the Linux I/O subsystem and scheduler. FIO is written by Jens Axboe and was originally written to save him time and energy on writing tests for every special case when measuring performance and finding bugs in a big program. FIO is able to simulate a given I/O workload without resorting to writing tailored test cases for every situation. A defined workload can be specified as a benchmark when evaluating a computer system in terms of performance [29]. FIO is used to perform a specific workload on a system to measure disk performance.

It is possible to create specific job files that match the I/O load that needs to be simulated using the parameters listed in section 1.9. of FIO's documentation [28]. When using FIO it is possible to use multiple threads or processes when performing a particular type of I/O action, which is specified by the user. The recommended way to run FIO is to create a job file, that details which actions are to be performed, along with other relevant instructions or parameters.

FIO provides an output on the run. The output is displayed so it is possible to know what each process or thread is currently doing, like how many reads and writes were performed, the rate of I/O, and estimated time until completion for the current running group. After all jobs have finished, FIO yields another output that provides information about the status of each thread, group of threads, and disks.

4.3 Mimircache

Mimircache [30] is a platform for flexible and efficient analysis of cache traces, created by Juncheng Yang, who is currently a student at Emory University. Mimircache is written in Python (PyMimircache) and has a submodule written in C (Cmimircache). To be able to use Mimircache, a trace is required. The trace needs to contain an ID, but other information can be included as well, such as time stamps or the size of the referenced block. Mimircache can be used to obtain statistics about the trace and evaluate access patterns and design algorithms, as well as visualize and analyze workloads.

PyMimircache allows the user to run the tracefile with different types of replacement algorithms and compare them. The Mimircache interface has some already implemented algorithms available, but it also allows users to design their own algorithms or make changes to the existing ones. PyMimircache supports several different kinds of calculations, as well as comparing outputs of the trace being run using different algorithms. Below is a list of algorithms that have been implemented in Mimircache:

- Optimal algorithm.
- First In First Out algorithm (FIFO).
- Least Recently Used algorithm (LRU).
- Least Frequently Used algorithm (LFU).
- Most Recently Used algorithm (MRU) (Similar to LRU, but the page with the newest reference is evicted).
- Clock algorithm.
- Random algorithm (Pages for eviction picked randomly).
- Segmented Least Recently Used algorithm (SLRU).
- Adaptive Replacement Cache algorithm (ARC).

Note that it is possible to implement the Optimal algorithm in Mimircache, as it accepts a trace and is able to calculate all its future access patterns, which is not possible in practice.

Mimircache includes a variety of visualization tools, like heatmaps and plots. Visual representation of data can make understanding the information the trace provides easier.

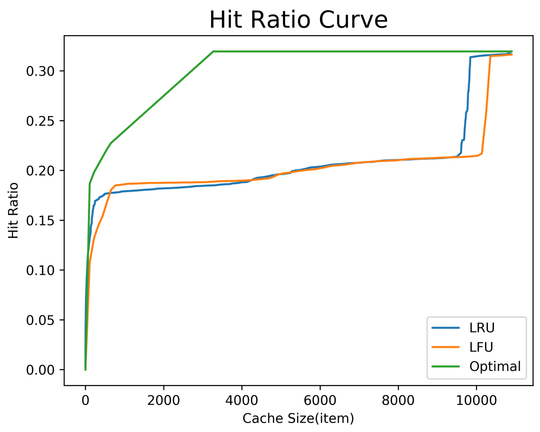
An example of a plot available in Mimircache is the hit ratio curve plot. It compares the hit ratio over cache size of different algorithms when a single trace has been analyzed. Figure 4.1a shows an example of such a plot, comparing the hit ratios of three different replacement algorithms, LRU, LFU, and Optimal, as the size of the cache grows.

Figure 4.1b shows an example of a heatmap in Mimircache. This plot is of the type *hit_ ratio_ start_ time_ end_ time*, which measures the hit ratio at a certain point in time. The scale of the hit ratio can be seen to the heatmap's right. As the hit ratio increases, the colour becomes increasingly warmer, that is, blue means that the hit ratio is low, red means that the hit ratio is high. The x-axis of the heatmap denotes the start time as a percentage of total time, and the y-axis denotes the end time.

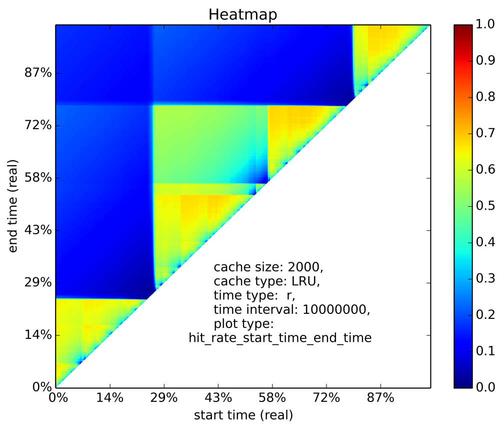
The heatmap is composed of pixels, which each denote how high a hit ratio is. A pixel at (x, y) shows the hit ratio obtained from time x to time y. The entire execution can be observed by looking at the column from $(0\%, 0\%)$ up to $(0\%, 100\%)$. If a time interval between x and y is recorded, the pixel at (x, y) shows the number of hits that occurred divided by the number of requests for that time interval. A pixel effectively shows the average hit rate for the execution from when it was started to when it ended. For example, you want to observe an execution from start time 50% to end time 75%. The hit ratio of the pixel at $(50\%, 75\%)$ would be the number of hits divided by the number of requests for that entire time interval.

The time is measured in percentages of the total run-time. Currently, there are two types of time available in Mimircache, virtual time and real time. Virtual time assumes that the number of requests per time interval is constant, while real time makes no such assumption.

The motivation behind displaying the traces in these heatmaps is that at the beginning of each trace, the hit ratio is quite affected by every single cache hit. In contrast, near the end of the trace, each hit ratio has a much less significant effect on the outcome, because a large number of cache hits is being divided by a large number of requests. Therefore it is interesting to see how different things would be if the trace had started and ended at a different time and measured on different time intervals.



(a) Hit ratio curve plot of different algorithms' performance.



(b) A heatmap showing hit ratio over time.

Figure 4.1: Example hit ratio curve plot and hit ratio heatmap.

5 Related Work

Over the last few decades, there has been much research done on the subject of the cache, particularly cache replacement algorithms. Most of this research is aimed at improving already existing algorithms, whether it is through the means of altering algorithms to improve upon failing access patterns, or to create an entirely new algorithm. This is done to either create a better general algorithm, or to optimize the algorithm to certain circumstances. For example, a general algorithm might perform okay in two different caches, but having two specialized algorithms for each case is preferable. Many replacement algorithm improvements are aimed to enhance LRU, described in section 2.1.2.

We have discussed multiple algorithms that are improvements on LRU in the previous chapters. In this section, other improvements on LRU, that have not been mentioned yet, will be featured. These are algorithms that we either did not have time to look into but are worth investigating in the future, or turned out to be irrelevant to our research upon inspection.

Sai Huang *et al.* [31] proposed the Lazy Adaptive Replacement Cache (LARC) algorithm, which is an improvement on ARC, described in section 2.1.6, and LRU. The goal of LARC is to improve flash-based disk cache, like Solid State Drives (SSD). LARC filters out seldomly accessed blocks and prevents them from entering the cache. As we want the page cache to behave better when working with SSD, this could be an ideal improvement on the page cache, but the page cache should work well with both SSDs and hard disk drives.

In 2004, Jaafar Alghazo, Adil Akaaboune and Nazeih Botros proposed SF-LRU [32], which improved both LRU and LFU by combining them, using the concept of second chance. This is done by comparing the frequency of a block with the block next to it in the set, and the block with lower frequency will be evicted. The idea of this algorithm is similar to ARC since the improvement is to extend the caching of blocks that are more often referenced. The goal of SF-LRU is not directly to improve the page cache, just to make an improved general cache replacement algorithm. SF-LRU could improve the efficiency of the page cache, and is worth investigating in the future.

The Low Inter-reference Recency Set (LIRS) [33] algorithm is an improvement on LRU, created by Song Jiang and Xiaodong Zhang. LIRS uses recent Inter-Reference Recency (IRR) to record history information for each block. IRR of a block is the reuse distance of the block, or the number of other blocks accessed between two consecutive references to the block. LIRS effectively addresses the limits of LRU by using recency to evaluate IRR to make a replacement decisions. By using IRR, the LIRS algorithm often outperforms LRU, among other existing replacement algorithms.

The Clock-Pro [14] algorithm by Feng Chen and Xiaodong Zhang is an improvement on the Clock algorithm. The implementation Clock-Pro uses to improve upon Clock is similar to the implementation LIRS uses to improve LRU. Clock-Pro uses a circular list to store information about recently referenced blocks. Clock-Pro additionally categorizes blocks into hot and cold blocks based on reuse distance. Both Clock-Pro and LIRS are very interesting, and could possibly be used to improve the page cache's performance, but investigating these algorithms further awaits another time.

6 Tracing The Page Cache

To be able to understand how the Linux kernel's page cache currently behaves, adding functionality that creates trace information is needed, which was discussed in section 4.1. Once a tracing mechanism has been added to the kernel, it becomes possible to properly analyze how the cache behaves under different circumstances. To generate these circumstances, FIO is used, as it is able to force the cache to perform operations as requested. After these traces have been created, they may be given as input to a tool that performs analysis, such as Mimircache. In this section we discuss tracing of the page cache in more detail, as well as highlight the challenges that arise when implementing such a mechanism.

6.1 Naïve Approach

The simplest way of implementing such a trace is to find critical locations in the cache's source code and perform a *printk()* operation, which is the print function for kernel messages, to log information about cache activity. The message from the *printk()* operation is then fetched from the message buffer of the kernel. The operation to display this message buffer in console is *dmesg*. In that print operation, basic information about what happened can be detailed, such as information about the page that has just been referenced. After these print operations have been created, different workloads can then be tested by running FIO, and analyzed using Mimircache.

6.2 Challenges of Tracing

The solution proposed in section 6.1 does have some problems. Recall that the Linux kernel contains millions of lines of code. Much of the source code is hardly documented and many functions contain no comments to supply the complicated programming. This makes finding the locations of the relevant cache operations a difficult task. When the programmer's understanding of the kernel source code is limited, it is hard to find documentation that describes the relevant code. Without understanding the functionality well enough, attempting to log information correctly becomes nearly impossible. A method that worked well for us when trying to improve our understanding of the source code is to get a clear picture of how the source code is connected. This was acheived with the help of the Bootlin [34] website, where it is possible to search for a variable or a function, and trace the function calls to help further our understanding of the kernel source code.

The *printk()* operation is slow, and if used to frequently, it can have an effect on the performance of the code under inspection and impede the system's execution [35]. To mitigate this problem, the *trace_printk()* function, that is part of the *ftrace* module, can be used instead of *printk()*. The *trace_printk()* was specially designed to mitigate the overhead by writing into a specific *ftrace* buffer that can be viewed a file instead of the the console output. The specialized *trace_printk()* outperformes *printk()* in terms of speed, as writing into a ring buffer using *trace_printk()* takes around a tenth of a microsecond while *printk()* operations may take several milliseconds.

The kernel has a mechanism, called shrinker, by which the memory management subsystem can request that cached items are discarded and their memory freed for other uses. This is done to prevent the page cache from taking up all available RAM. The actions of the shrinker can have an impact on the cache behaviour and when working with changes in the source code the developer must be aware of this. Two similar traces can yield different outputs because the shrinker changes the size of the page cache.

6.3 The Process of Tracing

There are many ways to obtain traces in the Linux kernel. In this section, our approach to tracing will be described.

The first thing that had to be done was to find the locations where print-operations should be placed, so that we could measure cache hits and cache misses. When a cache miss occurs in the page cache, there is a page fault that needs to be taken care of by a page fault handler. The handler will subsequently fetch the requested data from the disk and place it into the cache. When there is a cache hit, no such operation is required, as data only needs to be fetched from the page cache directly.

Since the page cache is a part of the Memory Management system, we looked at that source code and found files called *swap.c*, and *filemap.c* which contain the main functions used by the cache. The primary functions we looked at are *mark_page_accessed()* and *add_to_page_cache_lru()*. The *mark_page_accessed()* function in file /mm/swap.c measures total cache accesses, and marks cache as having seen activity. Once that function is called, pages get moved around in the active and the inactive lists as follows:

- If a page is inactive and unreferenced, it will become inactive and referenced.
- If a page is inactive and referenced, it will become active and unreferenced.
- If a page is active and unreferenced, it will become active and referenced.
- If a page is active and referenced, it will remain so.

The *add_to_page_cache_lru()* function in file /mm/filemap.c measures additions to the page cache.

Adding tracing to these functions allows us to observe all cache references. When a new page is being added to the page cache the *add_to_page_cache_lru()* function is called. Therefore, a cache miss must have occurred, as it was not present in the cache before this operation was performed. If *mark_page_accessed()* is called, a cache hit must have occurred, and the page must have been present in the page cache before an operation can reference it.

In order to create traces, information about the activity of the page cache had to be gathered. The information that we wanted to fetch was a page ID and a time-stamp. The page ID could be found, as the page is passed into both functions mentioned above as a parameter. The kernel has a time-stamp function built in that we used to see the order in which pages were referenced. We then used *trace_printk()* to print this information to a specific file, which will be referred to as the *tracefile*.

When we had placed print operations in suitable locations, collecting the appropriate information, traces were generated using a FIO jobfile specifying the workload we wanted to test. These tracefiles were then analyzed using Mimircache.

7 Analyzing the Page Cache

In this section, we will feature different experiments that were performed to be able to analyze the behaviour of the page cache. The methods that we used will be explained in further detail, such as specifying the required workload and generation of the traces. The expected outcome of each experiment will be discussed, and results will be shown.

7.1 Experimental Setup

Experiments were run on a Supermicro X8DTU machine with Intel(R) Xeon(R) E5645 @ 2.40GHz processor, on a virtual machine with a QEMU Virtual CPU version 2.5+, RAM size 103MB, on a single core. The virtual machine is running a Linux kernel version 4.15.0+. Each experiment was run by turning on tracing while a workload, specified by a FIO jobfile, was executed, to make sure that traces have as little noise as possible. The cache was flushed (emptied) before each workload was run, so previous entries in the cache will not influence the results of the experiment. The cache size of the virtual machine is around 70MB, but it is impossible to give a specific size, since the cache is dynamically sized because of the shrinker interface mentioned in section 6.2.

To test the behaviour of the cache, files that I/O operations can be performed on need to be generated. Two files were generated using the *dd* Linux command [36], and its input is as such:

```
dd if=/dev/zero of=10Mfile.txt  bs=10M  count=1
```

Where *if* is the input file, *of* is the output file, *bs* is the size of the block read from the input, and *count* denotes how many input blocks should be copied. In this example, we chose to copy one block of size 10MB. The two files we generated were of different sizes, one of size 10MB, and the other of size 200MB. The file sizes were chosen because 10MB is much smaller than the cache size, while 200MB is much larger. The 10MB file will henceforth be referred to as the smaller file, and the 200MB file as the larger file. Since the smaller file will always be smaller than the cache, the entire file will fit into the cache and nothing needs to be evicted during a read of the entire file. Therefore, the replacement policy will not be enforced. The larger file is chosen because it is much larger than the expected cache size, so more pages need to be referenced than is memory for. To make sure these pages get added to the page cache, the replacement policy is enforced.
Below is an example of a FIO jobfile created to perform experiments:

```
[cache-run]
rw=read
ioengine=sync
direct=0
invalidate=0
size=200MB
filename=200Mfile.txt
```

The parameters that were changed while performing experiments were *rw*, which is the I/O operation performed, *size*, which is the size of the file the I/O operation is performed on, and *filename*, which is the name of the file I/O operations were performed on. The I/O operation performed was set to read, write, randread and randwrite, based on what was being tested. The size and filename was changed in accordance with what experiment was being performed.

7.2 Experiment 1: Sequential Reads

In the first experiment, two tracefiles were generated by reading the smaller and the larger files sequentially. The tracefiles were then run in Mimircache to generate hit ratio curve plots and hit ratio heatmaps, so behaviour could be analyzed.
These are our expectations of the performance of different replacement policies run on the page cache:

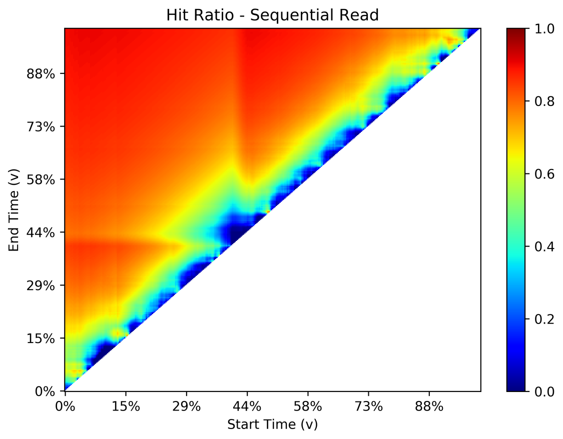
- The Optimal algorithm will have the best performance in all cases. However, it cannot be implemented in practice, but it is included to serve as a reference that other algorithms can be compared to.
- For the smaller file, all algorithms are expected to perform similarly, as the size of the file is smaller than the cache, so replacement policies are not enforced.
- Since ARC is an improvement on LRU, ARC will have better performance than LRU for the larger file.
- LFU is expected to have the worst performance when reading the larger file, as it is not scan-resistant.

We will now discuss our expectations for the hit ratios for both files. For the smaller file, we expect that reading sequentially will generate a high hit ratio, because prefetching from a hard drive makes sequential reading more efficient, and because the entire file fits into the cache. Prefetching is a technique that attempts to guess which pages will soon be needed and obtain it in advance [37]. Due to prefetching, there should be very few cache misses. The larger file is expected to have a lower hit ratio than the smaller file, because it does not fit into the cache. Therefore, more data needs to be read than there is memory for, and more cache misses will occur.

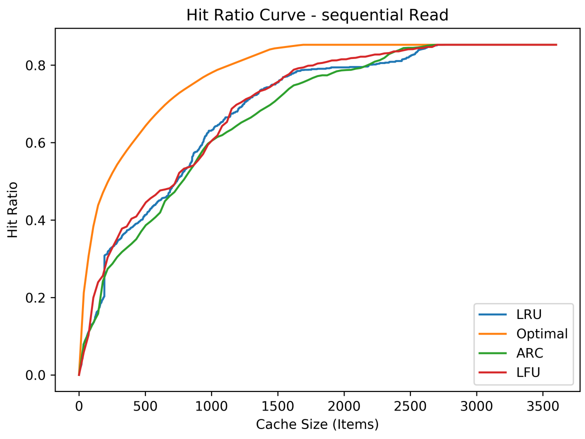
Recall that in section 4.3 we mentioned that the entire execution of an I/O workload can be seen by observing the column between $(0\%, 0\%)$ and $(0\%, 100\%)$. From now on, when we mention that an execution is at $y\%$, we are referring to the coordinate at $(0\%, y\%)$.

Figure 7.1a shows a heatmap generated by executing a read workload on the smaller file. By observing the figure, we can see that the hit ratio is quite low at first, but it increases quickly after around 15% of the execution is complete. The sequential read of the larger file shows a different behaviour. As can be seen in figure 7.1c, the start of the execution of a read workload on the larger file is quite similar to that of the smaller file. However, while the hit ratio for the smaller file remains high, the hit ratio for the larger file starts to decrease after 40% of the execution is done, and continues to do so for the remainder of the execution.

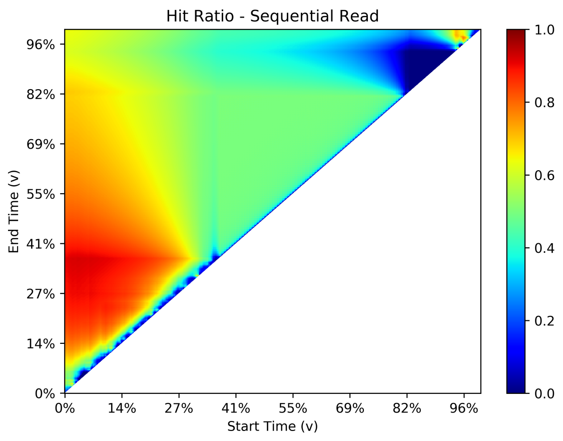
If we look at the hit ratio curve plot in figure 7.1b, we can see that Optimal has the best performance when reading the smaller file. LRU, LFU and ARC all have similar performance to each other. Figure 7.1d shows the performance of algorithms when reading



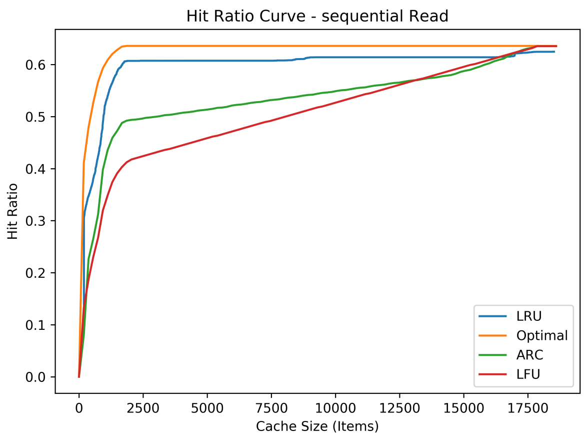
(a) Heatmap: sequential read of smaller file.



(b) Hit ratio curve: sequential read of smaller file.



(c) Heatmap: sequential read of larger file.



(d) Hit ratio curve: sequential read of larger file.

Figure 7.1: Plots and heatmaps of sequential reads for the smaller and the larger files.

the larger file. Optimal still has the best performance. LRU, LFU and ARC perform quite differently from each other. LRU performs well, its hit ratios are close to Optimal's hit ratios. LFU performs a prominently worse than the other algorithms for smaller cache sizes, but as the size increases, so does LFU's performance. ARC has better performance than LFU until cache size grows large, where they perform similarly.

7.3 Experiment 2: Random Reads

In the second experiment, two tracefiles were generated by reading the smaller and the larger files randomly. FIO uses a completely uniform random distribution when asked to perform random I/O, where each page is only accessed once.

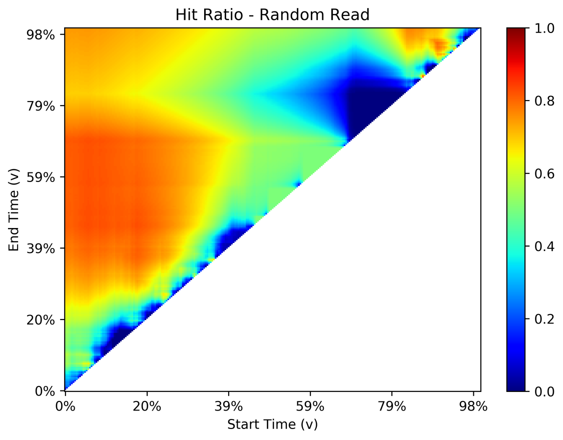
Listed below are our expectations of the performance of different replacement policies run on the page cache:

- The Optimal algorithm will still have the best performance in all cases.
- For the smaller file, all algorithms are expected to perform similarly, as the size of the file is smaller than the cache, so replacement policies are not enforced.
- Since ARC is an improvement on LRU, ARC will have better performance for the larger file.
- Because the file is read randomly and each block is only read once, LFU is expected to have bad performance.

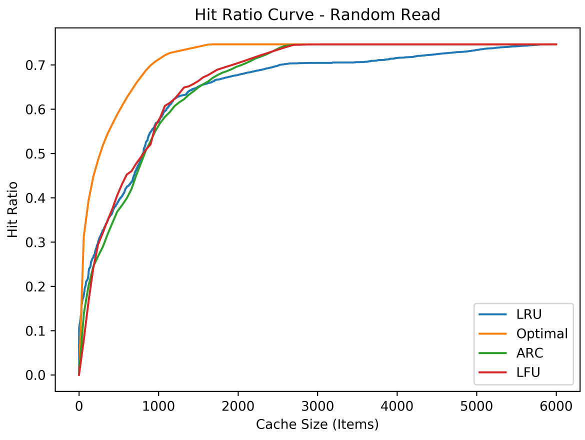
For the smaller file, we expect that reading randomly will generate a lower hit ratio than if the file was read sequentially, because prefetching does not function in favour of random access patterns. Likewise, we expect hit ratios to be lower for random reads of the larger file than for sequential reads.

Figure 7.2a shows a heatmap generated by running a random read workload on the smaller file. From the start, its hit ratio gradually increases until around 65% of the execution is completed. After that, the hit ratio decreases until around 80%, and then it starts to increase again until the execution is complete. The heatmap generated by running random read workload on the larger file can be seen in figure 7.2c. The hit ratio steadily increases from the start until around 15% of the execution is done, when it evens out. The hit ratio remains consistent for the rest of the execution.

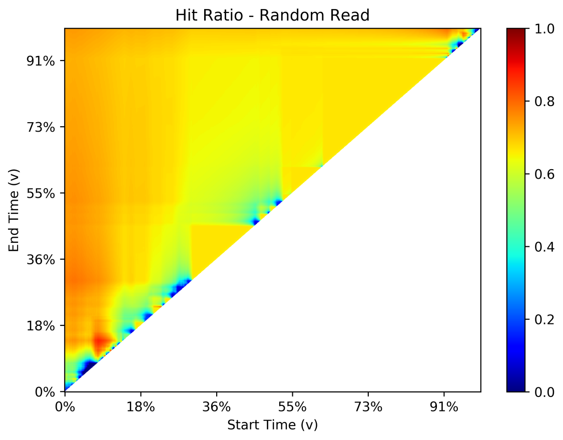
If we look at figure 7.2b, we can see that Optimal has the best performance when reading the smaller file. LRU, LFU and ARC all have similar performance to each other. Figure 7.2d shows a hit ratio curve plot for a tracefile generated by running a random read workload on the larger file. It shows the Optimal has the best performance. LRU, LFU and ARC all have similar performance for smaller cache sizes, but LFU's and ARC's performance increases as the cache size grows.



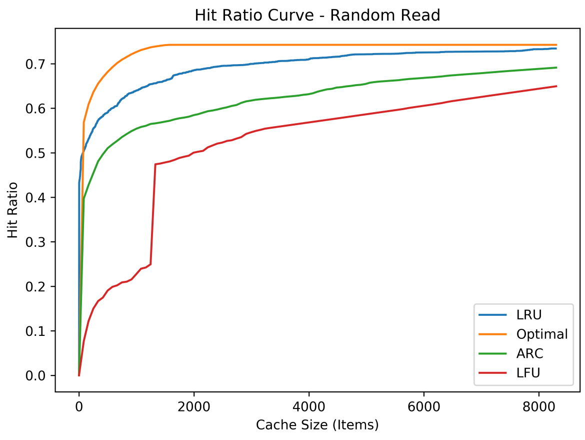
(a) Heatmap: random read of smaller file.



(b) Hit ratio curve: random read of larger file.



(c) Heatmap: random read of larger file.



(d) Hit ratio curve: random read of larger file.

Figure 7.2: Plots of random reads for the smaller and larger files.

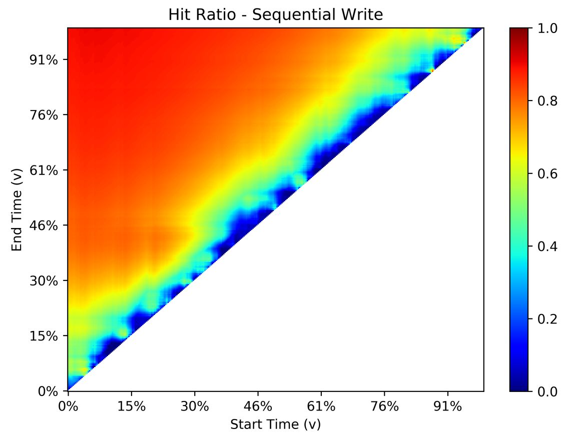
7.4 Experiment 3: Writes

In the third experiment, the smaller file was written randomly and sequentially, and analyzed using Mimircahce graphs. A FIO jobfile was created to simulate this workload. The expected outcome of this experiment is that both tracefiles, for sequential and random writes, will have a low hit ratio, as the pages being written do not exist in the cache.

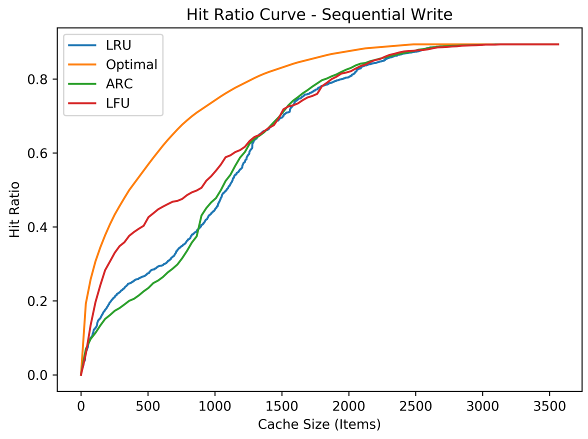
Figure 7.3a shows a heatmap generated by running a write workload sequentially on the smaller file. The hit ratio steadily increases throughout the execution, until it becomes consistently high at around 60%, and remains as such until the execution is complete. Figure 7.3a shows a heatmap generated by running a write workload randomly on the smaller file. The hit ratio steadily increases from the beginning of the execution to around 60%, where it starts decreasing again.

The hit ratio curve plot generated by running a write workload sequentially on the smaller file can be seen in figure 7.3b. Optimal has the highest hit ratio. For smaller cache sizes, LFU performs better than LRU and ARC, but as the cache size increases, performance of all algorithms becomes very similar.

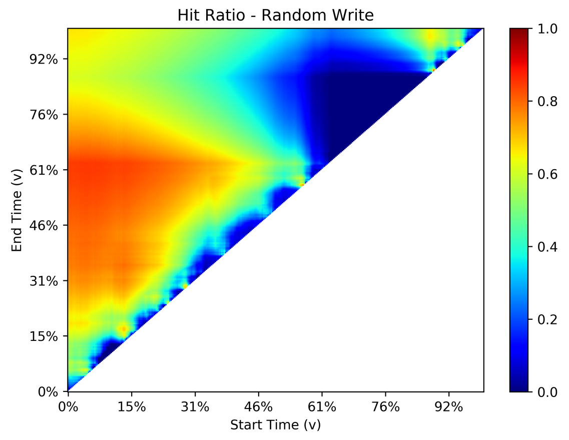
Figure 7.3d shows the hit ratio curve plot generated by running a write workload randomly on the smaller file. Optimal has the highest hit ratio. For smaller cache sizes, LFU has a higher hit ratio than LRU and ARC. As the cache size approaches 2000, the three algorithms perform similarly, but as the cache size increases, LRU starts performing worse than the others.



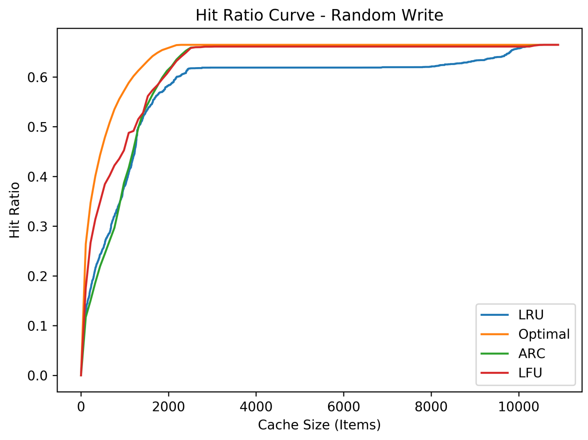
(a) Heatmap: sequential write of smaller file.



(b) Hit ratio curve: sequential write of smaller file.



(c) Heatmap: random write of smaller file.



(d) Hit ratio curve: random write of smaller file.

Figure 7.3: Plots of sequential and random writes for the smaller file.

8 Discussion

In this section, we will be discussing the results we obtained in section 7 and whether or not our expectations were met. We will also discuss shortcomings of the experiments, and how the experiment process could potentially be improved.

8.1 Results From Experiments

First off, we will take a look at the results from Experiment 1 on sequential reads, which can be seen in figures 7.1a and 7.1c. The results are similar to what was expected. The heatmap for the smaller file's sequential read shows that hit ratio increases very quickly. This is because the page cache is scan-resistant. The algorithm realizes what kind of workload it is dealing with, and quickly adapts by prefetching data it thinks will be referenced. It helps that the entirety of the smaller file fits inside of the cache, nothing has to be evicted so the hit ratio remains high. It is worth noting that when the execution is at around 40%, the hit ratio decreases for a while. We are not certain of why this happened, but it might be that the operating system performed another operation that interfered with the execution of the cache.

An execution similar to that of the smaller file can be observed in the heatmap for the larger file's sequential read. The hit ratio is low at the beginning, until prefetching starts to pay off and then the hit ratio becomes higher. The hit ratio remains high until the cache runs out of memory and needs to start evicting pages to be able to add new pages. After that, the hit ratio steadily decreases until the execution is complete.

If we compare the hit ratio curve plots in figures 7.1b and 7.1d, we see a drastic difference in the performance of algorithms. Optimal always performs best, as was expected. However, the other algorithms behave very differently if the size of the file is changed. First, we note that the scale on the x-axis differs between plots. The x-axis in figure 7.1b ranges from 0 to 3500, while the x-axis in figure 7.1d ranges from 0 to 17500. That is because Mimircache stops extending the axes of the plots when the algorithm's performance has reached a balance in hit ratio. By looking at the plot for the sequential read of the smaller file, we see that all the algorithms have very similar performance. This is because the file fits into the cache and the replacement policies are never enforced. However, there is a big difference in performance of algorithms when analyzing the larger file, where the replacement algorithm is enforced. We can see that LRU performs much better than ARC, which surprised us, as we thought the opposite would happen. However, we realized that ARC was implemented to improve upon the weak locality access patterns of LRU, that do not occur when the read is sequential, so this behaviour makes sense. LFU did not performed poorly, as we expected. This is normal behaviour, as no page is frequently accessed.

Now we will look at the results from Experiment 2 of the random reads. Looking at figures 7.2a and 7.2c, we can see that their behaviour is similar to what was expected. Randomly reading the smaller file will have similar results to its sequential counterpart, because the entire file fits in the cache and the prefetching mechanism will still fetch pages it think will be referenced soon. These prefetched pages will remain in the cache, and even though they are not referenced right away, they will be eventually. Therefore, reading the smaller file randomly has fairly good performance. Randomly reading the larger file, however, has a lower hit ratio than its sequential counterpart. This was also to be expected, since we do not have the advantage of prefetching when randomly reading the larger file. Prefetched pages might be referenced soon after they are placed into the cache, but it is also possible that they remain unreferenced and eventually age out of the cache. Those pages then have to be fetched again later, when they are referenced directly.

We must be cautious when making assumptions about random access patterns. The example of a large file read randomly provided in section 7.3, is only the result of one tracefile, so it does not give a good representation of the cache's behaviour. The tracefile generated for this experiment may have been "lucky" and accidentally gotten a high hit ratio, or it may have been "unlucky" and the hit ratio is lower than usual. Therefore, this example must not be made into a representation of how every run of random reading should look like. Multiple runs are needed to make an assertion on how random operations behave on average.

Now we will compare the hit ratio curve plots displayed in figures 7.2b and 7.2d. Notice that due to the smaller file being small enough to fit into the cache, the hit ratio curve plots for random reads and sequential reads look quite similar. In both cases, the highest hit rate for all algorithms is achieved when the size of the cache is at around 2000, although that hit rate is slightly lower for the random read, but that was to be expected.

What is more interesting is the fact that the hit ratio curves for the larger file also looks very similar for both sequential and random reads, considering the difference in scale on the x-axis. When the cache size is around 2000, the order of algorithms with respect to performance is the same. Optimal performs best, then LRU, then ARC, and lastly LFU, which has the worst performance in both cases. LRU even seems to perform better for the random reads than it does for the sequential reads as the cache size increases. However, since the file is being read randomly, this could be a "lucky" execution for LRU. The main difference between the plots is that LFU performs much worse when read randomly, which is understandable, as random reading in FIO only references each page once, which is disadvantageous for LFU.

Lastly, we will discuss the results from Experiment 3. The heatmap generated by writing the smaller file sequentially can be seen in figure 7.3a, and the heatmap generated by writing the smaller file randomly can be seen in figure 7.3c. Notice that the heatmap generated for sequential writes of the smaller file is very similar to the heatmap generated for sequential read of the same file, but has a slightly lower hit ratio. The same applies to the heatmaps for random reads and writes. This seemed like weird behaviour, as we expected that the hit ratio would be much lower than it turned out to be. We have two possible explanations that might be the cause for this higher hit ratio.

Firstly, when write operations occur every page is read into a buffer, called write buffer, which is then flushed down to the disk. That would lead to more cache hits because the same page is referenced more than once: at least once when it is placed into the buffer, and again when flushed down to the disk.

Secondly, it might be possible that Mimircache does not automatically make a distinction between read and write operations. The predefined algorithm might think that the tracefile that is being analyzed, generated by performing a write workload, is full of read operations. As of right now, we have not found any parameter that could be altered to make this distinction. However, the example *.csv* file displayed on the Mimircahce Github repository has an additional header named *op*, which can have two values, one represents read operations and the other write operations. This could have an effect on the output of Mimircache, but as we have not tested ourselves, we cannot make any statements about that. We did not specify what operation was going to be performed, so this is a likely explanation for the unexpectedly high hit ratio.

At this moment, we are not willing to say anything for certain. Further investigation on this behaviour is required for us to make any definitive statements. However, write operations are less relevant to our research than read operations. The page cache is primarily used to decrease latency when reading pages. When a write operation is performed, the page is placed into the cache and should not be fetched until the modifications have been written to the disk. Write operations for file level I/O is something that needs to be looked at in the context of application requests.

8.2 Tracing and Mimircache

We have come to realize that using Mimircache for analyzing data from the Linux kernel's page cache may not be ideal right now, because the replacement algorithms readily available in Mimircache are not the same as the one used by the page cache. The Mimircache interface allows the user to implement their own algorithm. To be able to get a more accurate representation of how the page cache operates, we will have to write a new algorithm in Mimircache that simulates the one in the page cache.

As has been discussed, the Linux kernel is extremely complicated and devising a completely new replacement algorithm would take much time, time that could potentially be wasted if it turns out that the performance of the replacement algorithm is worse than the current one. Mimircache could be used as a proof-of-concept to see if the new replacement algorithm works better than the current one.

The method we used to implement our traces for the experiments was described in section 6.3. Another way that traces could be implemented is logging where page faults occur. If a page fault happens, there is a cache miss, and the page will need to be fetched from the disk. This is implemented in a page fault handler. Finding cache hits and cache misses directly would allow us to perform statistical analysis without using Mimircache, and therefore getting more accurate results with respect to the current behaviour of the replacement policy.

9 Future Work

We have come up with several potential improvements to the page cache that might improve its efficiency. Below, a few of our ideas are listed. In this section, these ideas will be explained.

- Eviction of huge pages.
- Sizes of the inactive and the active lists.
- Attempt to predict the future of a page's activity.

Since support for huge pages is a recent addition in the page cache, proper eviction has not been implemented yet. Huge pages are pages that are larger than regular pages [38]. Regular pages are of size four kilobytes, while huge pages are usually two megabytes. When a huge page is added to the page cache, it is split into many subpages (which all have the same size as a regular page). A Radix tree is used to track down a subpage inside of a huge page. When a huge page reaches the end of the inactive list, only one subpage gets evicted, instead of the whole huge page. It may be possible to improve upon this behaviour, and allow huge pages behave like a regular page in the page cache. The Radix tree could be used to find all subpages of a huge page, so they can all be removed in one operation. Improving the behaviour of huge pages could save time when a new page needs to be added, as multiple regular pages can be added in place of one huge page.

Another possible improvement on the page cache is to alter the size of the active and the inactive lists, which currently have no predetermined size. We would like to try to add size constraints and see what sort of an effect that would have. For example, we could set a maximum size to each list, or require one list to be larger than the other. A possible downside to this is if a page is referenced more than once in a short time span, and then never again, it could remain longer in the inactive list than it otherwise would have. Experimentation would be required to see if this has an impact on the cache under real circumstances.

It may be possible to make the cache learn. The idea is to predict future access patterns based on past experiences. Listed below are examples of situations where a learning cache could be useful:

- When a page is referenced, the cache can infer that this page will be used again in the near future and add it immediately to the active list instead of the inactive list.
- If a page is referenced twice and the cache can infer that the page will age out of the cache, let the page stay in the inactive list and get evicted earlier. There would be no point in allowing the page to be upgraded to the active list, if it will not be referenced again in the near future.
- The cache can infer that a referenced page will only be referenced once, so the page will not be added to the cache.
- If page k is referenced, the cache can infer that it is likely that page $k + 1$ will be referenced sometime in the near future, and fetches it immediately.

The idea of a learning cache would require careful implementation, as the algorithm must be compact to not take memory away from the cache itself.

10 Conclusion

In this report, we have discussed how latency in computing is becoming an increasingly big problem in today's world, and improving caching, even by just a little bit, could have a great impact on performance. During the course of this project, we got intimately familiar with the Linux kernel, the page cache in particular, which was hard to get a handle on because of its cryptic documentation and intricate source code. We were able to effectively create tracefiles that were analyzed by Mimircache, and draw conclusions about the behaviour of the current state of the page cache. While we did not manage to make any improvements on the Linux kernel's page cache, we proposed some ideas that might have an impact on the cache's performance, which we will be able to investigate further at a later date.

Finally, we have managed to piece the components of the page cache together, and document these findings in a coherent manner. We believe that the contents of this report will serve as a guide to others that want to get familiar with the Linux kernel's page cache, but are not sure where to start. Hopefully, our work will aid future Linux kernel programmers as they take their first steps.

Bibliography

[1] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing Web Latency: The Virtue of Gentle Aggression", *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 159–170, Aug. 2013, ISSN: 0146-4833. DOI: 10.1145/2534169.2486014. [Online]. Available: http://doi.acm.org/10.1145/2534169.2486014.

[2] C. Tang, "Cache System Design in the Tightly Coupled Multiprocessor System", in *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, ser. AFIPS '76, New York, New York: ACM, 1976, pp. 749–753. DOI: 10.1145/1499799.1499901. [Online]. Available: http://doi.acm.org/10.1145/1499799.1499901.

[3] "Kernel Definition.", 2005, [Online]. Available: http://www.linfo.org/kernel.html. [Accessed: May 1, 2018].

[4] "Preemption Under Linux.", 2017, [Online]. Available: https://kernelnewbies.org/FAQ/Preemption. [Accessed: May 1, 2018].

[5] "Linux kernel source code", [Online]. Available: https://github.com/torvalds/linux. [Accessed: May 9, 2018].

[6] R. Love, *Linux Kernel Development*, 3rd. Addison-Wesley Professional, 2010, ISBN: 0672329468, 9780672329463.

[7] "Interactive Map of Linux Kernel", [Online]. Available: www.makelinux.com/kernel_map. [Accessed: April 16, 2018].

[8] L. Belady, "A Study of Replacement Algorithms for a Virtual-storage Computer", *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, Jun. 1966, ISSN: 0018-8670. DOI: 10.1147/sj.52.0078. [Online]. Available: http://dx.doi.org/10.1147/sj.52.0078.

[9] D. Peter, "The Locality Principle", *Commun. ACM*, vol. 48, no. 7, pp. 19–24, Jul. 2005, ISSN: 0001-0782. DOI: 10.1145/1070838.1070856. [Online]. Available: http://doi.acm.org/10.1145/1070838.1070856.

[10] A. Tanenbaum, *Modern Operating Systems*, 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007, ISBN: 9780136006633.

[11] S. Jiang and X. Zhang, "Making LRU friendly to weak locality workloads: a novel replacement algorithm to improve buffer cache performance", *IEEE Transactions on Computers*, vol. 54, no. 8, pp. 939–952, Aug. 2005, ISSN: 0018-9340. DOI: 10.1109/TC.2005.130.

[12] A. Blankstein, S. Sen, and M. Freedman, "Hyperbolic Caching: Flexible Caching for Web Applications", in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA: USENIX Association, 2017, pp. 499–511, ISBN: 978-1-931971-38-6. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/blankstein.

[13] ——, "Hyperbolic Caching: Flexible Caching for Web Applications", in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '17, Santa Clara, CA, USA: USENIX Association, 2017, pp. 499–511, ISBN: 978-1-931971-38-6. [Online]. Available: `http://dl.acm.org/citation.cfm?id=3154690.3154738`.

[14] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement", in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05, Anaheim, CA: USENIX Association, 2005, pp. 35–35. [Online]. Available: `http://dl.acm.org/citation.cfm?id=1247360.1247395`.

[15] R. Karedla, S. Love, and B. Wherry, "Caching Strategies to Improve Disk System Performance", *Computer*, vol. 27, no. 3, pp. 38–46, Mar. 1994, ISSN: 0018-9162. DOI: `10.1109/2.268884`. [Online]. Available: `http://dx.doi.org/10.1109/2.268884`.

[16] N. Megiddo and D. Modha, "Outperforming LRU With an Adaptive Replacement Cache Algorithm", *Computer*, vol. 37, no. 4, pp. 58–65, Apr. 2004, ISSN: 0018-9162. DOI: `10.1109/MC.2004.1297303`. [Online]. Available: `http://dx.doi.org/10.1109/MC.2004.1297303`.

[17] ——, "ARC: A Self-Tuning, Low Overhead Replacement Cache", in *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, ser. FAST '03, San Francisco, CA: USENIX Association, 2003, pp. 115–130. [Online]. Available: `http://dl.acm.org/citation.cfm?id=1090694.1090708`.

[18] J. Moellenkamp, "Some Insight Into the Read Cache of ZFS - or: The ARC.", 2009, [Online]. Available: `http://www.c0t0d0s0.org/archives/5329-Some-insight-into-the-read-cache-of-ZFS-or-The-ARC.html`. [Accessed: May 1, 2018].

[19] E. Azar and M. Alebicto, in *Swift Data Structure and Algorithms*, 1st, Birmingham, UK, UK: Packt Publishing, 2016, ch. Trie tree, ISBN: 1785884506, 9781785884504.

[20] M. Velikov, C. Hellwig, N. Piggin, and K. Khlebnikov, "radix-tree.h", 2012, [Online]. Available: `https://github.com/torvalds/linux/blob/eb93d973/include/linux/radix-tree.h`. [Accessed: May 2, 2018].

[21] C. Silvers, "UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD", in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '00, San Diego, California: USENIX Association, 2000, pp. 54–54. [Online]. Available: `http://dl.acm.org/citation.cfm?id=1267724.1267778`.

[22] "mmap(2) - Linux manual page.", [Online]. Available: `http://man7.org/linux/man-pages/man2/mmap.2.html`. [Accessed: April 30, 2018].

[23] R. van Riel, "Page Replacement in Linux 2.4 Memory Management", in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, Berkeley, CA, USA: USENIX Association, 2001, pp. 165–172, ISBN: 1-880446-10-3. [Online]. Available: `http://dl.acm.org/citation.cfm?id=647054.715629`.

[24] R. Bryant and D. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 2nd. USA: Addison-Wesley Publishing Company, 2010, ch. VM as a Tool for Caching, ISBN: 0136108040, 9780136108047.

[25] D. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd. United state of America: Oreilly & Associates Inc, 2005, ISBN: 0596005652.

[26] J. Corbet, “Trees I: Radix Trees”, 2006, [Online]. Available: https://lwn.net/Articles/175432/. [Accessed: April 28, 2018].

[27] “Workingset Detection.”, 2013, [Online]. Available: https://github.com/torvalds/linux/blob/973a5291/mm/workingset.c#L154. [Accessed: April 24, 2018].

[28] J. Axboe, “Fio - Flexible I/O Tester rev. 3.6.”, [Online]. Available: http://fio.readthedocs.io/en/latest/fio_doc.html. [Accessed: April 26, 2018].

[29] “Workload definition”, [Online]. Available: https://searchdatacenter.techtarget.com/definition/workload. [Accessed: May 10, 2018].

[30] J. Yang, “PyMimircache”, [Online]. Available: http://pymimircache.readthedocs.io/. [Accessed: April 3, 2018].

[31] S. Huang, Q. Wei, D. Feng, J. Chen, and C. Chen, “Improving Flash-Based Disk Cache with Lazy Adaptive Replacement”, *Trans. Storage*, vol. 12, no. 2, 8:1–8:24, Feb. 2016, ISSN: 1553-3077. DOI: 10.1145/2737832. [Online]. Available: http://doi.acm.org/10.1145/2737832.

[32] J. Alghazo, A. Akaaboune, and N. Botros, “SF-LRU Cache Replacement Algorithm”, in *Proceedings of the Records of the 2004 International Workshop on Memory Technology, Design and Testing*, ser. MTDT ’04, Washington, DC, USA: IEEE Computer Society, 2004, pp. 19–24, ISBN: 0-7695-2193-2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1078036.1079737.

[33] S. Jiang and X. Zhang, “LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance”, *SIGMETRICS Perform. Eval. Rev.*, vol. 30, no. 1, pp. 31–42, Jun. 2002, ISSN: 0163-5999. DOI: 10.1145/511399.511340. [Online]. Available: http://doi.acm.org/10.1145/511399.511340.

[34] “Bootlin, Elixir Cross Referencer”, [Online]. Available: https://elixir.bootlin.com/linux/latest/source/mm. [Accessed: May 3, 2018].

[35] S. Rostedt, “Debugging the Kernel Using Ftrace”, 2009, [Online]. Available: https://blogs.dropbox.com/tech/2012/10/caching-in-theory-and-practice/. [Accessed: May 5, 2018].

[36] “dd(1) - Linux manual page.”, 2017, [Online]. Available: http://man7.org/linux/man-pages/man1/dd.1.html. [Accessed: April 28, 2018].

[37] A. Smith, “Cache Memories”, *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, Sep. 1982, ISSN: 0360-0300. DOI: 10.1145/356887.356892. [Online]. Available: http://doi.acm.org/10.1145/356887.356892.

[38] J. Corbet, “Transparent huge Pages in 2.6.38”, 2011, [Online]. Available: https://lwn.net/Articles/423584/. [Accessed: May 10, 2018].