RE-REFERENCE INTERVAL PREDICTION SWAP POLICY IN THE LINUX KERNEL

An Undergraduate Research Scholars Thesis

by

ANDREW SINGER

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Paul Gratz

May 2017

Major: Computer Engineering

TABLE OF CONTENTS

ABSTRACT

Re-Reference Interval Prediction Swap Policy in the Linux Kernel

Andrew Singer
Department of Electrical & Computer Engineering
Texas A&M University

Research Advisor: Dr. Paul Gratz
Department of Electrical & Computer Engineering
Texas A&M University

Modern computing systems are placing ever greater pressure on their memory management systems. The current means of managing the page cache in the Linux kernel is a binary ranking standard through which cached pages are stored either in an active list or an inactive list and managed by an approximation of a least recently used (LRU) algorithm. Recent endeavors in processor caching have revealed the opportunity for increased performance resulting from refining LRU memory management algorithms. I sought to determine the feasibility of replacing the current pseudo-LRU page cache system with one based on re-reference interval prediction (RRIP). This was achieved this by exploring the current Linux kernel to understand how exactly the page cache is managed, programming and implementing the custom RRIP page level replacement policy, and performing benchmark tests to determine the change in performance of the new system. Conducting this research determined the potential of RRIP in the Linux kernel's page cache system to be at least on par with the current architecture and paved the way for future opportunities to expand and refine RRIP in the context of the page caching.

NOMENCLATURE

DRAM	Dynamic Random-Access Memory
LRU	Least Recently Used
NRU	Not Recently Used
OS	Operating System
RRIP	Re-Reference Interval Prediction
RRPV	Re-Reference Predictor Value

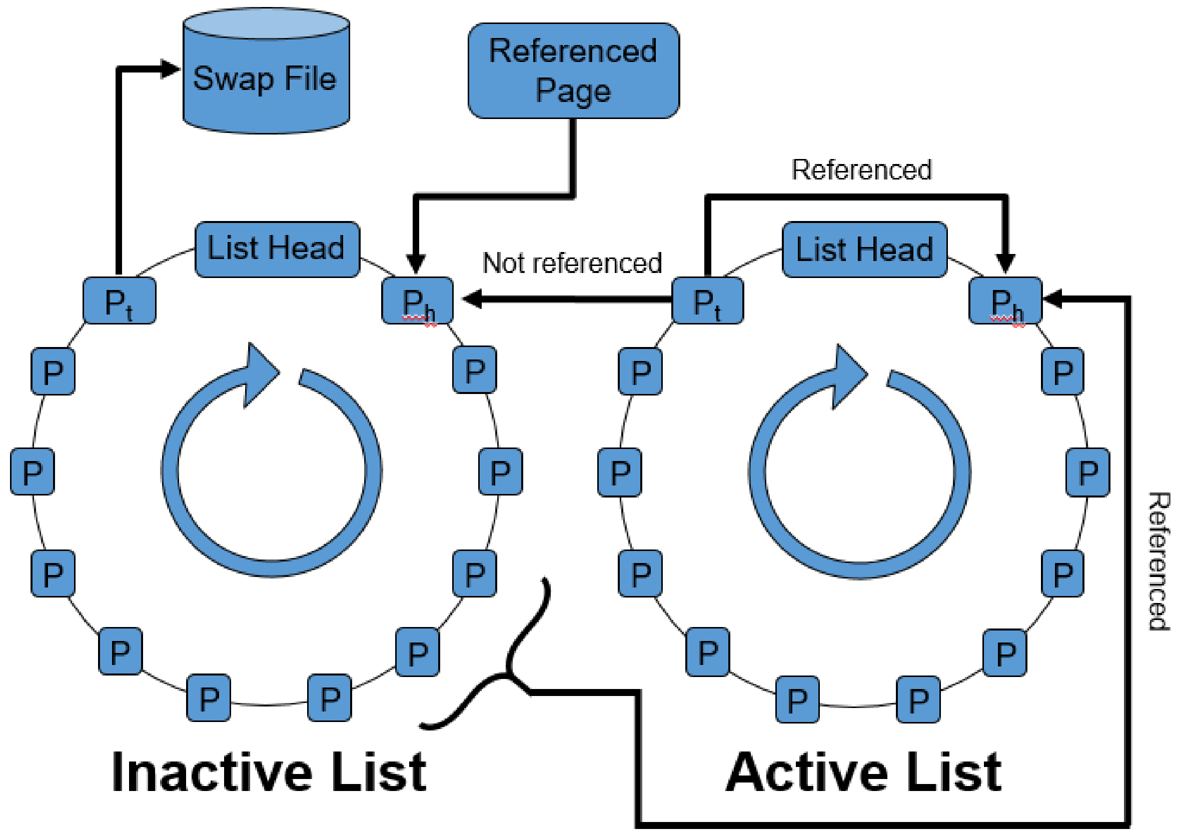
CHAPTER I

INTRODUCTION

The Linux kernel has always been one of the most powerful tools for research on behalf of the extent to which it has been developed over the past few decades and its availability for use in the hands of the public. One area of research in which it has been very instrumental is the architecture and functionality of memory systems. Direct access to the Linux kernel allows developers and researchers to make changes to the source code to evaluate the efficacy of certain policies within a computer's memory management system. Page caching [3], a relevant topic within this realm of study, is the management of certain clusters of memory in the DRAM on the magnitude of pages that the kernel is likely to access again in a relatively short amount of time. I have chosen to delve into this area of study because the system that currently governs the Linux kernel's page cache has not incurred significant architectural changes in quite some time and could be improved with the introduction of recently discovered optimizations in memory systems research.

Replacement Policy Design

Applications these days are using more and more memory, and with the advent of higher complexity memory systems, using page swapping as a means of caching draws attention to the realm of page level replacement policy for possible solutions. However, it has been quite some time since Linux has made significant adjustments to its brand of this policy. Upon progressing from version 2.2 to 2.4, nearly 15 years ago, the kernel saw many changes associated with its page management such as the implementation of page aging and the introduction of the active/inactive list system [8]. Under this new system, cached pages are stored either in an active list or an inactive list and managed by an approximation of a least recently used (LRU) algorithm visualized in Figure 1. The kernel's unique architecture of doubly linked lists contributes significantly to the way in which the active and inactive lists are structured and managed [1]. This overarching method of using more than one list for memory management emerged a few decades ago, and the Linux kernel's instance of this structure is quite similar to that of the two queue (2Q) algorithm proposed by Johnson and Shasha [6]. However, there are a few aspects of the kernel's design that cause it to differ fundamentally from the 2Q algorithm.



[Figure 1 –Linux kernel's active/inactive list management system]

Current Page Caching Algorithm

When a page is first accessed, it is added to the inactive list on account of many file accesses only occurring once in a relatively long period of time. However, after the first access request to a page on the inactive list, the page in question will be promoted to the active list, where it will be managed on a pseudo-LRU basis [9].This page management system is does not abide by true LRU standards, because pages on the active list are not eligible for promotion to its head until they reach the list's tail. This means that at any point in time, the most recently used page may not be at the head of the list. Once pages on the active list reach the tail of this list, the kernel determines whether or not the page has been accessed since it was last added to the front of the list. If the page has been accessed since then, it is added to the head of the active list again. Otherwise, it is added to the head of the inactive list, where it will reside until it is either evicted at the tail or promoted to the head of the active list upon being accessed again. According to this architecture, when the page cache needs to free up space, it will only evict pages at the end of the inactive list. Pages removed from memory at this point are sent to the swap file on the disk. By nature of the way the kernel manages these lists, it is safe to say that while the structure of the page cache is similar to that of 2Q, the management methodology is more similar to that of a Clock algorithm [4].

Potential for RRIP

This version of Linux that upgraded the page caching system in the aforementioned ways reaped several improvements to the system's overall performance because of its improved memory management. While this page level replacement policy is more effective than the previous one, additional developments in the field of memory systems suggest that there are performance improvements to be reaped by adjusting the swapping algorithm to favor pages with slightly prolonged, yet regular re-reference patterns. A newer style of replacement policy in the realm of processor caching named re-reference interval prediction (RRIP) has emerged, and such a policy implemented in page caching could prove to be an arguably preferable alternative to the existing pseudo-LRU system utilized in the current version of Linux [5]. On behalf of the improvements in performance reported by the research regarding RRIP policies, I sought to evaluate the feasibility and efficacy of implementing such a system in the page cache of the Linux kernel.

Further Endeavors in Page Caching

In past years, researchers have sought to improve performance in page caching in many different ways. On one hand, some researchers seek to restructure the page cache's architecture like with the development of the multi-queue replacement algorithm [10]. On the other hand, especially in recent years, some researchers pursue advances in perfecting the page replacement policy. In recent years, the field of memory systems has seen many publications seeking to improve upon specific types of access patterns and memory usage. However, many of the design paradigms utilized in my project most similarly align with those adopted by Megiddo and Modha in the development of their Adaptive Replacement Cache Algorithm (ARC) [7]. This algorithm, also designed on the magnitude of pages, contains two page lists managed in different ways to capture different types of pages that the cache cares about keeping. The distinction of my algorithm is the introduction of RRIP into the existing Linux page cache in a way that capitalizes on the existing efficiencies and the potential improvements.

CHAPTER II

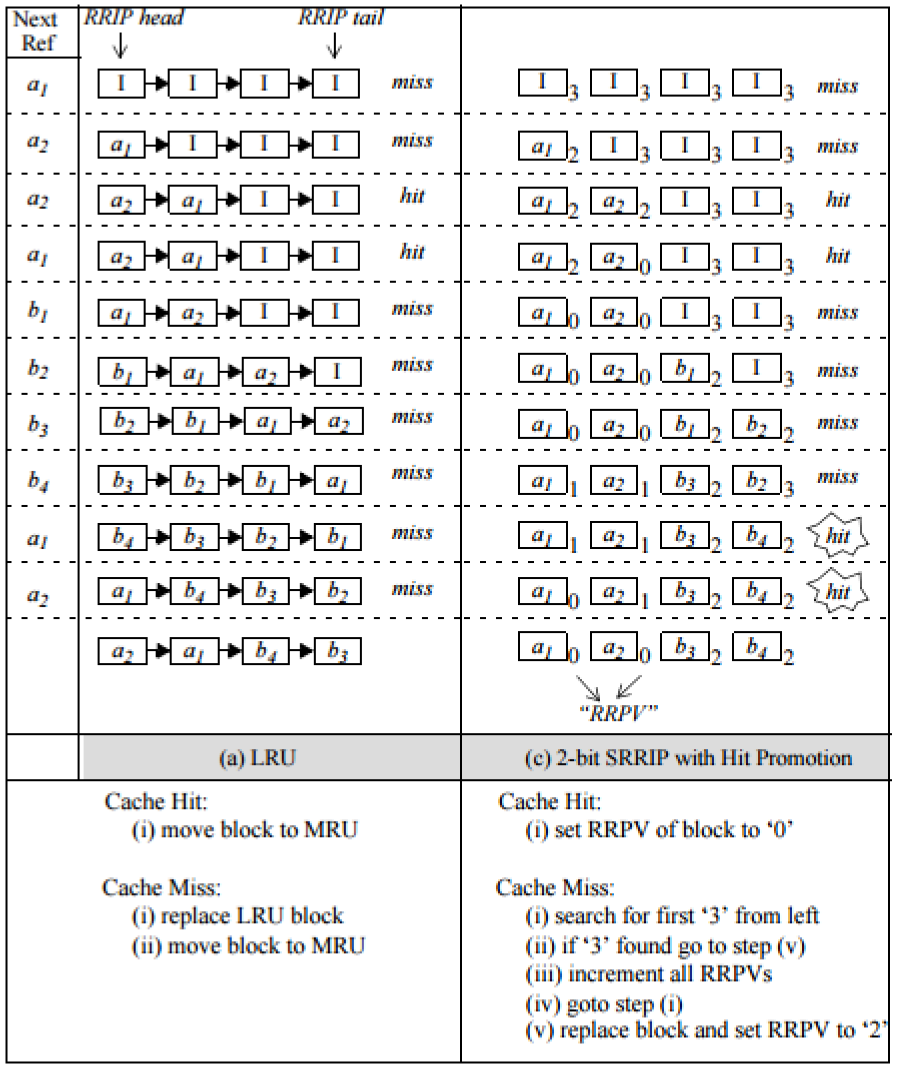
METHODS

Literature Review

The very first step of getting started in this project was the literature review. The preliminary motivations for the literature review in this case were twofold. First of all, it was necessary to know about the existing endeavors in memory systems especially in recent years to ensure the uniqueness of this project. In the event that other researchers had conducted similar experiments, it was important to use their procedures and results not only to inform the way in which I approached my project, but also to determine any adjustments that needed to be made to the very question that I was hoping to answer. Secondly, exploring developments in the research of memory systems allowed me to gain a greater understanding of how to address complications which may have arisen throughout the project. It also helped me figure out how to most effectively invest my focus in the experiments I was performing.

RRIP Research

One aspect of the literature review was more significant than the rest in this case, however. The foundation of this project was the implementation of a RRIP style algorithm in the Linux kernel's page management system, and this started with having a deep understanding of how RRIP works. Consequently, spending quite a bit of time delving into the associated journal article [1] allowed me to achieve this. RRIP is a processor cache policy that attempts to improve computing performance by effectively predicting the re-reference patterns of data stored in processor caches. The basic notion is that memory access requests exhibit patterns that do not necessarily abide by simple policies such as LRU and NRU. These policies, while slightly different, both function on the notion that information which has been used most recently is always more likely to be used again than other information in the cache. However, this is not always the case, and the RRIP policy aims to improve upon such shortcomings. RRIP gives a great deal of favor to information that has been used more than once and continues to be reused in a reasonably short period of time. Data collected from tests conducted on the RRIP policy pointed to its effectiveness in processor caching and suggests the same results may be possible on the magnitude of page caching. Figure 2 shows a trace of the RRIP algorithm which I used in the design of my custom kernel.



Cache Hit:
(i) move block to MRU

Cache Miss:
(i) replace LRU block
(ii) move block to MRU

Cache Hit:
(i) set RRPV of block to '0'

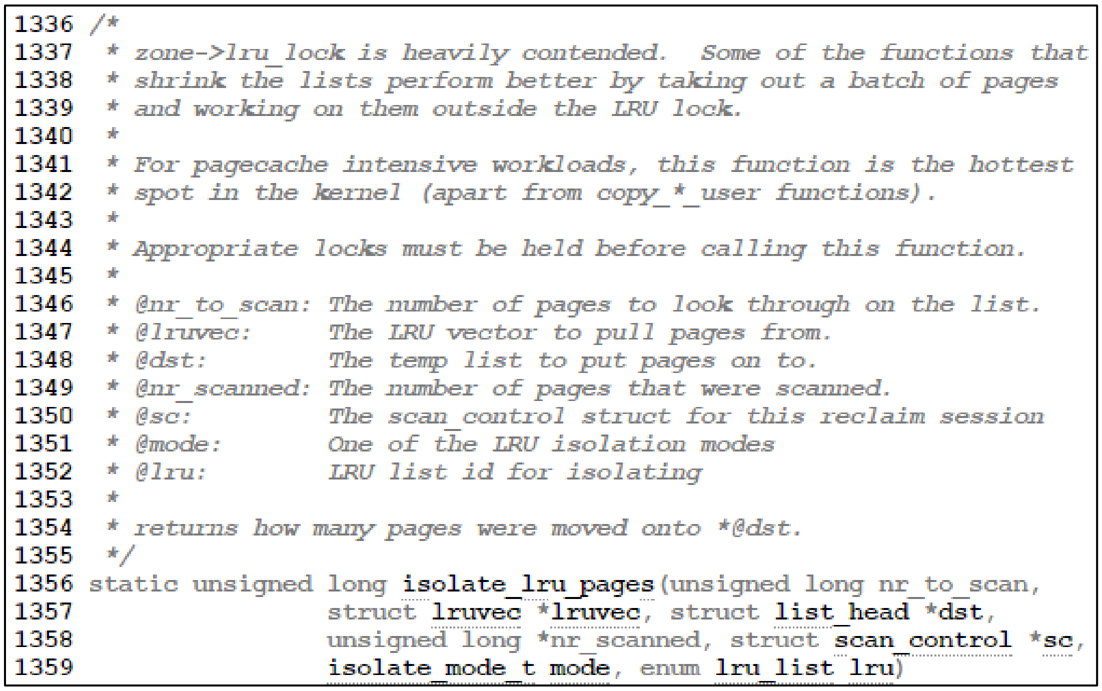
Cache Miss:
(i) search for first '3' from left
(ii) if '3' found go to step (v)
(iii) increment all RRPVs
(iv) goto step (i)
(v) replace block and set RRPV to '2'

[Figure 2 – Trace of RRIP algorithm, taken from Figure 3 from [5]]

Linux Kernel Evaluation

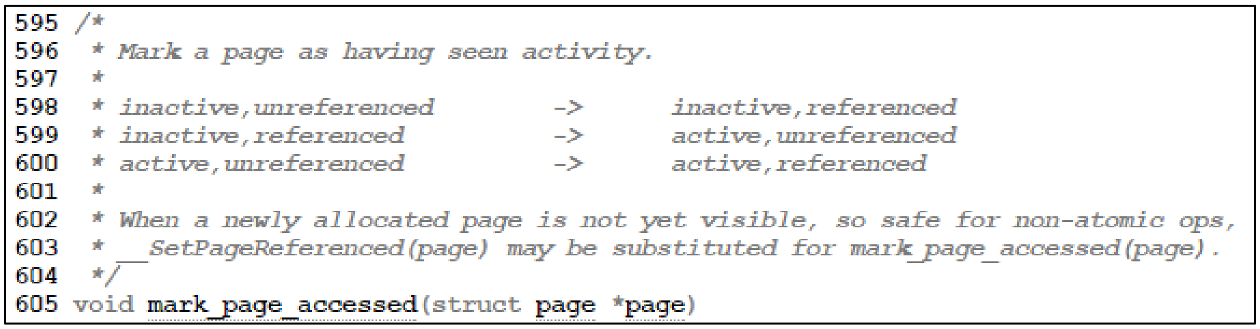
The next step in the literature review process was to dive into the Linux source code to understand how the kernel manages the page cache at a much lower level. High level descriptions of the page cache are useful for understanding how the memory is generally managed, but to make adjustments to the code itself, I needed to know where to make necessary adjustments to certain function calls. Nearly all of the pertinent files are located inside the /mm/ folder within the source of the file tree. A few of the files with code relevant to file swapping include, but are not limited to the following: swapfile.c, frontswap.c, swap_state.c, shmem.c, and filemap.c. However, most of the function calls in these files are related to the population or eviction of pages in the page cache. These methods were not vital to adjust in the scope of this project, as I was interested in the algorithm that manages the pages within the page cache.

There were only two relevant files in this project's scope: vmscan.c, which contains functions responsible for moving pages between the active and inactive lists, and swap.c, which contains functions responsible for identifying page references. Figure 3 shows the section of vmscan.c that contains the API for the function that isolates a collection of pages from the active list to move to the inactive list. Figure 4 shows the section of swap.c that contains the API for the function that marks pages as having been accessed recently and determines what to do with them. After reading through the source code, it became evident that the implementation of the RRIP policy would occur almost completely within these two functions. Additionally, rather than removing the active/inactive list system and building a RRIP-style system from the ground up, I decided to use a RRIP algorithm in the place of the LRU-style one that previously managed the active and inactive lists. In other words, pages would be moved from the active list to the inactive list according to a RRIP basis.



```
/*
 * zone->lru_lock is heavily contended.  Some of the functions that
 * shrink the lists perform better by taking out a batch of pages
 * and working on them outside the LRU lock.
 *
 * For pagecache intensive workloads, this function is the hottest
 * spot in the kernel (apart from copy_*_user functions).
 *
 * Appropriate locks must be held before calling this function.
 *
 * @nr_to_scan: The number of pages to look through on the list.
 * @lruvec:     The LRU vector to pull pages from.
 * @dst:        The temp list to put pages on to.
 * @nr_scanned: The number of pages that were scanned.
 * @sc:         The scan_control struct for this reclaim session
 * @mode:       One of the LRU isolation modes
 * @lru:        LRU list id for isolating
 *
 * returns how many pages were moved onto *@dst.
 */
static unsigned long isolate_lru_pages(unsigned long nr_to_scan,
                     struct lruvec *lruvec, struct list_head *dst,
                     unsigned long *nr_scanned, struct scan_control *sc,
                     isolate_mode_t mode, enum lru_list lru)
```

[Figure 3 – isolate_lru_pages API from vmscan.c]



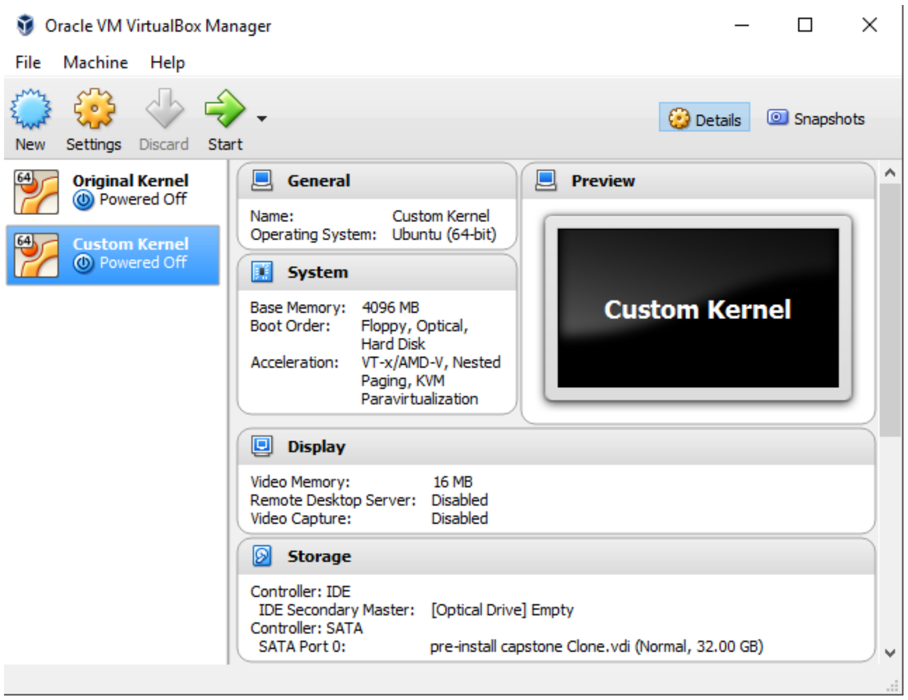
```
/*
 * Mark a page as having seen activity.
 *
 * inactive,unreferenced        ->      inactive,referenced
 * inactive,referenced          ->      active,unreferenced
 * active,unreferenced          ->      active,referenced
 *
 * When a newly allocated page is not yet visible, so safe for non-atomic ops,
 * __SetPageReferenced(page) may be substituted for mark_page_accessed(page).
 */
void mark_page_accessed(struct page *page)
```

[Figure 4 – mark_page_accessed API from swap.c]

Virtual Machine Setup

Utilizing a virtual machine in this project as opposed to a physical machine with a Linux-based operating system (OS) installed allowed for the freedom to adjust certain system conditions without the concern of losing or destroying anything valuable. This is necessary in a project in which adjustments are being made to the very source code of the operating system. In the event that an OS built with faulty code is installed and booted in a non-virtual environment, files, software, and even possibly hardware may begin to suffer and potentially break down.

Consequently, it was much safer to proceed enlisting the use of a virtual machine for this project. Additionally, building a version of Linux with or without a custom page caching policy is necessary to be performed in a Linux environment on account of having easier access to the source trees and other utilities needed conduct the build. For this project, I chose to use Ubuntu, a popular Debian-based Linux operating system. After downloading the OS from the developer, I installed it on VirtualBox, a popular virtual machine utility. This configuration is shown in Figure 5.



Original Kernel
Powered Off
Custom Kernel
Powered Off

General

```
Name:              Custom Kernel
Operating System:  Ubuntu (64-bit)
```

System

```
Base Memory:   4096 MB
Boot Order:    Floppy, Optical,
               Hard Disk
Acceleration:  VT-x/AMD-V, Nested
               Paging, KVM
               Paravirtualization
```

Display

```
Video Memory:            16 MB
Remote Desktop Server:   Disabled
Video Capture:           Disabled
```

Storage

```
Controller: IDE
  IDE Secondary Master:  [Optical Drive] Empty
Controller: SATA
  SATA Port 0:           pre-install capstone Clone.vdi (Normal, 32.00 GB)
```

[Figure 5 – Virtual machine configuration for this project]

Preliminary Kernel Construction

In order for this project to see its completion, I needed to be able to take the Linux source code and build it into a usable OS. Without doing this, it would not be possible to conduct benchmark performance testing on it to determine its efficiency compared to an OS with the unaltered version of the Linux kernel. Performing a preliminary control build before adjusting the source code served two purposes. First of all, it allowed me to get a good feel for this process and understand how to carry it out in the future. Secondly, it gave me a frame of reference for pinpointing compilation and building errors so that I would be able to know that they were not the result of improper kernel construction or configuration, but instead because of issues in adjustments made to the kernel code.

Building Ubuntu from Bare Linux

In addition to using the Ubuntu OS as an environment in which to build my custom Linux OS, I also selected Ubuntu to be the type of Linux to build. This process began by isolating the Linux kernel source code for the version with which I was going to be working (4.4). After that, I ascertained the build environment that allowed me to construct the kernel on my own by linking all of the object files together after compiling the sources. Next, I downloaded the packages and source trees that allow the kernel to be built into Ubuntu specifically. After getting a hold of all of these utilities, I was finally able to perform the construction of the kernel and test it to verify the success of this procedure.

Programming the Custom Page Replacement Policy

There were many goals I sought to achieve in the design of my custom kernel. First and foremost was to effectively introduce RRIP as a means of managing the page cache, but the biggest question was how to properly do so. In the scope of this project, the most basic aim was to determine the feasibility of using RRIP on the magnitude of pages and establish whether or not page cache access patterns are linked to performance increases or decreases in a system managed by RRIP. So rather than reinvent the wheel and completely remove the existing pseudo-LRU 2Q, I opted to incorporate RRIP logic into the existing system as the means of selecting pages for eviction from the active list to the inactive list. Not only did this allow me to see on a small scale the relative performance of a page cache with RRIP logic, but it was also the way in which I took advantage of a vital design paradigm of the ARC algorithm [7]. One of the page lists managed by the ARC algorithm seeks to capture recently used pages, whereas the other seeks to capture pages which have been more frequently used. In the same way, in my design, the kernel's active page list managed by RRIP will capitalize on frequently used pages, and the inactive page list managed by the conventional Clock algorithm will capitalize on recently used pages.

One challenge inherent in implementing RRIP in the page caching mechanism of the Linux kernel is the fact that referencing a page and making an eviction is not as simple and clear cut as the example in Figure 2. This trace functions on two assumptions that do not line up with how the Linux kernel's page cache operates. First of all, it assumes that page references and evictions take place at the same time, and secondly, it assumes that multiple pages are never up for eviction at the same time. Both of these are not the case in the kernel, so I needed to determine how to adjust the original RRIP algorithm to accommodate this. In some cases, this was as simple as splitting up certain RRIP operations into different files, and in other cases, the logic needed to be changed.

Miscellaneous Additions

There were a few minor utilities to add before diving into the details of the RRIP logic. First and foremost, the source code needed to be adjusted to incorporate the re-reference predictor value (RRPV) that describes the re-reference likelihood or "age" of a given page. I accomplished this with the simple addition of an integer data member named "rrip_counter" in the page struct architecture located in the mm_types.h file in the /include/linux/ directory. Additionally, as this value would be accessed and changed relatively often, I added getter and setter functions, named "rrip_counter" and "set_rrip_counter," respectively. These functions were added in the mm.h file also in the /include/linux/ directory. All other changes to be made related to the logic of the RRIP algorithm.

Page Eviction

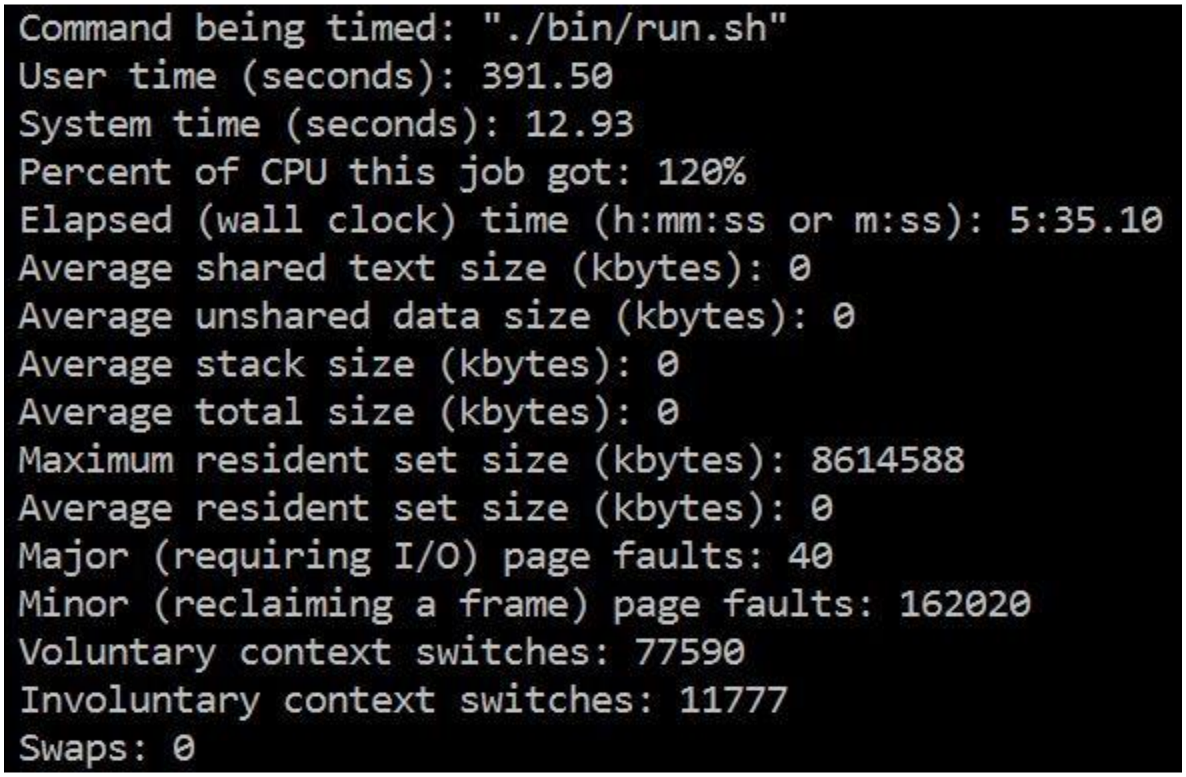
The portion of the Linux source code responsible for evicting pages from the active list to the inactive list is found in the file, /mm/vmscan.c. Figure 3 shows the API of one of the relevant functions in question, isolate_lru_pages. This function originally pulled clusters of pages from the tail of the active list to be tested for eviction. To introduce RRIP policy to page eviction, I adjusted the logic of this function to pull pages from the active list according to RRIP standards. Namely, the function was to continue pulling pages from the end of the active list and hang onto them if their rrip_counter data member was at its maximum value. Otherwise, they would be added back to the active list, and their rrip_counter was to be incremented by 1. This process was to continue until a certain amount of pages, as dictated by other processes in the kernel, were collected or scanned.

Page Referencing

The file, /mm/swap.c, contains the code related to identifying pages as having been accessed. This is very useful in the scope of this project, because the RRPVs of objects being managed by RRIP are adjusted whenever said objects are accessed. As shown in Figure 2, whenever an item in a list incurs a cache hit, its RRPV is set to 0. Additionally, whenever an item is added to the list for the first time, its RRPV is set to its maximum value minus 1. Logic for both of these scenarios is included in the function, mark_page_accessed, whose API is shown in Figure 5. If the page was already on the active list, this corresponded to the former scenario, so the rrip_counter would be set to 0. If the page was being added to the active list from the inactive list, this corresponded to the latter scenario, so the rrip_counter would be set to the maximum value minus 1. Logic is also included for a page that is being added to the inactive list due to not currently being in memory, but since only the active list was to be monitored on an RRIP basis, I was not concerned with this case.

Benchmark Performance Testing

Once the changes to the kernel source code had been made and the custom kernel had been compiled and built into a functional OS, it was time to determine its relative performance. This process consisted of running a benchmarking application on both the original and the custom RRIP kernel and timing the tests. In this project, I used SparkBench, an acclaimed and comprehensive benchmarking suite with capabilities for conducting tests concerned with a wide range of memory operations. I recorded data for each of the different benchmarks supported by SparkBench in order to get as complete of a perspective as possible on the performance of the RRIP kernel with respect to the original.



```
Command being timed: "./bin/run.sh"
User time (seconds): 391.50
System time (seconds): 12.93
Percent of CPU this job got: 120%
Elapsed (wall clock) time (h:mm:ss or m:ss): 5:35.10
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 8614588
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 40
Minor (reclaiming a frame) page faults: 162020
Voluntary context switches: 77590
Involuntary context switches: 11777
Swaps: 0
```

[Figure 6 – Output snippet of verbose Linux time command]

Data Collection

Conducting any one of the SparkBench benchmark tests was as easy as typing in a command, but it would not necessarily provide useful information in the scope of what I was trying to determine. By using the Linux “time” command with the “verbose” flag, users are able to see how much time a given process takes, in what space that time was distributed, how many page faults occurred, how many swaps occurred, and much more. For my purposes, I only really needed to know these fields on account of how they informed me on the performance of the respective kernels and why certain performance patterns were persisting. For the most part, I cared about the system time on account of the source code changes existing entirely in the kernel. However, the user and wall clock time were not negligible as performance must be adequate in the user space as well as the kernel space to have a truly efficient kernel. Additionally, understanding the patterns of page faults in certain tests allowed me to gain a better understanding of why certain benchmarks did better or worse in the custom RRIP kernel compared to the original.

CHAPTER III

RESULTS

Preliminary Builds

Through a bit of trial and error, I was able to develop a build of the kernel which performs quite well compared to my expectations. Bugs and inefficient source code in preliminary versions of the custom RRIP kernel caused yields of very poor performance. For example, I experimented with including a spinlock system [2] that would secure the active list while searching for eviction candidates. While this may have cut down on the amount of improperly executed evictions, the performance increases of doing so did not outshine the overhead of locking the list so often. Developments like this paved the way to the final implementation of the custom kernel with which I recorded the data presented in this section. These inefficiencies also informed me on which data are most significant to consider in the scope of this project. As mentioned previously, user time and wall clock time are important to evaluate. However, where previous, incomplete versions of the kernel reported user and wall clock times nearly identical to those of the classic kernel, they also reported system times that were significantly slower (30x on average) than those of the classic kernel. Consequently, I was able to determine that system time is likely the most significant factor in my results.

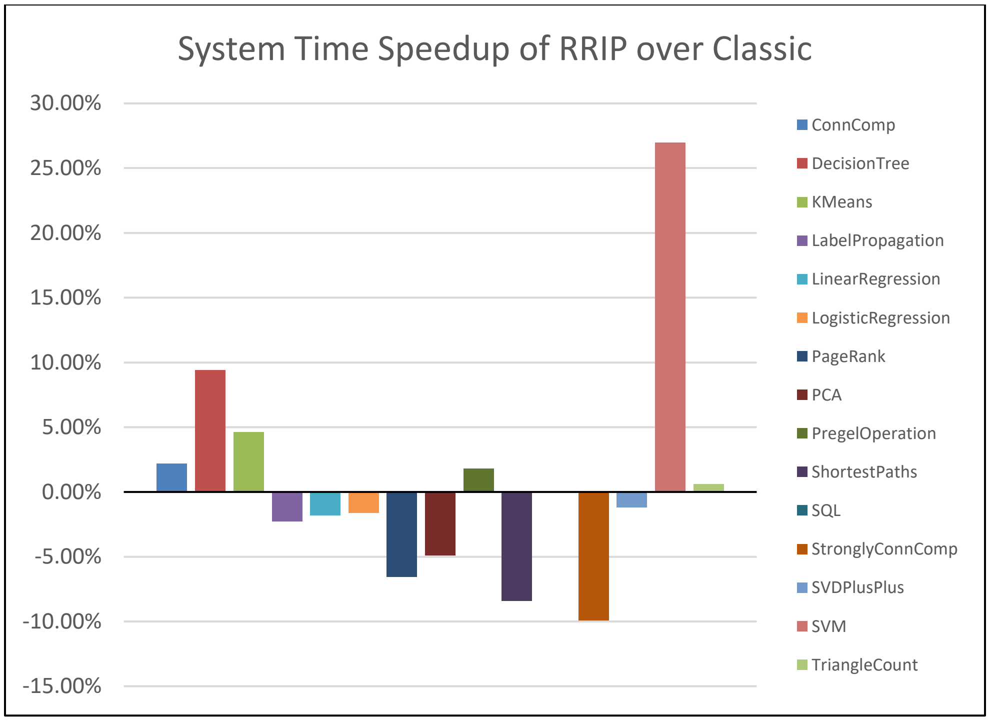
Timing Analysis

On average, the custom RRIP kernel performed very similarly to the original kernel. There were many different benchmark tests I performed through SparkBench, and each kernel yielded different results from test to test, however. Table 1 presents the corresponding relative speedup of the RRIP kernel over the classic kernel recorded for each benchmark. Values in green represent tests in which the custom RRIP kernel yielded better performance, and each percentage is recorded relative to 100% (i.e. 2.21% implies the new kernel operates at 102.21% of speeds reported by the original kernel).

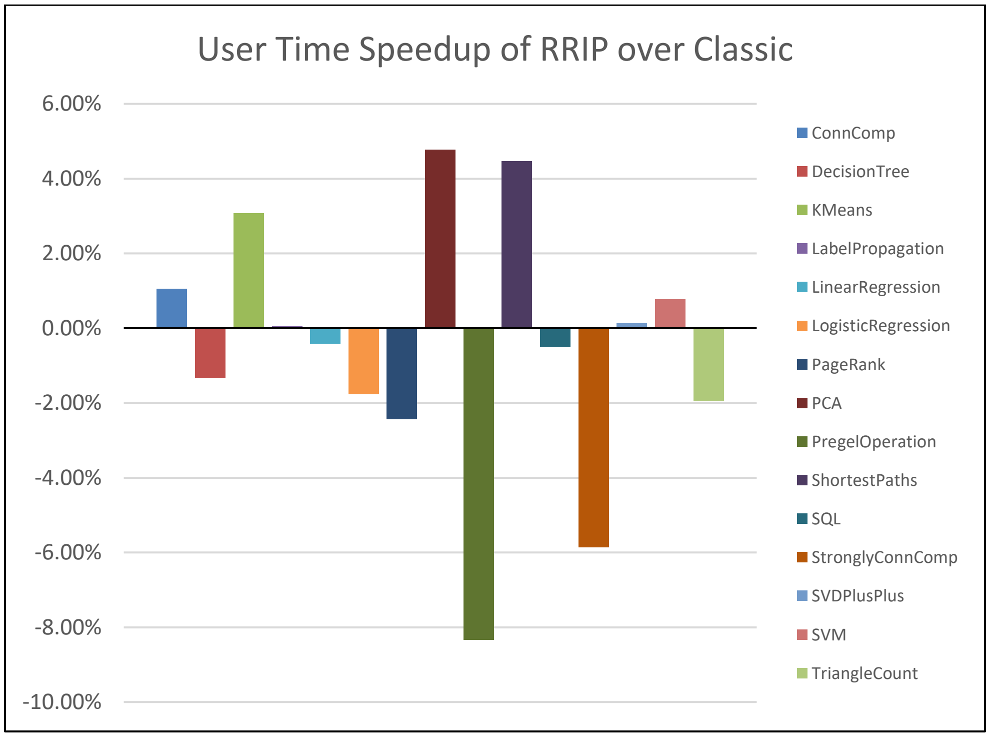
[Table 1 – RRIP kernel speedup data]

Benchmark Test	Relative Speedup of RRIP over Classic			
	System Time	User Time	Total Time	Wall Time
ConnComp	2.21%	1.05%	1.09%	0.02%
DecisionTree	9.42%	-1.32%	-1.17%	-0.99%
KMeans	4.64%	3.07%	3.10%	4.08%
LabelPropagation	-2.30%	0.05%	-0.02%	-2.89%
LinearRegression	-1.81%	-0.40%	-0.45%	-1.40%
LogisticRegression	-1.61%	-1.76%	-1.76%	-2.47%
PageRank	-6.56%	-2.43%	-2.56%	-3.12%
PCA	-4.88%	4.77%	4.48%	1.63%
PregelOperation	1.80%	-8.34%	-8.08%	-4.37%
ShortestPaths	-8.41%	4.47%	3.96%	5.36%
SQL	0.00%	-0.50%	-0.48%	-0.38%
StronglyConnComp	-9.93%	-5.85%	-6.05%	-6.45%
SVDPlusPlus	-1.20%	0.13%	0.04%	0.08%
SVM	27.00%	0.78%	1.47%	0.59%
TriangleCount	0.61%	-1.96%	-1.83%	-1.63%
Geomean	**0.27%**	**-0.61%**	**-0.60%**	**-0.84%**

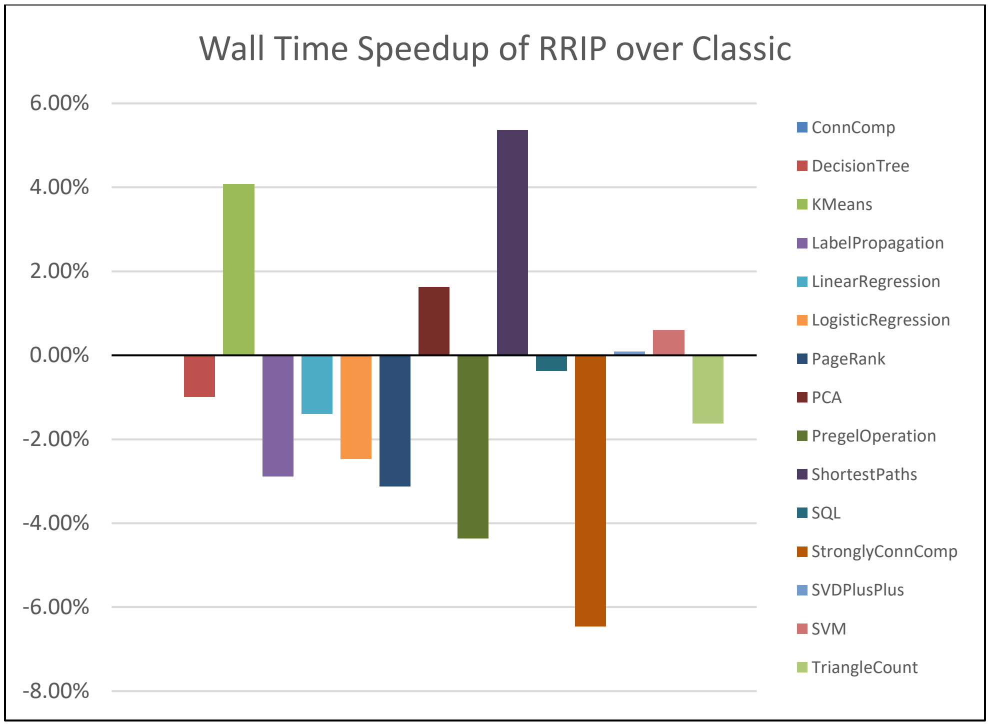
There is a strong correlation between user time, total time, and wall time. This makes sense because of how connected the three metrics are. Total time is simply a combination of system time and user time, taking into account multiple CPU usage as user time does. It is nearly identical to user time on account of how small of a contribution system time makes to the sum. Wall time simply reports actual real world time. On the other hand, there is nearly no correlation between system time and the other timing metrics. In only about half of the benchmarks, good system time actually aligns with good user, total, and wall time. Figure 7, 8, and 9 show the relative speedup of system, user, and wall time, respectively.



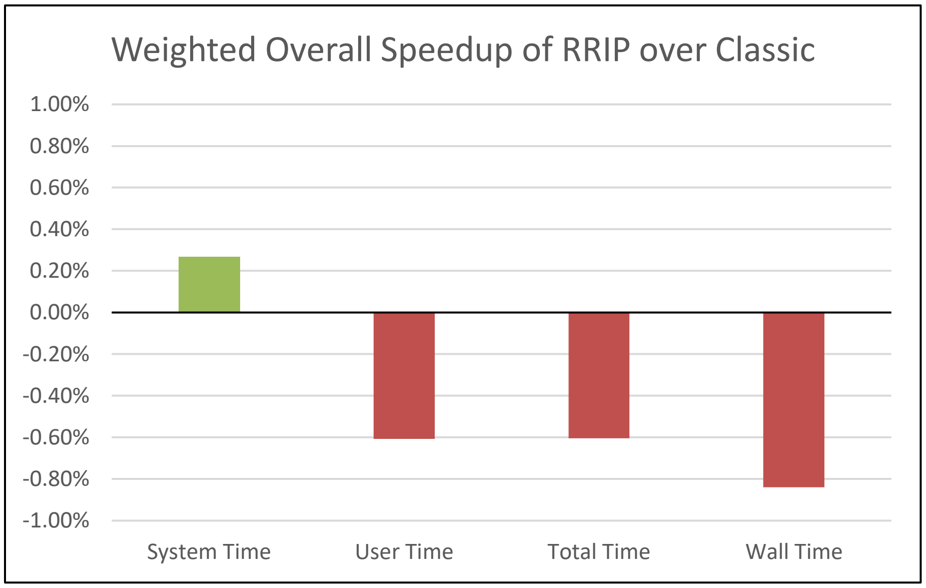
[Figure 7 – System time speedup of RRIP kernel over classic kernel]



[Figure 8 – User time speedup of RRIP kernel over classic kernel]



[Figure 9 – Wall time speedup of RRIP kernel over classic kernel]



[Figure 10 – Weighted overall speedup of RRIP kernel over classic kernel]

These figures show the scattered differences in performance between the two kernels. In some cases, the custom RRIP kernel is much better. In others, the custom RRIP kernel is much worse. And some tests yield extremely similar results in the two kernels. However, by calculating ratio of the geometric mean of all of the benchmark timing data for each metric, we can approximate the overall weighted speedup of each type of time shown in Figure 10. Keep in mind that all of these differences, both positive and negative, are very small as none of these values have magnitude greater than 1%.

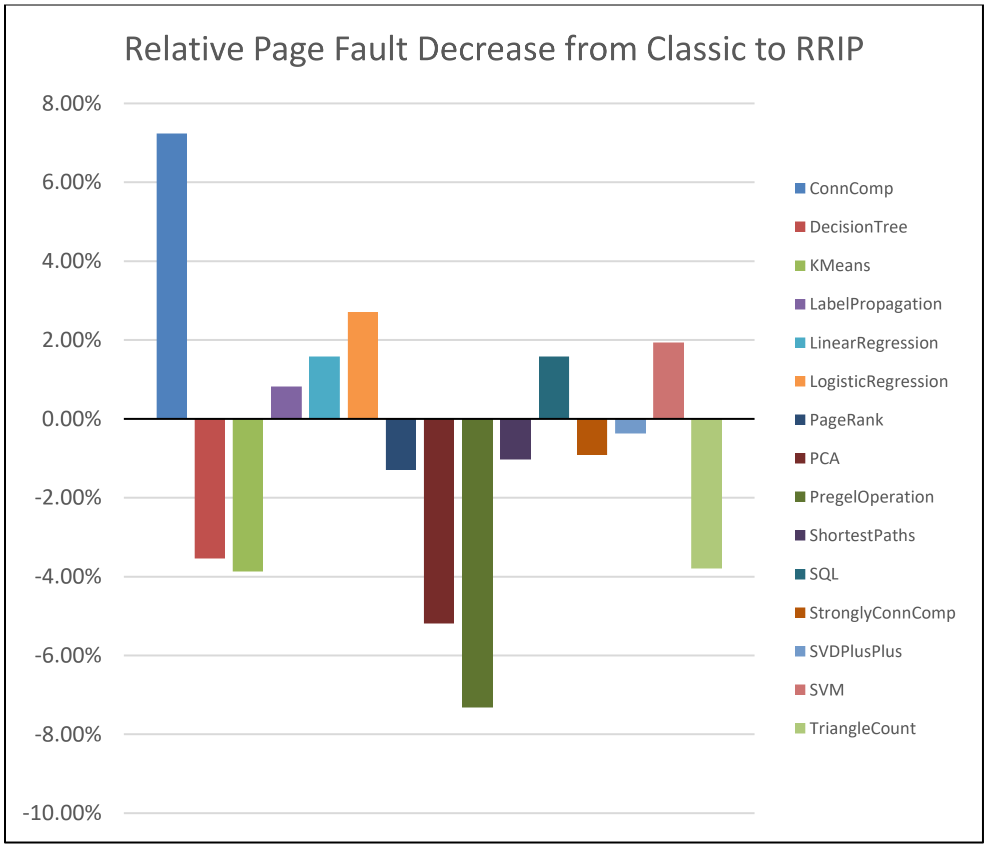
Page Fault Behavior

Another informative vein of data from which I was able to collect is that of the amount of reported page faults. A page fault occurs when the kernel attempts to access information that does currently reside in the DRAM. Therefore, well designed swap policy should allow the kernel to incur page faults as infrequently as possible. The custom RRIP kernel performed roughly as well as the classic kernel in this realm. Table 2 shows the reported differences in major and minor page faults and the percentage difference for minor faults.

[Table 2 – Page fault comparison data]

Benchmark Test	Difference from Classic to RRIP		Percentage Difference of Minor Faults
	Major Faults	Minor Faults	
ConnComp	-1	11725	7.24%
DecisionTree	-3	-4573	-3.53%
KMeans	1	-5556	-3.87%
LabelPropagation	1	860	0.81%
LinearRegression	-1	2092	1.57%
LogisticRegression	1	3234	2.71%
PageRank	0	-2345	-1.28%
PCA	0	-4622	-5.18%
PregelOperation	0	-9938	-7.32%
ShortestPaths	0	-1446	-1.03%
SQL	0	1857	1.57%
StronglyConnComp	0	-2069	-0.90%
SVDPlusPlus	-1	-671	-0.37%
SVM	0	2111	1.93%
TriangleCount	1	-6447	-3.79%
Geomean	**-0.23**	**-976.69**	**-0.70%**

Recall that the two versions of the kernel exhibited performance across the board throughout all benchmarks in the timing metrics. The same can be said about the data they reported with respect to page faults. In some benchmark tests, the custom RRIP kernel incurred significantly fewer page faults. In others, the classic kernel had fewer page faults. And in some cases, the kernels had very similar amounts of page faults. Figure 11 exhibits the relative decrease in minor page faults for the custom RRIP kernel. For example, according to the chart, the RRIP kernel incurred just over 7% fewer minor page faults than the classic kernel. Finally, by calculating the percentage difference of the geometric means of minor page faults for all benchmarks with each kernel, I determined the overall difference in minor page faults incurred to be -0.70%, slightly favoring the classic kernel. Once again, as with the timing metrics, keep in mind that with a magnitude of less than 1%, this value is nearly insignificant.

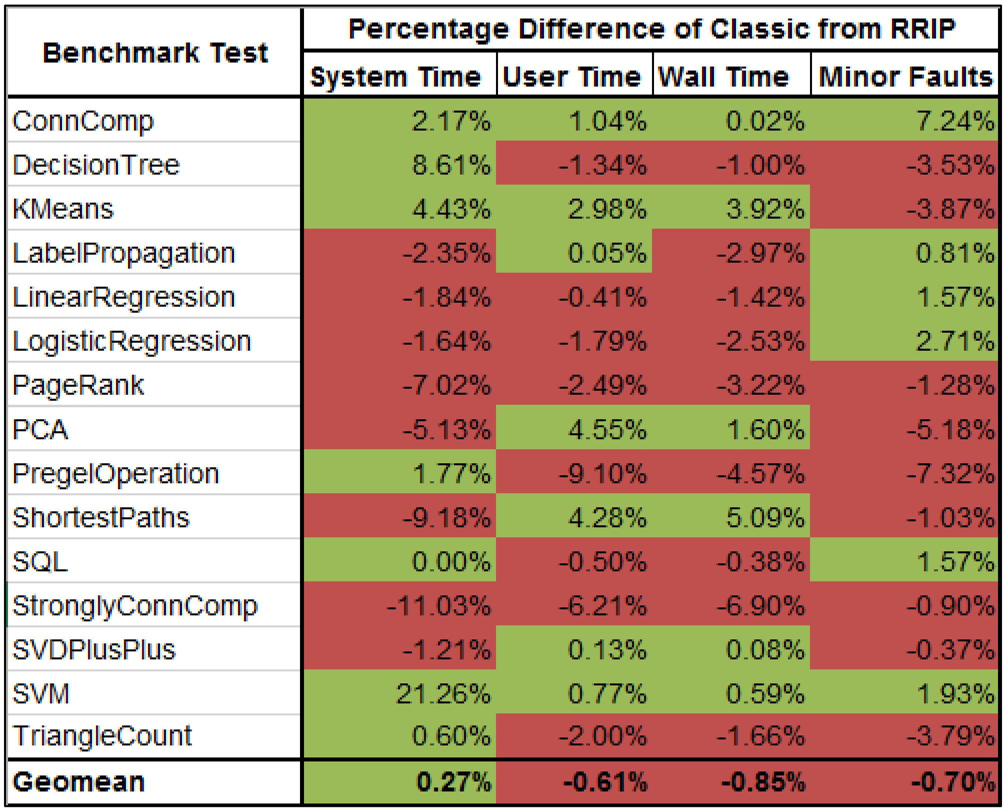


[Figure 11 – Percentage decrease of page faults from classic kernel to RRIP kernel]

Correlations

As mentioned previously, while there is a strong correlation between user time, total time, and wall time, there is no evident correlation between system time and any of the other timing metrics. This is true not just of whether or not the difference is positive or negative, but also of the magnitude of the difference. Across the data, custom RRIP kernel performance is associated with strong, weak, and neutral classic kernel performance. What can be said about correlations between the minor page faults and the timing data, though? Coincidentally, as is shown in Table 3, there is no evident correlation here either. Between minor fault data and system time data, there are multiple instances of positive relationships and negative relationships throughout the different benchmarks. The same can be said of minor fault data paired with user/wall time data.

[Table 3 – Timing and page fault comparison data]



Benchmark Test	Percentage Difference of Classic from RRIP			
	System Time	User Time	Wall Time	Minor Faults
ConnComp	2.17%	1.04%	0.02%	7.24%
DecisionTree	8.61%	-1.34%	-1.00%	-3.53%
KMeans	4.43%	2.98%	3.92%	-3.87%
LabelPropagation	-2.35%	0.05%	-2.97%	0.81%
LinearRegression	-1.84%	-0.41%	-1.42%	1.57%
LogisticRegression	-1.64%	-1.79%	-2.53%	2.71%
PageRank	-7.02%	-2.49%	-3.22%	-1.28%
PCA	-5.13%	4.55%	1.60%	-5.18%
PregelOperation	1.77%	-9.10%	-4.57%	-7.32%
ShortestPaths	-9.18%	4.28%	5.09%	-1.03%
SQL	0.00%	-0.50%	-0.38%	1.57%
StronglyConnComp	-11.03%	-6.21%	-6.90%	-0.90%
SVDPlusPlus	-1.21%	0.13%	0.08%	-0.37%
SVM	21.26%	0.77%	0.59%	1.93%
TriangleCount	0.60%	-2.00%	-1.66%	-3.79%
Geomean	**0.27%**	**-0.61%**	**-0.85%**	**-0.70%**

CHAPTER IV

CONCLUSION

There are a number of different conclusions that can be drawn from the development and results of this project. From the outset, the original question that was being addressed was whether or not it was even possible to implement RRIP as a Linux kernel swap policy. This question was answered as soon as the first version of the kernel successfully booted, even though its performance at the time was quite unimpressive. After that threshold was crossed, however, the question evolved into, "Is it viable to use RRIP as a Linux kernel swap policy?" From the data presented in the previous section, it is safe to say that this question can confidently be answered affirmatively. Additionally, on account of the remaining potential for future research and development of this project, it is likely that a more finely tuned RRIP swap policy implemented in the Linux kernel could probably outperform the classic kernel in speed and/or page fault incurrence in most cases.

Performance

The data recorded from the benchmark testing offers much insight on the performance of the custom RRIP kernel with respect to the classic one. As mentioned in the previous section, performance increases and decreases were recorded all across the board in the benchmark tests. However, it is challenging to determine whether or not the custom RRIP kernel had objectively better performance in a majority of the benchmarks on account of inconsistencies between system time, user/wall time, and page fault performance. As a matter of fact, there were only four benchmarks about which definitive statements could be made. In all relevant aspects, the custom RRIP kernel outperforms the classic kernel in the ConnComp and SVM benchmarks, and the classic kernel outperforms the custom RRIP kernel in the PageRank and StronglyConnComp benchmarks.

If the page fault metric is removed, it is possible to come to a few more conclusions regarding kernel speeds. If the incurrence of page faults were removed from the equation, it would be safe to say that the custom RRIP kernel outperforms the classic kernel in the ConnComp, KMeans, and SVM benchmarks. It would also be safe to say that the classic kernel outperforms the custom RRIP kernel in the LinearRegression, LogisticRegression, PageRank, and StronglyConnComp benchmarks. However, that still does not account for the remaining majority of benchmarks about which judgements cannot be made. It is due to all of this inconsistency and uncertainty that it is hard to make a definitive claim one way or another about the performance of the custom RRIP kernel compared to the classic one. However, according to the bottom row of Table 3, although the grand percentage difference of the geometric means for system time is in favor of the custom RRIP kernel, it is probably fair to say that the classic kernel still performs at least slightly better on account of the other grand geometric mean values for user time, wall time, and minor page faults. Once again, it must not be ignored that all of these percentages are of magnitude less than 1%, so this performance difference is quite nearly negligible.

Strengths and Weaknesses

One useful way to gain understanding of the strengths and weaknesses of the custom RRIP kernel with respect to the classic one is to understand the similarities of the benchmarks with which it has similar performance patterns. However, it is quite challenging to establish such correlations given the somewhat disconnected nature of the data recorded with these two kernels. In other words, there are a number of different benchmarks that may be clustered together on account of executing similar tasks such as StronglyConnComp with ShortestPaths and LinearRegression with LogisticRegression. However, not all of these potential groupings of benchmarks exhibit similar performance characteristics. Additionally, because of the fact that there is very little performance consistency from metric to metric for each of these groupings, it is hard to use these relationships as a basis for claiming that the custom kernel performs objectively well or poorly at these things. For example, according to Table 3, the first two benchmarks listed above (both graph algorithms) do not perform similarly, whereas the second two benchmarks listed above (both regression algorithms) do perform similarly. Consequently, there is not much that can definitively be said about the custom RRIP kernel's performance with similar tasks within a given realm.

Future Endeavors

This research is anything but fully explored. There are so many different ways in which RRIP may be developed as a swap policy in the Linux kernel. First of all, there is room for adjusting the management style of the algorithm itself. Throughout this project, the value assigned to RRIP_COUNTER_MAX was always 4, meaning RRPVs of pages in the active list only moved back and forth between 0 and 3. Increasing or decreasing this value could change the benchmarks which are favorable for the custom RRIP kernel or even increase performance overall. Secondly, the kernel source code of the RRIP implementation has room for improvement. Optimizing the way in which the kernel selects pages for eviction and increments RRPVs according to RRIP standards would increase performance and may be achieved in a number of different ways. On one hand, adjusting the logic behind these processes may prove fruitful. On the other hand, changing the location of this logic in the hierarchy of active list management function calls has quite a bit of potential too.

Finally, after all other avenues are explored, and the true worth of RRIP at the level of page caching has been established, it would be worthwhile to change the scope of the question altogether. If RRIP is found to perform very well in a page caching context, researchers in this field are granted a huge opportunity. Rather than simply seeking to implement it as a policy for moving pages from the active list to the inactive list, it would be worthwhile to explore the possibility of replacing the active/inactive list system with something optimized to work well with RRIP. The results of this project as it stands right now are nothing if not encouraging, as the most recent version of the custom RRIP kernel has incurred very minimal fine tuning. Since it is currently just below breaking even in performance when compared with the classic kernel, my results suggest that it will only take a bit more effort in development to allow it to surpass the average performance of the classic kernel. Once that threshold has been crossed, there is no knowing what kinds of performance may be achieved by continuing to explore, expand, and optimize RRIP as a swap policy in in the Linux page cache.

REFERENCES

[1] J. Corbet, G. Kroah-Hartman and A. Rubini, "Linked Lists," in Linux Device Drivers, O'Reilly, 2005.

[2] J. Corbet, G. Kroah-Hartman and A. Rubini, "Spinlocks," in Linux Device Drivers, O'Reilly, 2005.

[3] G. Duartes, "Page Cache, the Affair Between Memory and Files," Feb 9, 2009.

[4] M. Gorman, "Page Frame Reclamation," in Understanding the Linux Virtual Memory Manager, Prentice Hall, 2004.

[5] A. Jaleel, K. Theobald, J. Steely Simon and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," ACM SIGARCH Computer Architecture News, vol. 38, pp. 60-71, June 19,. 2010.

[6] T. Johnson and D. Shasha, "2Q : a low overhead high performance buffer management replacement algorithm".

[7] N. Megiddo and D.S. Modha, "Outperforming LRU with an adaptive replacement cache algorithm," Mc, vol. 37, pp. 58-65, 2004.

[8] Rik Van Riel, "Page replacement in Linux 2.4 memory management," Proceedings of the USENIX Annual Technical Conference, pp. 165-172, 2001.

[9] Rik van Riel, "PageReplacementDesign," October 18, 2013.

[10] Y. Zhou and J.F. Philbin, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," Proceedings of the 2001 USENIX Conference, 2001.