IMPROVING MEMORY MANAGEMENT IN THE LINUX KERNEL

An Undergraduate Research Scholars Thesis

by

DYLAN SIEGERS

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. Chia-Che Tsai

May 2020

Major: Computer Science

TABLE OF CONTENTS

ABSTRACT

Improving Memory Management in the Linux Kernel

Dylan Siegers
Department of Computer Science
Texas A&M University

Research Advisor: Dr. Chia-Che Tsai
Department of Computer Science
Texas A&M University

In this research I intend to analyze the Linux OS memory management system for inefficient segments and work to improve the source code to decrease the inefficiency. This is primarily important due to the nature of the Linux kernel being a constantly developing project. This operating system is extremely prominent in the field of computer science, and is always being improved, however most of these improvements come from academia, not industry. This makes it important to spend resources researching and continuing the process of perfecting this large body of code. There is an expanse of previous research on the Linux kernel in general, but the purpose of my research is to build on or improve the work of what has been done before. This is accomplished by looking at the composite of all the research of others (the Linux kernel as a whole), and adapting it to be more efficient in certain situations. The outcome I am aiming for is just that, locating and altering specific cases of the code to allow for a greater efficiency of the OS as a whole.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Tsai, and all of the other students who have worked with him during my time as a researcher. As an undergraduate student, I did not expect to be so well received when I was interested in doing research, but everyone along the process has helped me out tremendously in pursuing a better knowledge of kernel hacking.

CHAPTER I

INTRODUCTION

My research will be part of a much larger project that has been contributed to by thousands of others before me. This is because the entire Linux operating system, which my research is centered around, is an open source project. This means that the code is public property, available to anyone for free online.

The Linux Kernel

The Linux kernel is a popular operating system that is used by many throughout the world of computer science. It is not owned or managed by a company like Windows or MacOS, but instead is a ‘owned’ by the public. This is to say that there is no group being paid for the upkeep and improvement of this language, rather the collective efforts of the public, mostly by companies such as Microsoft, Intel, Ubuntu and Fedora, working to continue its development. The research described in this paper is driven by an effort to contribute to that continued development. In this particular case, I am focused on improving the design of memory management.

Previous Work on this Topic

It is difficult to have a clearly defined review of what work has previously been done in regards to my research, as the entire source code is the result of previous research. While there may not be previous projects that directly map to the work I will be doing, as all updates to Linux should be unique or they would not hold much value, there are some resources that have value as a reference. This includes those that simply acknowledge the need for continued work, such as Johnston claiming “more work should be done to identify and study unusual problematic

program behaviors not represented in our sample" (26). Additionally, from the research of Huang, Qureshi, and Schwan at the Georgia Institute of Technology, "In-sights derived from this study concern the development process of the virtual memory system, including its patch distribution and patterns, and techniques for memory op-timizations and semantics" (465). This type of report offers insight on what sections of code are most frequently patched, which can help guide which areas I should focus my efforts on.

Guide to Additional References

Beyond this, my references are a variety of sources that offer deeper understanding of the common choices of how to approach memory management. The most straightforward of these is directly sourced from the Linux kernel user guide, with fields like "Virtual Memory Primer", "Reclaim", and "Compaction", all integral to the functionality of the Linux memory system (Concept Overview). I will use these to consider how I could rework segments of the source code after locating pieces that are less efficient or have a lower latency (response/processing time). The idea of a buddy pairing system for example, where "Whenever possible, an unallocated buddy is merged with a companion buddy in order to form a larger free block", which I would not have conceptualized on my own (Silva 1). While these may not be quite as valuable as having the old research of others to reference, as mentioned before, the presence of prior research in any one topic for Linux is all but impossible as each improvement should be unique. This is one of the fascinating parts of this project for me though, as it means that I am doing something new, even if it is just an adaptation of the existing work.

Memory Management: Corner Cases

With this research project, I will locate corner cases in the Linux OS memory management code that have low latency and redesign the source code to make these cases more

efficient. I will use various timing techniques to see the comparative delays of different aspects of memory allocation and reclamation. Following this process of data collection, I will select a particular section of the source code to attempt to improve. This will be a piece of code that is not necessary for all cases of memory allocation or reclamation, but rather is used in certain cases when memory allocation has a very high time cost or delay. This is what is referred to as a corner case. I am electing to focus on targeting corner cases over other more common situations since most opportunities to improve those cases are likely to have been done previously.

CHAPTER II

METHODS

This project consisted of three primary sections: preparation, data collection, and improvement. Each section has a distinct focus, as it is entitled. There were various methods required for the different sections, as they represent strong shifts in the manner I spent my efforts. The first section consisted primarily of reading, learning, and, as the title suggests, all around preparing. In the second section I collected all of the timing data that I used to justify the need for change in certain locations. This data is also valuable outside of my research, should other projects be interested in a similar concept. The third and final of these sections is the improvements. This is the body of where the results will come from as it is the final goal of my work. These are the changes that will be made to the Linux source code, and may be considered to be pulled into the official Linux OS.

Preparation

The initial stages of this project were centered around learning the ways of Linux memory management, as well as familiarizing myself with the environment. This began with reading different resources, including previous research, and the Linux kernel manual. Various textbooks and articles also aided in supporting my understanding of the nature of this topic and can be found in the references concluding this paper. After sufficient research on the theory and concepts behind Linux memory management, I went on to design some simple projects inside the kernel. This included simple kernel drivers and user end modules that could communicate or write to the logs on the kernel side. These tools allowed me to learn more about the manners with which the user and kernel side of Linux can and cannot interact. The final step in this first stage

was immersing myself in the source code of the operating system. I now firmly believe that this is integral to the success of any project based on this large of an amount of code. While I certainly did not go through all of the source code, I did work to understand the context of the functions I was focusing on in this research. This proved extremely useful as my work progressed, as I knew more about what was actually happening whenever certain functions were being called. It also helped me to know which functions were prioritized or most frequently called in the process of page allocation.

Data Collection

All work is done using Linux 4.17.8 + Ubuntu 16.04, *Machine Specs: Intel Core i5-8500 CPU @ 3 GHz, 6 core with 9MB cache, DRAM 16GB 2x8GB DDR4 2666MHz UDIMM Non-ECC*

The first step in the process of data collection was to design two pieces of software. The first of these is the ioctl driver for the kernel side. This program is designed to call the kernel functions alloc_pages() and free(), which allocate and free memory respectively, as well as collect the actual time data from per_cpu variables that are located throughout the source code when these functions are triggered. These are broken into ‘read’ and ‘write’ ioctl calls that are used by the second piece of software, the user program. This program interacts with the driver module to make system calls in the kernel, and receives back all of the time data which is placed into a .csv file to be analyzed.

The ‘times’ referenced throughout this document are in terms of cycles received from rdtsc_ordered() on the linux kernel. The outlier values have been removed from the following data to remove the high time value cases from consideration for averages. This is because these cases are fairly uncommon and tend to have functions that are not normally used that take up the

majority of their processing time. The table below (Fig. 1) contains the averages from trials for the first 60,000 allocations of size 2^6 pages, with data centering around the chosen function get_page_from_freelist, which is the primary path of page allocation in the alloc_pages function.

	Freelist Total time	Try Loop time	Rmqueue time
Average in cycles	1001.00762	864.174783	824.949933
% of Freelist time	100%	86.33049%	82.411954%

*Figure 1: Table of data for get_page_from_freelist

As the table shows, the vast majority of the overall time that is spent in the function get_page_from_freelist is inside of the Try loop in the same function. As the stress test data shows, this try loop is generally called only once. While it consists of several functions, the time data in the table shows that this loop is almost purely dependent on the rmqueue function, comprising over 82% of the entire get_page_from_freelist function and over 95% of the try loop in terms of the number of cycles to run the function.

Looking within the rmqueue function, it becomes slightly more difficult to target a function or section of code with a heavy timeshare for processing time. The table below (Fig. 2) shows that the Do While loop comprises the majority of the processing time for rmqueue, and that the function for if the order is likely 0 is a fairly small portion of the total time. The data from this data set also showed that this loop generally only occurs once, except in the later cases after there have been a large number of allocations already completed.

	Rmqueue Total time	Do While Loop time	Likely order 0 time
Average in cycles	856.575356	672.71922	36.9592373
% of Rmqueue time	100%	78.535906%	4.314768%

*Figure 2: Table of data for rmqueue

When you look inside of the Do While loop that makes up the majority of the rmqueue function, the difficulty in locating the source of the greatest processing time in this function becomes apparent. There are two main sections, each an if statement. The two if statements are centered around the functions rmqueue_smallest and __rmqueue. The issue with determining the main source of processing occurs when timing these functions. Inside of the main rmqueue function, rmqueue_smallest was never called in during any stress test run, making it 0% of the rmqueue total time. The function __rmqueue was consistently called, however it only made up about 10% of the rmqueue total time on average. While this is not a majority, there was no other clearly defineable function that took up a larger portion of the processing time. That being said, the following is a breakdown of the times inside of __rmqueue.

The entire body of __rmqueue is a retry process, however the stress test measuring this section again showed that it tends to only occur once. This ‘looping’ process was required to repeat more frequently than the ones referenced previously, however the cases where it did so were still only in the latter ones of the stress test. Inside the retry, there is only one function that appears as though it would take up a decent amount of processing to complete, which is a call to the same rmqueue_smallest that was never actually called in the main rmqueue function. This function alone generally comprised 53.5% of the overall time for __rmqueue. Looking at the data itself, especially in the earlier cases, rmqueue_smallest almost always accounts for around 50%

in the ‘standard’ case (most cases run in around 400-600 cycles). In the early cases where the overall __rmqueue function takes a larger number of cycles (generally 600-1300), the increase in rmqueue_smallest accounts for almost all of the increase. This shows that while rmqueue_smallest is not necessarily a significant majority of the processing that occurs in __rmqueue, it generally is the vast majority of the increased processing time when __rmqueue takes longer than base case.

At this stage in the data collection, there was a shift in my focus. The function get_page_from_freelist was initially chosen because it is the primary path of page allocation, however in order to find the worst cases, I chose to redirect my efforts towards the “slowpath” functions. Most specifically, the reclaim and compact pages functions, which are used in both slowpaths and standard allocations, seemed to have room for improvement. Figure 3 shows the strong relationship between reclamation and compaction, which seem to always occur in a 1:2 ratio, favoring compaction. These functions are a consistent source of delay whenever they are needed for an allocation, as they attempt to locate pages that can be re-collected if there is not a block currently existing of the appropriate size.

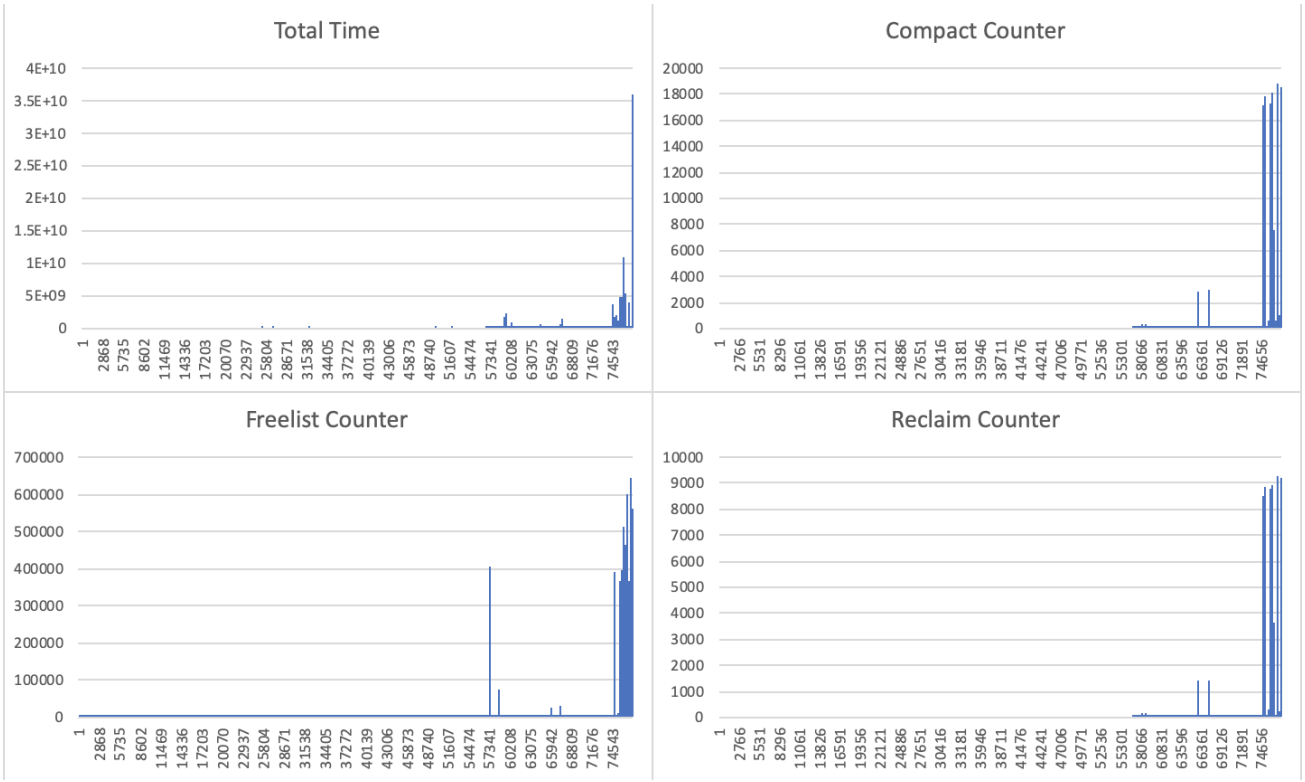

*Figure 3: Graphs of data counting the number of occurrences of Compact, Reclaim, and Freelist

Improvement

Following the shift in focus that was noted in the previous section, I began to look at how the processes of compaction and reclamation could be improved upon. The most prominent of the ideas for improvement was the concept of continuous compaction and/or reclamation. This would be accomplished by performing these processes every time a page was allocated or freed, utilizing a large number of "microtransactions", rather than the large cost these functions generally require. In order to accomplish this, I hope to design a simplified version of the two functions responsible for the slowpath compaction and reclamation. These functions would then be called each time that a page is allocated or freed, which would be one of the "microtransactions". The cost of these should be relatively negligible to the latency of each allocation or freeing of pages, however it could in theory prevent the need for the long delays

that occur when the system is closer to being full of memory, as is seen in Figure 3. The functions try_to_compact_pages and __perform_reclaim exist in the the primary functions I am considering, __alloc_pages_direct_compact and __alloc_pages_direct_reclaim, respectively. I believe that these currently existing functions may be the key to the abbreviated versions of compact and reclaim I hope to implement. The design of these functions has not been completed or tested, however upon that completion, I intend to update/append that data to this paper.

CHAPTER III

RESULTS

The primary results of my research come from the various timings that were performed in the data collection stages of the previously described methods. Some sections of this data collection center around different areas of memory management than the final area where I elected to focus my improvements, however the data itself can still be valuable in understanding how different functions' latency is impacted as the available memory space decreases.

Data Analysis

*Figure 4: Graphs of the latency of alloc_pages

In Figure 4, you can see the simple initial timing that I performed to see when problems occurred assuming the entire memory was being allocated. The stress test that I designed used

the alloc_pages function to completely fill the memory, then uses the free function to return all of the memory that was claimed in the process. Because of the high load on the computer, it is very common for this program to crash the computer, which then requires a hard reboot. The collected data was still protected as the file time_data.csv is continuously updated throughout the process of allocation. A hard reboot will also free any allocated pages that have not been freed already, so there is no concern for wasted memory caused by early termination of the program.

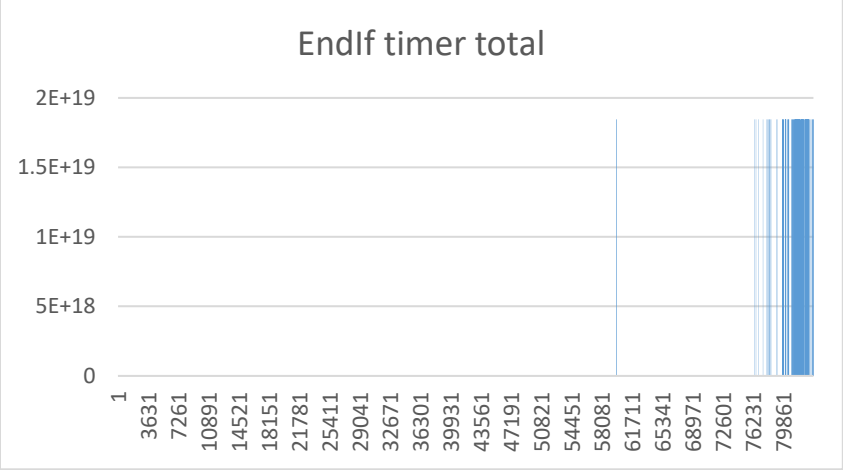

*Figure 5: Example of looping of time data due to the max value of long integers

One issue I ran into along the process of timing was the limit on the length of a long integer. The values that I recorded were the difference between the value of a long int before and after a function was run, which meant occasionally, the long int would be overflown and loop back to zero. This led to data that appeared like Figure 5, where several values would be near the max value of a long int. This was solved in my timing functions by implementing a check for if a loop had occurred, and then triggering corrections for that if one did.

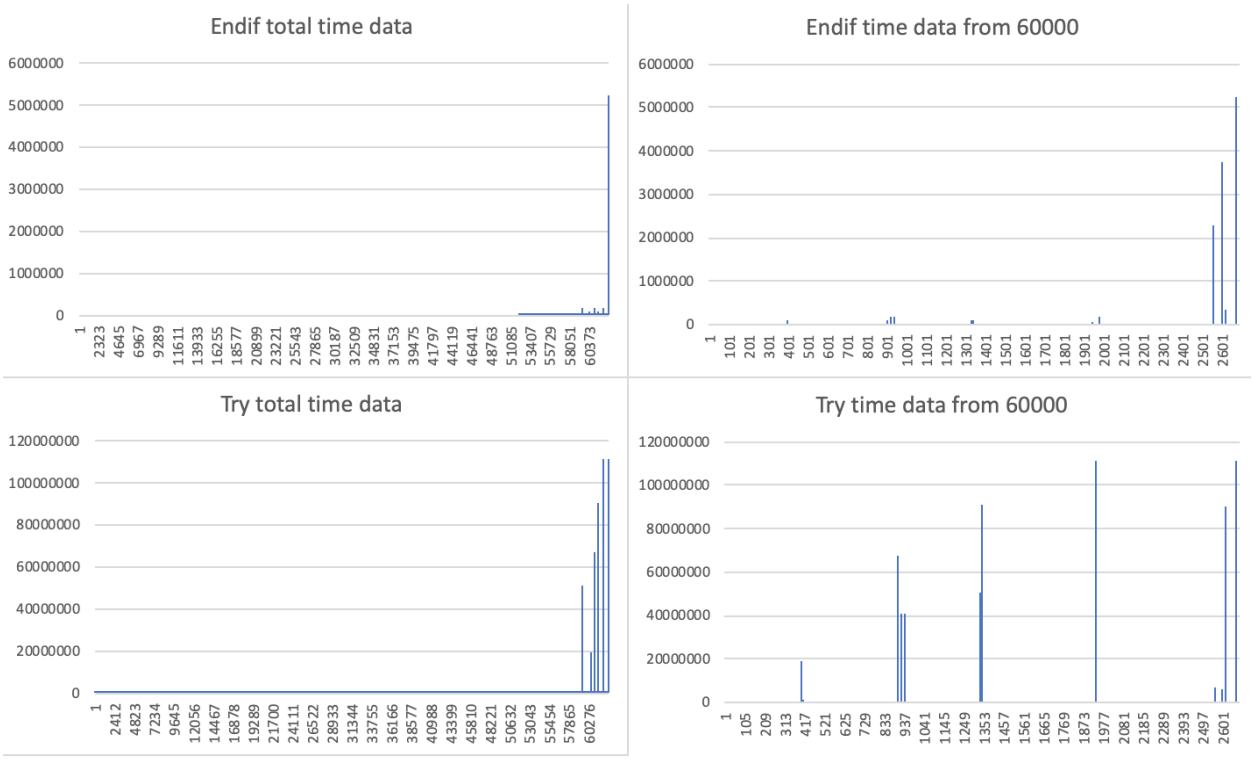

*Figure 6: Time data from inside the get_page_from_freelist function

Following the correction of the issue with looped data, I was able to collect accurate time values for the interior sections of the get_page_from_freelist function. An example of this data can be found in Figure 6, which shows that the majority of the time spent in the freelist function, as discussed previously, is inside of the try loop it contains. It was at this point that I turned to the function rmqueue for my timing process. Timing inside of rmqueue supports the data that is found in Figure 2 inside of the Data Collection subsection of the Methods chapter.

Data Collection Form

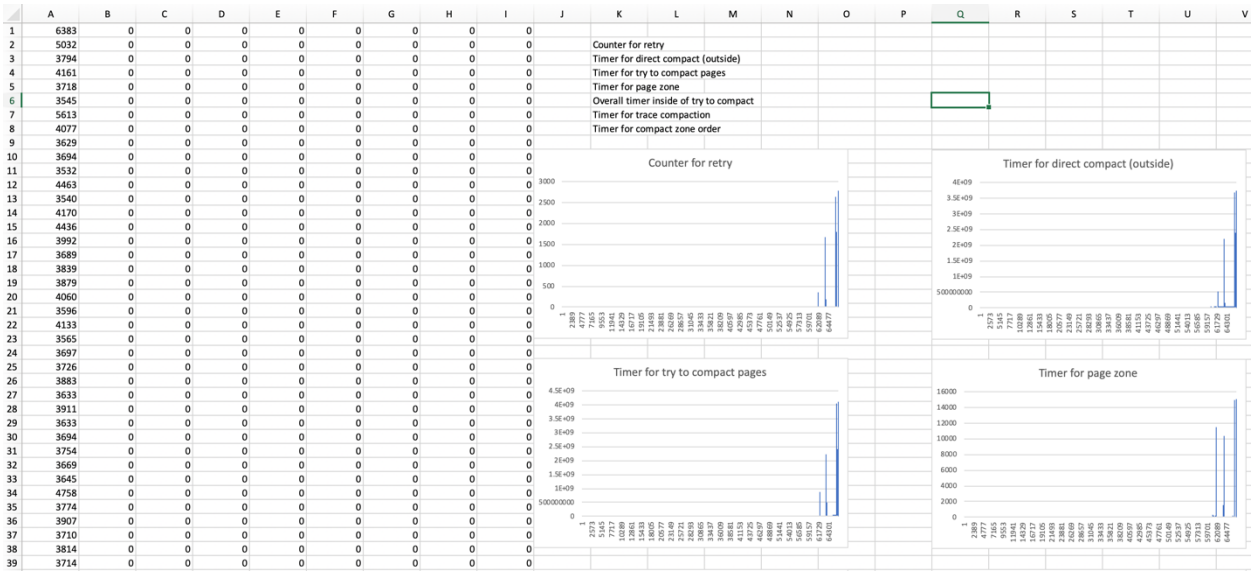

*Figure 7: Example of the format used for data collection

The columns of data that can be seen on the left side of Figure 7 are what the timing tool produces when opened in excel. I generally created forms in the format above in order to look back and interpret the data. The list of elements above the graphs corresponds to the columns of data, in order to keep track of what is being referenced in this particular data set. The graphs below were different for each of the data sets collected, however could contain any seemingly useful correlation between columns. While only the left side of this figure is generated by the program, I highly recommend using this form or something similar should you use the timing tool, as it allows for easy interpretation and reference to the data.

CHAPTER IV

CONCLUSION

Timing Program

The most important product of this research at this stage is the timing program and driver that were designed to work alongside per_CPU variables. This tool can be used throughout many other projects to accumulate data on the latency of various kernel functions. This is an invaluable tool, as I determined through the course of my own work. While this alone is not a solution to the latency issues that I had initially noted, it is certainly a part of the solving process. I will include the two files, driver and user program, in the appendix, as well as a link to the github repository where they are maintained. These are both publicly licensed, and I hope that others can benefit from them to aid future research.

In order to properly use these two pieces of code, one should familiarize themselves with drivers in general. This code is an IOCTL driver, so experience/understanding of how such a driver works would be most valuable. In addition to understanding the driver, it is important to understand per_CPU variables and how they work. While I still highly recommend doing your own research and learning on these variables, here is a short guide to their use in my work:

1. Define the variable at the start of the kernel source code file where you plan to time a function
2. Export the variable so it can be accessed outside of the current file
3. Use either increment or add commands where you are counting occurrences or timing respectively

a. If timing, use rdtsc_ordered, and capture the number of cycles before and after what you would like to time, then add the difference to the per_CPU variable

Corner Cases

One of the initial goals of this research was the locating of corner cases that may cause issues with the latency of memory allocation. Despite my choosing to shift the direction of the efforts for actual improvement of the Linux kernel, there was still a good amount of effort committed to locating the functions that I deemed corner cases. This is most thoroughly discussed in the Data Collection segment of my Methods chapter; however, my primary finding was that the number of calls to the function rmqueue showed a respectable correlation with the heightened latency cases. Due to the refocusing of my work, I elected not to make any changes to this particular section of code, however this is an opportunity for future work on improving memory management design. The timing work done in my stage of data collection, along with the timing program itself can certainly facilitate later work, more centered around corner cases.

Improving Compaction and Reclamation

The redirected end goal of this research is to find a way to limit the need of compaction and reclamation when allocating pages. This is specifically intended to be done by calling simplified versions of the two functions whenever pages are allocated or freed. The data from Figure 3 shows that these functions have enormous cost when the end of memory is being neared. Due to this, there is a clear benefit to the “microtransactions” described in the Improvement segment of the Methods chapter, as these will minimize the number of occurrences of high cost compactions and reclamations. While this process has not yet been completed, I intend to continue my efforts on this research and include the results of my work as an extension of this paper.

REFERENCES

Huang, Jian, Moinuddin Qureshi, and Karsten Schwan. "An Evolutionary Study of Linux Memory Management for Fun and Profit." Usenix. 24 June 2016. 16 Sept. 2019 <https://www.usenix.org/system/files/conference/atc16/atc16_paper-huang.pdf>.

Silva, Fernando. "Buddy Memory Allocator." Cs.fsu.edu. 2001. FSU. 10 Sept. 2019 <https://www.cs.fsu.edu/~engelen/courses/COP402003/p827.pdf>.

Johnstone, Mark, and Paul Wilson. "The Memory FragmentationProblem: Solved?*." Cs.tufts.edu. 1998. 10 Sept. 2019 <https://www.cs.tufts.edu/~nr/cs257/archive/paul-wilson/fragmentation.pdf>.

Adi, Prima. "Stress test CPU and Memory in Linux." Medium. 27 July 2018. Medium. 10 Sept. 2019 <https://medium.com/@primaadipradana/stress-test-cpu-and-memory-in-linux-d17bfa5e8887>.

"Concepts overview¶." Concepts overview - The Linux Kernel documentation. 10 Sept. 2019 <https://www.kernel.org/doc/html/latest/admin-guide/mm/concepts.html#mm-concepts>.

Galov, Nick. “111 Linux Statistics and Facts - Linux Rocks!” *Hosting Tribunal*, Hosting Tribunal, 10 Jan. 2020, <hostingtribunal.com/blog/linux-statistics/#gref>.

APPENDIX

Item A: Stress_test.c (user program)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <cmath>


#define WR_PAGE _IOW('a','a',struct page*)
#define RD_PAGE _IOR('a','b',struct page*)

struct my_data
{
        struct page *current_page;
        unsigned long time;
        unsigned long int freelist_inc, if_time, endif_time, try_time;
};

long int get_meminfo() {
        FILE *pf;
        long int memFree;

        pf = fopen("/proc/meminfo", "r");

        if (pf == NULL) {
                printf("Unable to open the file\n");
                return 0;
        } else {
                fscanf(pf, " MemTotal: %li kB ", &memFree);
                fscanf(pf, "MemFree: %li ", &memFree);
                fclose(pf);
                return memFree;
        }
}

void array_mem_process(int fd) {
        unlink("time_data.csv");
        FILE *f = fopen("time_data.csv", "a");
        if (!f){
                perror("fopen failed");
                return;
        }
        struct my_data *data = malloc (sizeof (struct my_data));
        int j = 0;
```

```
        int size = 1000;
        struct page* *pages = malloc(size*sizeof(struct page*));
        while (get_meminfo()>1000) {
                if (j==size-1) {
                        size = size*2;
                        pages = realloc(pages, size*sizeof(struct page*));
                }
                ioctl(fd, RD_PAGE, (struct my_data*) data);
                pages[j++] = data->current_page;
                if (data.freelist_inc > 18000000000000000000) {
                         data.freelist_inc = pow(2,64)-1 - data.freelist_inc;
                 }
                if (data.if_time > 18000000000000000000) {
                        data.if_time = pow(2,64)-1 - data.if_time;
                }
                if (data.endif_time > 18000000000000000000) {
                         data.endif_time = pow(2,64)-1 - data.endif_time;
                 }
                if (data.try_time > 18000000000000000000) {
                         data.try_time = pow(2,64)-1 - data.try_time;
                 }
                fprintf(f, "%lu, %lu, %lu, %lu, %lu\n", data->time, data-
>freelist_inc, data->if_time, data->endif_time, data->try_time);
                fflush(f);
        }
        fprintf(f, "done");
        fflush(f);
        fclose(f);
        while (j>0) {
                data->current_page = pages[--j];
                ioctl(fd, WR_PAGE, (struct my_data*) data);
        }
}


int main() {
        int i = 0;
        int fd;

        fd = open("/dev/etx_device", O_RDWR);
        if(fd < 0) {
                printf("Cannot open device file...\n");
                return 0;
         }
        array_mem_process(fd);

        close(fd);

        return 0;
}
```

Item B: freelist_driver.c (kernel driver)

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include<linux/slab.h>                  //kmalloc()
#include<linux/uaccess.h>               //copy_to/from_user()
#include <linux/ioctl.h>



#define WR_PAGE _IOW('a','a',struct page*)
#define RD_PAGE _IOR('a','b',struct page*)


DECLARE_PER_CPU(unsigned long int, freelistCounter);
DECLARE_PER_CPU(unsigned long int, freelistIf);
DECLARE_PER_CPU(unsigned long int, freelistEndif);
DECLARE_PER_CPU(unsigned long int, freelistTry);


dev_t dev = 0;
static struct class *dev_class;
static struct cdev etx_cdev;

unsigned long int check = 18000000000000000000;
unsigned long int fix_loop = 18446744073709551615;
unsigned long int freelist, if_start, endif_start, try_start;

static int __init etx_driver_init(void);
static void __exit etx_driver_exit(void);
static int etx_open(struct inode *inode, struct file *file);
static int etx_release(struct inode *inode, struct file *file);
static ssize_t etx_read(struct file *filp, char __user *buf, size_t
len,loff_t * off);
static ssize_t etx_write(struct file *filp, const char *buf, size_t len,
loff_t * off);
static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long
arg);

struct my_data
{
        struct page *current_page;
        u64 time;
       unsigned long int freelist_inc, if_time, endif_time, try_time;
};

struct my_data data;
```

```
static struct file_operations fops =
{
        .owner          = THIS_MODULE,
        .read           = etx_read,
        .write          = etx_write,
        .open           = etx_open,
        .unlocked_ioctl = etx_ioctl,
        .release        = etx_release,
};

static int etx_open(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "Device File Opened...!!!\n");
        return 0;
}

static int etx_release(struct inode *inode, struct file *file)
{
        printk(KERN_INFO "Device File Closed...!!!\n");
        return 0;
}

static ssize_t etx_read(struct file *filp, char __user *buf, size_t len,
loff_t *off)
{
        printk(KERN_INFO "Read Function\n");
        return 0;
}
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t
len, loff_t *off)
{
        printk(KERN_INFO "Write function\n");
        return 0;
}


static void get_per_cpu(void){
        freelist = get_cpu_var(freelistCounter);
        if_start = get_cpu_var(freelistIf);
        endif_start = get_cpu_var(freelistEndif);
        try_start = get_cpu_var(freelistTry);
}

static void compare_per_cpu(void){
        data.freelist_inc = get_cpu_var(freelistCounter) - freelist;
        if (data.freelist_inc > check) {
                data.freelist_inc = fix_loop - data.freelist_inc;
        }
        data.if_time = get_cpu_var(freelistIf) - if_start;
        if (data.if_time > check) {
                data.if_time = fix_loop - data.if_time;
        }
        data.endif_time = get_cpu_var(freelistEndif) - endif_start;
        if (data.endif_time > check) {
                data.endif_time = fix_loop - data.endif_time;
        }
        data.try_time = get_cpu_var(freelistTry) - try_start;
```

```
        if (data.try_time > check) {
                data.try_time = fix_loop - data.try_time;
        }
}

static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
        u64 start_time;
        u64 end_time;

        switch(cmd) {
                case WR_PAGE:
                        copy_from_user(&data.current_page ,(struct my_data*)
arg, sizeof(data));
                        __free_pages(data.current_page, 6);
                        break;
                case RD_PAGE:
                       get_per_cpu();
                       start_time = ktime_get_ns();
                       data.current_page = alloc_pages(GFP_KERNEL, 6);
                       end_time = ktime_get_ns();
                       data.time = end_time - start_time;
                       compare_per_cpu();
                        copy_to_user((struct my_data*) arg,
&data.current_page, sizeof(data));
                       break;
        }
        return 0;
}


static int __init etx_driver_init(void)
{
        /*Allocating Major number*/
        if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
                printk(KERN_INFO "Cannot allocate major number\n");
                return -1;
        }
        printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

        /*Creating cdev structure*/
        cdev_init(&etx_cdev,&fops);

        /*Adding character device to the system*/
        if((cdev_add(&etx_cdev,dev,1)) < 0){
            printk(KERN_INFO "Cannot add the device to the system\n");
            goto r_class;
        }

        /*Creating struct class*/
        if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
            printk(KERN_INFO "Cannot create the struct class\n");
            goto r_class;
        }

        /*Creating device*/
        if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
```

```
            printk(KERN_INFO "Cannot create the Device 1\n");
            goto r_device;
        }
        printk(KERN_INFO "Device Driver Insert...Done!!!\n");
    return 0;

r_device:
        class_destroy(dev_class);
r_class:
        unregister_chrdev_region(dev,1);
        return -1;
}

void __exit etx_driver_exit(void)
{
        device_destroy(dev_class,dev);
        class_destroy(dev_class);
        cdev_del(&etx_cdev);
        unregister_chrdev_region(dev, 1);
    printk(KERN_INFO "Device Driver Remove...Done!!!\n");
}

module_init(etx_driver_init);
module_exit(etx_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("DSiegers");
MODULE_DESCRIPTION("A simple device driver for kernel function timing");
MODULE_VERSION("1.0");
```

Item C: Github Repository for IOCTL code

https://github.com/dsiegers/memory_modules