PERFORMANCE ANALYSIS AND CONTROL OF LATENCY UNDER MEMORY PRESSURE IN THE LINUX KERNEL FOR EDGE COMPUTING

by

Smita Koralahalli Channabasappa

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2019

ABSTRACT

SMITA KORALAHALLI CHANNABASAPPA. Performance Analysis and Control of Latency under Memory Pressure in the Linux Kernel for Edge Computing. (Under the direction of DR. ARUN RAVINDRAN)

The Edge computing paradigm seeks to bring Cloud-like compute capabilities close to the Edge of the network, next to where the data is generated, so as to minimize the data communication latency. Edge applications such as autonomous driving, surveillance for accident and crime detection, and robotics are latency sensitive. To ensure low end-to-end latency, the impact on latency of all layers of the computing stack needs to be considered. In this thesis, we investigate the impact of multiple applications sharing the memory on the compute latency. We present a comprehensive experimental evaluation of the impact of different types of co-located memory applications on the latency sensitive application. We choose YOLOv3, a deep learning based object recognition system as an example of a latency sensitive application. We synthesize microbenchmarks that capture the various characteristics of memory intensive background applications. We show that at a high memory utilization due to the co-located microbenchmark, YOLOv3 suffers a latency degradation of 20x compared to low memory utilization situations. To mitigate the impact on latency due to memory intensive applications, we propose and evaluate latency control strategies based on the recently available Pressure Stall Information feature in the Linux kernel. We show that using *latd* our proposed user space latency controller, the latency constraints of YOLOv3 are not significantly violated despite the high memory pressure exerted by background memory intensive microbenchmarks. The thesis thus makes it possible to practically deploy latency sensitive applications along with memory intensive background applications on the same physical machine at the Edge while efficiently utilizing the memory.

ACKNOWLEDGEMENTS

I would first like to express my sincere gratitude to my thesis advisor Dr.Arun Ravindran for his motivation and support in my Master's study and thesis. This work would have not been possible without him who always directed me with patience whenever I ran into a trouble or had a question about my research or writing. Even though he allowed it to be my own work he steered me in the correct direction continuously.

Besides my advisor, I would like to thank rest of my thesis committee: Dr. Hamed Tabkhi and Dr. James M Conrad for their valuable inputs and comments.

Finally, I must express my gratitude to my parents, my brother for their continuous support and encouragement throughout my years of study.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

BCC	BPF Compiler Collection.
BPF	Berkeley Packet Filters.
CDF	Cumulative Distribution Function.
OOM	Out of Memory.
PID	Process ID.
PSI	Pressure Stall Information.
RAM	Random Access Memory.
RSS	Resident Set Size.

CHAPTER 1: INTRODUCTION

Edge computing paradigm aims to minimize latency and also have cloud-like compute capabilities close to the edge of the network. [1]. Most of the edge applications such as autonomous driving, surveillance for accident and crime detection, and robotics are latency sensitive. To ensure low end-to-end latency, the impact on latency of all layers of the computing stack needs to be considered. In this thesis, we investigate the impact of multiple applications sharing the memory on the compute latency. Cloud like approaches of isolating latency sensitive applications in separate virtual machines (or even physical machines) are not applicable at the Edge due to limited hardware available at the Edge owing to space, power, and cost constraints.

A rich body of work in real-time computing exists on scheduling and synchronization algorithms that investigates how compute resources can be shared effectively between latency sensitive, and background applications [2]. However, memory is a shared resource, virtualized and managed by the operating system, and is not directly controlled by applications. Modern operating systems such as Linux kernel uses main memory as a cache for disk based files using the page cache mechanism [3]. Additionally, memory is used as a store for non-disk based storage allocations such as those involving automatic variables (stack), and user memory allocation (heap). The policies employed by the Linux kernel are targeted at maximum system throughput, rather than maintaining the latency constraints of individual applications. Unfortunately, as demonstrated in this thesis, when latency critical Edge applications (for example real-time object detection), share the system with memory intensive, but non-latency critical application (for example database for storing images), under conditions of high memory utilization, the latency sensitive applications can experience large tail latencies.

1.1 Summary of approach

In this thesis, we present a comprehensive experimental evaluation of the impact of different types of co-located memory applications on the latency sensitive application. We choose YOLOv3, a deep learning based object recognition system as an example of a latency sensitive application. YOLOv3 is used as a stand-alone as well as a part of many machine vision computing pipelines at the Edge. We synthesize microbenchmarks that capture the various characteristics of memory intensive background applications. While these co-located applications are important, they are not typically latency sensitive. Examples include datastores such as MongoDB, and batch analytics such as Spark. We show that at a high memory utilization due to the co-located microbenchmark, YOLOv3 suffers a latency degradation of 20x compared to low memory utilization situations. To mitigate the impact on latency due to memory intensive applications, we propose and evaluate latency control strategies based on the recently available Pressure Stall Information feature in the Linux kernel. We show that using *latd* our proposed userspace latency controller, the latency constraints of YOLOv3 are not significantly violated despite the high memory pressure exerted by background memory intensive microbenchmarks. The thesis thus makes it possible to practically deploy latency sensitive applications along with memory intensive background applications on the same physical machine at the Edge while efficiently utilizing the memory.

1.2 Key Contributions

The thesis makes the following contributions

1. Experimental characterization of YOLOv3 as a latency critical application under different memory intensive background applications.

2. Reduce latency under high memory pressures with the proposed userspace la-

tency controller *latd* which makes use of Pressure Stall Information and tunable Linux kernel parameters.

3. Demonstration of the usability of *cgroups* depending on the behavior of background application or microbenchmark.

4. Writing tracing tools for extracting system information needed for the research.

1.3 Organization of thesis

The rest of the thesis is organized as follows. Chapter 2 presents a brief background on the Linux kernel, tools for performance analysis, and prior work related to this research. Chapter 3 describes the experimental evaluation of latency degradation under high memory pressure. Chapter 4 proposes control strategies to mitigate the latency violations allowing multiple applications to coexist on the same hardware at the Edge. Chapter 5 concludes the thesis summarizing our results, and with suggestions for future research directions.

CHAPTER 2: BACKGROUND

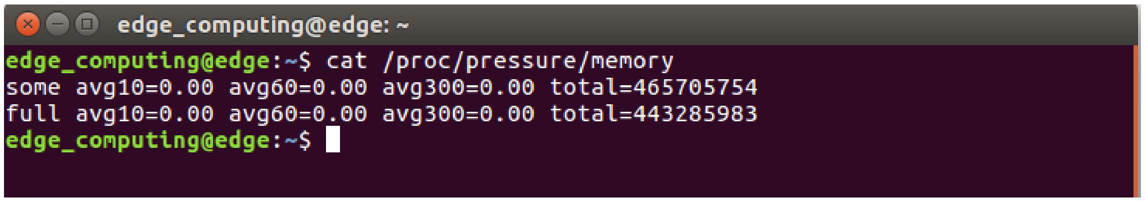
In this chapter we provide a brief background of the different topics used in this thesis including the page cache, the pressure stall information kernel feature, virtual memory tuning parameters, kernel tracing tools, and kernel resource management.

2.1 Page Cache

The page cache is a layer between the kernel memory management code and the disk I/O system. Pages read from a file or block device are generally added to the Page Cache to avoid further disk I/O. The operating system keeps a page cache in otherwise unused portions of the main memory (RAM), resulting in quicker access to the contents of cached pages and overall performance improvements. The kernel writes the cache pages out to disk as necessary in order to create free memory. Page caches are motivated by temporal locality, and disk being much slower than memory. The kernel maintains a number of page lists which collectively comprise the page cache. The *active_list*, the *inactive_dirty_list* and the *inactive_clean_list* [4] are used to maintain a least-recently-used sorting of user pages. Usually, all physical memory not directly allocated to applications is used by the operating system for the page cache. Anonymous pages (those without a disk file to serve as backing storage - pages of malloc memory, for example) are assigned an entry in the system swapfile, and do not get added to the page cache [3] whereas pages mapped from files begin life in the page cache.

2.2 PSI-Pressure Stall Information

PSI [5] [6] is a newly available Linux kernel feature (kernel version >= 4.20) that aggregates and reports the overall wallclock time in which the tasks in a system wait for contended hardware resources. It tracks three major system resources CPU, memory and I/O over time. The percentage of time the system is stalled on the CPU, memory, or I/O is exposed via /proc/pressure/ as pressure percentages. The patch can be added to older kernels too manually which requires recompilation and rebuilding the kernel by [7] and setting CONFIG_PSI=y in the build configuration. A kernel with CONFIG_PSI=y will create a /proc/pressure directory with 3 files: CPU, Memory, and I/O. Memory file contains two lines:



```
edge_computing@edge:~$ cat /proc/pressure/memory
some avg10=0.00 avg60=0.00 avg300=0.00 total=465705754
full avg10=0.00 avg60=0.00 avg300=0.00 total=443285983
edge_computing@edge:~$
```

Figure 2.1: PSI Metric Display

The averages give the percentage of wall clocktime in which one or more tasks are delayed. These are recent averages over 10 sec, 1 min, 5 min windows. The *total* statistic gives the absolute stall time in microseconds. The *some* statistic gives the time where some (one or more) tasks were delayed due to lack of resource. The *full* statistic, indicates time in which no task is using the resource productively due to resource pressure, that is all non-idle tasks are waiting for resource in one form or another.

2.3 Virtual Memory kernel parameters

We briefly describe the different control knobs (kernel parameters) that can be used to tune page cache behavior in Linux. These kernel parameters can be set from user space using the *sysctl* [8] tool.

2.3.1 Swappiness

Swappiness is a Linux kernel parameter whose range can be set to values between 0 and 100 inclusive. A low value causes the kernel to prefer to evict pages from the page cache. The default value of *swappiness* of anonymous pages is 60. The priority value of file backed pages is calculated as $200 - priority\_of\_anonymous\_pages$. Note that lower values indicate higher priority. The default values thus favor the anonymous pages over the file backed pages. The code in Appendix A shows how to set this *swappiness* value using *sysctl* tool.

2.3.2 Dirty_ratio and Dirty_background_ratio

Dirty_ background_ ratio gives the percentage of total available memory which when dirty, the system via flusher threads write dirty data to disk. The default value is 10. On the other hand *dirty_ ratio* is the percentage of total available memory which when dirty, the process generating the dirty pages, stops further I/O, and starts writing out dirty data. A low value causes frequent I/O pauses affecting performance. By keeping the *dirty_ background_ ratio* to a low value, and *dirty_ ratio* to a high value, flusher threads can be invoked frequently without pausing the process. The dirty pages can be monitored through /proc/vmstat or using biotop BCC tool (explained in Section 2.4).

2.3.3 Dirty_expire_centiseconds

Dirty_ expire _ centisecs is the time based service which writes back all modified data once every specific time interval which is defaulted in the Linux kernel to 30 seconds. Tuning this to wake up frequently incurs greater overhead when there are few dirty pages, due to context switching overhead.

2.3.4 Overcommit_memory

It contains a flag that enables memory overcommitment. When this flag is 0, the kernel attempts to estimate the amount of free memory left when userspace requests more memory. At value 1, the kernel pretends there is always enough memory until it actually runs out. At value 2, the kernel uses a never overcommit policy that attempts to prevent any overcommit of memory. We set it at value 0 for enabling default over-commitment for complete memory utilization in the system.

2.4 BPF Compiler Collection

BCC (BPF Compiler Collection) is a powerful set of tools for kernel tracing. It utilizes extended Berkeley Packet Filters (eBPF) introduced in Linux 3.15. Most of the components used by BCC requires Linux 4.1 or above. eBPF can be used to run user defined sandboxed programs in the kernel safely and efficiently without the need for kernel modules. eBPF has been used to implement a number of kernel performance monitoring tools under the BCC toolkit. The thesis uses the following BCC tools downloaded from the repository [9] -

- *cachetop* shows Linux page cache hit/miss statistics including read and write hit % per processes.
- *cachestat* shows hits and misses in the page cache.
- *biotop* summarizes block device I/O per process.

2.5 cgroups

A *cgroup* is a logical grouping of processes that can be used for resource management in the kernel. Once a *cgroup* has been created, processes can be migrated in and out of the *cgroup* via a pseudo-filesystem API. Resource usage within *cgroup* is managed by attaching controllers to a *cgroup*. The CPU controller mechanism allows a system manager to control the percentage of CPU time given to a *cgroup*. The memory controller mechanism can be used to limit the amount of memory that a process uses. This work requires *cgroup* version 2.

2.6 Signals

A signal is a very short message that may be sent to a process or a group of processes to make a process aware that a specific event has occurred. The thesis uses the following signals -

- *SIGSTOP* suspends the execution of the current process until resumed. This signal cannot be ignored by the process.
- *SIGCONT* resumes the execution of the paused process by *SIGSTOP*.
- *SIGTERM* causes program termination. This signal can be blocked, handled and ignored. It terminates the process gracefully by cleaning its process tables in memory.

2.7 OOM Killer

Out of Memory killing is a process which is employed by the Linux kernel when its typically low on memory and the corresponding killer is called by kernel to free some memory. It is often encountered on servers which have a number of memory intensive processes running and processes request for more memory than is physically available. Linux kernel basically reviews all running processes, assigns them a badness score called /oom_score [10] and kills one or more of them in the decreasing order of their badness score in order to free up system memory and keep the system running.

The file in */proc/PID/oom_score* displays the current score that the kernel gives to the mentioned PID process for the purpose of selecting a process by the OOM-killer. A higher score means that the process is more likely to be selected by the OOM-killer. The oom-killer can be disabled in target application by adjusting the score in */proc/PID/oom_adj*. The valid values are in the range -16 to +15, the special value -17 disables OOM-killing altogether for the process. A positive score increases the likelihood of this process being killed by the OOM-killer whereas a negative score decreases the likelihood.

2.8 Related Work

Managing the available resources and satisfying the memory demands for various applications running in the system has always been a topic of research in the systems level. But they are more oriented towards maximizing system throughput as a whole rather than maintaining latency constraints of each applications.

Oomd [11] developed by Facebook takes corrective action in userspace before an oom-killer occurs in kernel space. By default, this involves killing offending processes. It leverages PSI [5] and cgroupv2 [12] to monitor a system holistically and offers flexibility where each workload can have custom protection rules.

Earlyoom [13] an OOM preventer with minimum dependencies checks the amount of available memory and free swap 10 times a second and by default if both are below 10% it will terminate the largest process with the highest oom_score. The percentage value is configurable via command line arguments. It will send the SIGTERM signal to the process that uses the most memory in the opinion of the kernel by looking into the */proc/(pidof processes)/oom_score* [10].

Nohang [14] does the similar action of terminating the process with more useful information in terms of system data and printing statistics as compared to Earlyoom [13]. It claims to correctly prevent out of memory (OOM) and keep system responsiveness in low memory conditions by terminating the application.

All approaches try to overcome oom livelocks and freeze by minimizing the time spent by the kernel in freeing pages which is a slow and painful process because the kernel can spend an unbounded amount of time swapping in and out pages and evicting the page cache. Though system responsiveness and increased throughput can be achieved, the latency suffered by individual applications under pressure is not paid significant attention. Hence this thesis presents a comprehensive evaluation on the inference latency of the latency sensitive application under pressure and tries to minimize the latency and also achieve high memory utilization.

CHAPTER 3: EXPERIMENTAL CHARACTERIZATION OF LATENCY UNDER MEMORY PRESSURE

In this chapter we present our experimental study of the impact of memory pressure on latency. As outlined in Chapter 1, the need to achieve low latency is one of the key motivations behind Edge computing. To gain a better understanding of memory pressure on application latency, we quantitatively characterize the impact of a typical latency sensitive Edge application using microbenchmarks that simulate memory pressure. The application we choose is YOLOv3 [15][16], a Deep Learning based real-time object detector.

Our experimental setup consists of an a Dell laptop Intel Core i7 CPU and Nvidia GEFORCE GTX 1060 GPU with 16GB memory, running Ubuntu 16.04 LTS with Linux kernel version 4.20. Given the power and space constraints at the Edge, the specifications above are close to a typical server available at the Edge.

3.1 Characterizing YOLOv3

Our first objective is to characterize the latency and memory requirements of YOLOv3 on a lightly loaded system. YOLOv3 uses the GPU for running the Deep learning network. PyTorch and other dependencies required by YOLOv3 are installed on the machine. Pre-trained weights are downloaded from the YOLOv3 repository, and stored in the file system[17]. The workload consists of a series of 1000 images that YOLOv3 reads from the file system. The images are 100 KB to 200 KB in size, and consists of natural scenes, people, and animals. The output is the classes of the objects detected in the image, and the latency incurred in each detection event.

We first characterize the image inference latency of YOLOv3 application running across 1000 images on a lightly loaded machine. Figure 3.1 plots the latency Cumulative Distribution Function (CDF). The latency ranges from 0.03 - 0.045 seconds, with the 95th percentile at 0.04 seconds.

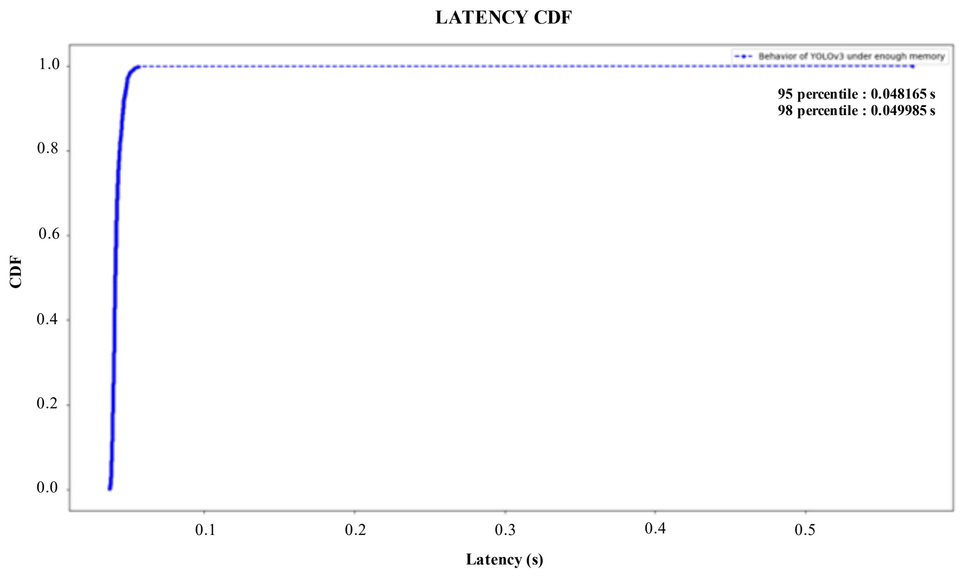


Figure 3.1: Latency CDF for YOLOv3 on a lightly loaded machine

We then determine the memory resident set size (RSS) of YOLOv3, both to understand the Deep Learning model requirements, as well as the the memory of the image data. We observe that prior to processing the images, the memory RSS is 0.98 GB. The measurement is done by examining the individual process under $proc/(pidof(yolov3))/statm$. The measurement script is included in Appendix B. We then plot the memory RSS as YOLOv3 infers objects on images. As shown in Figure 3.2, the total memory RSS(including anonymous and file backed) increases linearly with the number of images processed to a maximum of 2.1 GB. We also determine the page cache size using *vmstat* (script included in Appendix D, and note that the page cache size increases from 1.5 GB to 2.7 GB. The average page cache miss rate is 9%. The 9% miss rate accounts for presence of anonymous pages in YOLOv3. The total anonymous and file backed pages in an application running can be determined dynamically by the script included in Appendix F. The above measurements show that YOLOv3 makes extensive use of the page cache. Interestingly, the page cache remains in use despite the images being only processed once.

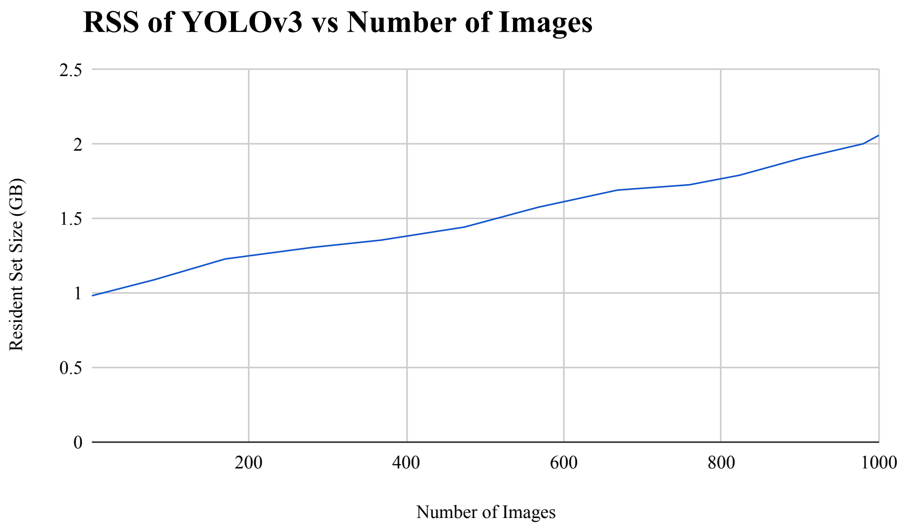


Figure 3.2: Memory Consumption of YOLOv3 (Resident Set Size) over number of Images

3.2 Memory pressure microbenchmarks

Our goal is to determine the impact on YOLOv3 inference latency under memory pressure. Memory usage could result either from use of anonymous pages (through malloc), or from file backed pages. For example, the in-memory database Memcached [18][19] allocates memory chunks, while persistent databases such as MySQL [20][21] allocate file backed pages. To simulate these two behaviors, we devise two microbenchmarks

1. Anonymous page memory consumer workload (code included in Appendix C)
2. File backed page memory consumer workload (code included in Appendix G).

3.2.1 Anonymous page memory consumer microbenchmark

The microbenchmark and YOLOv3 are run simultaneously on the system. The microbenchmark allows the tuning of anonymous pages consumed. We experiment with different page consumption levels, and note that at 99% of total memory consumed (by both applications), the YOLOv3 inference latency increases substantially. Figure 3.3 shows the time series plot of the inference latency exhibited by YOLOv3.

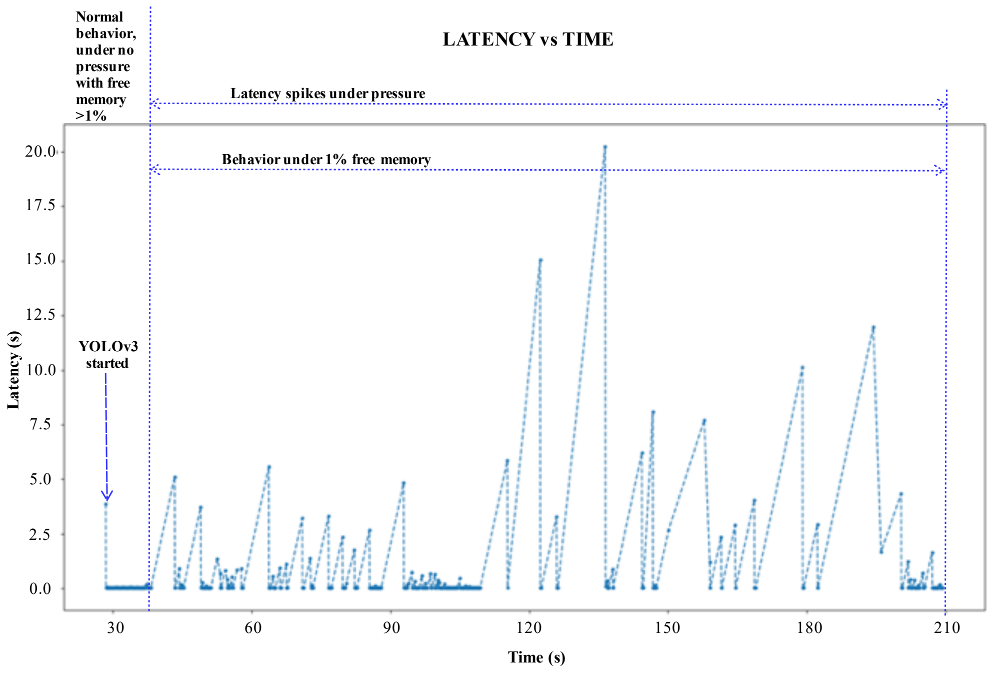


Figure 3.3: Time series plot of the inference latency suffered by YOLOv3 under the effect of Anonymous Page Memory Consumer Microbenchmark

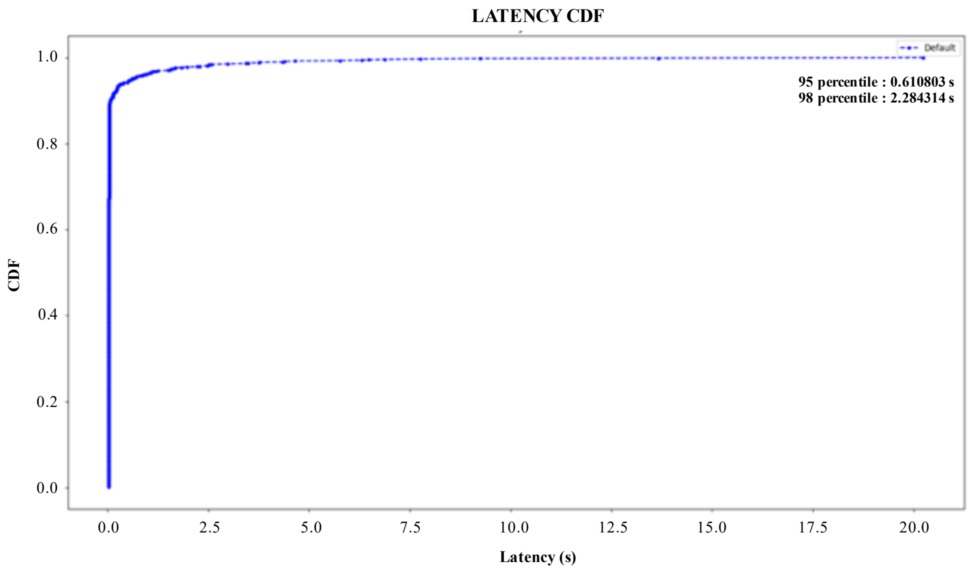


Figure 3.4: Latency CDF for YOLOv3 under memory pressure simulated by Anonymous page memory consumer microbenchmark

In Figure 3.4, the CDF plot of the latency is shown. At the 95th percentile, the latency is 0.6 seconds, which represents a 20x increase in latency compared to YOLOv3 running in isolation.

We also examine the memory RSS of YOLOv3 as shown in Figure 3.5, and note that it decreased from 1.72 GB to 1.62 GB. The page cache size also showed a corresponding decrease under pressure from 2.3 GB to 1.94 GB. The average page cache miss rate is 38.64% which is thrice more than YOLOv3 running in isolation. The above results indicate that the consumption of pages by the microbenchmark, decreases the page cache size, increasing the page cache miss rate, and resulting the increased latency experienced by YOLOv3. Interestingly, the latency does not show appreciable increase before the 99% memory consumption level indicating that certain critical pages needed for the application (for example, model weights) were being evicted from page cache beyond a certain memory pressure level.

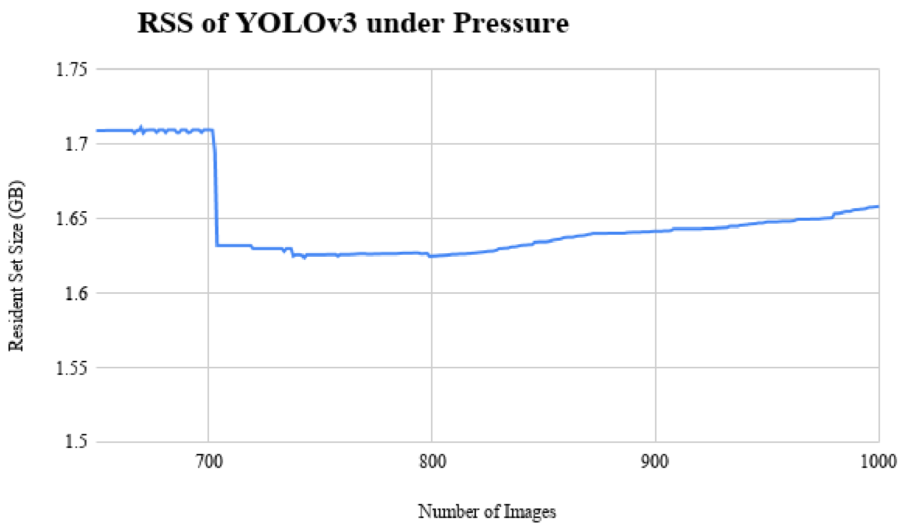


Figure 3.5: Effects on Resident Set Size of YOLOv3 under the influence of Anonymous Page Memory Consumer Microbenchmark

3.2.2 File backed page memory consumer microbenchmark

The *mmap* API with either MAP_PRIVATE flag is used by the microbenchmark to create file backed pages. The workload and YOLOv3 are run simultaneously on the system. Unlike the anonymous page benchmark, the inference latency depends on the page access pattern. If the pages allocated by the microbenchmark are only accessed once, there is no impact on the YOLOv3 latency despite high memory usage. This is supported by page cache miss rate experienced by YOLOv3 which at 9.69% is almost similar to the the miss rate on a lightly loaded machine. The page replacement mechanism preferentially chooses the inactive pages from the microbenchmark as compared to YOLOv3 pages. This is confirmed by measurements of the RSS size of the microbenchmark which decreases from 12.8 GB to 12.2 GB.

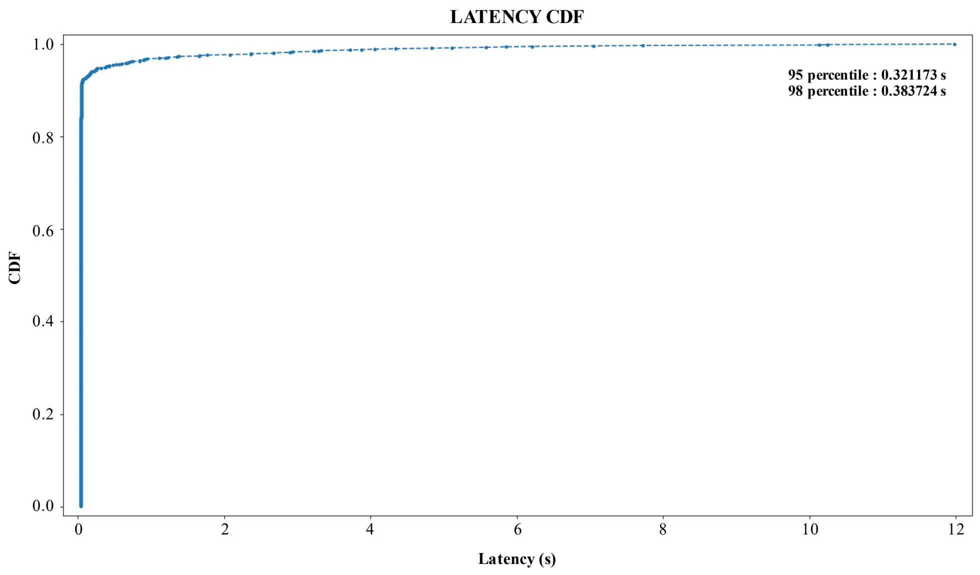


Figure 3.6: Latency CDF for YOLOv3 under memory pressure simulated by File backed page memory consumer microbenchmark

On the other hand, if the pages allocated by the microbenchmark is accessed frequently, at 98.9% of memory utilization, YOLOv3 experiences a substantial increase in latency of 0.32 seconds at the 95th percentile which represents a 10x increase in latency compared to YOLOv3 running in isolation. In Figure 3.6, the CDF plot of the latency is shown. Figure 3.7 shows the time series plot of the inference latency exhibited by YOLOv3.

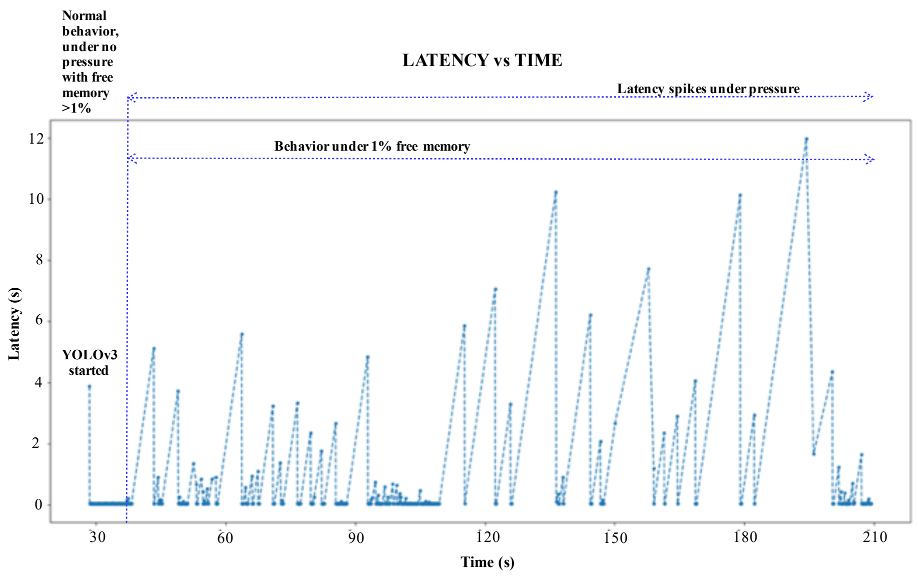


Figure 3.7: Time series plot of the inference latency suffered by YOLOv3 under the effect of File backed Page Memory Consumer Microbenchmark

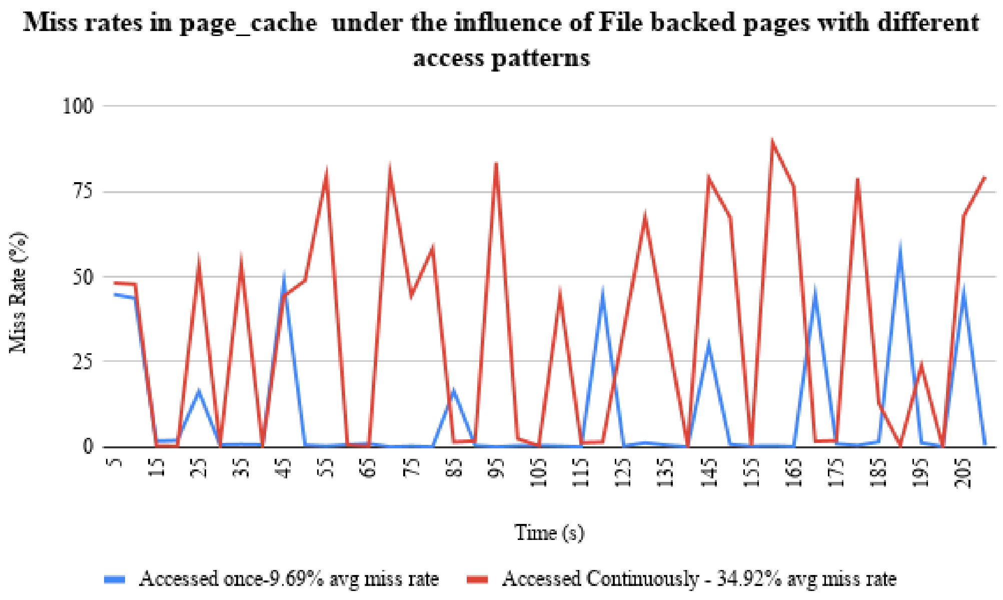


Figure 3.8: Miss rates in page_cache under the influence of file backed page memory consumer microbenchmark accessed once and accessed continuously

This observation corresponds to a substantial increase in miss rate experienced by

YOLOv3 which increased to 34.92% as compared to the lightly loaded case. Figure 3.8 shows the miss rates suffered by YOLOv3 under the influence of file backed pages accessed once and accessed continuously. Also, we observe that the page cache size decreases and the corresponding RSS for both YOLOv3 and microbenchmark, indicating the pages are now being evicted from both the applications.

3.3 Discussion

From the above experiments we note that the impact on latency on YOLOv3 depends not only on the amount of memory consumed, but also on the type of pages backing the memory, and the access patterns. Anonymous pages have higher preference to be retained in the memory despite the presence of a swap space, and hence impacts YOLOv3 adversely. On the other hand, with file backed pages, the impact depends on the access patterns. If the pages are accessed infrequently, then the pages are evicted from the page cache for use by YOLOv3. However, if the pages are accessed frequently, the latency of YOLOv3 is affected, since the page cache is now being used by both applications.

CHAPTER 4: DESIGN OF LATENCY CONTROLLER

We have shown in Chapter 3, the adverse impact of high memory pressure on latency of an object-detection application (YOLOv3). In this chapter, we present the design of a user space latency controller (*latd*), that ensures that the high latency events are avoided in the presence of memory intensive background applications. We use the newly available feature in the kernel (available from kernel version >= 4.20) known as Pressure Stall Information (PSI)[5][7] (see Section 2) that reports the overall wall clock time in which the tasks in a system (or cgroup) wait for contended hardware resources. While PSI has been used to tackle Out-Of-Memory (OOM) problems [6] [11], in this thesis we use PSI to monitor memory pressure and take corrective action before it impacts latency. We study the two causes of memory pressure, that is consumption of anonymous pages, and those of file backed pages separately.

4.1 *Latd* Controller Design for Anonymous Page Memory Dominant Microbenchmark

Algorithm 1 and Figure 4.1 shows the architecture of the Latd controller. See Appendix A for the complete code. The design of the latd controller is as follows-

- This controller is run as *root* to assign higher priority over other processes. Root process is given 3% higher priority while running which helps CPU and the scheduler to prioritize latd process over others under pressure and take immediate actions.
- *Swappiness* and *overcommit_memory* kernel parameters are set to 90 and 0 respectively since YOLOv3 is mostly file backed. The swappiness value of 90 gave the best latency inference. Overcommit value of 0 instructs the kernel to

overcommit memory by default.

Algorithm 1 Architecture of Latd

```
1: Initialize latd as root
2: Set sysctl -w vm.swappiness = 90
3: Set sysctl -w vm.overcommit_memory = 0
4: Start yolov3 and disable oom-killer as echo '-17' > /proc/(pidof yolov3)/oom_adj
5: while $\delta t$==0.1s do
6:     Scan oom-scores of all processes and extract process with highest oom-score
7:     if some PSI increase then
8:         sysctl -w vm.dirty_background_ratio = 2
9:         sysctl -w vm.dirty_ratio = 100
10:        if full PSI increase then
11:            Stop process with highest oom-score
12:        end if
13:    end if
14:    if some PSI stable && dirty_ratio && background_ratio not default then
15:        if full PSI is stable and process is stopped then
16:            Restart the stopped process
17:        end if
18:        sysctl -w vm.dirty_background_ratio = 10
19:        sysctl -w vm.dirty_ratio = 20
20:    end if
21:    if available memory is 0.1% of total then
22:        Terminate highest oom-scored process
23:        break
24:    end if
25: end while
```

- Disables OOM killing for YOLOv3 by writing the value of -17 to */proc/(pidof yolov3)/oom_adj*.
- Resource pressure in the system is determined by tracking absolute stall time of *some* and *full* PSI statistics, sampling ten times per second.
- If there is an increase in *some* PSI statistics indicating memory pressure, *dirty _background_ratio* is changed to 2, and *dirty_ratio* to 100, to awaken kernel flusher threads more frequently to write back dirtied pages to disk.
- If there is an increase in *full* PSI statistic indicating lack of memory, stop the

process with highest OOM score (scan the list of all *PID* in *proc* directory and extract OOM score from each process and determine highest OOM score).

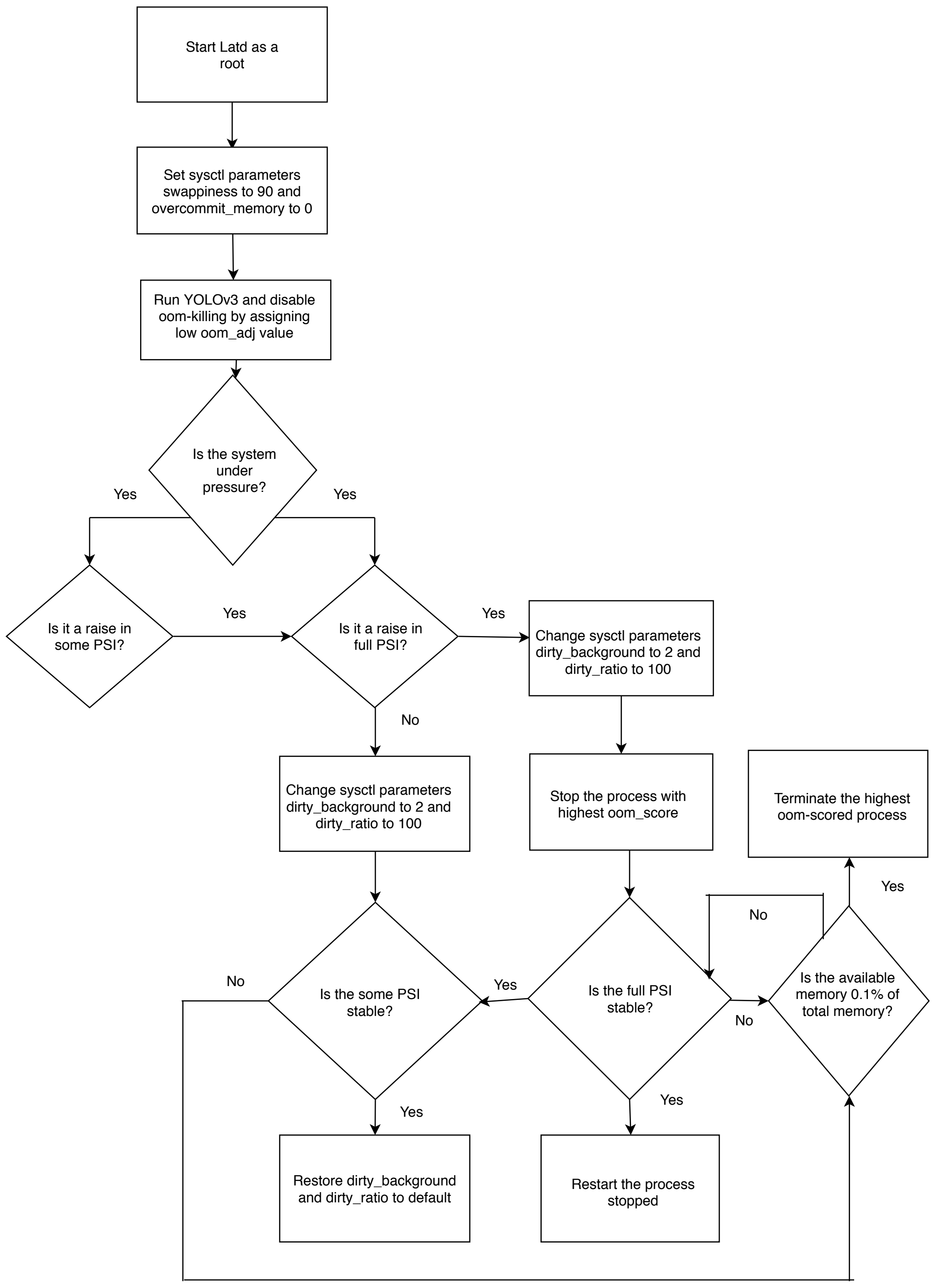

Start Latd as a root
Set sysctl parameters swappiness to 90 and overcommit_memory to 0
Run YOLOv3 and disable oom-killing by assigning low oom_adj value
Is the system under pressure?
Yes
Yes
Is it a raise in some PSI?
Yes
Is it a raise in full PSI?
Yes
Change sysctl parameters dirty_background to 2 and dirty_ratio to 100
No
Change sysctl parameters dirty_background to 2 and dirty_ratio to 100
Stop the process with highest oom_score
Terminate the highest oom-scored process
Yes
No
Is the some PSI stable?
No
Yes
Is the full PSI stable?
No
Is the available memory 0.1% of total memory?
Yes
Yes
Restore dirty_background and dirty_ratio to default
Restart the process stopped

Figure 4.1: Flowchart of the working of *Latd* to reduce memory pressure

- Restart the process if *full* PSI statistic is stable and restore the default dirty _background_ratio, dirty_ratio if *some* PSI is stable.

- Continue doing this iteratively until available memory is 0.1% in the system. If available memory hits 0.1% of total then terminate the process with highest oom_score using SIGTERM (see Section 2.6).

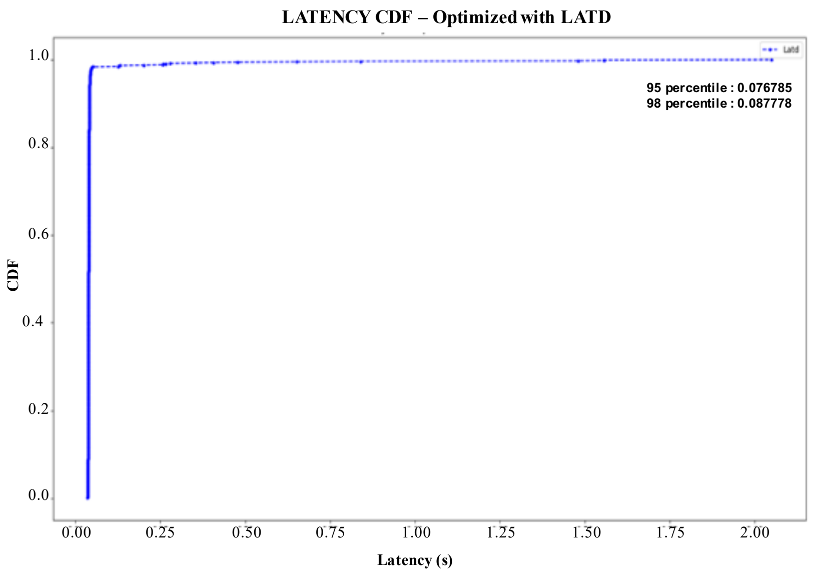


Figure 4.2: CDF Plot of Inference Latency of YOLOv3 with *Latd*

Figure 4.2 shows the CDF plot of YOLOv3 latency at 99% memory usage. We note that the 95^{th} percentile latency decreased from 0.6 second to 0.07 seconds. This represents a 10x improvement compared to non-control case. Also, it represents only a 2x degradation in latency compared to the lightly loaded case. With only 2x degradation in latency we are able to achieve a higher memory utilization.

4.2 *cgroups* for File backed Page Memory Dominant Microbenchmark

As we saw in Chapter 3 YOLOv3 makes extensive use of the page cache. For co-located workload that puts pressure on the page cache, the *sysctl* based VM parameters are of limited use, since we do not have control over per-process use of the page cache in the Linux kernel. As a result flushing pages to the disk due to memory pressure, will impact the latency sensitive YOLOv3 applications as well. We, therefore, use the Linux *cgroups* capability described in Chapter 2, to isolate the two processes. *cgroups* allows us to specify memory resource limit per process. If the process exceeds the memory limit, it is killed by the kernel. Note that the page cache is still shared between the two processes. As a result, even if one of the processes has spare memory capacity, the kernel will kill the other process that exceeds its memory limit, since from the kernel's view the system is not under memory pressure. This is determined as *some* PSI statistic doesn't evidence any increase when the microbenchmark is close to its memory limit. Appendix H includes the code for creation and management of *cgroups*.

Algorithm 2 *cgroup* management for latency reduction

```
 1: Create two cgroups for yolov3 and microbenchmark
 2: Set swappiness to 90 for YOLOv3 assigned cgroup
 3: Set memory.limit_in_bytes to microbenchmark assigned cgroup
 4: Start both applications
 5: Assign the applications to cgroups using their PID
 6: while $\delta t$==0.1s do
 7:     if some PSI of the system increases then
 8:         Stop the highest oom-scored process
 9:     end if
10:     if some PSI of the system is stable then
11:         Restart the highest oom-scored process
12:     end if
13:     if available memory is 0.1% of total then
14:         Terminate highest oom-scored process
15:         break
16:     end if
17: end while
```

Motivated by these observations, and then need to maintain the latency of YOLOv3, our strategy is to place memory limits only for the non-latency sensitive application. YOLOv3 is thus able to utilize the full memory available (minus the memory reserved for the co-located application), and the page cache management system will flush pages appropriately as the memory pressure rises. Also, we set the *swappiness* parameter to 90 for YOLOv3 to ensure that pages are not swapped out frequently from the page cache. Algorithm 2 shows our implementation.

Algorithm 3 Possible direction of *cgroup* management for latency reduction

```
 1: Create two cgroups for yolov3 and microbenchmark
 2: Set swappiness to 90 for YOLOv3 assigned cgroup
 3: Set memory.limit_in_bytes to microbenchmark assigned cgroup
 4: Start both applications
 5: Assign the applications to cgroups using their PID
 6: while $\delta t$==0.1s do
 7:     if some PSI of file backed microbenchmark cgroup increases then
 8:         Spin up another cgroup with higher memory limits
 9:         Assign the process affected by rise in some PSI to new cgroup
10:         Destroy unused cgroup by cgdelete memory:mmap to free memory
11:     end if
12:     if some PSI of the system increases then
13:         Stop the highest oom-scored process
14:     end if
15:     if some PSI of the system is stable then
16:         Restart the highest oom-scored process
17:     end if
18:     if available memory is 0.1% of total then
19:         Terminate highest oom-scored process
20:         break
21:     end if
22: end while
```

The drawback of our approach is that the need to accurately predict the memory requirements of the non-latency sensitive and latency sensitive applications. Even if free memory is available, exceeding the resource limits will cause the kernel to kill the cgroup. The solution to this is in being able to monitor the memory pressure per cgroup. If the pressure exceeds in a particular *cgroup* with enough system memory available, the latd controller could spin up a new cgroup with increased memory. Fortunately, this capability has been made available as a patch of May 10, 2019 by Facebook[22]. Unfortunately, we did not have enough time to explore this option. With this capability the *latd* daemon would be able to make fine grained adjustments to the cgroup capability. Algorithm 3 depicts the possible direction of implementation.

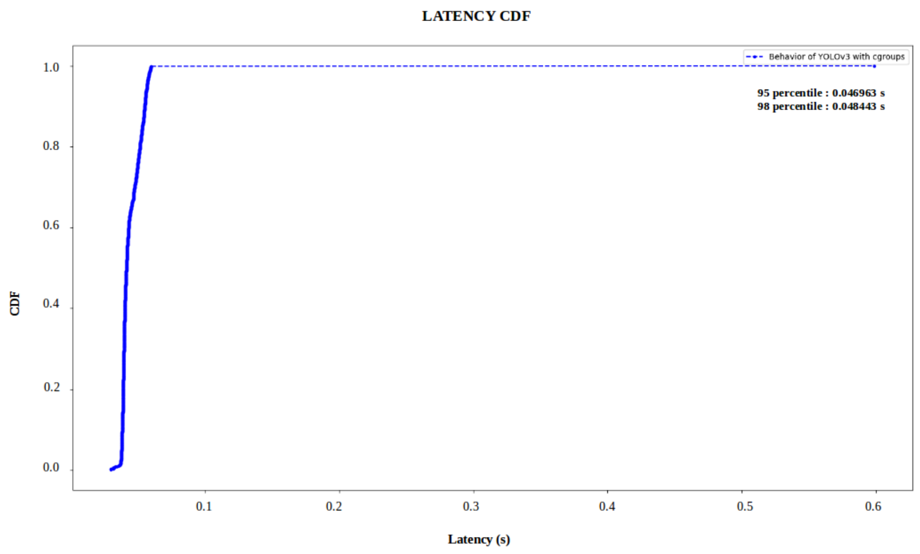


Figure 4.3: CDF Plot of Inference Latency of YOLOv3 with cgroups

In Figure 4.3, the CDF plot of the latency is shown. The 95^{th} percentile latency is 0.04 seconds which is same as the lightly loaded system seen in Chapter 3. We are thus able to achieve high memory utilization with minimal impact on the latency of the application.

CHAPTER 5: CONCLUSIONS AND FUTURE WORK

In this thesis, we have shown the adverse impact memory intensive background applications can have on the latency of real-time object detection applications such as YOLOv3. In an Edge computing system, where resources are limited, co-locating applications on the same physical machine while simultaneously hosting latency sensitive, and background applications becomes a necessity. We show that the recently available Pressure Stall Information in the Linux kernel can be used to monitor memory pressure. A userspace controller can utilize the memory pressure information to take corrective action. We observe that the corrective actions taken, depends on the type of memory consumed by the background application - anonymous or file backed. Using tunable Linux kernel parameters available via the *sysctl* tool, as well as the *cgroups* kernel feature, we are able to effectively limit the impact on the inference latency of YOLOv3, despite high memory pressure. The thesis thus makes it possible to practically deploy latency sensitive applications along with memory intensive background applications on the same physical machine at the Edge while efficiently utilizing the memory. Additionally, the proposed techniques are applicable to Cloud computing systems, to consolidate workloads on fewer machines, saving costs and energy.

Future research directions include a more comprehensive evaluation of our proposed latency controller with realistic background benchmarks. Also, the newly available enhancement to *cgroup* called *cgroupv2*, allows per *cgroup* monitoring of resource pressure, allowing containers to be resized dynamically based on memory availability.

REFERENCES

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, pp. 637–646, Oct 2016.

[2] B. Brandenburg and J. H. Anderson, *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[3] R. Love, *Linux Kernel Development*. Developer's Library, 2010.

[4] M. Gorman, *Understanding The Linux Virtual Memory Manager*. Prentice Hall, 2004.

[5] J. Weiner, "psi: pressure stall information for cpu, memory, and io v2." https://lwn.net/Articles/759658/, 2018. [Online; accessed 20-Feb-2019].

[6] "Pressure stall information." https://facebookmicrosites.github.io/psi/docs/overview. [Online; accessed 20-Feb-2019].

[7] J. Weiner, "Psi patch." http://git.cmpxchg.org/cgit.cgi/linux-psi.git, 2018.

[8] "Sysctl parameters." https://www.kernel.org/doc/Documentation/sysctl/vm.txt. [Online; accessed 20-Feb-2019].

[9] "bcc." https://github.com/iovisor/bcc, 2015. [Online; accessed 20-Feb-2019].

[10] "Linux mm: Oom killer." https://linux-mm.org/OOM_Killer, 2017. [Online; accessed 20-Feb-2019].

[11] "Oomd." https://github.com/facebookincubator/oomd, 2018. [Online; accessed 20-Feb-2019].

[12] "Cgroup-v2." https://www.kernel.org/doc/Documentation/cgroup-v2.txt. [Online; accessed 20-Feb-2019].

[13] "Earlyoom." https://github.com/rfjakob/earlyoom, 2018. [Online; accessed 20-Feb-2019].

[14] "Nohang." https://github.com/hakavlad/nohang, 2018. [Online; accessed 20-Feb-2019].

[15] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[16] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[17] "Pytorch-yolov3." https://github.com/eriklindernoren/PyTorch-YOLOv3, 2019. [Online; accessed 20-Feb-2019].

[18] J. Petrovic, "Using memcached for data distribution in industrial environment," in *Third International Conference on Systems (icons 2008)*, pp. 368–372, IEEE, 2008.

[19] K. A. Bakar, M. H. M. Shaharill, and M. Ahmed, "Performance evaluation of a clustered memcache," in *Proceeding of the 3rd International Conference on Information and Communication Technology for the Moslem World (ICT4M) 2010*, pp. E54–E60, IEEE, 2010.

[20] A. Pareek, M. Lakshminarayanan, A. Dubey, and S. Corbin, "Mysql database heterogeneous log based replication," Aug. 13 2013. US Patent 8,510,270.

[21] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *ACM SIGARCH Computer Architecture News*, pp. 631–644, ACM, 2017.

[22] D. Schatzberg, "psi: Expose pressure metrics on root cgroup." https://lkml.org/lkml/2019/5/10/545, 2019. [Online; accessed 16-May-2019].

APPENDIX A: Latd Daemon

```
#!/bin/bash
# ensure running as root
if [ "$(id -u)" != "0" ]; then
  exec sudo "$0" "$@"
fi

#Defaults changed after evaluating performance and the best fit
bash -c 'sudo sysctl -w vm.overcommit_memory=0'
bash -c 'sudo sysctl -w vm.swappiness=90'

#PSI statistics and memory availability statistics
gnome-terminal --tab --command="bash -c 'cd /home/edge_computing/
    Documents/THESIS_SMITA/thesis; chmod +x memstat.sh; ./memstat.sh
    $SHELL'" --tab --command="bash -c 'cd /home/edge_computing/Documents
    /THESIS_SMITA/thesis; chmod +x psi.sh; ./psi.sh $SHELL'"

#pytorch application
gnome-terminal --tab --command="bash -c 'cd /home/edge_computing/
    Documents/THESIS_SMITA/PyTorch-YOLOv3; python3 detect1.py; $SHELL'"
sleep 5

#Individual Memory stastics for pytorch application
gnome-terminal --tab --command="bash -c 'cd /home/edge_computing/
    Documents/THESIS_SMITA/thesis; chmod +x memory.sh; ./memory.sh
    $SHELL'"

#Extract PID of YOLOV3
PID1="$(pidof python3)"

#Disable oom-kill for YOLOv3
sudo bash -c "echo '-17' | tee /proc/$PID1/oom_adj"
```

```
#Run microbenchmark memory hog
gnome-terminal --tab --command="bash -c 'cd /home/edge_computing/
    Documents/THESIS_SMITA/thesis; ./oomkill; $SHELL'"
PID2="$(pidof oomkill)"

#Individual Memory statistics for microbenchmark memory hog
gnome-terminal --tab --command="bash -c 'cd /home/edge_computing/
    Documents/THESIS_SMITA/thesis; chmod +x memory1.sh; ./memory1.sh
    $SHELL'"

#PSI metric as a control
some=`cat /proc/pressure/memory| cut -d' ' -f5| sed -n 1p | cut -d= -f2`
     #Memory stall in us (accumulated some)
full=`cat /proc/pressure/memory | cut -d' ' -f5| sed -n 2p | cut -d= -f2
    ` #Memory stall in us (accumulated full)
some1=$(echo $some)
full1=$(echo $full)

process_mem ()
{
     #Extract oom-score of all processes from /proc
    oom= eval 'for i in /proc/*/oom_score; do pid=$(echo "${i}" | cut -d/
         -f3); echo "$(cat "${i}"), PID=${pid}, exe=$(readlink -e /proc/$
        {pid}/exe)"; done 2> /dev/null | sort -rn -t, -k 1.11 | head |
        sed -n 1p'  #scans entire proc directory and list process with
        highest oom_score and name

    #Determine PID of highest oom-scored process
     PID= eval 'for i in /proc/*/oom_score; do pid=$(echo "${i}" | cut -d
         / -f3); echo "$(cat "${i}"),${pid}, exe=$(readlink -e /proc/${
         pid}/exe)"; done 2> /dev/null | sort -rn -t, -k 1.11 | head |
         sed -n 1p | cut -d "," -f 2'  #extracts PID of highest
         oom_scored process

     some=`cat /proc/pressure/memory| cut -d' ' -f5| sed -n 1p | cut -d=
         -f2` #Memory stall in us (accumulated some)
    full=`cat /proc/pressure/memory | cut -d' ' -f5| sed -n 2p | cut -d=
        -f2` #Memory stall in us (accumulated full)
    #echo "${some1} ${some}"

    #If change in some PSI change sysctl parameters
    if [ "${some1}" -lt "${some}"]; then
         bash -c 'sudo sysctl -w vm.dirty_background_ratio=2'
        bash -c 'sudo sysctl -w vm.dirty_ratio=100'
    fi

    #If change in full PSI stop the process
    if [ "${full1}" -lt "${full}" -a  ! "$(ps -o state= -p $PID)" = T  -a
         "$PID" -ne "$PID1"]; then
    #If pressure changes, and its not already stopped (T=Stopped process)
        ; avoids recursive stop and deadlock
        kill -SIGSTOP $PID
    fi

    #Restart the process if full PSI is stable
    if [ "${full1}" -eq "${full}" -a  "$(ps -o state= -p $PID)" = T ];
        then    #If pressure becomes constant restart the stopped process
        , work continued
        kill -SIGCONT $PID
    fi

    #Restore the default sysctl settings if some PSI is stable
    if [ "${some1}" -lt "${some}"]; then
         bash -c 'sudo sysctl -w vm.dirty_background_ratio=10'
        bash -c 'sudo sysctl -w vm.dirty_ratio=20'
    fi

    some1=$(echo $some)
    full1=$(echo $full)
    TotalSwap=`free -k | tr -s ' ' | cut -d' ' -f2 | sed -n 3p`          #
        Total Swap
    UsedSwap=`free -k | tr -s ' ' | cut -d' ' -f3 | sed -n 3p`
         #Used Swap
    Usedswapp=$( printf '%.02f' $(echo "$UsedSwap / $TotalSwap * 100" |
        bc -l) )    #Used Swap
    result1=${Usedswapp/.*}

    if [ "$result1" -gt 99.9 ]; then
        kill $PID
    fi
}

while :
do
    process_mem
    sleep 0.1
done
```

APPENDIX B: Individual processes memory consumption

```
#!/bin/bash
PID="$(pidof python3 | cut -d' ' -f1)"
echo "--------" | tee -a /home/edge_computing/Documents/THESIS_SMITA/
    thesis/Results/memory.txt
echo Individual Memory Statistics | tee -a /home/edge_computing/
    Documents/THESIS_SMITA/thesis/Results/memory.txt
echo "--------" | tee -a /home/edge_computing/Documents/THESIS_SMITA/
    thesis/Results/memory.txt
x=0
process_mem ()
{
#we need to check if 2 files exist
if [[ -f /proc/$PID/status ]];
then
    if [[ -f /proc/$PID/smaps ]];
    then
        #count memory usage, Pss, Private and Shared = (Pss-Private)
        time1=`date +"%T.%6N"` #Time in hrs, min, sec and milliseconds
        #PSS = Private(Clean+Dirty) + Shared(Clean+Dirty)/Number of
            Processes
        Pss=`cat /proc/$PID/smaps | grep -e "^Pss:" | awk '{print $2}'|
            paste -sd+ | bc `
        #RSS = Private(Clean+Dirty) + Shared(Clean+Dirty)
        Rss=`cat /proc/$PID/smaps | grep -e "^Rss:" | awk '{print $2}'|
            paste -sd+ | bc `
        Private=`cat /proc/$PID/smaps | grep -e "^Private" | awk '{print
            $2}'| paste -sd+ | bc ` #Private(Clean+Dirty)
        Shared1=`cat /proc/$PID/smaps | grep -e "^Shared" | awk '{print $2
            }'| paste -sd+ | bc ` #Shared(Clean+Dirty)
        Swap=`cat /proc/$PID/smaps | grep -e "^Swap" | awk '{print $2}'|
            paste -sd+ | bc `          #SwapUsed
        SwapPss=`cat /proc/$PID/smaps | grep -e "^SwapPss" | awk '{print
            $2}'| paste -sd+ | bc ` #SwapUsed-PSS

        if [ x"$Rss" != "x" -o x"$Private" != "x" ];
        then
                let Shared=${Pss}-${Private}            #Shared = PSS-Private
                Name=`cat /proc/$PID/status | grep -e "^Name:" |cut -d':' -
                    f2`
                let Sum=${Shared}+${Private}            #Sum = PSS
                x=$(( $x + 1 ))
                echo "${x},${time1},${Name},${Pss}, ${Sum}, ${Rss},${Private
                    },${Shared1},${Shared},${Swap},${SwapPss}"
        fi
    fi
fi
}

while :
do
    if [[ ! -d /proc/$PID ]];
    then
        break               #If process terminated
    fi
    process_mem
    sleep 0.1
done | tee -a /home/edge_computing/Documents/THESIS_SMITA/thesis/Results
    /memory.txt
```

APPENDIX C: Anonymous page memory consumer workload

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#define SIZE 104857600 //1 MB=1024*1024 bytes//

int main(void)
{
        int count = 0;
   sleep(20);
        while (1)
        {
                char* p = malloc(SIZE);
                if (p==NULL)
      {
         printf("Allocation done to all %d MB\n", count);
         break;
      }
      for(int i=0; i<=SIZE; i++)
      p[i] = 1;
                printf("Allocating %d MB\n", ++count);

        }

}
```

APPENDIX D: Memory Consumption Statistics of the System

```
#!/bin/bash
# Memory Statistics of the system when process running
echo "------" | tee -a /home/edge\_computing/Documents/THESIS\_SMITA/
    thesis/Results/memstat.txt
echo Memory Statistics | tee -a /home/edge\_computing/Documents/THESIS\
    _SMITA/thesis/Results/memstat.txt
echo "------" | tee -a /home/edge\_computing/Documents/THESIS\_SMITA/
    thesis/Results/memstat.txt
x=0
process\_mem ()
{
    time1=`date +"%T.%6N"`                              #Time in hours,
        minutes, seconds and milliseconds
    TotalRAM=`free -k | tr -s ' ' | cut -d' ' -f2 | sed -n 2p`          #
        Total RAM
    FreeRAM=`free -k | tr -s ' ' | cut -d' ' -f4 | sed -n 2p`          #
        Free RAM
    Freep=$( printf '%.02f' $(echo "$FreeRAM / $TotalRAM * 100" | bc -l)
        )        #Free RAM percentage
    UsedRAM=`free -k | tr -s ' ' | cut -d' ' -f3 | sed -n 2p`          #
        Used RAM
    Usedp=$( printf '%.02f' $(echo "$UsedRAM / $TotalRAM * 100" | bc -l)
        )        #Used RAM percentage
      AvailableRAM=`free -k | tr -s ' ' | cut -d' ' -f7 | sed -n 2p`
                  #Available RAM
    Availablep=$( printf '%.02f' $(echo "$AvailableRAM / $TotalRAM * 100"
         | bc -l) ) #Available RAM percentage
      SharedRAM=`free -k | tr -s ' ' | cut -d' ' -f5 | sed -n 2p`
                  #Shared RAM
    Sharedp=$( printf '%.02f' $(echo "$SharedRAM / $TotalRAM * 100" | bc
        -l) )        #Shared RAM percentage
    BufferRAM=`vmstat -S K | tr -s ' ' | cut -d' ' -f6 | sed -n 3p`
                         #Buffer RAM
    Bufferp=$( printf '%.02f' $(echo "$BufferRAM / $TotalRAM * 100" | bc
        -l) )  #Buffer RAM percentage
    CacheRAM=`vmstat -S K | tr -s ' ' | cut -d' ' -f7 | sed -n 3p`
         #Cache RAM
    Cachep=$( printf '%.02f' $(echo "$CacheRAM / $TotalRAM * 100" | bc -l
        ) )          #Cache RAM percentage
    TotalSwap=`free -k | tr -s ' ' | cut -d' ' -f2 | sed -n 3p`          #
        Total Swap
    UsedSwap=`free -k | tr -s ' ' | cut -d' ' -f3 | sed -n 3p`
         #Used Swap
    AvailableSwap=`free -k | tr -s ' ' | cut -d' ' -f4 | sed -n 3p`
             #Availabale Swap
    Usedswapp=$( printf '%.02f' $(echo "$UsedSwap / $TotalSwap * 100" |
        bc -l) )    #Used Swap
    x=$(( $x + 1 ))
    echo "${x},${time1},${TotalRAM},${FreeRAM},$Freep,${UsedRAM},$Usedp,$
        {AvailableRAM},$Availablep,${SharedRAM},$Sharedp,${BufferRAM},
        $Bufferp,${CacheRAM},$Cachep,${TotalSwap},${UsedSwap},${
        AvailableSwap},$Usedswapp"

}
while :
do
    process\_mem
    sleep 0.1
done | tee -a /home/edge\_computing/Documents/THESIS\_SMITA/thesis/
    Results/memstat.txt
```

APPENDIX E: PSI Statistics

```
#!/bin/bash
echo "-----" | tee -a /home/edge_computing/Documents/THESIS_SMITA/thesis
    /Results/psi.txt
echo PSI Statistics | tee -a /home/edge_computing/Documents/THESIS_SMITA
    /thesis/Results/psi.txt
echo "----" | tee -a /home/edge_computing/Documents/THESIS_SMITA/thesis/
    Results/psi.txt
x=0
process_mem ()
{
   time1=`date +"%T.%6N"`  #Time in hr, min, sec, millisec
   output1=`cat /proc/pressure/memory | cut -d' ' -f2| sed -n 1p | cut -
      d= -f2` #Memory(some) 10s
   output2=`cat /proc/pressure/memory | cut -d' ' -f2| sed -n 2p | cut -
      d= -f2` #Memory(full) 10s
   output3=`cat /proc/pressure/memory | cut -d' ' -f3| sed -n 1p | cut -
      d= -f2` #Memory(some) 1m
   output4=`cat /proc/pressure/memory | cut -d' ' -f3| sed -n 2p | cut -
      d= -f2` #Memory(full) 1m
   output5=`cat /proc/pressure/memory | cut -d' ' -f4| sed -n 1p | cut -
      d= -f2` #Memory(some) 5m
   output6=`cat /proc/pressure/memory | cut -d' ' -f4| sed -n 2p | cut -
      d= -f2` #Memory(full) 5m
   output7=`cat /proc/pressure/memory | cut -d' ' -f5| sed -n 1p | cut -
      d= -f2` #Memory stall in us (accumulated some)
   output8=`cat /proc/pressure/memory | cut -d' ' -f5| sed -n 2p | cut -
      d= -f2` #Memory stall in us (accumulated full)
   output9=`cat /proc/pressure/cpu | cut -d' ' -f2| sed -n 1p | cut -d=
      -f2`      #CPU(some)10s
   output10=`cat /proc/pressure/cpu | cut -d' ' -f3| sed -n 1p | cut -d=
       -f2`     #CPU(some)1m
    output11=`cat /proc/pressure/cpu | cut -d' ' -f4| sed -n 1p | cut -d=
        -f2`    #CPU(some)5m
    output12=`cat /proc/pressure/cpu | cut -d' ' -f5| sed -n 1p | cut -d=
        -f2`    #CPU stall in us (accumulated some)
    output13=`cat /proc/pressure/io | cut -d' ' -f2| sed -n 1p | cut -d=
       -f2`    #I/0(some) 10s
    output14=`cat /proc/pressure/io | cut -d' ' -f2| sed -n 2p | cut -d=
       -f2`    #I/O(full) 10s
    output15=`cat /proc/pressure/io | cut -d' ' -f3| sed -n 1p | cut -d=
       -f2`    #I/O(some) 1m
    output16=`cat /proc/pressure/io | cut -d' ' -f3| sed -n 2p | cut -d=
       -f2`    #I/O(full) 1m
    output17=`cat /proc/pressure/io| cut -d' ' -f4| sed -n 1p | cut -d= -
       f2`     #I/O(some) 5m
    output18=`cat /proc/pressure/io | cut -d' ' -f4| sed -n 2p | cut -d=
       -f2`    #I/O(full) 5m
    output19=`cat /proc/pressure/io | cut -d' ' -f5| sed -n 1p | cut -d=
       -f2`    #I/O stall in us (accumulated some)
    output20=`cat /proc/pressure/io | cut -d' ' -f5| sed -n 2p | cut -d=
       -f2`    #I/O stall in us (accumulated full)
    x=$(( $x + 1 ))
    echo "${x},${time1},${output1},${output2},${output3},${output4},${
       output5},${output6},${output7},${output8},${output9},${output10},
       ${output11},${output12},${output13},${output14},${output15},${
       output16},${output17},${output18},${output19},${output20}"
}
while :
do
    process_mem
    sleep 0.1
done | tee -a /home/edge_computing/Documents/THESIS_SMITA/thesis/Results
    /psi.txt
```

APPENDIX F: Determination of Total Anonymous and File Backed Pages in a Process

```
#!/bin/bash

# Page type of the process when process running
echo "----" | tee -a /home/edge_computing/Documents/THESIS_SMITA/thesis/
    Results/pagetype.txt
echo Page Statistics | tee -a /home/edge_computing/Documents/
    THESIS_SMITA/thesis/Results/pagetype.txt
echo "----" | tee -a /home/edge_computing/Documents/THESIS_SMITA/thesis/
    Results/pagetype.txt

x=0
PID="$(pidof python3)"

process_mem ()
{
   time1=`date +"%T.%6N"`  #Time in hr, min, sec and milliseconds

   pmap -x PID >> file1.txt   # read mappings of entire process

   tail -n +3 file1.txt >> file2.txt #discard first 2 rows

   awk '$6~/[ anon ]/&&$3>0{print > "file3";next}{print > "tmp"}' file2.
       txt && mv tmp file2.txt
   tr -s ' ' < file3.txt | cut -d' ' -f3 >> file4.txt #extract RSS

   while read -r num; do ((sum += num)); done < file4.txt;  #gives total
        anonymous pages at that instant of a process

    tail -n1 file1.txt >> total.txt
    totalRSS=`tr -s ' ' < total.txt | cut -d' ' -f4` #total pages of a
        process
    filebacked=$(echo "$totalRSS-$sum")  #remaining file backed

     x=$(( $x + 1 ))

    echo "${x},${time1},$sum,${totalRSS},${filebacked}"  #updates 10
        times a second to determine which page dominant is the process
}

while :
do

    process_mem
    sleep 0.1

done | tee -a /home/edge_computing/Documents/THESIS_SMITA/thesis/Results
    /pagetype.txt
```

APPENDIX G: File backed page memory consumer workload

G.1 Accessed once

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>

int main(int argc, const char *argv[])
{
    const char *filepath = "file.txt";

    int fd = open(filepath, O_RDONLY, (mode_t)0600);

    if (fd == -1)
    {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }

    struct stat fileInfo = {0};

    if (fstat(fd, &fileInfo) == -1)
    {
        perror("Error getting the file size");
        exit(EXIT_FAILURE);
    }

    if (fileInfo.st_size == 0)
    {
        fprintf(stderr, "Error: File is empty\n");
        exit(EXIT_FAILURE);
    }

    printf("Size is %i\n", (intmax_t)fileInfo.st_size);

    char *map = mmap(0, fileInfo.st_size, PROT_READ, MAP_PRIVATE, fd, 0)
        ;
    if (map == MAP_FAILED)
    {
        close(fd);
        perror("Error mmapping the file");
        exit(EXIT_FAILURE);
    }

    for (off_t i = 0; i < fileInfo.st_size; i++)
    {
        printf("Found character %c at %ji\n", map[i], (intmax_t)i);
    }
    close(fd);
    return 0;
}
```

G.2 Accessed continuously

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
```

```
#include <string.h>
#define SIZE 104857600 //1 MB=1024*1024 bytes//
int main(int argc, const char *argv[])
{
  char fName[16];
  char* text = malloc(SIZE);
  for (int z=0; z<2; z++)
 {
    sprintf(fName,"%d.txt",z);
    int count = 0;
    if (text==NULL)
    {
    printf("Allocation done to all %d MB\n", count);
    break;
    }
    for(int i=0; i<1024*1024*100; i++)
    text[i] = 1;
    printf("Allocating %d MB\n", ++count);
   int fd = open(fName, O_RDWR | O_CREAT | O_TRUNC, (mode_t)0600);

    if (fd == -1)
    {
        perror("Error opening file for writing");
        exit(EXIT_FAILURE);
    }
    size_t textsize = strlen(text) + 1;

    if (lseek(fd, textsize-1, SEEK_SET) == -1)
    {
        close(fd);
        exit(EXIT_FAILURE);
    }
    if (write(fd, "", 1) == -1)
    {
        close(fd);
        exit(EXIT_FAILURE);
    }

    char *map = mmap(0, textsize, PROT_READ | PROT_WRITE, MAP_PRIVATE,
        fd, 0);
    if (map == MAP_FAILED)
    {
        close(fd);
        perror("Error mmapping the file");
        exit(EXIT_FAILURE);
    }

    for (size_t i = 0; i < textsize; i++)
    {
        printf("Writing character %c at %zu\n", text[i], i);
        map[i] = text[i];
    }

    close(fd);
   }
    return 0;
}
```

APPENDIX H: Processes Management through cgroups

```
# Create cgroups
sudo mkdir /sys/fs/cgroup/memory/yolov3
sudo mkdir /sys/fs/cgroup/memory/mmap

#Specify limits and configurations
echo 2621440 | sudo tee /sys/fs/cgroup/memory/mmap/memory.limit_in_bytes
    #allocation happens in pages; 10GB assigned
echo 90 | sudo tee /sys/fs/cgroup/memory/yolov3/memory.swappiness

#Assign processes to cgroups
echo $(pidof yolov3)$ > /sys/fs/cgroup/memory/yolov3/cgroup.procs
echo $(pidof mmap)$ > /sys/fs/cgroup/memory/mmap/cgroup.procs

#Verify whether process is assigned correctly
ps -o cgroup $(pidof process)$

#Delete cgroups once work is accomplished
sudo cgdelete memory:yolov3
sudo cgdelete memory:mmap
```

APPENDIX I: Plots

I.1 Swappiness Variation

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
from cycler import cycler
import matplotlib
matplotlib.rcParams['pdf.fonttype'] = 42
matplotlib.rcParams['ps.fonttype'] = 42

# Read latencies
data1 = np.loadtxt('pytorchlatency20.txt')
x1 = np.sort(data1)
y1 = np.arange(1, len(x1)+1)/float(len(x1))
f1 = interp1d(x1, y1)
print("95 percentile of 20 swap: %f" % np.percentile(x1,95))
print("98 percentile of 20 swap: %f" % np.percentile(x1,98))
print("median of 20 swap: %f" % np.percentile(x1,50))

data2 = np.loadtxt('pytorchlatency40.txt')
x2 = np.sort(data2)
y2 = np.arange(1, len(x2)+1)/float(len(x2))
f2 = interp1d(x2, y2)
print("95 percentile of 40 swap: %f" % np.percentile(x2,95))
print("98 percentile of 40 swap: %f" % np.percentile(x2,98))
print("median of 40 swap: %f" % np.percentile(x2,50))

data3 = np.loadtxt('pytorchlatency60.txt')
x3 = np.sort(data3)
y3 = np.arange(1, len(x3)+1)/float(len(x3))
f3 = interp1d(x3, y3)
```

```
print("95 percentile of 60 swap: %f" % np.percentile(x3,95))
print("98 percentile of 60 swap: %f" % np.percentile(x3,98))
print("median of 40 swap: %f" % np.percentile(x3,50))

data4 = np.loadtxt('pytorchlatency80.txt')
x4 = np.sort(data4)
y4 = np.arange(1, len(x4)+1)/float(len(x4))
f4 = interp1d(x4, y4)
print("95 percentile of 80 swap: %f" % np.percentile(x4,95))
print("98 percentile of 80 swap: %f" % np.percentile(x4,98))
print("median of 80 swap: %f" % np.percentile(x4,50))

data5 = np.loadtxt('pytorchlatency90.txt')
x5 = np.sort(data5)
y5 = np.arange(1, len(x5)+1)/float(len(x5))
f5 = interp1d(x5, y5)
print("95 percentile of 90 swap: %f" % np.percentile(x5,95))
print("98 percentile of 90 swap: %f" % np.percentile(x5,98))
print("median of 90 swap: %f" % np.percentile(x5,50))

data6 = np.loadtxt('pytorchlatency100.txt')
x6 = np.sort(data6)
y6 = np.arange(1, len(x6)+1)/float(len(x6))
f6 = interp1d(x6, y6)
print("95 percentile of 100 swap: %f" % np.percentile(x6,95))
print("98 percentile of 100 swap: %f" % np.percentile(x6,98))
print("median of 100 swap: %f" % np.percentile(x6,50))


f, ax = plt.subplots(2, 3)
ax[0,0].plot(x1,f1(x1),marker='.', linestyle='-')
ax[0,0].set_title('Latency CDF-20% swap')
ax[0,0].set_xlabel('Latency(s)')
```

```
ax[0,0].set_ylabel('CDF')

ax[0,1].plot(x2,f2(x2),marker='.', linestyle='-')
ax[0,1].set_title('Latency CDF-40% swap')
ax[0,1].set_xlabel('Latency(s)')
ax[0,1].set_ylabel('CDF')

ax[0,2].plot(x3,f3(x3),marker='.', linestyle='-')
ax[0,2].set_title('Latency CDF-60% swap')
ax[0,2].set_xlabel('Latency(s)')
ax[0,2].set_ylabel('CDF')

ax[1,0].plot(x4,f4(x4),marker='.', linestyle='-')
ax[1,0].set_title('Latency CDF-80% swap')
ax[1,0].set_xlabel('Latency(s)')
ax[1,0].set_ylabel('CDF')

ax[1,1].plot(x5,f5(x5),marker='.', linestyle='-')
ax[1,1].set_title('Latency CDF-90% swap')
ax[1,1].set_xlabel('Latency(s)')
ax[1,1].set_ylabel('CDF')

ax[1,2].plot(x6,f6(x6),marker='.', linestyle='-')
ax[1,2].set_title('Latency CDF-100% swap')
ax[1,2].set_xlabel('Latency(s)')
ax[1,2].set_ylabel('CDF')
plt.show()
```

I.2 CDF Plot

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
```

```
from scipy.interpolate import interp1d
from cycler import cycler



# Read latencies
data1 = np.loadtxt('dirty.txt')
x1 = np.sort(data1)
y1 = np.arange(1, len(x1)+1)/float(len(x1))
f1 = interp1d(x1, y1)
print("95 percentile: %f" % np.percentile(x1,95))
print("98 percentile: %f" % np.percentile(x1,98))
print("median: %f" % np.percentile(x1,50))




# Plot CDF of latencies
f, ax = plt.subplots()
cy = cycler('color', ['blue'])
ax.set\_prop\_cycle(cy)
ax.plot(x1,f1(x1), '--', marker= '.')
plt.xlabel('Latency (s)')
plt.ylabel('CDF')
plt.legend(['Latd'], loc='best')
plt.title('Latency CDF— Optimized with LATD')

plt.show()
```

I.3 Time Series Plots

```
import math
import datetime
import matplotlib
import matplotlib.pyplot as plt
import csv
with open('tlat.txt','r') as f\_input:
    csv\_input = csv.reader(f\_input, delimiter=',', skipinitialspace=
        True)
    x = []
    y = []
    for cols in csv\_input:
        x.append(matplotlib.dates.datestr2num(cols[0]))
        y.append(float(cols[1]))

# naming the x axis
plt.xlabel('Time(s)')
# naming the y axis
plt.ylabel('Latency(s)')
# giving a title to my graph
plt.title('Latency vs Time')
# plotting the points
plt.plot\_date(x, y, marker='.', linestyle='--')
# beautify the x-labels
#plt.gcf().autofmt\_xdate()
#plt.gca().axes.get\_xaxis().set\_visible(False)
#plt.xticks([])
ax = plt.axes()
ax.xaxis.set\_major\_formatter(plt.NullFormatter())
# function to show the plot
plt.show()
```