**The answers for the questions**

Things to be considered :

[1] The word applications and processes are used interchangeably

[2] **Virtual Memory or Virtual Addressing** is the combined sizes of the physical memory and the swap space where swap space is pre configured space in hard-disk. Seen through swapon -s. If the swap is created as partition then it is an independent section of the hard disk used solely for swapping, no other files can reside there.

**Performance improvement of Latency critical applications under Virtual addressing /Virtual Memory during memory pressure in Linux Kernel.**

**1)What is the problem?**

Improved research to run multiple applications in parallel using multicore and multiprocessor systems paved way to run many processes simultaneously with multiple CPUs thereby reducing the latency of an application to greater extent.

Due to increased number of applications running simultaneously there is also an increase in the memory consumption rate of the system as a whole. Especially in the shared memory systems even with huge RAM configuration there are chances of going low on RAM when the applications running are large and memory intensive, thus making other latency critical applications suffer under virtual addressing scheme as they start using the swap.

The different levels of caches and TLB play an important role in improving latency of fetching information from Main memory and there are lot of researches being done in terms of reducing the latency of an application under memory bottlenecks in compiler and hardware levels by extensively studying the memory controller and DRAM behavior and their access patterns.

But how well can the latency be improved in virtual addressing scheme in the system software or the kernel level under low memory situations is the question of interest.

This is the part of my research.

**2) Why is new research needed?**

If we study the behavior of the applications in system software level under low memory situations we must definitely understand what are the measures taken by the kernel under traditional approach to handle out of memory situations and free back the memory thereby not affecting all applications progress.

But this traditional approach of freeing back the memory by killing least important process might not be well suited for latency critical applications which are running simultaneously.

So what is the current scenario under low memory and how the applications latency affects at that time?

Considering shared memory systems the operating system starts allocating memory dynamically and over committing memory when the processes start running. Everything is fair and good and all applications are running smoothly until there are enough free pages in RAM to satisfy the requests.

What happens when you are out of RAM?

Applications start thrashing pages from hard-disk if the swap is configured in the system freeing back the pages from the RAM which are least recently used.

Do the applications behave the same as they were earlier when you are low on RAM and start using swap?

No, as most of the processes spend time thrashing pages from swap with less progress made thereby affecting latency.

Thus if our system has an application running whose latency is of utmost important to us then definitely it is a serious problem as we can’t wait for the OOM-killer to get invoked and kill the process with highest oom\_score.

This is not a good scenario too as you can’t just sit and delay the applications doing excessive swapping until oom killer is invoked. Additionally due to current improvement in technology there is always some clean and upto date cache to reclaim so OOM killer never kicks in even if tasks spend 90% of the time thrashing pages which seriously affects latency critical applications.

There are few researches done to terminate the application ahead of time before oom-killer gets invoked by sending SIGTERM signal and terminating the application gracefully thereby preventing system unresponsiveness. I have written an user-daemon too which will constantly monitor the free memory in the system, calculate the oom\_score dynamically and terminate the process with highest oom\_score under low memory just before oom\_killer invocation thus preventing system unresponsiveness.

But will that solve our problem of reducing latency of other applications? Answer is no, because of intensive thrashing. In that aspect there must be some measures taken ahead of time even before OOM-Killer is invoked and also before gracefully terminating the application through SIGTERM.

So my research dwelves on investigating more under this situation where I have taken measures to reduce latency to some extent under low memory virtual addressing scheme before SIG TERMINATION in user space done by current researches to overcome OOM Problem.

**3) How are you solving this problem?**

My research involves

[1] Studying the latency CDF graphs under various memory availability of the system. Different memory availabilities are simulated by the micro benchmark I have written.

[2]Usage of PSI patch upstreamed Linux Kernel 4.20 for monitoring memory pressure under low memory which effectively reports absolute stall time in microseconds with the some and full metric where some indicates the atleast one processes stall and full with no processes making progress which means that the processes spend time in thrashing pages.

[3] Fine tuning the vm\_parameters like dirty\_bytes, dirty\_background\_ratio, swappiness and dirty\_expire\_centisecs and observing the latency effects.

[4] Monitor memory availability dynamically, individual memory consumption of the processes over time to understand their memory consumption patterns, where I have written bash scripts to monitor all.

[5] Usage of bcc tools to understand more on the access behavior the latency critical application has on page cache, buffer and disk by monitoring the hits it makes.

I have simulated a low memory situation by running a microbenchmark which keeps allocating memory without freeing it and running an latency critical application alongside where I have used pytorch-YOLOv3 for object detection which normally takes around 300-400 microseconds for each object detection under normal situations. I have like 1000 image samples to be detected for the pytorch application.

During low memory where your system extensively is thrashing pages, the absolute stall time in memory of PSI keeps rising when there is no progress made by any application(full) and at least one(some).

I have effectively used this increased some metric from PSI patch under low memory where I have been dynamically tracking my system memory consumption rate, processes memory consumption rate individually and the processes with highest oom\_score alongside

I have temporarily stopped the process with highest oom\_score when this some pressure rises, then I have changed the sysctl vm parameters dynamically at this situation by extensively studying their behavior through simulating different memory availabilities and studying through biosnoop and biostat bcc tool for different applications and applying the best effects for this latency critical application. I have increased dirty\_ratio to reduce the processes doing more frequent block I/O, lessening the dirty\_background\_ratio by awakening kernel\_flusher\_threads frequently so that they do background block I/O rather than invoking latency critical application to do the I/O. Increase in swappiness as the latency critical application is using more disk pages rather than anonymous pages which I have tracked down. And then I have dropped\_caches dynamically to release memory and not affecting the latency critical application where I have tracked down the hits made through cachetop bcc tool. Finally cleared the swap space by flushing out the least recently used pages in swap. And then when the pressure is stable, with all these measures I have continued back the application with highest oom\_score.

Through this I could decrease the latency of the applications significantly in virtual addressing scheme under swap usage. These metrics I have embedded dynamically in a bash script and observed the latency decrease.

So my approach tries to overcome the problem of system unresponsiveness and latency improvement under low memory by stopping the process, adjusting the sysctl parameters dynamically and thus pushing user termination a little ahead rather than invoking oom\_killer or sigterminating the process right ahead without taking other precautionary measures unlike the current researches where the work done is lost.

After taking above measures, with pressure being stable, after termination of latency-critical application OR after significant rise in free memory the highest oom\_scored process is continued again. (Highest oom\_scored process is most of the time least important). Then I have used same measure to SIG terminate the process after running low on swap additional to RAM with highest oom\_score now if stopping taking some extra measures and then continuing didn’t solve the problem.

Through this latency is reduced significantly before sig- termination and extra time and a bit of more memory for memory hog process is provided too before sig- termination.

**4) What results do you plan to show to prove that you have solved the problem?**

[1] Latd daemon where I have written script for that with variations in sysctl parameters after increase in some metric, stopping the process then continuing as explained above.

[2] Significant latency decrease in terms of CDF after tuning sysctl parameters as a whole.

[3] Increase in free memory rate when measures are taken as mentioned above after rise in some psi, process stopped and compare with free memory after rise in some psi without the measures mentioned above.

[4] Cache hit rate graphs without and with clearing page cache monitored using cachetop through which I can say whether hits are decreasing with the measure I have taken.

[5] Graph of increased dirty page swap out with kernel flusher threads with decrease in dirty\_background\_ratio monitored through bcc tool.

[6] Table to indicate what the corresponding memory availability is when some psi starts rising.

[7] Graph of latency of pytorch application as to what happens when highest oom\_Scored process is allowed to run after some psi rise vs stopped oom\_scored process.

**without swap :** I will show it by making comparison of 3 different cases:

[1] run the memory hog process until it is **killed** even after psi some patch rise----plot latency of pytorch

[2] run the memory hog process and **stop** immediately after psi patch rise ---- effect of latency plot

[3] run the memory hog process and **terminate** it gracefully after psi patch rise -- effect of latency plot

Compare all 3 and compare latency and say which is best and conclude stop is best as you wont loose work done by other process and latency affected is also small.

**With swap :** I will show this by making comparison of 2 different cases -- I will be excluding the case unlike swapoff of killing (1st case) as oom\_killing will take hours here because of os trying to search at least one free page available until it invokes oom\_kill and termination ahead is far better than this.

[1] First running the hog after some psi without any change in vm parameters and plot latency

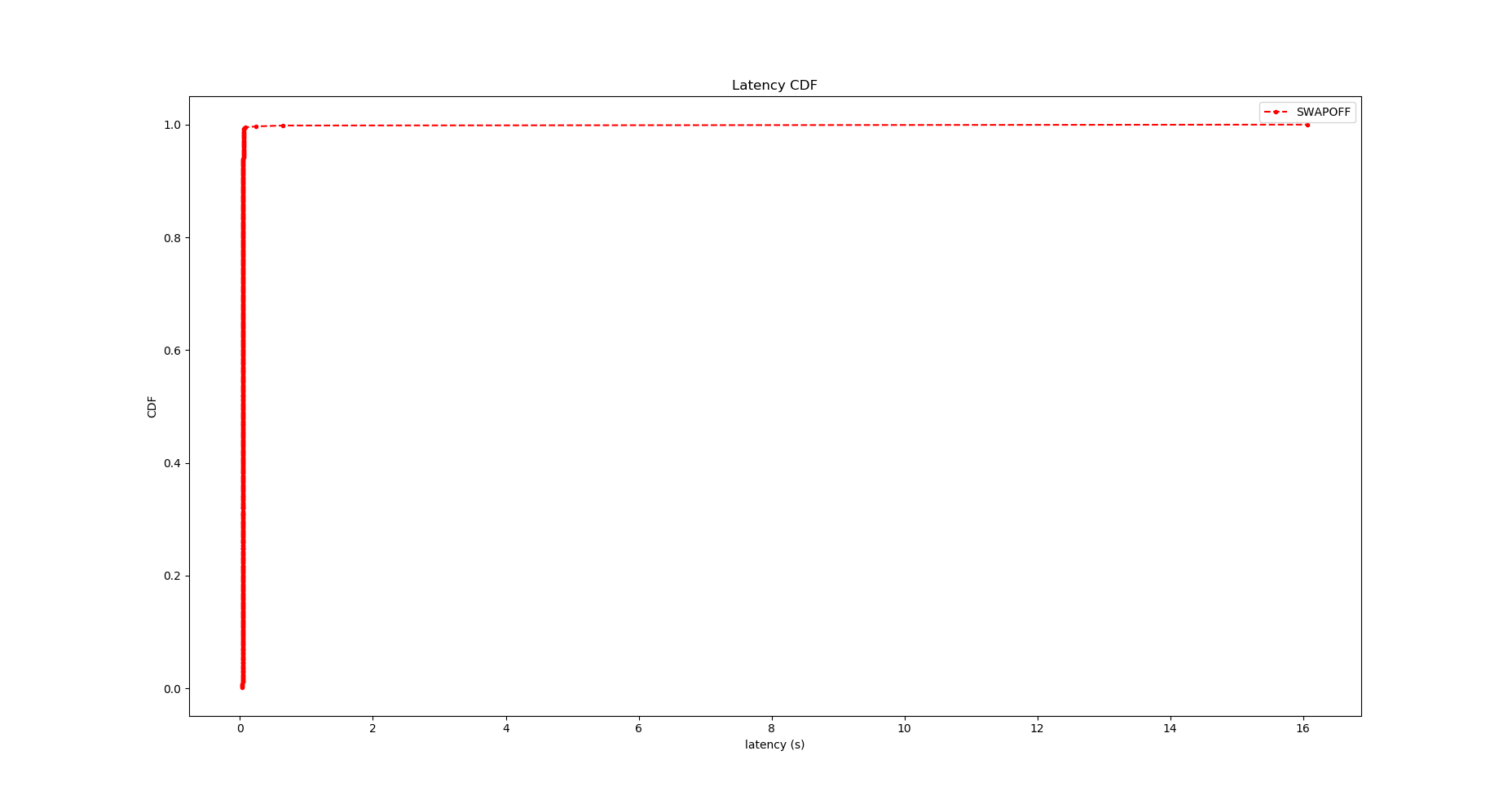
[2] Stopping hog after some psi change vm parameters, drop caches determined by hits mentioned above and then continue process if psi is stable --- plot latency here

Compare those two and show significant latency decrease unlike termination of highest oom\_Scored process right ahead without taking these measures.

So my research essentially employs to take these measures after some rise in psi pressure and try to bring latency down with variation in vm parameters making room to more free memory, stop process until pressure is stable and then continue. If things won’t work out this way too then terminate the application ahead of oom\_killer.

**The question asked with swapoff at what % of memory availability**

**Why PyTorch application latency is not getting affected under low memory with SWAPOFF. Why it is 300-400us only(means the 95th percentile or 98 percentile latency is 300-400us) and only goes as high as 16s when oom-killer is getting invoked (freeze time) and there is no latency affect till then?**



The reason is PyTorch application has good spatial locality. Hence it has its pages cached in page cache read recently for faster access in the near future. But memory hog application doesn’t have pages cached in page cache observed through hits and misses of cachetop bcc tool.

So under low memory situation just before oom-killer is getting invoked **AND** after rise in psi some pressure the processes that are getting stalled because of no memory are other processes not pytorch process as it is having its pages cached in page cache ahead of time and it will be making use of those pages only.

**But the question which arises is won’t the page cache be evicted too before oom-killer is invoked? So at that time there must be increase in latency of application!!!!**

But it is not happening in case of swapoff. **Your page, buffer cache and shared memory is completely dropped before oom-killer is getting invoked during SWAPON**

**But in case of SWAPOFF its not dropping page -cache pages before oom-killer invocation. Hence there is no latency variation .**

**Why it is happening so has no proper answer anywhere.**

**SUPPORTING VIEWS**

[**https://serverfault.com/questions/898417/oom-killer-kills-process-though-page-cache-is-never-evicted**](https://serverfault.com/questions/898417/oom-killer-kills-process-though-page-cache-is-never-evicted)

**I’m looking over c code in kernel again to find answer as to what happens to page-cache pages during swapoff before oom-killer is getting invoked.**

**Additional Information :**

**OOM**

Out of Memory killing is a process which is employed by the linux kernel when its typically low on memory and processes request for more memory than is physically available.

**How does it work?**

Linux kernel reviews all running processes, assigns them a badness score based on some criteria 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.

**What happens when oom killer is invoked? Disadvantages.**

* The process with highest oom score-badness score is killed which might be the most important process.
* Work done by the process is lost.
* Increased overhead as the system becomes unresponsive for a specified time period.

**How badness is calculated?**

* Recovers a large amount of memory
* Kills the minimum number of processes forked. The process which share mm\_struct gets killed if the main process is killed.
* Loses minimum amount of work done.
* Not innocent process which eat tons of memory.
* User expected process ----- which is being improved meticulously to select the process to kill of least surprisable
* Root, kernel and important system processes are given much lower scores.

**The only solution to avoiding oom killer currently it is to either reduce the memory requirements or increase the available memory.**

**GOAL OF THESIS:**

**OVERCOMING THE DISADVANTAGES MENTIONED ABOVE CAUSED BY OOM-KILLER.**

**Work Done as a part of thesis**

**[1] *Overcoming system unresponsiveness***

**Written an user-daemon** which kicks in before the oom-killer and terminates the process gracefully thereby overcoming the system freeze/system unresponsiveness.

**How this user-daemon works?**

It employs the same strategy which kernel oom killer employs in calculating oom-score but does it in user space by reading through proc filesystems, monitoring RAM and swap periodically, notifying user over periodic intervals about system memory and then terminating the process with highest oom-score gracefully rather than killing it.

Since this user-daemon gets invoked just prior to oom-killer does its action and employs SIGTERM signal for graceful process termination in user space without involving kernel interference(switching of stacks, saving processor context) thus preventing overhead of system unresponsiveness.

**Work in progress**

**Can we take more corrective action?**

That is neither terminate or kill the process rather stop the process with highest oom-score if it is important and wait for other processes in the system to complete their work and exit, thereby resuming this process and continuing it for completion.

**The question which arises:**

How long the process with highest oom-score needs to be stopped?

Can we predict how long the other processes needs to run to complete their work and exit?

**To determine:**

To determine how long the other processes might run to completion we need to determine how long the process has run till time, what kind of work its doing, what resources has it used, how much memory has it consumed and other system level data related to the process.

Through this can we estimate how much more memory it might consume for next 5-10 seconds or can it finish its work for next 5-10 seconds. In general can we predict the behaviour of the process for the next 5-10 seconds?

If its accomplished we can make more intelligent decisions in either killing/terminating/stopping the process with highest oom-score accordingly.

**To predict process behaviour**

1. Select benchmark
2. Determine feature vectors that data will be collected for
3. Collect data (values of features vectors, and memory consumed) at sampling interval
4. Figure out how to preprocess the data
5. Build a deep learning model
6. Evaluate

**2/18/19**

For Object Detection Application - YOLOv3.

Run the application at highest priority.

Interferer 1 - Memory hog

1. For different memory availability (controlled with mem hog with swap enabled), characterize the “some” psi. Relate that to inference latency. This shows how much memory pressure can the application sustain while providing acceptable performance.
   1. Run the hog first to consume certain % of memory
   2. Run the application
   3. Repeat for different memory % usage (50, 70, 80, 90, 95, 98)
2. Investigate the same with different sysctl VM tuning parameters - [https://www.kernel.org/doc/Documentation/sysctl/vm.tx](https://www.kernel.org/doc/Documentation/sysctl/vm.txt)t

swapiness: low makes kernel keep pages in memory

vfs\_cache pressure: high makes kernel free caches from RAM

dirty\_background\_ratio: keep more dirty pages in memory

dirty\_ratio: if dirty pages exceeds this process to blocks

Also: swap turned on (total = swap + mem\*overcommit\_ratio): default is 50;

Plots - latency CDF under different memory pressures, and different sysctl parameters

Memory stall CDF (not cumulative)

Adjust the sampling 100ms, 1s, 5s

Time series studies -

1. Let the application run for some time, and then turn the interferer on all the way till latency is high (say 98%), Set the sysctl parameters to be the best found above.
   1. Plot mem stall for each sampling interval vs. time (not cumulative).
   2. Plot inference latency vs. time
   3. I/O and CPU stall vs. time
   4. page cache hit/miss (cachestat) vs. time
   5. per-process page cache hit/miss (cachetop) vs. time
   6. block I/O (biosnoop) vs. time
   7. mem available vs. time

Try to explain how these plots are correlated.

Design a controller (call it latd) that monitors the above metrics, and stops (SIGSTOP) lower priority user applications if there is a possibility that the highest priority application will suffer from increased latency due to resource pressure.

Time series plots to show that the controller is working.

Interferer 2 - (Database behavior)

Mmap large files, access them once, and thus pollute the page cache. See link below for read and write patterns

<https://engineering.linkedin.com/performance/optimizing-linux-memory-management-low-latency-high-throughput-databases>

Repeat all of above.

Thesis organization

Chapter 1. Intro - Motivation, Brief Description of experiments and results, organization of thesis

Chapter 2. Background - PSI patch, BCC Tools, Other perf. measurement tools, workload (YoloV3), Linux page cache

Chapter 3. Interferer-1 experiments, explanation of what’s going on, design of controller, results for control

Chapter 4. Interfere-2, Same organization as Chapter 3

Chapter 5. Summary of work, possible future directions.