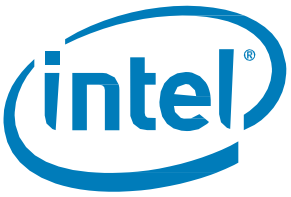


Utilizing Linux Swap with Intel® Optane™ DC SSDs as a Memory Overcommit Technique

Solutions Blueprint

June 2019
Version 1

Team Contacts:

Andrzej Jakowski andrzej.jakowski@intel.com Kernel Development

Tim C. Chen tim.c.chen@intel.com Kernel Development

Ying Huang ying.huang@intel.com Kernel Development

Frank Ober frank.ober@intel.com Testing and Outreach

David J. Leone david.j.leone@intel.com Testing and Outreach

Andrew Ruffin andrew.ruffin@intel.com Market Analysis and Outreach

Pragathi Narendra pragathi.narendra@intel.com Performance Test and Test Development

Mariusz Barczak mariusz.barczak@intel.com Kernel Development

Gert Pauwels gert.pauwels@intel.com Field Technical Support EMEA Region

Steven Briscoe steven.briscoe@intel.com Field Technical Support EMEA Region

Farib Khondoker farib.khondoker@intel.com Testing and Support

Document Number: 340608-001US

Revision History

Revision Number	Description	Revision Date
001	• Initial release.	June 2019

Contents

1 Introduction

This solutions blueprint explains how to use Intel® Optane™ DC SSDs in memory extension configurations, or as memory replacement. We'll describe recent performance improvements that were first introduced in version 4.11 and completed in version 4.14 of the Linux* kernel. For simplicity, we will refer to version 4.14 or newer, as the kernel version needed to evaluate high performance swap usage. Very high endurance and low latency devices like Intel® Optane™ DC SSDs can be efficiently used as swap devices, thereby enabling the system to exceed its minimum required system level performance in various memory overcommit use cases. Intel® Optane™ SSDs used as swap devices are expected to have a long life span of five or more years in this usage. For those who intend to immediately implement and test the use cases outlined in this document, please jump to the Appendix sections, and visit the following GitHub link for tools, instructions, and test code. http://github.com/fxober/LinuxSwap

1.1 Scope

We will focus on how the Linux operating system (OS) can utilize Intel® Optane™ DC SSDs as swap devices, thereby allowing storage device capacity to be used in conjunction with DRAM to store memory pages on both DRAM and non-volatile memory type media. The process of moving memory pages between the storage device and main memory is called *paging*. Paging allows system administrators to perform efficient management of system resources (memory, CPU, storage) at desired cost and service levels. With recent advancements in storage media and Linux kernel improvements, Intel® Optane™ DC SSDs provide a new opportunity to offset DRAM costs and allow for more flexible process memory oversubscription, at higher performance levels than before. This solutions blueprint will explore those usages.

1.2 Target Audience

Targeted for system administrators, system operators, DevOps teams, and application developers wanting to configure their underlying software and hardware resources to maximize system performance at a better cost. This document assumes familiarity with basic computer architecture terminology and techniques in OS usages to manage physical resources such as CPU, memory and storage. It also explains fundamental concepts of memory management techniques utilized in modern OSs, focusing on the Linux environment. The improved implementations of Linux Swap* and better higher endurance memory media, such as Intel Optane memory, is essentially what enables such a solution to be effective in a modern data center environment.

1.3 Document Organization

First, this document introduces use cases in which the Intel® Optane™ DC SSD is used as memory augmentation. Later, a server cost model is presented, which can be adopted or adjusted to calculate potential cost savings when leveraging an Intel® Optane™ DC SSD as DRAM replacement. Next, we describe the OS upgrades necessary to maximize system performance when using an Intel® Optane™ DC SSD as a swap device. Specifically we provide guidance on minimum required versions of common Linux distributions that utilize swap and memory management subsystem improvements, along with details on building the Linux kernel manually to maximize swap performance. The Additional Considerations for Software Configuration section explores system configuration details for maximizing swap performance. Then we compare the performance of the different swap devices. Finally in the Appendix sections the details of the memory management subsystem and details of Linux kernel innovations that improve swap performance are explained. Finally, a kernel patch list is provided for advanced users willing to backport the changes into their own kernel fork.

1.4 Glossary

Term	Definition
Physical memory	Fast memory, byte addressable (as opposed to disk storage which is sector or block addressable). This fast, dynamic system memory is typically provided by DRAM technology.
Swap device	Dedicated space on a storage device for storing memory pages of process data or process code. It can be whole block storage device or its partition or a file in filesystem (swap file).
Virtual memory	Memory management technique implemented in modern OSs. It provides an illusion to the running process that it operates on a contiguous block of memory, while in reality hardware and the OS manage translations between virtual addresses to physical addresses, and transfers of memory pages from storage device to physical memory. OS virtual memory hides those complexities from the application programmer.
Total Cost of Ownership (TCO)	A defined, but often not standardized approach to analyzing the financial impact of a purchase, and perhaps ongoing expenses of hardware and software infrastructure over its life cycle. TCO models typically includes various factors impacting cost, e.g. cost to purchase HW (capital spending), operational cost related to electricity used to power and cool a building, and Data Center equipment. This paper focuses on a simplified server cost model. You can consider it Bill of Material optimization, since the target is not full analysis of all server operation or acquisition costs.

§

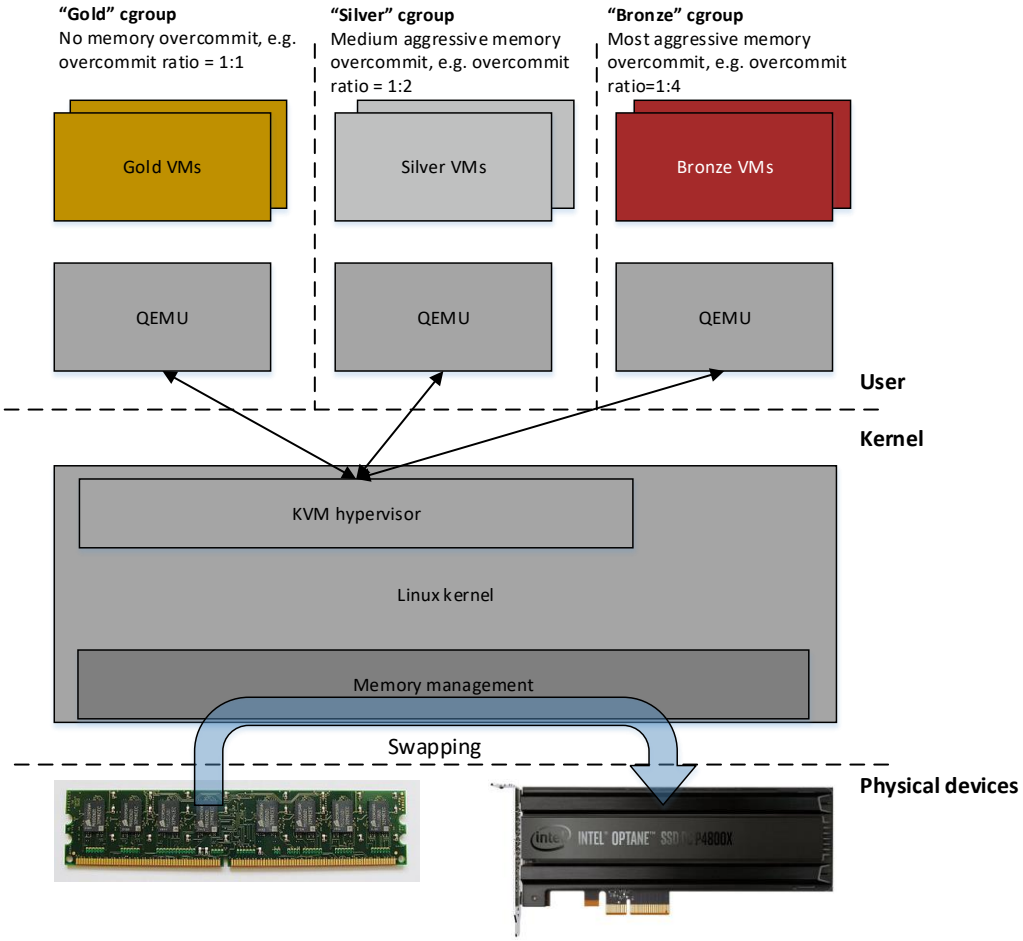
2 Memory Overcommit Use Cases

This chapter introduces example use cases in which an Intel® Optane™ DC SSD can be used as memory extension, or as memory replacement by using the Linux swap mechanism. This chapter also provides an example server cost model that has been developed to illustrate potential cost savings when considering the purchase of a new HW infrastructure. Use this server cost model as a framework to calculate potential cost savings at the server capital expenditure level.

2.1 Memory Overcommit for Virtualizes Environments

One common technique widely used among cloud service providers (CSPs) is to perform physical resources over-commitment including physical CPU, storage, and memory. The following figure illustrates virtual machine differentiation based on ratio, and how much of the guest physical memory is actually backed up by physical DRAM. For example "Gold" VMs' guest physical memory is fully backed up by DRAM, while for "Silver" VMs half of its guest physical memory is backed up by the swap device. Finally, for "Bronze" VMs, a quarter of the guest physical memory is backed up by DRAM, the remaining portion can be paged out to the swap device. With Linux based hypervisor (KVM) this type of differentiation can be achieved using the mechanism called control groups (cgroup) which controls resource usage (e.g. system memory) to a group of process – in this case a class of VMs.

Figure 1: Example of Virtual Machine Differentiation Based on Memory Overcommit Ratio



§

3 Example Server Cost Model

This chapter focuses on deriving an example server cost model, from a system memory hardware costs perspective, for two example configurations of servers: server "A" and server "B." The server cost model does not take into account the varied and unique operational expenses or other capital expenditures related to the larger scope of running a data center. For simplicity of our comparison, differences in space, power, operating costs, and other variable factors are ignored.

Server "A" and server "B" configurations are almost identical with regards to CPU, networking, and storage (both boot disks and data volumes). There are only 2 differences between them:

- Server "A" total physical DRAM is 384 GiB (24 x 16 GB RDIMMs), while server "B" is populated with only 192 GiB (12 x 16 GB RDIMMs) of physical DRAM
- Server "A" does not use Intel® Optane™ DC SSD as a swap device; instead server "B" uses Intel® Optane™ DC SSD (2 x 100 GiB devices) as swap devices

One of the data points most interesting to a system administrator is the relative cost of server "B" to server "A" which illustrates the potential hardware component cost savings on the purchase or lease of new servers for the data center. Additional server cost calculations focus on the relative costs of server "B" configuration compared to server "A". For simplicity, this costing model takes into account only the memory components (DRAM + Intel® Optane™ DC SSD capacities), because all other components of those server configurations are identical. Relative cost comparison of server "B" configuration to server "A" configuration can be defined as follows:

$$Relative_{cost} = \frac{server_B}{server_A} = \frac{cost_{DRAM} * capacity_{DRAM\_B} + cost_{Optane} * capacity_{Optane\_B}}{cost_{DRAM} * capacity_{DRAM\_A}}$$

Now simply dividing numerator and denominator of above equation by cost $_{Optane}$ leads to the following formula:

$$Relative_{cost} = \frac{\frac{cost_{DRAM}}{cost_{Optane}} * capacity_{DRAM\_B} + capacity_{Optane\_B}}{\frac{cost_{DRAM}}{cost_{Optane}} * capacity_{DRAM\_A}}$$

Substitution of $\frac{cost_{DRAM}}{cost_{Optane}}$ with normalized per GiB DRAM to Optane price ratio (DRAM_to_Optane) will lead to this final formula:

$$Relative_{cost} = \frac{DRAM\_to\_Optane * capacity_{DRAM\_B} + capacity_{Optane\_B}}{DRAM\_to\_Optane * capacity_{DRAM\_A}}$$

Note: Please do your own price calculations using the formula above to calculate your server cost savings.

§

4 The Kernel Build Process

4.1 Recommended Software Upgrades

In order to maximize Intel® Optane™ DC SSD performance in a memory extension configuration (as a swap device) Intel recommends upgrading your Linux distribution to a recent version containing the backported series of patches that were added to the upstream Linux kernel in versions 4.11 and later.

The following table contains information on the common Linux distribution versions that adopted performance improvements pertaining to swap performance.

Table 1: Linux Distribution Containing Swap Performance Improvements

Linux Distribution	OS Version
RHEL/CentOS	Starting version 7.5 and forward Starting version 8.0 and forward
Ubuntu	Starting version 18.10 and forward
SLES	Starting version SLES15, SLES12 SP4 and forward
Oracle* Linux	Starting version Oracle Linux 7.5 and later with UEK R5 and RHCK

4.2 How to Build your Kernel Based on Upstream Linux Kernel

This section provides instructions on building a Linux kernel image based on the upstream Linux kernel project. This may be especially useful for those interested in further exploration of Linux kernel improvements relating to swap device performance, and who are willing to upgrade their infrastructure's Linux kernel.

Please note that these instructions are based on Ubuntu* server 18.04.2 system build, the exact steps may differ between different Linux distributions, e.g. usage of distribution package manager.

Approximate time needed: 1 hour

4.2.1 Development Tools Required for menuconfig (Possible Pre-requisites)

In order to clone, compile, and build a new kernel/driver, the following packages must be installed.

You must be logged in as root to install these packages.

Dependencies needed to run kernel menuconfig

```
# apt-get install flex bison
# apt-get install libncurses5-dev libncursesw5-dev
```

Dependencies needed to perform kernel build

```
# apt-get install libssl-dev libelf-dev
# dpkg -i linux-*.deb
```

4.2.2 Build New Linux Kernel with RCU Setting for Swap

Download Linux kernel 4.14 or 5.x or newer from this repository: https://www.kernel.org/pub/linux/kernel/ into your Linux distribution. It is the best to choose the latest stable kernel.

From a working directory:

Use wget to download the kernel and unpack it (here the example is 4.18.20)

```
# wget https://mirrors.edge.kernel.org/pub/linux/kernel/v4.x/linux-4.18.20.tar.xz
# tar -xvf linux-4.18.20.tar.xz
```

Alternatively clone whole Linux kernel git repository and checkout specific branch

```
# git clone https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
# git checkout -b v4.18.20_local v4.18.20
```

Build and install

To create the kernel configuration file (.config) based on the running kernel, and use the default setting for all new options, run the following command:

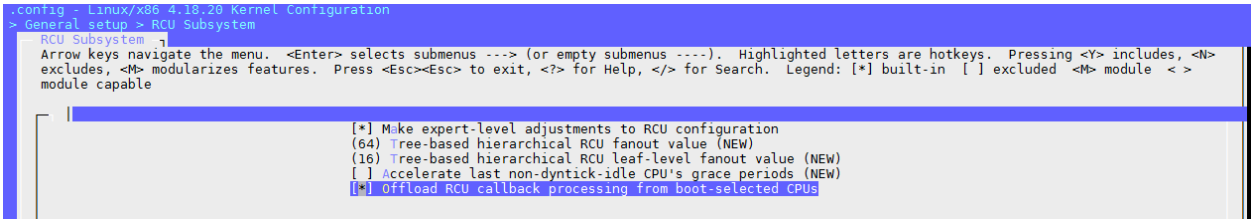
```
# yes "" | make oldconfig
```

To obtain maximum performance, avoid read-copy-update (RCU) callback processing as this may introduce delays. To avoid RCU, edit "CONFIG_RCU_NOCB_CPU=y" setting in your local kernel .config file. See Offloading RCU Processing to Dedicated Kernel Threads for details on editing RCU settings.

Alternatively, you can make changes by running menuconfig to select that option using the user interface as shown in the image below.

```
# make menuconfig
```

Under "General Setup and Features > RCU Subsystem" set the "Offload RCU callback …" flag as shown in the image below:



Save and Exit menuconfig. Build the kernel and kernel modules, and install the new kernel on the system.

To build kernel image and loadable kernel modules invoke

```
# make
# make modules_install
```

Install newly built kernel into operating system

```
# make install
```

After successful install, reboot the system to load the new kernel image and kernel modules. Usually the new kernel becomes the default boot selection. After booting the OS, use "uname -a" to verify that the running kernel version matches the newly installed kernel version.

If a different kernel version is loaded, you can modify this by reconfiguring the system loader, usually grub2. Refer to the system loader documentation for your specific distribution.

4.3 Additional Considerations for OS Configuration

This section explores OS configuration considerations for maximizing performance of the swap device(s).

4.3.1 Offloading RCU Processing to Dedicated Kernel Threads

To offload RCU processing to dedicated kernel threads, edit the kernel command line option in the system loader. When using Grub2 as system loader, navigate to /etc/default/grub file and add "rcu_nocb=<all_cpus>" to the GRUB_CMDLINE_LINUX_DEFAULT entry. See below /etc/default/grub file listing for example:

```
. . .

GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`

GRUB_CMDLINE_LINUX_DEFAULT="rcu_nocbs=0-n maybe-ubiquity"

GRUB_CMDLINE_LINUX=""

. . .
```

Note: n is the number of cpus (or hw threads) in your system

After saving edits, run either the "update-grub" or "grub2-mkconfig" command to update your grub2 settings in the boot partition.

Reboot the system and verify that the new settings have been applied to the kernel.

```
# dmesg | grep -i offload
[    0.000000]  Offload RCU callbacks from CPUs: 0-63.
```

The reason for this step is to avoid RCU processing in an IO completion path, as RCU processing will likely increase paging latency.

4.3.2 Turning Off Transparent hugepages

To minimize the overhead of coalescing memory pages into hugepages and later breaking them up on the swap device, perform the following commands:

```
# echo 'never' > /sys/kernel/mm/transparent_hugepage/enabled

# echo 'never' > /sys/kernel/mm/transparent_hugepage/defrag
```

4.3.3 Watermark Scale Factor

It is important to increase the watermark scale factor in /proc/sys/vm as this is the level where available memory is checked by kswapd. We recommend setting it to 400 or 4% of available memory, doing so will set kswapd to automatically kick off swapping at 4% of available system memory.

```
# echo '400' > /proc/sys/vm/watermark_scale_factor
```

4.3.4 NUMA Considerations

When dealing with multiple swap devices on a multi-socket system we recommend distributing swap devices evenly among different CPU sockets to avoid QPI/UPI transfers. Moreover to avoid software overhead we recommend creating many swap devices on a partitioned NVMe device. Each swap partition must have the same priority. In most cases there can be at least 28 partitions, depending on the kernel configuration. When setting up your system, we recommend adhering to the NUMA locality rules for maximum performance.

4.4 Performance Data of 4.18.20 Linux Swap

We used the pmbench utility to test the allocation and access of 4KiB memory pages on a Linux system. Our test system utilized an Ubuntu 18.4.2 distribution of Linux which we initially upgraded to the 4.18.20 version of the kernel, as the Ubuntu release comes with 4.15.x kernel version. We upgraded using the methods noted in Appendix A - Automation Scripts and How-to Guide. There should be no issue running kernel 4.14 or newer as the kernel patches to Linux swap are upstreamed (public on kernel.org) in 4.14. You cannot gain this level of performance on kernels prior to 4.14. We tested the in-box kernel of Ubuntu 18.04.2 (kernel 4.15.0-46-generic) and saw minimal difference (<5% variation) from the performance documented in this paper. Performance testing results presented in this paper were run on kernel version 4.18.20. All security patch mitigations were current at the time of this publication.

Table 2: Test Server System Configuration

Parameter	System Configuration May 4, 2019
System/Intel Board ID/ Maker	S2600WFT Intel
CPU	Intel® Xeon® Gold 6142 CPU @ 2.60GHz with 16 cores, 32 threads
Memory	192 GB DDR4 DIMM @ 2666 MHz, 12 x 16 GB
NVMe SSDs	(2) 375 GB NVMe PCIe Intel® Optane™ SSD DC P4800X, (2) 1 TB NVMe PCIe Intel SSD DC P4510
Linux Distribution Installation	Ubuntu* 18.04.2 LTS
Kernel Version	4.18.20
Intel® Optane™ SSD DC P4800X Firmware Version	E2010435
Intel® SSD DC P4510 (TLC NAND) Firmware Version	VDV10131
BIOS Version	Intel Version: SE5C620.86B.00.01.0016.020120190930

Performance is dictated by how much write pressure is exerted on the drive. The read ratio parameter “r” is used to drive most write changes to memory pages, thereby making this a high performance synthetic memory “walk” test of the hardware combination of DDR4 DRAM and P4800X (Optane memory media) versus the P4510 (NAND media) devices. In a future revision of this document we will test SATA NAND devices, but as these results indicate SATA NAND is not the best option for displacing memory. For our tests “-r 0” or “-r 50” parameters were passed to pmbench, “-r 0” is the more write intensive and should show lower scale numbers. Please see the whitepapers from Jisoo Yang about how pmbench is designed, you will find hyperlinks in the References section.

The following scaling tables, related graphs, and access latency histograms show the bi-modal latency distribution between pages accessed from DRAM versus swapped out pages.

Running a single thread on just DRAM without any swap yields over 4 million accessed pages per second, with swap you get over 100 thousand, while NAND provides just over 17 thousand. This is detailed below in Table 3. Bandwidth of DRAM is much higher than any SSD on the market, hence the DRAM baselines are very high. In a future revision of this document we will explore testing an application usage, see how an application using overcommitted memory performs, and how well swap can perform for your application usage.

The following table provides performance results of 4 KiB page access at specified levels of write only, or 50/50 read/write threads, doing the page accesses (multiply data point by 4,096 to get bandwidth).

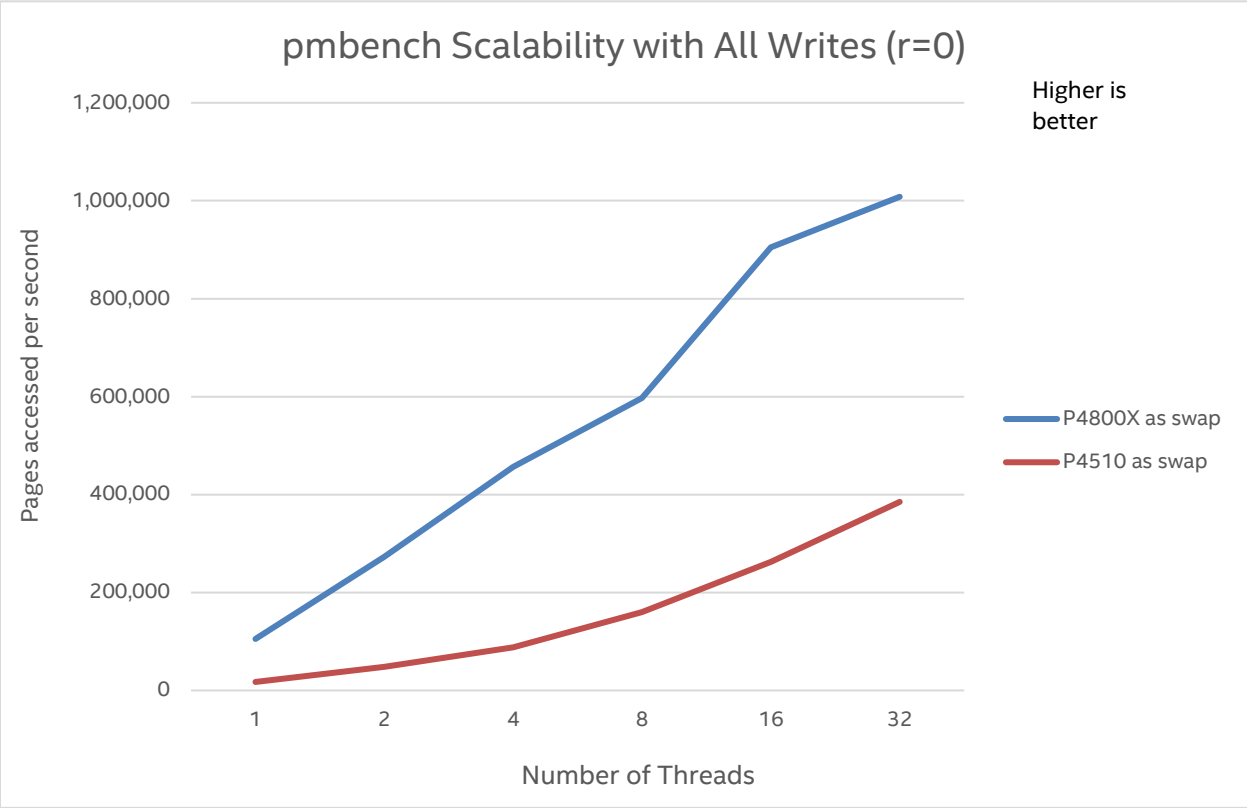
Table 3: Performance Results of 4 KiB 100% write and 50/50 Read Writer

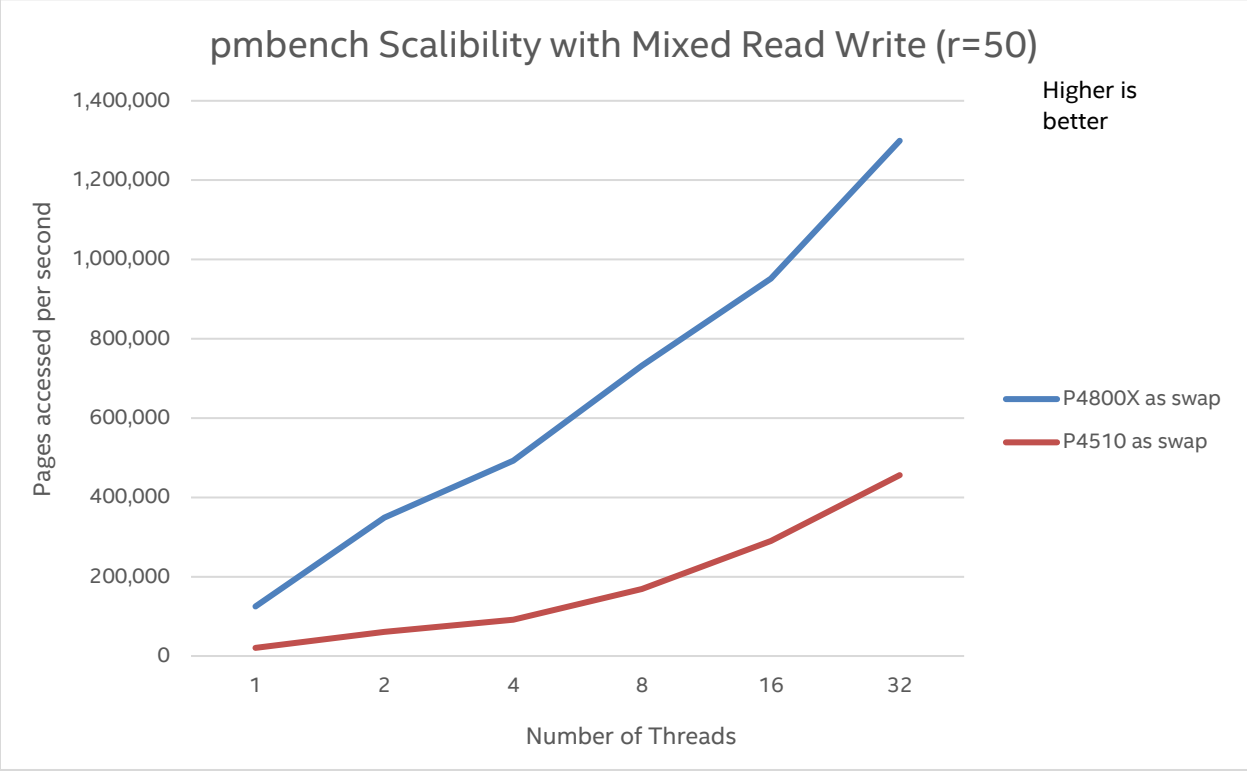
Results[1] May 3, 2019		Intel® Optane™ SSD DC P4800X			Intel SSD DC P4510 (TLC NAND Memory Media)			DRAM		
Read Ratio	Threads	Pages per second[2]			Pages per second[2]			Pages per second[2]		
		Numa0	Numa1	Total	Numa0	Numa1	Total	Numa0	Numa1	Total
0	1	105,166		105,166	17,333		17,333	3,496,333		3,496,333
0	2	136,250	136,416	272,666	24,416	23,833	48,249	4,494,833	4,506,000	9,000,833
0	4	229,916	226,833	456,749	45,666	42,333	87,999	8,909,083	9,017,166	17,926,249
0	8	300,666	296,333	596,999	84,500	75,333	159,833	17,514,416	17,638,000	35,152,416
0	16	534,416	370,666	905,082	144,666	117,750	262,416	34,856,916	35,036,583	69,893,499
0	32	623,166	384,916	1,008,082	205,666	179,416	385,082	68,430,750	68,706,666	137,137,416
50	1	124,916		124,916	20,500		20,500	3,048,833		3,048,833
50	2	173,166	175,416	348,582	30,666	30,166	60,832	4,222,750	4,245,083	8,467,833
50	4	248,166	244,500	492,666	46,916	44,333	91,249	8,721,500	8,881,500	17,603,000
50	8	366,555	365,388	731,943	88,000	81,166	169,166	17,270,000	17,475,000	34,745,000
50	16	484,277	467,166	951,443	158,611	131,222	289,833	34,353,000	34,276,583	68,629,583
50	32	704,388	594,888	1,299,276	265,666	190,444	456,110	67,860,666	67,949,083	135,809,749

NOTES:

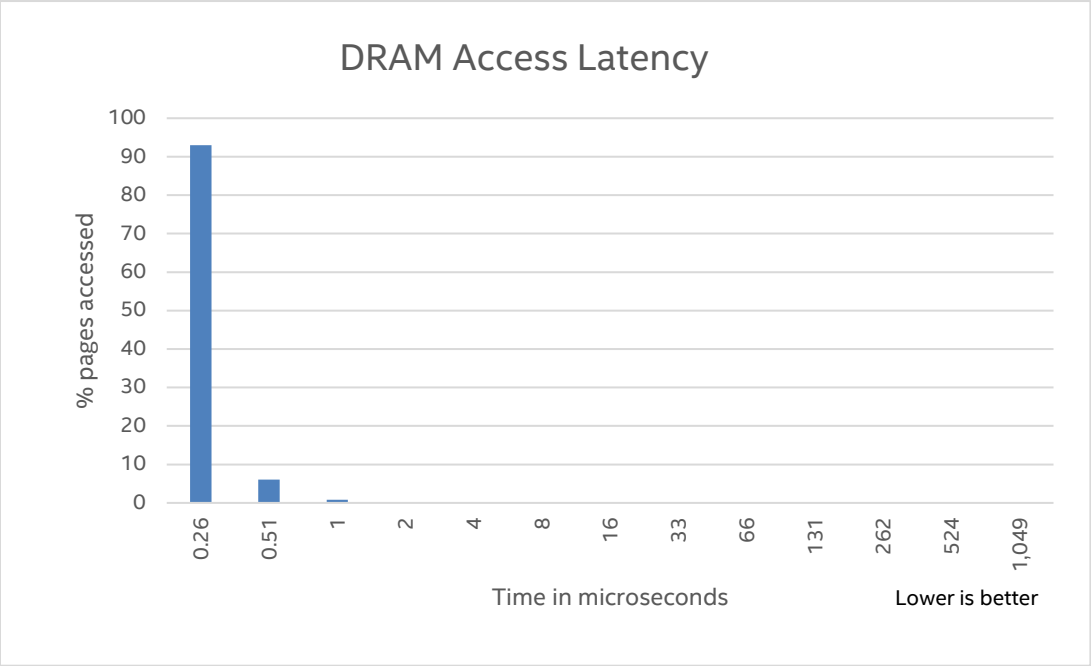
1. Performance result values are a combination of direct DRAM and swapped pages, plus software overhead; therefore direct comparison with Product Specification values are not valid.
2. All memory accesses are in 4,096 Byte page sizes.

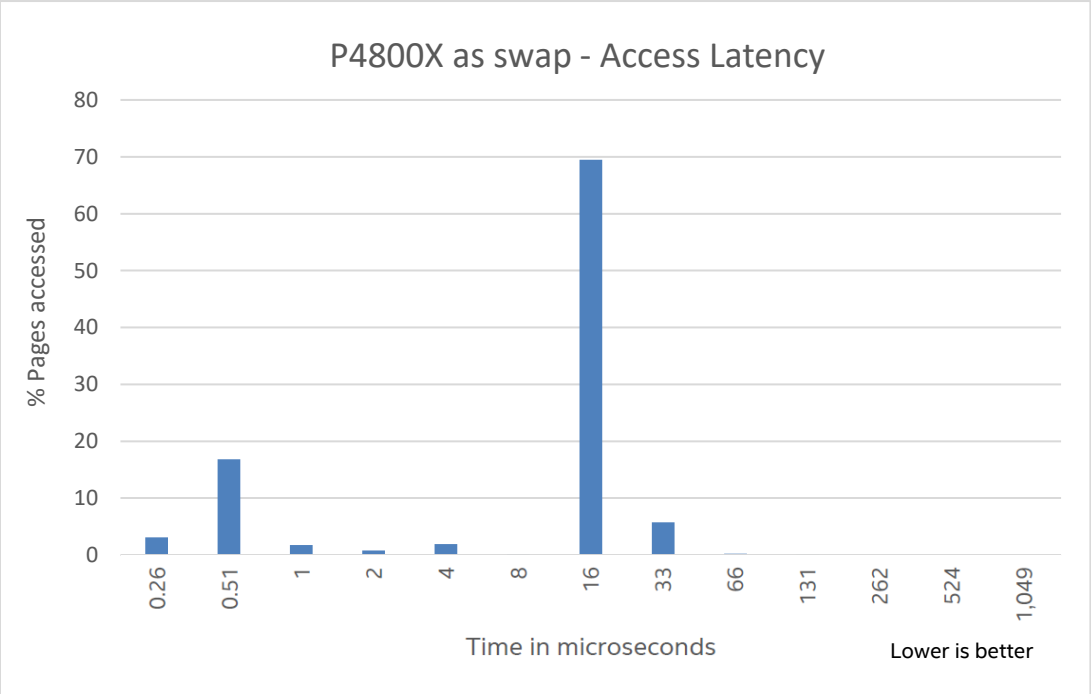
Next we'll explore how swap can scale across in the thread count. The following graphs provide visual representations of the data in Table 3, above.

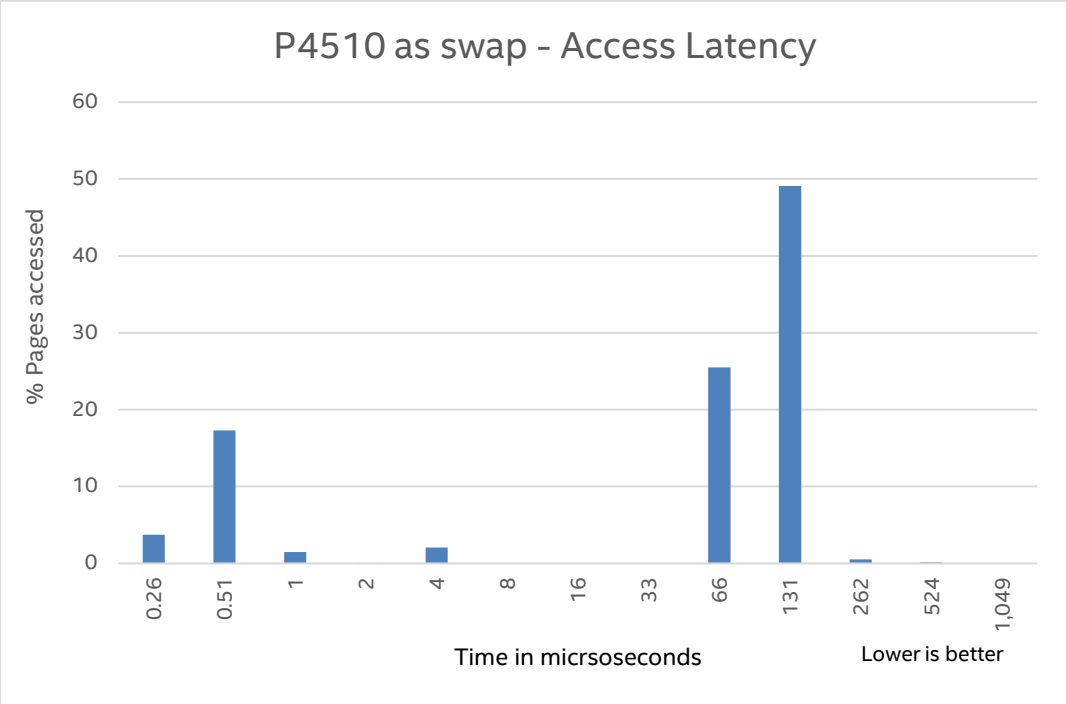




Access latency of the memory pages is significantly improved by using a high performance Intel® Optane™ DC SSD compared to a standard high performance NVMe-based NAND device. In the following graphs DRAM-only access result is provided first, then the swap access times of the Intel® Optane™ and NAND SSDs, which are in full microseconds. DRAM takes roughly 250 nanoseconds to access 4,096 memory pages. These graphs are provided by the histogram output of the pmbench tool and are critically important in understanding how closely a second tier media compares to DRAM. These graphs also clearly illustrate how using swap improves the application's response time.







A key takeaway is the access latency and the ability to scale effectively over a number of threads, which the data above illustrates. The NAND SSD provides a significant number of pages at 131 microseconds of access latency (that's very high), while the higher performance Intel® Optane™ SSD DC P4800X performs the same task at 16 microseconds, closer to the goal of <10 microseconds. This is about an 8x (800%) difference in page access latency between the two SSDs tested (131/16 is ~8). This is quite significant, as the closer to DRAM performance, the better the ability to oversubscribe memory successfully and maintain a good level of latency for requests of a real application, such as Redis in-memory key value data stores.

§

Appendix A Automation Scripts and How-to Guide

Please visit the repository for Linux swap testing instructions and testing code:

https://github.com/fxober/LinuxSwap

The repository contains process instructions and the software to test swap on Linux and x86 platforms.

1. The rar file contains scripts needed for achieving the performance data
2. The how-to-guide details setting up the benchmark and processing the results.

Best Practices for Configuration:

1. Use one (1) Intel® Optane™ DC SSD, or a Storage Class Memory SSD that supports high write traffic, for each NUMA node (i.e. CPU socket). For example, in a two socket system you would use two SSDs.
2. The Linux kernel can support 28 "swap partitions" so 14 partitions per SSD, assuming you use two SSDs for a two socket server. We recommend 28 swap partitions; each partition should be set with equal priority. Do this using syntax like this: / swapon –p 10 /dev/nvme01p1.
   Partitions are important to use as this will allow for the best scalability of the current (kernel 4.14 or greater) version of swap.
3. RCU call back processing causes long latency due to the amount of time spent on blocking, therefore it is important to avoid RCU call back processing in softirq.

   Here are a couple of options that enable you to offload the rcu processings to dedicated kthreads, depending on your kernel version:

   Set the following kernel configuration parameters.
   a. CONFIG_RCU_NOCB_CPU=y
   b. CONFIG_RCU_NOCB_CPU_ALL=y

   Depending on your kernel version, you may need only line **a.** above.

   The following kernel command line argument will need to be added to the process used to start the kernel:

   ```
   rcu_nocbs=<number of vcores/cores>
   ```

   Here is an example variable setting from /etc/default/grub, CPU count specific:

   GRUB_CMDLINE_LINUX_DEFAULT="rcu_nocbs=0-[n] maybe-ubiquity"

   Where [n] is the number of total CPU cores or virtual CPU threads in your system.

Configure the kernel with these .config settings if you are able to compile your own kernel.

4. EXPERIMENTAL: Generally speaking, it is best to set the NVMe scheduler to [none] on the NVMe SSDs which you are testing the mq block or kyber scheduler.
   In most cases your build shows [none], which is fine.

   ```
   # more /sys/block/nvme1n1/queue/scheduler [none]
   ```

5. Newer kernels allow an NVMe queue size of 1,023, which is sufficient and recommended.
6. If you are seeing NVMe block merges, change your NVMe block size to 4Kib (not 512b) sectors. If block merges are still occurring after making this change, try the following.

   First, check the nomerges value:

   ```
   # cat /sys/block/queue/nomerges
   ```

   The nomerges value should be set to 2. Verify and change if necessary:

   ```
   echo 2 > /sys/block/queue/nomerges
   ```

§

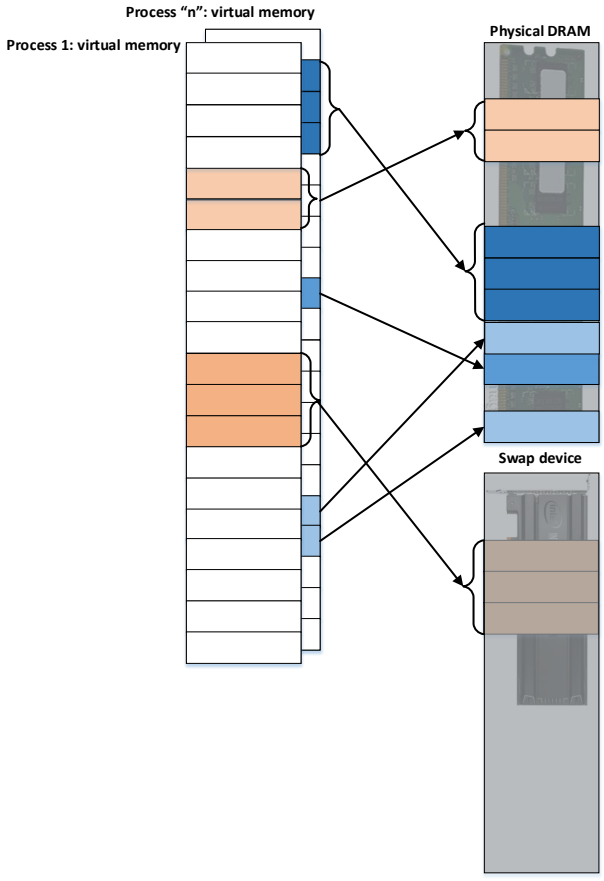
Appendix B Memory Management Fundamentals

This chapter introduces the basic memory management concepts used in the Linux kernel. It explains system level bottlenecks observed when Intel® Optane™ DC SSDs are used as swap devices with Linux versions prior to v4.14 of the upstream Linux kernel. Finally, it explains techniques to overcome those bottlenecks in version 4.14, so users can experience improved performance and utilize Intel® Optane™ DC SSDs as swap devices.

B.1 Memory Management System Overview

Modern operating systems implement a virtual memory model which provides many advantages to application developers. Virtual memory model simplifies software development, it leaves physical memory allocation and data placement complexity to the underlying operating system. The operating system kernel deals with that complexity by providing an impression to any running process that has a big chunk of memory available (usually 4GiB) for its exclusive use. In reality OS kernel maps process virtual memory to physical DRAM, and potentially overflows to a swap device, which extends available physical memory. The process of transferring data between the swap device and physical memory is called *paging* and consists of *page-ins* when the data is read from the swap device into physical memory, and *page-outs* when data is moved out of memory. It should be noted, *page-outs* may require data to be written out to the swap device, based on the state of the page. Figure 2 below provides a conceptual diagram of virtual memory and paging

Figure 2: Virtual Memory Concept through Paging



The paging process is managed by the OS and is heavily supported by CPU hardware through the memory management unit (MMU). For example, MMU contains translation lookaside buffer (TLB) cache which contains recent information on virtual-to-physical memory translations. This enables a significant reduction in time needed to access data in memory.

Another CPU feature that assists the OS with memory management is a mechanism called ***page fault.*** Page fault is an exception raised by CPU hardware when a process tries to access a virtual memory location that is not mapped to a physical address. There are different types of page faults:

- **Minor –** is risen when a page exists in main memory but there is no entry indicating virtual-to-physical address mapping. The page fault handler is implemented in the OS creates a new mapping entry.
- **Major** – is risen when a page does not exist in main memory. The page fault handler needs to bring required data from the swap device into memory and create corresponding mapping entry. For example, this happens in a freshly loaded process which causes the OS kernel to delay loading the whole program into memory. This technique, called on-demand paging, accelerates process start up. A major page fault is a performance draining procedure that requires the OS page fault handler to find an available location in physical memory, which can potentially involve paging-out and loading content of the program from the swap device into memory, before the process can continue its execution.

There are two different types of pages:

- **Filesystem pages, or pages backed up by the files.** These are memory pages that contain file data; for example, database files directly mapped into to process address space, or library files containing executable program code. These pages can be paged-in to physical memory; for example, when the program starts executing instructions stored on the disk (i.e. program usage of a shared library). The Linux page cache is a cache of these pages destined for files – both resident to-be-read, and changed (dirty) that need to be synchronized to some storage device. Direct access IO routines for which there is no page cache usage are also available on Linux. Since the page cache is an opportunistic and general usage cache, it is not appropriate for all usages.

- **Anonymous pages.** These are memory pages that contain private process information, that is heap or stack, and have no device or filesystem backing them. When the system is running into low memory conditions (high memory pressure) anonymous pages can be paged-out (swapped out) to the swapping file or swap device by OS process kswapd and its related kernel threads. This process can be more or less aggressive based on the configuration of the swappiness parameter, as this parameter sets the target of when swapping should become more active. The parameter can be set from 0 to 200; the higher the value, the more swap is utilized over page cache memory reclamation. In our performance study the OS is configured to its default value of 60, which is the typical production recommended setting. Value of 100 means that OS will reclaim memory pages using page cache and swap equally. You can print out proc variable /proc/sys/vm/swappiness to view its current value. Another important parameter used to control when kswapd kernel threads are activated is watermark_scale_factor. The user can set a lower limit of available memory that specifies when kswapd activity will be started. More details are available in Watermark scale factor section.

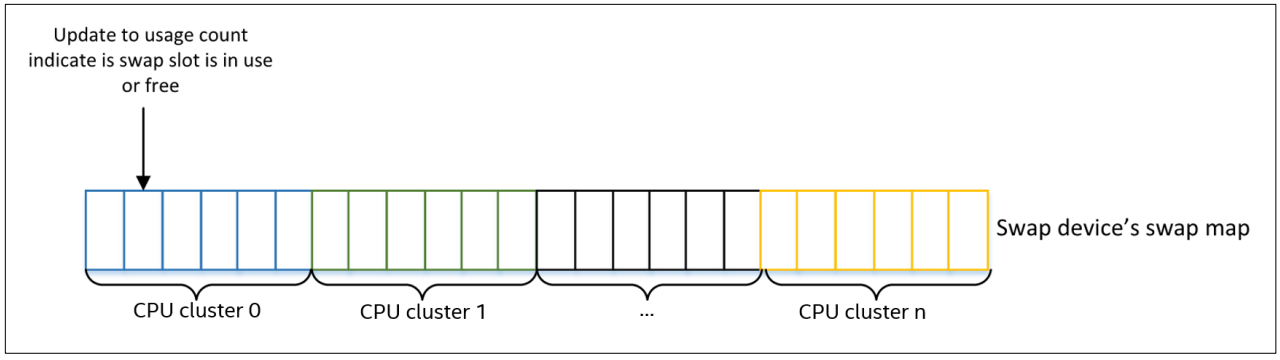
§

Appendix C Linux Kernel Innovations to Leverage Fast SSDs as Memory Extension

Until recently the Linux kernel had been primarily optimized for rotational disks because they were the predominant storage devices. One of the techniques used to maximize swap performance for rotational hard disk drives (HDDs) was to maintain swap data in the contiguous location on the disk to minimize disk seek time. The performance yields of this technique were fine for rotational hard disk drives (HDDs) but inadequate for solid state drives (SSDs). With recent advancements in non-volatile memory (NVM) technologies like Intel® Optane™ technology, new techniques and methods are needed to take advantage of the increased performance of the media and devices. While testing Linux swap against these new devices, many system-level bottlenecks were discovered in Linux swap. Kernel developers have addressed some of the performance bottlenecks in the release of Linux kernel 4.14. In this section we explore some of those enhancements.

Swap device in the Linux kernel is represented by a dedicated data structure (*swap_info_struct*) that contains information on how memory pages are stored on the swap device, see Figure 3 below. This information is stored in an array, called *swap_map* which is part of *swap_info_struct*. *Swap_map* stores information on usage count for a page stored on the swap device. *Swap_map* entries are aggregated into clusters, these clusters effectively assign specific portions of the swap device to the specific CPU core. Updates to the usage count of individual *swap_map* entries require per cluster locks to be taken instead of holding a single lock protecting the whole *swap_map.*

Figure 3: Primary Swap Device Data Structures



Even though there are dedicated swap entries per CPU cluster, accesses to the *swap_map* are protected by a single lock which is a scalability and performance limiter when concurrent attempts to the swap device are made. The negative impact of this single lock is especially visible in high memory pressure conditions.

When the single lock is used to protect critical information in the *swap_info_struct* data structure, latencies for handling page faults from the swap device are significantly increased. This heavily impacts end user performance and renders the latest HW latency improvements ineffective due to system level bottlenecks.

The next section explains techniques to minimize lock contention on the single lock that protects *swap_info_struct* data structure, and to improve system level latencies. As previously discussed in the Performance Data section, access latencies on swap average below 20 microseconds when utilizing a higher performance drive.

C.1 Swap Improvements Completed in v4.14 of Linux Kernel

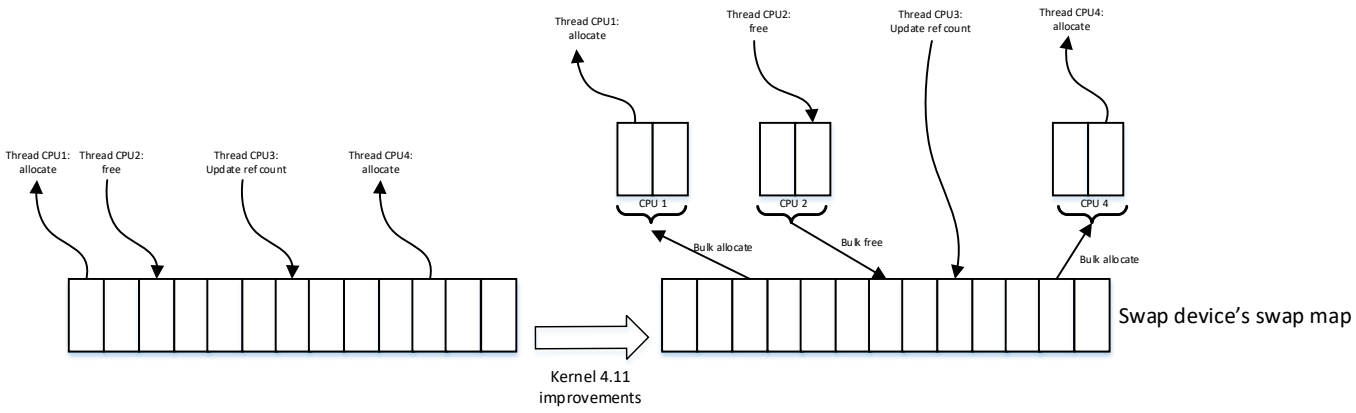
There are many software techniques to address performance problems related to lock contention. These approaches typically rely on the following principles:

- **Replacement of single coarse-grained lock on swap partition with multiple finer-grained locks on the swap cluster** – when many pieces of data are protected from concurrent accesses by a single, big lock, the concurrent threads that are attempting to read or write data are serialized in a queue while awaiting their turn. In such cases, to improve parallelism, a big lock can be split into many smaller locks to protect independent sub-pieces of data. This approach may yield significant performance improvements especially when multiple threads access independent pieces of data, however when more than one thread attempts to access the same piece of data, those attempts will be serialized in a queue.
- **Reduction of time spent when holding lock (or time spent in critical section)** – when there are multiple threads attempting to access a critical section that is protected by an exclusive lock held by another thread they are all paused until lock is released. The longer the critical section is, the longer the other threads will wait before they can continue. Reduction of time that given thread spends in the critical section is another useful technique increasing parallelism and reducing latency.

Kernel Developers determined that the occurrence of increased system level latencies while swapping to Intel® Intel® Optane™ DC SSD were caused by a single lock protecting *swap_info_struct* data structure. They have applied the principles discussed above into the series of swap improvements that are available in Linux kernel version 4.14 and later. The following techniques have been developed to reduce lock contention on the *swap_info_struct* lock.

1. **Bulk operations and per CPU lock cluster improvements** – multiple *swap_map* entries that represent free space on the swap device have been aggregated in larger units and stored in swap slot cache. Swap slot cache is managed by a specific CPU core, because of that it is called “per cpu swap slot cache”. When a SW thread requests new swap space it first tries to allocate it from swap slot cache on the given CPU. This operation does not require locking. Because single swap slot cache contains multiple *swap_map* entries it is likely that *swap_map* entry will successfully be allocated from it. When allocation from swap slot cache is not possible, swap software needs to perform bulk allocation of multiple *swap_map* entries from *swap_map,* and assign those entries to swap slot cache. *Swap_info_lock* is acquired when doing bulk operations on the *swap*_map data structure. Please refer to Figure 4 below for details of the changes.

Figure 4: Swap Bulk Operations Improvements



2. **Radix tree split –** another source of lock contention that existed in Linux kernel prior to version 4.14 was radix tree used for swap cache. Swap cache is an optimization in a swapping behavior that reduces the number of writes to swap device or swap file and maintains mapping between memory page and swap map entry when memory page is swapped in or swapped out. Swap write is considered unnecessary when a page exists in a swap device or swap file, as well as in main memory, because both of those locations contain the same data. When Linux considers page for reclamation it can simply check if it exists in both swap device or swap file, and in main memory and data in those two locations match. In such case page in main memory can be simply marked as invalid and reclaimed. To perform check if swap entry has corresponding page stored in main memory radix tree data structure is used. Swap cache radix tree prior to version 4.14 of Linux used to be protected by single swap cache lock which reduced parallelism. In version 4.14 single swap cache radix tree has been split into multiple smaller trees. This modification introduced separate locks per each smaller radix tree and increased parallelism. The current design method is best implemented with many swap partitions on the physical swap device. See Appendix A and the automation scripts on github to implement the maximum number of Linux swap partitions, typically 28.

§

Appendix D Swap Improvements Patch Lists

This section provides a list of kernel patches pertaining to swap improvements that were introduced in the Linux kernel 4.11 and in 4.14. This list of patches may be useful when considering creating a unique kernel image based on kernel versions older than 4.11, and backporting swap improvements into it.

```
commit 322b8afe4a65906c133102532e63a278775cc5f0
Author: Huang Ying <ying.huang@intel.com>
Date: Wed May 3 14:52:49 2017 -0700

    mm, swap: Fix a race in free_swap_and_cache()

commit 0ccfece6ed507738c0e7e4414c3688b78d4e3756
Author: Huang Ying <ying.huang@intel.com>
Date:   Wed May 3 14:56:16 2017 -0700

    mm/swapfile.c: fix swap space leak in error path of swap_free_entries()

commit 322b8afe4a65906c133102532e63a278775cc5f0
Author: Huang Ying <ying.huang@intel.com>
Date:   Wed May 3 14:52:49 2017 -0700

    mm, swap: Fix a race in free_swap_and_cache()

commit ba81f83842549871cbd7226fc11530dc464500bb
Author: Huang Ying <ying.huang@intel.com>
Date:   Wed Feb 22 15:45:46 2017 -0800

    mm/swap: skip readahead only when swap slot cache is enabled

commit 039939a65059852242c823ece685579370bc574f
Author: Tim Chen <tim.c.chen@linux.intel.com>
Date:   Wed Feb 22 15:45:43 2017 -0800

    mm/swap: enable swap slots cache usage

commit 67afa38e012e9581b9b42f2a41dfc56b1280794d
Author: Tim Chen <tim.c.chen@linux.intel.com>
Date:   Wed Feb 22 15:45:39 2017 -0800

    mm/swap: add cache for swap slots allocation

commit 7c00bafee87c7bac7ed9eced7c161f8e5332cb4e
Author: Tim Chen <tim.c.chen@linux.intel.com>
Date:   Wed Feb 22 15:45:36 2017 -0800

    mm/swap: free swap slots in batch
```

```
commit 36005bae205da3eef0016a5c96a34f10a68afa1e
Author: Tim Chen <tim.c.chen@linux.intel.com>
Date:   Wed Feb 22 15:45:33 2017 -0800

    mm/swap: allocate swap slots in batches

commit e8c26ab60598558ec3a626e7925b06e7417d7710
Author: Tim Chen <tim.c.chen@linux.intel.com>
Date:   Wed Feb 22 15:45:29 2017 -0800

    mm/swap: skip readahead for unreferenced swap slots

commit 4b3ef9daa4fc0bba742a79faecb17fdaaead083b
Author: Huang, Ying <ying.huang@intel.com>
Date:   Wed Feb 22 15:45:26 2017 -0800

    mm/swap: split swap cache into 64MB trunks

commit 235b62176712b970c815923e36b9a9cc05d4d901
Author: Huang, Ying <ying.huang@intel.com>
Date:   Wed Feb 22 15:45:22 2017 -0800

    mm/swap: add cluster lock

commit 6a991fc72d1243b8da0c644d3147d3ec41a0b281
Author: Huang, Ying <ying.huang@intel.com>
Date:   Wed Feb 22 15:45:19 2017 -0800

    mm/swap: fix kernel message in swap_info_get()

commit f6498b3f33123a6ee1c81a1b29b9c07964cb95c1
Author: Huang Ying <ying.huang@intel.com>
Date:   Fri Oct 8 16:59:30 2016 -0700

    mm: don't use radix tree writeback tags for pages in swap cache
```

D.1 References

See the following links for important reference information.

Most of the original patches: https://kernelnewbies.org/Linux_4.11#Memory_management

Second step swap optimization notes: https://kernelnewbies.org/Linux_4.14#Memory_management

Whitepaper on PMBench (2018): https://www.semanticscholar.org/paper/Pmbench%3A-A-Micro-Benchmark-for-Profiling-Paging-on-Yang-Seymour/dd0adcde7d074a414a9df76fb20d52a0d8aa8c71#paper-header

Whitepaper with deeper analysis of persistent memory's applicability to memory page access performance: https://web.cs.unlv.edu/jisooy/paper/yang_pmbench.pdf

§