Design Guideline for Disaggregated Persistent Memory System

Project Report

Xincheng Xie xx2365

Zhejian Jin zj2324

Abstract

The emergence of Persistent Memory (PM) provides a new choice for memory and storage. Disaggregated architectures provide better hardware elasticity and efficient resource utilization. However, there has been few works working on disaggregated memory system designed for PM. In this paper, we did experiments on the throughput and latencies among different types of memory/storage such as hard disks, DRAM, ramdisk, PM to have a better understanding of PM. We also implemented and deployed Infiniswap[4] over DRAM and PM on an RDMA network and evaluated its performance using workloads running on Memcached[1]. Based on the results, we proposed insights and seven guidelines on how to design a disaggregated memory system designed for PM.

1 Introduction

It is difficult for storage speed catch up with CPU speed. Therefore, storage hierarchy is designed to make up this gap, but misses in top level of the hierarchy is still a serious problem. Some applications solve this problem by putting their data into memory as much as possible, like Memcached[1], DNN[5], etc. However, as applications get larger, it is hard to fit entire applications into memory. In addition, DRAM is expensive, limited in size and consumes a lot of power. With the emergence of Persistent Memory (PM), it gives chances to solve this problem. PM has larger memory capacity (typically several TBs). It is also persistent but byte addressable, and has lower latency and higher bandwidth than SSD.

Monolithic machines are hard to fit various applications, because their hardware configurations are hard to change without interrupting services[2]. Disaggregated architectures can provide better hardware elasticity, and efficient resource utilization[2].

Hardwares are connected with networks, making it easier to add or drop single components and dealing with failure parts without suffering any downtime[2].

Therefore, disaggregated persistent memory system with can be an opportunity for applications with large memory needs. This work studied different configuration of disaggregated persistent memory system and provided seven guidelines for designing a high-performance system.

2 Related Work

LegoOS[2]: LegoOS divides a monolithic machine into three components, process component, memory component, and storage component. These three components' functionality is separated and different components are connected by a customized RDMA-based network stack.

pDPM[3]: It designs passive disaggregated PM storage systems, where it separates the location of metadata and data and uses different access mechanism.

Infiniswap[4]: Infiniswap[4] is a remote memory paging system designed specifically for an RDMA network. Because one-sided RDMA operations bypass remote CPUs, Infiniswap[4] leverages the power of many choices to perform decentralized slab placements and evictions.

3 Experiment Design

In order to figure out the rules for design of disaggregated persistent memory system, two kinds of tests are designed: unit tests and integral tests.

3.1 Test Devices

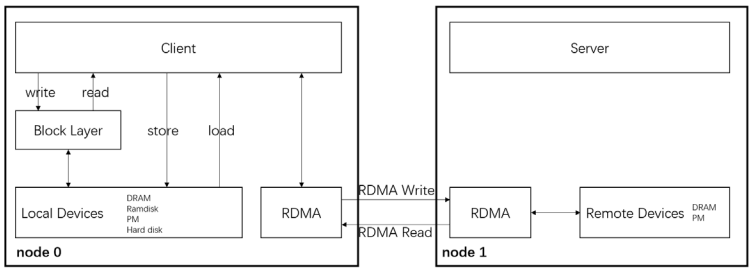

Figure 3.1 Test Devices.

Devices we tested are divided into two types: devices located in the local and devices located in the remote. (Figure 3.1)

Local devices include hard disk, ramdisk, DRAM and PM. Hard disks and ramdisks are seemed as block devices, which are accessed by `read()` and `write()` APIs. Hard disks use `ext4` file system while ramdisks use `tmpfs`. DRAM and PM are byte-addressable devices, which can be accessed by `load()` and `store()` APIs. However, in real implementation, PMs generally use a file system with direct access mode (`dax`), which directly mapped PMs into userspace[6]. We use `xfs` with `dax` mode open here.

Remote devices are accessed through RDMA over infiniband. Therefore, for remote part, we only test DRAM and PM. We use only RDMA write and RDMA read verbs to access remote devices.

3.2 Unit Test

The purpose of unit tests is to test the single operations' performance in both local and remote devices. These results can be used to benchmark and explain integral test results.

Unit tests mainly test a device's latency and throughput. Latency is tested by single thread with different bytes of objects. To test throughput, first measure the latency of the same size of objects (like 4KB) with different numbers of threads, then compute the throughput by:

$$\text{Throughput} = \frac{\#\text{ threads}}{\text{latency}}$$

3.3 Integral Test

We used Infiniswap[4] as our baseline for integral tests. Memcached[1] was running on top of Infiniswap[4]. To measure the performance, `memaslap`[7] was used to configurable workload. Memory size was limited by `lxc` container. The workload generated has 10% SETs and 90% GETs. We run `memaslap`[7] with fixed time 20 seconds.

4 Evaluation

This section presents the result of our experiments and state the rules we find to better implement disaggregated PM systems. We run experiments in CloudLab[8] with 2 nodes, each with one Intel Xeon E5-2450 processor, 16GB DRAM and one Mellanox ConnectX-3 InfiniBand network adapter[12]. We used `pmem.io`[10]

to emulate PM, which assigns 4GB of DRAM as PM device. The Linux version we used is v4.4.23. LXC containers[11] were used to restrict a process's memory.

4.1 Unit-test Results

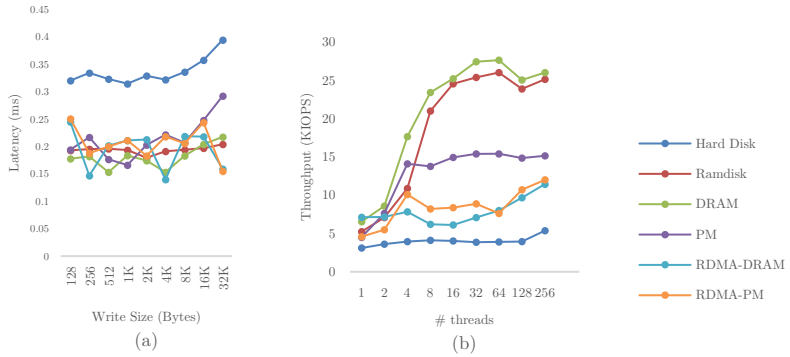

Figure 4.1 Unit-test Results for small object size. (a) is write latency of different size objects to different devices. (b) is write throughput of 4KB objects to different devices.

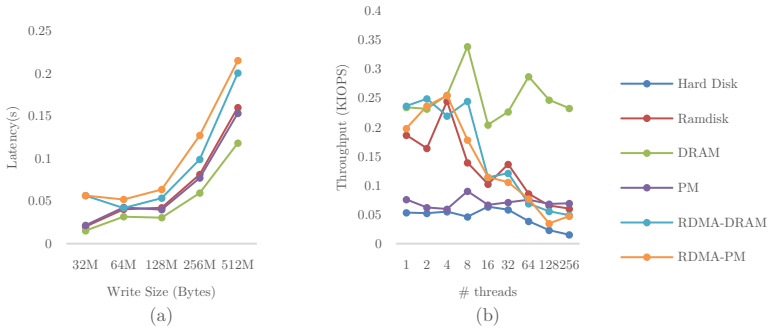

Figure 4.2 Unit-test Results for large object size. (a) is write latency of different size objects to different devices. (b) is write throughput of 8MB objects to different devices.

Latencies. Figure 4.1(a) shows write latencies for small objects. Except for hard disks, write latencies of other devices are similar for small writes. This is because latencies are dominant by other factors such as thread creation overhead, cache misses, etc. Figure 4.2(a) shows latencies for large objects, where write latencies dominate the overall latencies. In this case, the latencies acted as expected: RDMA has the highest latencies while DRAM has the lowest. Comparing these two results, we get our first design guide:

Guide 1: It is beneficial to aggregate small swaps before flushing to remote side in order to reduce the impact of fixed overheads. These fixed overheads include thread creation, cache misses, user to kernel switch overhead, etc.

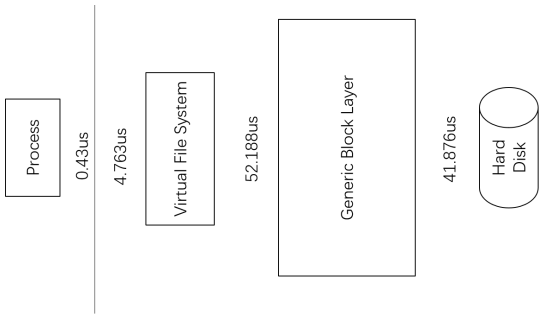

Figure 4.3 Latency breakdown of 4KB write to hard disk

There are two other interesting observations regards to the above results. The first observation is that in large writes latencies, ramdisk's latency is larger than DRAM's. These extra latencies are induced by file systems. In Linux, the file system for ramdisk is `tmpfs`. Latency induced by file systems composes half of the overall latency. We used `ftrace`[9] to measure the latency breakdown of 4KB write to hard disks, which is shown in Figure 4.3. This leads to our second guide:

Guide 2: It is beneficial to avoid frequent interaction with kernel in order to reduce the overhead. The overhead includes file systems overhead, context switch overhead, etc.

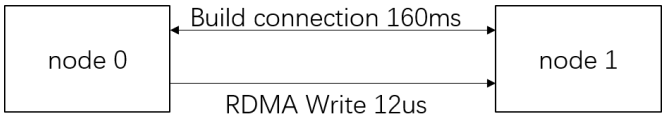

Figure 4.4 Latency breakdown of 4KB RDMA Write

The other counterintuitive observation is that latency of RDMA write is nearly as low as DRAM's latency when write size is not large. We think the reason is that after registered a memory area for RDMA, RDMA NIC may actively cache the data in that area, which may significantly reduce the latency. The most time-consuming part in RDMA is its complicated connection building, where nodes in each side exchange various information. We did not take this part of costs into account. Figure 4.4 shows the latency breakdown of writing a 4KB bytes through RDMA write. This leads to another guide:

Guide 3: It is beneficial to pre-register a pool of memory for RDMA in order to reduce connection building overhead.

Throughputs. We tested the throughputs of 4KB and 8MB with different number of threads. Generally, DRAM has the highest throughput while hard disks have the lowest. For 4KB, bandwidth has not saturated, so throughputs increase with the increment of number of threads; but for 8MB, because bandwidth has already saturated, throughputs decrease with the decrement of number of threads except for DRAM. Therefore, data can be partitioned into fixed-sized slices in order to saturate the bandwidth.

Guide 4: For aggregated data in guide 1, it is better to partitioned into slices in order to saturate the network bandwidth.

Another interesting observation is that RDMA-DRAM and RDMA-PM's latencies and throughputs are similar. This is mainly because RDMA writes and reads are bottlenecked by the network.

4.2 Integral-test Results

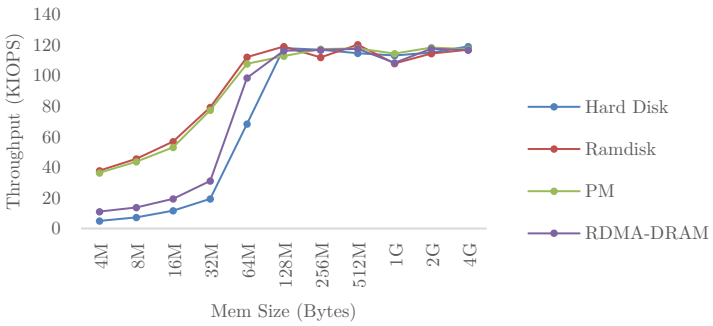

Figure 4.5 Infiniswap performance with different limited memory sizes.

The testing results are similar with the Memcached[1] results in the "Infiniswap" paper[4]. From Figure 4.5, the whole data set size is about 128M. When 50% working set is in memory, i.e., memory size is about 64M, the performance is about 85% of when whole working set is in memory. (The paper is about 80%.)

There is an explicit cliff in the graph, that is, when the proportion of data set in memory exceeds some threshold, performance will improve significantly.

Guide 5: Putting workload size slightly greater than threshold in the local memory and the remaining in the remote sides will not

only minimize memory consumption in the local, but also maximize performance.

A contradiction also presents in this integral test: As in the unit test, performance of RDMA is close to performance of local DRAM. However, in this integral test, when memory size is in range 4M to 32M, performance is close to local hard disks. There are mainly two reasons. The first one is that Infiniswap[4] doesn't start to swap the pages to the remote side until the page fault's rate exceeds 20 pages per second. When page fault is less than 20 pages per second, page is swapped to the local hard disks. Infiniswap[4] uses a passive swap-in and swap-out methods: only start swapping when page fault occurs. This will pause the execution of a program.

Guide 6: Actively instead of passively swap in and swap out will improve the overall performance. For example, a system can pre-flush a page and pre-fetch a page.

The second reason is that once a slab in the remote side is full, Infiniswap[4] needs to request a registration of a new slab from remote servers. This apparently violates guide 3. As presented in the unit test, register a memory will cost more time than RDMA write or read.

4.3 Additional Concern for Persistent Memory

Figure 4.6 Throughput when guarantee persistency to PM

When swapping to Persistent Memory, either local or remote, we do not guarantee data consistency, i.e., whether swapped data are persisted or not is undefined. If we guarantee the persistency, large amount of overhead would be induced. We write a simple simulation to demonstrate this idea. This simulator emulates page swaps with clock algorithm for page replacement. Each page flush, we guarantee that the page is persisted to PM. Access pattern of this simulation

is totally random. Figure 4.6 shows the result, which presents this huge overhead.

However, data written to PM are desired to have the ability to recover, which is one of the advantages towards DRAM. This indicates we may need to design a new protocol for the disaggregated persistent memory system.

Guide 7: New protocol is needed to guarantee data consistency when swapping to remote devices, in purpose of failure recovery.

5 Reference

[1] Nishtala, Rajesh, et al. "Scaling memcache at facebook." *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 2013.

[2] Shan, Yizhou, et al. "Legoos: A disseminated, distributed {OS} for hardware resource disaggregation." *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018.

[3] Tsai, Shin-Yeh, Yizhou Shan, and Yiying Zhang. "Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores." *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 2020.

[4] Gu, Juncheng, et al. "Efficient memory disaggregation with infiniswap." *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017.

[5] Bae, Jonghyun, et al. "FlashNeuron: SSD-Enabled Large-Batch Training of Very Deep Neural Networks." *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*. 2021.

[6] Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt

[7] memaslap. http://docs.libmemcached.org/bin/memaslap.html

[8] CloudLab. https://www.cloudlab.us/

[9] ftrace - Function Tracer. https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[10] pmem.io. https://pmem.io/2016/02/22/pm-emulation.html

[11] LXC. https://linuxcontainers.org/lxc/

[12] MLNX_OFED. https://www.mellanox.com/products/infiniband-drivers/linux/mlnx_ofed

6 Appendix: Interesting Remarks

1. Dealing with legacy system is really annoying

Source codes in Infiniswap's repository says it verify with many linux kernels, but it only works fine in kernel 4.4. In addition, many versions of RDMA drivers cannot work well. Another constraint is that in order to emulate PM, we need kernel version larger than 4.2. Fortunately, kernel 4.4 is larger than 4.2. However, in order to find a proper configuration, we have compiled nearly 30 different kernels, which is the most time-consuming part of this project.

2. Testing is not easy.

Testing an operation or a system need to take different factors into accounts; otherwise, unexpected things will occur.

The most unexpected stuff to me is the performance of RDMA. I tested the latencies of writing 1GB to local DRAM using `memcpy` and to remote side using RDMA write. The result is astonishing: `memcpy` uses about 300µs but RDMA write only uses about 200µs. I first think the data is not actually write to the remote side. But after RDMA write, I verify that the remote server side's memory has changed and the content is what I sent. Therefore, I think maybe RDMA NIC do something for me. RDMA NIC may cache the registered memory area, and leverage multi-core to send data concurrently; while `memcpy` only copies that sequentially.

The source code of this project: https://github.com/xxcisxxc/dis-store