In-memory Compression for Page Swapping

Joel Nider
joel@ece.ubc.ca

April 28, 2021

Abstract

Swapping memory pages to disk is commonly used to extend main memory at the cost of a significant decrease in performance. Swap device I/O latency is the largest contributor to the performance degradation, even with fast block storage devices. Caching compressed pages in RAM that are swapped out significantly improves the time for both loads and stores from a swap device. The cost is both additional processing time that the CPU must spend on compression and the reduced amount of available RAM. I propose to overcome these costs by using PIM (processing-in-memory) enabled DRAM for caching swapped pages. Pushing compression and bookkeeping responsibilities to the PIM-enabled RAM frees the host CPU and main RAM. I investigate the decisions and trade-offs in storage allocation on the PIM enabled DRAM. I conclude that a fixed-size block allocator with a 256KB block size is the best choice for the amount of available storage (60MB) and input page size (4KB) on my test system.

1 Introduction

Main memory is a precious resource. We often run programs than cannot fit in physical memory - either too many programs competing for ram, or a single program that has high demands. Virtual memory was designed to overcome this limitation by dividing the memory into small, equally sized chunks called *pages*. Hardware to support virtual memory was added to the CPU called the MMU (memory management unit). This gives the OS the flexibility to move memory contents to other devices, called *swapping*. Through page tables, the MMU can determine if a virtual address is backed by physical memory or not. If the CPU tries to access memory that is not present, the OS is given an opportunity (through a page fault) to fetch the page - called 'swapping in'. If a program requires more memory, the OS can take pages that haven't been used in a while and move them to secondary storage - called 'swapping out'.

Swapping is a way to extend memory, but it is slow and hurts performance. Swapped pages are generally written to block I/O devices (secondary storage). This effectively extends the size of RAM by the size of the secondary storage, which can be orders of magnitude larger than installed RAM. The problem is that the secondary storage is often orders of magnitude slower than RAM. Together with the overhead of the OS managing all of this movement, paging is very slow.

Compressing swapped pages and caching them in a portion of main memory is a popular approach[2, 3, 5]. Swap caches can be built to use a dynamic amount of RAM. As memory pressure grows, more of the RAM gets added to the cache which is compressed. At some point however, the number of physical pages available for the working set becomes so small that even the cache is no longer effective as the overhead of compression and swapping overtakes application processing time. A nice feature is the graceful degradation - application performance gets slowly worse over time as the memory pressure increases. The swap cache still relies on the CPU for compressing and decompressing pages as they are swapped in and out. While this is still faster than relying on secondary storage, it is much slower than accessing RAM directly. The problem is that it still takes up space in RAM and it distracts the CPU from running the application. It is slower than RAM, but faster than disk.

A better solution is to have the RAM compress the swapped pages by itself, without involving the CPU. PIM (processing in memory) may provide us with a way to improve performance of swap caches. The main advantage is that the memory can operate on many pages in parallel, thus improving performance. In addition, the CPU is freed from having to perform the compress/decompress operations and any bookkeeping related to stored pages. This can have secondary effects such as reducing i-cache misses in the application.

PIM has been proposed in the architecture community for a wide range of applications. Some companies are now on the verge of releasing the first commercial products. I explore the design space of a compressed page swap cache for Linux, using a PIM architecture based on the UPMEM[1] design.

2 Background

Linux kernel modules The Linux kernel is a monolithic kernel that is composed of core functionality and optional functionality. Much of the optional functionality can either be compiled into the kernel directly, or loaded dynamically in modules. The advantage to loadable kernel modules is that they may be loaded on-demand, for example, a device driver that is only needed for a removable device such as a USB thumb drive. Selective loading of modules reduces the kernel's memory footprint and reduces complexity when the modules are not needed. When modules are loaded dynamically, their init function is called which is the entry point for the module to register itself to various kernel APIs.

Frontswap API The Linux memory manager decides when and which pages to swap out when the number of available pages hits a certain threshold. It offers each page to the register swap caches through the 'frontswap' api until one accepts. When the module is loaded, the module registers itself as a page swap cache through the frontswap API. Each time a page is to be swapped out, the kernel will offer it to each registered cache in turn. Each cache has the option of accepting (i.e. caching) the page, or rejecting it. If all cache modules reject the page, it is written to a swap device without being cached. When a page is to be swapped in, the kernel will ask each cache if it has the page in question. Each page is accompanied by a unique identifier.

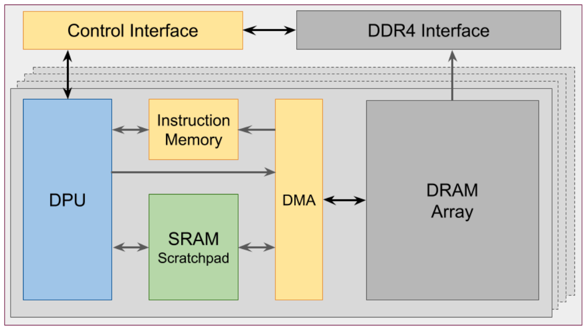

Figure 1: High level diagram of the UPMEM PIM architecture

Processing In Memory Processing in memory has a wide range of designs and implementations. My experiments are based on the UPMEM design, which is shown in figure 1. The following is a brief description of the major architectural points. This design embeds a simple, 32-bit, in-order processor (known as a DPU) with every 64MB slice of DRAM. Each DPU has exclusive access to its own DRAM (i.e. no other DPU may access this memory), although the DRAM may also be accessed by the host through the DDR interface. Due to the way 8-bit DRAM chips are attached to the 64-bit wide DDR bus, the DPUs are most efficiently controlled one rank at a time (8 chips = 64 DPUs). There is a access control which arbitrates access to the DRAM between the DPU (internal) and host (external), which also operates at rank granularity. The DPU can only directly access memory in the 64KB private SRAM. Any data that resides in DRAM must be first copied (with an explicit DMA fetch instruction) into SRAM. The DMA can move 8 bytes per cycle and stalls the processor during the copy. Since the processor is a simple in-order design with no prefetch, it is implemented as a barrel processor with multiple threads (called tasklets) to hide the latency. The DPU switches to the next (unblocked) tasklet every cycle, skipping over any tasklets that are waiting for a DMA transaction to complete.

3 Related Work

Existing Linux Solutions (1) zswap - in-kernel compressed cache for swapped pages. Zswap operates using the same API as our solution, and can compress pages using the same algorithms. Since the frontswap API supports multiple swap caches, zswap and PIM-swap could potentially be used at the same time. There is only a single zswap instance per machine, which means it is a point of serialization in the swap process. Zswap's allocator is called zpool. Zswap's page directory is a red-black tree (separate trees for each swap device).

(2) zram - a compressed, general purpose RAM disk. This is not a cache like zswap, but is a virtual storage device that compresses contents on-the-fly. It can be used as standalone swap device, or just a RAM disk (more use cases than zswap). The disk advertises a maximum uncompressed size, but it takes memory dynamically according to need. Zram could potentially be used as a backing device for zswap (or PIM-swap), although I don't know why you would want to do that.

Other Approaches One obvious solution is to swap to faster (usually flash-based) devices (SSD, NVMe). While faster, these devices are still much slower than RAM so it does not really solve the problem. Flash also tends to wear out quickly, which makes it not well suited for a swap device[15].

People have experimented with swapping to other machines over a network[9, 12, 13]. That works much better than local storage devices but depends on the availability of RAM on another machine nearby. If both machines are fully utilizing their local memory, then the memory manager must fall back to using a local swap device.

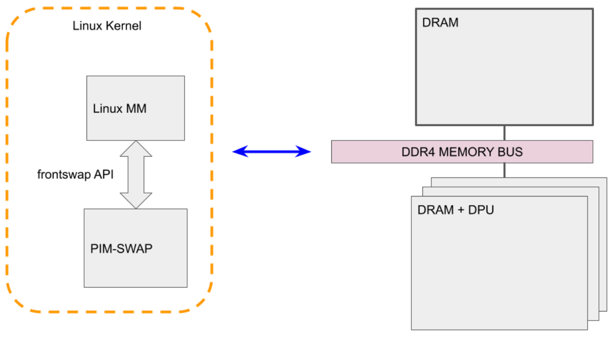

Figure 2: Overview

4 Design

I have implemented a page swap module called PIM-swap, which is much like zswap but with PIM support. The module gets requests from the Linux memory manager through the frontswap API to load, store or invalidate pages. Each page request is accompanied by a unique identifier which is used by the module for bookkeeping (i.e. finding the stored page). The module has two main components: the allocator which is responsible to manage the storage space, and the directory which is used to find stored pages during load or invalidation. Both of these components run on the DPU. The PIM-swap module operates on a single full rank of DPUs (64). It assigns pages to a DPU based on the lower 6 bits of the unique identifier (64 values), which is designed to maximize parallelism across a single scheduled hardware unit (the rank).

Page buffering Swapped pages are buffered in RAM order to increase parallelism in the rank by activating multiple DPUs simultaneously. There is a separate buffer for stored pages and for loaded pages, but they are constructed and used in the same way. Each buffer consists of 64 descriptors - one for each DPU in the rank, and a bitmap to check if a slot is free. Each descriptor holds a 4KB uncompressed page, the page ID and a status word.

Store During a store, the descriptor for the corresponding page ID is filled with the page contents and ID, if the descriptor is empty. The function can return immediately in this case, without writing anything to the DPUs. In the optimal case, this will occur 63 out every 64 stores. If the descriptor is already marked as used, a commit is triggered for the entire rank. When the rank is committed, all of the filled descriptors are copied to the DPU memory, and the corresponding DPUs are launched. Each DPU is then responsible for compressing its incoming page, and finding a place to store it in its DRAM. Since multiple DPUs are operating in parallel, the average latency of a store is relatively low (amortized across all active DPUs in the rank). The details about the compression, allocation and storage are explained later in this section.

Load A load is the reverse process of a store. The page ID is sent to the DPU, which looks up the page, decompresses the contents and returns the page. The page is copied from DPU memory, and returned to the kernel. Reading multiple pages in a single operation may provide further optimization and this is more fully described later in this section.

4.1 Allocator

The purpose of the allocator is to manage free storage space and provide a storage location of the correct size, on request. When a page is to be stored, the module must be able to allocate space to store the page. In general, an allocator may have variable size allocations (like malloc) or fixed size (like a block device such as a disk). To determine the best configuration for the allocator, we are interested in its efficiency in both space and time. First I give a brief comparison of *fixed size* vs. *variable size* allocation schemes. Following that is the result of an empirical evaluation of 2 'space' metrics: the efficiency of the allocated space, and the efficiency of the total used space.

Comparison with a file system This is not unlike a file system on a hard disk that allocates sectors for files. It may be possible to reuse a simple file system (i.e. FAT-16) for this purpose, but there are some important differences from the general file system use case that allows us to simplify the design even further. A complete file system has several features that are not necessary in a swapper: we don't have filenames, timestamps, extended attributes, dynamic resizing of files, or file locking. All of these features that are necessary in a full file system (even a simple one such as FAT) would only add complexity and reduce performance in our system.

Another important differentiating feature from a file system is the lifespan and usage of a stored object. In a file system, files may change sizes (grow or shrink) which means they need to have dynamically allocated space (usually implemented as chaining or extents). This is complicated to implement and hard to get right. It is also completely unnecessary in my case since a page can neither shrink nor grow once it is written.

Fixed-size allocation Fixed-size allocators always return blocks of the same size. This is often used in situations in which the media is not byte accessible, such as flash pages (often 4KB) or disk blocks (often 512B). Since our media (DRAM) is byte addressable, it is not immediately obvious why a fixed-size allocator is right choice. The subsequent explanations show that for the particular architecture I am modeling, it is more useful to view DRAM as a slow block device rather than random access, byte-addressable memory.

Variable-size allocation Variable size allocators are used when the free space (heap) is extremely large, and is therefore more efficient to manage with ranges rather than individually indexed blocks. The ranges are kept in a list, which must be searched linearly when looking for free space. Each range usually encodes its size in a hidden header at the beginning of the allocated space. In our case, that is not convenient since we would not know how large the range is (i.e. how much data to read in the DMA transfer) until the header is read and interpreted. This kind of allocator can also lead to fragmentation (unusable portions of memory) when ranges are removed and leave holes that are too small to be filled by new write requests. The design is also constrained by the DMA hardware which limits the minimum transferrable size to 8 bytes (with 8 byte alignment). I can therefore think of a variable size allocator as a special case of a fixed size allocator, with an 8-byte block size. The main difference is that a fixed-size allocator would use a bitmap to track the free blocks, while a variable size allocator would use a list of ranges. The 'storage' paragraph explains why the bitmap is more efficient for this architecture.

Efficiency of Allocated Space This metric measures how many bytes are actually being used out of the number of bytes allocated. This tells us how much space is being wasted on average in each allocation for a page. This number is influenced by the block size, because smaller blocks limits the potential waste. There is a tradeoff, because a smaller block size implies a larger number of blocks given a fixed amount of addressable space. More blocks requires more bits in the address and larger tables for managing the storage space. Figure 3 shows a comparison of the average utilization of allocated memory with different block sizes. There is a clear trend of decreasing utilization as the block size grows, which confirms the intuition that a larger block size has a larger potential for unused space. This is reflected in the number of stored (compressed) pages as well. An allocator with 2048-byte blocks stores nearly 20% fewer pages as compared to a 64 byte block (28248 vs 34156). Therefore, a design goal is to find the balance between a minimal block size, and low management overhead that increases with the number of blocks.

Efficiency of Total Space The second metric (efficiency of the total used space) measures the effect of fragmentation - the number of unused blocks that appear after deletions leave "holes" in the contiguous space. This effect can only been seen over a long time, after many inserts and deletes are performed on the data. Even after tests with over 200000 operations (300 seconds), there is hardly any fragmentation observed (less than 2% unusable blocks) across all block sizes. A small variation is seen with larger blocks having slightly lower rates but this does not have any significant effect since the variation is so small.

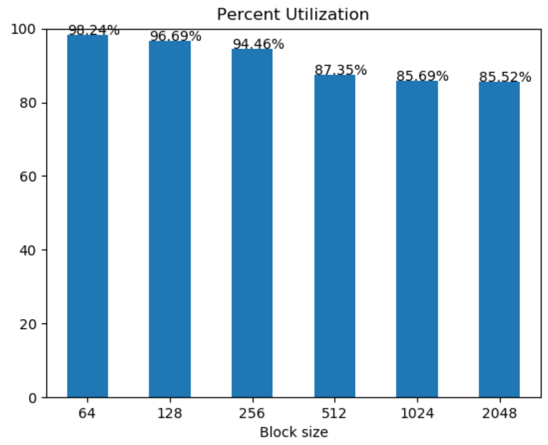

Figure 3: Average utilization of allocated memory with different block sizes

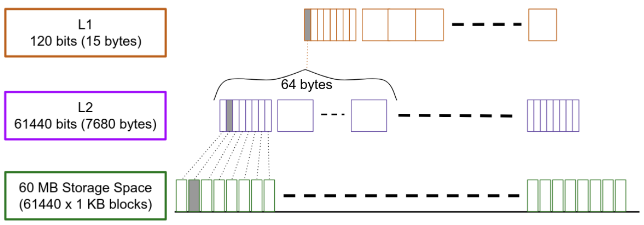

Figure 4: Multi-level bitmap allocator

Storage The design of the block allocation table is influenced by the microarchitecture of the DPU. With only a 64KB scratchpad, only small blocks may be moved at a time (no more than 2KB), and the time to fetch a block increases linearly with the size. Even with dedicated DMA engines, a low DPU clock speed and a private bus, fetching data from DRAM is expensive. To minimize the time required to find an empty block, I must minimize the number of data movement operations. To find a free block as quickly as possible, I use a two level bitmap allocator. Single level bitmap allocators have been used to improve performance over other structures in standard C++ libraries[4], storage applications[11] and file systems[8, 16]. Bitmaps work well with fixed-size blocks because the index of a bit in the bitmap can be calculated directly from the block number (and vice versa). Figure 4 shows the structure of the two level bitmap allocator with a 1KB block size. The bottom level (L2) represents each block with one bit.

$$bits = \frac{storage\_size(bytes)}{block\_size(bytes) * 8(bits\ per\ byte)}$$

A two level bitmap allows the allocator to skip over large groups of blocks that are already allocated. The upper level (L1) bitmap uses one bit to represent sections of 64 bytes in the L2 bitmap. Therefore, a single bit in the L1 can tell if there are any blocks free in a 512 block region. This ratio was chosen so the L1 bitmap could be loaded from DRAM as quickly as possible while still covering relatively small sections in the L2. The worst case for an allocation occurs when there are no runs of contiguous blocks to satisfy the allocation, but every section has at least one empty block (i.e. the L1 bitmap is completely empty). In that case, we must load every section of the L2 bitmap.

4.2 Page Directory

The purpose of the page directory is to find a stored page during a load operation. Since the allocator always allocates contiguous blocks for each stored page, we can retrieve a page by knowing the index of the first block, and the number of used blocks. It is important to have a fast lookup because the application is stalled while waiting for the page to be swapped in. Therefore, the primary concern is to keep retrieval latency to a minimum. I have considered 3 data structures for the page directory which are briefly compared below.

Hash Table A hash table is the simplest data structure to implement. The hash function can be performed in a constant time which makes this an attractive solution. Since both time and space are at a premium, I use a statically allocated hash table (simple array). The hash table runs in to problems when taking collisions into consideration. When there is a collision, the common solutions are chaining and finding the next available slot. Chaining requires dynamically allocated memory, which is complicated and time consuming in such a constrained system. Finding the next available slot can work with a statically allocated array but has the potential to deteriorate into an unordered list which no longer has a constant lookup time (searching the whole list on a hash miss in the worst case). Neither of these alternatives are particularly attractive, which encouraged me to look at other options.

B-tree A b-tree (nobody really knows what the B stands for) is a sorted tree that is a generalized form of a binary tree. Where each node in a binary tree has 2 children, a b-tree usually has more. Other (specialized) forms of b-trees include 2-3 trees and red-black trees. B-trees are often used in file systems and databases because they are known to be good when parts of the tree may not be in memory and retrieving them (i.e. reading from disk) is expensive. This is comparable with DPU architecture, since the data structure is likely not resident in the limited SRAM and access to the DRAM takes many cycles. Like the chaining solution in the hash table, b-trees also require dynamic node allocation (as the tree grows). The lookup is a bit slower (log n operations depending on the height of the tree) but has strict upper bounds on the worst case, and will not degenerate like the hash table.

Cuckoo Hash A cuckoo hash is a variation on the simple hash table that uses multiple hash algorithms[14]. Each hash algorithm is designed to return a different entry index for the same key. Like the simple hash, it can use a static array but differs in the collision policy. If the first hash algorithm causes a collision with an existing entry, another hash algorithm can be used to generate a different index. If all of the hash algorithms collide with existing entries, one of those entries may be rehashed using another algorithm and moved to an empty index, freeing up space for the incoming entry. This collision avoidance policy can be time consuming and is not guaranteed to converge. To reduce the possibility of collisions (and infinite loops of rehashing and relocating) the space in the static array should be substantially overprovisioned. In this case, I have ample space reserved for the hash table (approximately 390000 entries) which is sufficient for a cuckoo hash down to 256B block size (245760 entries). For smaller block sizes, the number of blocks exceeds the allocated space for the directory, and are therefore not compatible with this design choice.

Compression Algorithm I am using LZO[10], which is the same algorithm used by zswap by default. It gives a good balance of compression ratio vs processing complexity and is known to be quite fast[6]. LZO can compress a 4KB page from 44 bytes to 5200 bytes (dependent on the contents), with ~1900 bytes being the median. On a system with real PIM hardware (not emulated like mine), the compression algorithm would be implemented to run on the DPUs. In my case, I use the existing Linux kernel function since the DPUs are emulated. The algorithm can be changed by loading a different version of the executable into DPUs. In the future, I plan to test other algorithms before moving to real PIM hardware.

Optimizations As noted in the paragraph on compression, sometimes the compressed results are larger than the original page. In this case, I store the uncompressed page instead. Even in the case that the compressed page is smaller but uses the same number of blocks, I store the uncompressed page. This small optimization saves time during load by not having to decompress the page, without wasting additional storage space.

Another optimization that I mentioned earlier is reading multiple pages (one from each DPU) during a load. This is based on the observation that Linux swaps pages in and out in batches with sequential identifiers. The intuition is that a program (or portion of a dataset) is swapped out because it has not been used recently. When that program is later scheduled to run, the OS must then swap in the same portions that were swapped out, leading to sequential regions being swapped together. By reading multiple pages from sequential identifiers, the cache acts like a kind of prefetcher. Even if the hit rate is very low from the prefetched pages, it may be a huge success since the latency drops to nearly nothing by copying a page that is already uncompressed and waiting in memory. The cost is also minimal since we are already waiting for a single page to be decompressed (on-demand) and the DPUs operate in parallel. The additional cost is also minimal since the memory movement overhead is already covered by moving the single page.

4.3 PIM Simulator

In a system with real PIM-enabled memory, the PIM hardware is controlled by the PIM-swap module. In the interests of expediency, I have not experimented on a system with real hardware, and opted to simulate the hardware instead. There are several challenges when working on real hardware that are not necessary to overcome within the scope of this work (focused on the allocation of memory). For example, the in-kernel API is not as clear and well-documented as the userspace API. Because this is implemented as a kernel module, there is also a challenge of experimenting with the same machine that is used for development. Any small bug in the kernel module can destabilize the system, leading to lengthy reboots which slows down development. An alternative approach would have been to work in userspace on real hardware, but simulate the kernel paging mechanism. I believe running a real application and simulating the memory model of the machine in a simulated kernel would have been much more challenging and error prone. It would have necessitated reverse engineering the Linux kernel paging mechanism to understand the intricacies of the algorithm, and then faithfully reproducing it in a simulator. It may also have been possible to generate ”swap traces” by recording the activity of an unmodified kernel as it experiences memory pressure, and then replaying these traces in a simulator. My approach allows testing the applications "live" in a variety of configurations without having to record traces and without dealing with new hardware. I simulated the fundamental components of the PIM architecture as well as the API that a normal application would use to interface with it. The actual code that runs (i.e. DPU application) is compiled as part of the kernel module (x86_64 Linux code in C) but in separate functions that are only called as part of the DPU execution model. That means the simulation is able to test *functionality* but not *timing* since it executes on the same CPU as the rest of the kernel module whereas true DPU code would run in many separate processors of a different ISA.

API I chose to emulate the published userspace API from UPMEM for a few reasons. I am already well familiar with its use and the API is well documented on their website. I also believe the kernel API should follow the userspace API quite closely but likely includes additional details that are not necessary for my purposes at this time, and would only complicate the implementation. I only had to implement about 10 functions which are used for controlling the DPUs, and for moving data between DPU memory and host memory. The API allows for moving data by variable name (as would be exposed by symbol names in the DPU program's ELF executable file), which presented a small challenge. To address this, I created a struct that represented DPU memory, with each global variable as a field in the struct. By using the *offsetof* compiler intrinsic, I was able to simulate access to DPU memory by variable name.

DPU The main features that I simulate are the memories and execution model. The memories are allocated as kernel memory when the DPUs are allocated through the API. The size of the DRAM and SRAM available to the DPU are controlled by macros (i.e. can be altered by recompiling the module) so it is possible experiment with different memory sizes which is quite useful during debugging. The state of the DPUs is held in an array. The DPUs are executed sequentially (rather than in parallel as in real hardware) but this is equivalent since there is no communication allowed between DPUs. The only synchronization points are when the DPUs are launched and when they complete. I only allow synchronous operation at this time (the real API allows for asynchronous operation). The tasklets for each DPU are also executed sequentially. This is potentially more problematic since in the real hardware the tasklets may communicate and have dependencies. In my particular code, I only use a single tasklet which avoids this problem (possibly at the expense of performance in a real system).

5 Evaluation

5.1 Methodology

All tests were performed using a Linux kernel (v5.1) in Buildroot environment. The buildroot environment was hosted by a virtual machine (QEMU + KVM) on an x86_64 machine with 8GB of physical memory. I configured the virtual machine to have 2GB of physical memory and a 1GB swap partition which was backed by a virtual file on the host. All output was monitored and recorded through a virtual serial port in the virtual machine.

The experiments were performed with the stress-ng[7] tool. The tool starts a number of processes that consume a specified amount of memory, stressing the memory subsystem which causes paging. The majority of the tests used the following command:

```
stress-ng -vm 2 --vm-bytes 2200M --timeout 300s
```

6 Conclusion

Using PIM as a swap cache has many advantages, but has many design points that need to be explored before settling on a final design. The design is heavily dependent on the fundamentals of the PIM architecture, including amount of DRAM available to each processor as well as the algorithms used for allocation and page directory. The system that was tested in this report has 64MB of DRAM, which must be partitioned into storage space and directory space accordingly. Through experimentation, I show that a fixed-size block allocator with a 256 byte block size gives a good compromise between space efficiency (94.5%) and total number of blocks (245760) which fits into the available space for the directory. With this block size, the average case number of operations required to look up a stored page is reasonable. Further study is required to measure the performance of the design to show the overall benefit of PIM vs. a purely RAM based approach.

References

[1] F. Devaux. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–24, Los Alamitos, CA, USA, aug 2019. IEEE Computer Society. doi: 10.1109/HOTCHIPS.2019.8875680. URL https://doi.ieeecomputersociety.org/10.1109/HOTCHIPS.2019.8875680.

[2] F. Douglis. The compression cache: Using on-line compression to extend physical memory. In *USENIX Winter 1993 Conference (USENIX Winter 1993 Conference)*, San Diego, CA, Jan. 1993. USENIX Association. URL https://www.usenix.org/conference/usenix-winter-1993-conference/compression-cache-using-line-compression-extend-physical.

[3] M. Freedman. The compression cache: Virtual memory compression for handheld computers. 2000. URL https://www.cs.princeton.edu/~mfreed/docs/6.033/compression.pdf.

[4] GCC. *The GNU C++ Library*. Free Software Foundation, 2021. URL https://gcc.gnu.org/onlinedocs/libstdc++/manual/bitmap_allocator.html.

[5] S. Jennings. zswap. 2021. URL https://www.kernel.org/doc/html/latest/vm/zswap.html.

[6] I. Johnson. Why lzo was chosen as the new compression method. 2020. URL https://snapcraft.io/blog/why-lzo-was-chosen-as-the-new-compression-method.

[7] C. I. King. stress-ng: A stress-testing swiss army knife. Oct 2019. URL https://elinux.org/images/5/5c/Lyon-stress-ng-presentation-oct-2019.pdf.

[8] B. Matthews. *Homework Solutions*. Truman State University, 2021. URL http://matthews.sites.truman.edu/files/2017/01/chapter12.pdf.

[9] T. Newhall, E. R. Lehman-Borer, and B. Marks. Nswap2l: Transparently managing heterogeneous cluster storage resources for fast swapping. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, page 50–61, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343053. doi: 10.1145/2989081.2989107. URL https://doi.org/10.1145/2989081.2989107.

[10] M. F. Oberhumer. Lzo. Mar 2017. URL http://www.oberhumer.com/opensource/lzo/.

[11] N. Ojha. Rhcs 3.x - backport and make default bluestore allocator as bitmap. 2019. URL https://bugzilla.redhat.com/show_bug.cgi?id=1728069.

[12] J. Oleszkiewicz, L. Xiao, and Y. Liu. Parallel network ram: Effectively utilizing global cluster memory for large data-intensive parallel programs. In *Proceedings of the 2004 International Conference on Parallel Processing*, ICPP '04, page 353–360, USA, 2004. IEEE Computer Society. ISBN 0769521975.

[13] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramcloud. *Commun. ACM*, 54(7):121–130, July 2011. ISSN 0001-0782. doi: 10.1145/1965724.1965751. URL https://doi.org/10.1145/1965724.1965751.

[14] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004. ISSN 0196-6774. doi: 10.1016/j.jalgor.2003.12.002. URL https://doi.org/10.1016/j.jalgor.2003.12.002.

[15] T. Song, G. Lee, and Y. Kim. Enhanced flash swap: Using nand flash as a swap device with lifetime control. In *2019 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–5, 2019. doi: 10.1109/ICCE.2019.8662047.

[16] L. Technologies. Ntfs file system. URL https://www.ntfs.com/exfat-allocation-bitmap.htm.