M.S. THESIS

RapidSwap: An Efficient Hierarchical Far Memory

RapidSwap: 효율적인 계층형 Far Memory

AUGUST 2021

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

김 현 익

RapidSwap: An Efficient Hierarchical Far Memory

RapidSwap: 효율적인 계층형 Far Memory

지도교수 Bernhard Egger

이 논문을 공학석사 학위논문으로 제출함

2021 년 04 월

서울대학교 대학원

컴퓨터공학부

김 현 익

김현익의 공학석사 학위논문을 인준함

2021 년 06 월

위 원 장	이 재 진
부위원장	Bernhard Egger
위 원	김 진 수

Abstract

As computation responsibilities are transferred and migrated to cloud computing environments, cloud operators are facing more challenges to accommodate workloads provided by their customers. Modern applications typically require a massive amount of main memory. DRAM allows the robust delivery of data to processing entities in conventional node-centric architectures. However, physically expanding DRAM is impracticable due to hardware limits and cost. In this thesis, we present RapidSwap, an efficient hierarchical far memory that exploits phase-change memory (persistent memory) in data centers to present near-DRAM performance at a significantly lower total cost of ownership (TCO). RapidSwap migrates cold memory contents to slower and cheaper storage devices by exhibiting the memory access frequency of applications. Evaluated with several different real-world cloud benchmark scenarios, RapidSwap achieves a reduction of 20% in operating cost at minimal performance degradation and is 30% more cost-effective than pure DRAM solutions. RapidSwap exemplifies that the sophisticated utilization of novel storage technologies can present significant TCO savings in cloud data centers.

Keywords: Far Memory, Virtual Memory, Swap Systems, Cloud Datacenter, Operating Systems

Student Number: 2019-22487

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, memory resources have been intensively utilized and are actively recognized as a critical component for warehouse-scale computing (WSC) workloads, such as databases, distributed big-data processing, artificial intelligence, and many more [13, 20, 22]. These in-memory applications require an enormous amount of physical memory to avoid performance degradation from processing and retrieval latency [28]. For example, Google Cloud Platform, Microsoft Azure, and Amazon Web Services provide machines with up to 24 terabytes of main memory to operate a proprietary in-memory database [1, 2, 9].

Dynamic RAM (DRAM) enables the robust delivery of data to processing entities in conventional processor-centric architectures, and it is still maintaining its position as the fastest component in the storage tier. However, infinitely expanding memory capacity is unfavorable in terms of hardware limitations. Also, constituting overestimated memory resources in the warehouses yields resource under-utilization and high total cost of ownership (TCO). Therefore, cloud datacenter operators are challenged to allocate and harmonically orches-

Instance Type	vCPU	Memory (GiB)	Storage (TB)	Cost ($/hr)
c6g.metal	64	128	-	2.176
r6g.metal		512		3.2256
r5.metal	96	768		7.296
i3en.metal			60	10.848

Table 1.1: Amazon EC2 pricing. [8]

trate memory resources more than ever.

While it is ideal to store all the data in the fastest storage for maximum performance, maintaining all contents in DRAM heavily impacts the warehouses' TCO. This directly affects the pricing policy of cloud services. Table 1.1 analyzes the on-demand pricing of several Amazon EC2 instances. Instance *r6g.metal* offers 384 GiB more memory than *c6g.metal* for an additional $1.0496 per hour. Amazon also provides 60 TB of local NVMe SSDs for *i3en.metal* for an extra $3.552 per hour. Combining these two observations, Amazon is charging $\times$42.19 more on DRAM over SSD for the same capacity. In the market, the per-gigabyte cost of the former is about six times higher than the latter; this without accounting for the surge in DRAM price due to increasing demand and supply chain shortage [6, 7]. Our observation shows that warehouse operators try to minimize memory pressure by imposing a relatively large price penalty on DRAM.

Paging has been the core of the virtual memory management of operating systems. It dissolves the capacity gap between the virtual memory and the actual physical memory by extending physical memory to attached storage devices. The Linux kernel tries to reclaim pages in proactive way if the number of free pages drops below a certain threshold. If the memory pressure surpasses

what proactive reclamation thread can tolerate, on-demand reactive reclamation will be triggered and can cause a significant page reclaim latency [17]. The research community has suggested several solutions for this issue [14, 15, 16]. However, these solutions extend the failure domain, contribute to complex recovery, or do not show significant improvements in the TCO. In this thesis, we try to answer the following: Is it possible to reduce memory pressure and expose cheaper memory to cloud customers while minimizing the performance penalty?

This thesis presents RapidSwap, an efficient hierarchical far memory that leverages tiered storage to maintain a high memory utilization. RapidSwap exploits phase-change memory (Intel Optane persistent memory), a novel storage technology, in data centers to present near-DRAM performance at a significantly lower total cost of ownership. RapidSwap migrates cold memory contents to slower and cheaper storage devices by tracking the memory access frequency of applications. RapidSwap delivers 20% reductions in operating cost and is 30% more cost-effective the pure DRAM solutions when evaluated with well-known cloud benchmarks.

This thesis is organized as follows: Chapter 2 presents the background and related work for RapidSwap. Chapter 3 describes the motivation for our work. Chapter 4 address the design and implementation of our technique. Chapter 5 deliver the experimental setup and a detailed evaluation of RapidSwap. Chapter 6 discusses conclusion and future works.

Chapter 2

Background

2.1 Tiered Storage

Tiered storage, also known as hierarchical storage, is a widely adopted technique in computing devices [11]. Typically, these systems place faster high-cost storage devices in the upper portion of the hierarchy while slower low-cost media are located in the lower levels of the hierarchy. Placing all the data in high-performance devices is preferred, but may be unfavorable in terms of operating costs. Since access patterns of real-world applications are subject to locality, one way to decrease the operating cost is to optimize the system towards these characteristics [24]. In *tiered storage*, policy to arrange and place every data chunk must be designed exquisitely to create an illusion that there is no need to consider the trade-off between performance and cost. This is done by eventually segmenting all data by temperature (*hot* or *cold*). Data which has turned cold is not likely to be accessed near future and is available as a candidate for migration into slower devices.

2.2 Trends in Storage Devices

Recently, storage devices with dramatically improved performance characteristics have emerged in the market. Non-volatile memory (NVM) devices such as phase-change memory (PCM) used in Intel 3D XPoint show read/write theoretical latencies of 10 µs [12]. This figure is three orders of magnitude lower than conventional Hard Disk Drives (HDD). Aside from SATA interfaces, storage devices are available in different interfaces such as NVDIMM or PCIe. The NVDIMM interface allows direct load/store accesses from CPU cores and able to enjoy cache benefits. It is a technical trend to consider and efficiently support the emerging fast storage devices in file systems [27, 29]. By allowing direct access (DAX) to the device, they achieve better performance than block-based storage devices. Table 2.1 shows the read/write latencies that we measured on different storage devices.

Device	**4K Read Latency**	**4K Write Latency**
DRAM	434 ns	439 ns
PMEM	1183 ns	1789 ns
NVMe SSD	9837 ns	28 870 ns
SATA SSD	94 930 ns	36 143 ns

Table 2.1: 4K read/write latency in different storage devices.

2.3 Techniques Proposed to Lower Memory Pressure

2.3.1 Transparent Memory Compression

Zswap is an in-kernel feature that was officially merged in the Linux kernel v3.11 [18]. *Zswap* attempts to compress swapped-out pages with a lightweight

and robust compression algorithm. If the size of the compressed page including metadata is smaller than the uncompressed page, the compressed data is stored in local memory pool. Rejected pages from *zswap* are handed over to swap devices. *Zswap* serves as an efficient cache for swap devices especially when the read latency from the *zswap* cache is dramatically faster than the physical backing store as it can obtain the original page content from DRAM in a robust manner. Figure 2.1 shows the overall flow of *zswap*.

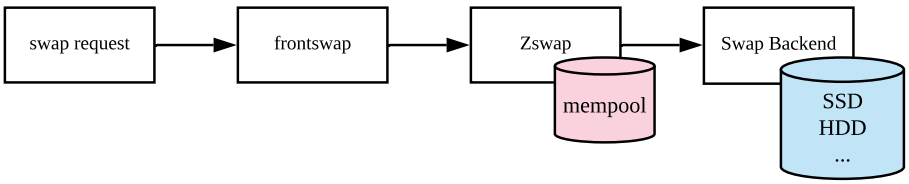


Figure 2.1: Zswap flow.

2.3.2 Far Memory

How to efficiently handle scarce memory resources has been one of the core topics studied by the research community. Memory that is not available as natural or *local* is defined as *far memory*. Currently, *far memory* is proposed through 2 different forms – 1) software-based approach and 2) memory disaggregation.

Software-Defined Far Memory

Software-based approach has been the classic method to implement *far memory* [4, 26]. It is offered as an intermediary tier between DRAM and secondary storage devices. Recently, Google researchers have revisited the idea and implemented in-DRAM compression in warehouse-scale without additional hardware

support [15] .

Google's software-defined far memory re-implements the existing *Zswap* mechanism in Linux to compress a certain portion of the original memory, which is still stored in DRAM. All memory pages in this case are classified as either hot or cold by their last access time. If a page has not been touched for more than T seconds, it will be considered as a cold page and ready to be *sent* to the *far memory*. Parameter T is dynamically adjustable and optimized towards Google's warehouses with both hands-on-dirty and machine learning approaches.

On average, Google reports that 20% of all pages are classified as cold, and among them, about 69% are compressible by 1/3. The other cold pages incur metadata overhead and there is no gain from compression. Overall, Google was able to achieve a 4-5% reduction in the overall TCO. However, the number of absolute compressible pages is limited and the figure in the savings is not overwhelming on a smaller scale.

Memory Disaggregation

Memory Disaggregation is another form of *far memory* to increase resource utilization. By pooling memory resources from different physical nodes over low-latency and high-throughput network, disaggregated memory can simply overcome the limitations of the current node-centric computation model. If local memory is not sufficient, remote memory will be used as a swap device.

Gu et al. [10] presented *Infiniswap*, a remote paging system under Remote Direct Memory Access (RDMA) network. Infiniswap divides and distributes swapped memory to memory spaces along with remote machines. *Infiniswap* is exposed as a block device and does not need any extra modifications after opening the device. Jo et al. [14] proposed *RackMem*, which implements a cus-

tom demand paging through disaggregated memory. *RackMem* has resolved the latency caused by slow page reclamation of the Linux kernel.

Previous studies [14, 16, 21] leverage remote memory to increase the overall memory utilization of the over-committed memory. They allow page size fine-grained distribution memory pressure. However, memory disaggregation extends the failure domain from local to remote machines which makes dealing with fault tolerance more complex. It also incurs a certain amount of network traffic to swap in/out memory pages to remote machines. Lee et al. [16] ensure fault tolerance by applying erasure coding rather than the full memory replication.

Additional replication, erasure coding [16], or a hybrid approach [25] may be applied to ensure data recoverability. Still, this requires either extra remote memory or computation.

Chapter 3

Motivation

3.1 Limitations of Existing Techniques

Pure *zswap* behaves like a temporary caching layer between DRAM and the backing store. If most of the memory pages are not compressible, *Zswap* becomes a redundant overhead layer for the backing store, wasting extra CPU cycles and energy. In the worst case (thrashing), the *zswap* layer may become the critical bottleneck contributing to page restore latency.

Recent *software-defined far memory* [15] secures extra memory space by utilizing *zswap* to apply in-memory compression. However, only a small fraction of memory contents are subject to compression. It does not dramatically improve memory pressure since it solely relies on DRAM to accommodate memory contents. It also causes additional computational overhead by attempting to compress all pages requested by the swap system. Moreover, Google's work requires hand-tuning or sophisticated machine learning model to optimize several parameters that affect the efficiency of the system and it is generally not

applicable in other environments.

Memory disaggregation through high-speed interconnect tackles the conventional memory limitation caused by node-centric architectures. Since the remote memory is available as a fast swap device, the optimized page fault handler [14] for fast devices resolves the problems that the current demand paging shows. However, *outsourcing* one's memory to other nodes extends the failure domain from a single machine to remote machines. In addition, memory contents are visible from other nodes. Therefore, *memory disaggregation* must consider additional *fault tolerance* or *security* aspects. In this case, redundant use of remote memory or computation is necessary.

3.2 Tiered Storage as a Promising Alternative

RapidSwap implements *far memory* by leveraging the *tiered storage* concept. It does not attempt to make any modifications to the original memory contents which consume CPU cycles. It physically alleviates memory pressure by migrating memory pages by their temperature, which allows more condensed use of the memory resources. Also, placing unnecessary pages in the cheaper improves the overall TCO. Our work does not consume any network resources or send any memory contents to other nodes. This greatly contributes to confining the failure and security domain to the local machine. RapidSwap only utilizes temporal and spatial locality to efficiently act as *far memory*.

Table 3.1 summarizes recently proposed system designs that aim at lowering memory pressure. All of these systems leverage page-sized granularity. All of the previous works may require additional CPU or network resources to directly compress data to either save storage capacity or the amount of the data transferred through the network. Some of these works may utilize memory on remote

nodes and may require additional fault tolerance or security considerations. On the other hand, RapidSwap confines the failure domain to the local node only and does not require any fault tolerance techniques which consume additional computational resources.

Type	**Granularity**	**Failure Domain**	**Overhead**
Software-defined Far Memory [15]	Page	Local	CPU
Hydra [16]		Remote	CPU & Network
RackMem [14]		Remote	Network
RapidSwap		Local	-

Table 3.1: Comparison of various work that lowers memory pressure.

Chapter 4

RapidSwap Design and Implementation

4.1 RapidSwap Design

RapidSwap utilizes different non-volatile devices in the storage hierarchy by the performance of each storage entity. Figure 4.1 shows the overall architecture of RapidSwap. The two core components of RapidSwap are the **Storage Frontend** and the **Storage Backend** layer.

4.1.1 Storage Frontend

RapidSwap contains a *storage frontend*, which exposes the tiered storage as a single device to the *swap handler*. The main role of the *swap frontend* is to swap in/out pages from/to different storage devices. Therefore, the *storage frontend* maintains the relevant metadata of mapping between the virtual address and its actual location in the storage hierarchy. *Storage frontend* groups and manages pages into certain sized *slabs* for spacial locality. As RapidSwap aims to

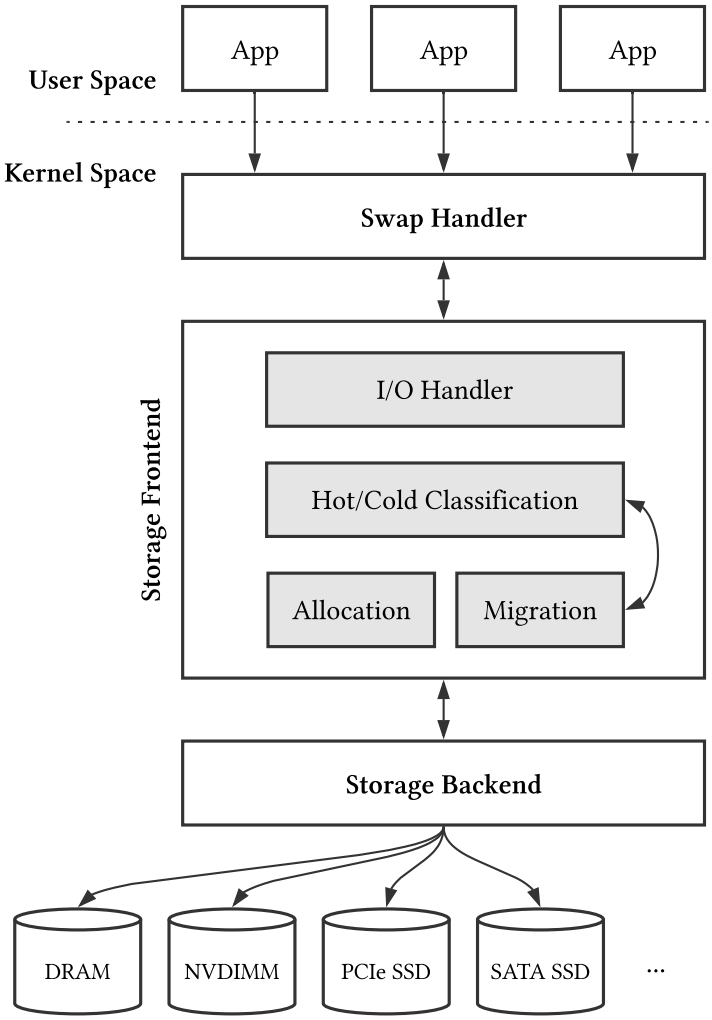


Figure 4.1: Overall RapidSwap architecture.

lower the TCO, it is also responsible for (1) classifying swapped-out pages into hot/cold pages and (2) keep them in appropriate storage backends according to their temperatures.

I/O Handler and Allocation

The *storage frontend* manages metadata of all allocated slabs. When any piece of data in the *far memory* needs to be accessed, the *swap handler* requests the *I/O handler* in the *swap frontend* with the unique id of the slab and the virtual address. The *I/O handler* then calculates the relative offset of the page within the slab and requests the appropriate *storage backend* for the physical read or write access to the storage device. The *I/O handler* is responsible to *turn on* the *accessed bit* when a slab is accessed. Slabs are not allocated until they are first-accessed (lazy-loaded). In this case, the *allocation* task. A new slab is always allocated from the fastest device available.

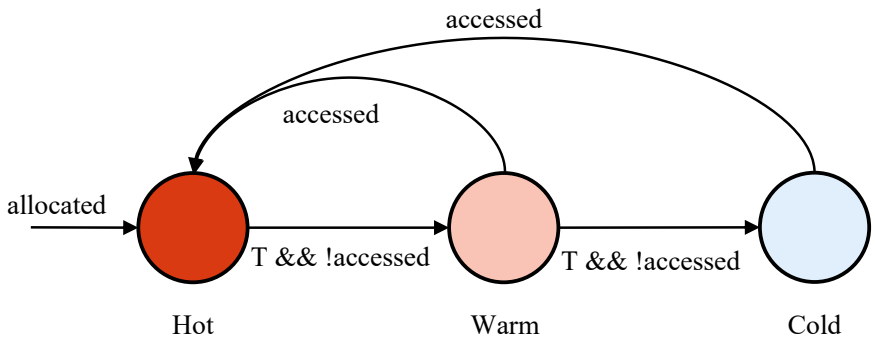


Figure 4.2: State transition diagram of RapidSwap temperatures.

Hot/Cold Classification

During the lifetime of RapidSwap, *Hot/Cold classifier* repeatedly checks the *accessed bit* of all slabs every second and categorizes them into one of the following three states: *hot*, *warm*, and *cold*. If a slab is found *accessed*, it is marked as *hot*. Slabs that have not been accessed more than T seconds are classified as *cold*. *Warm* is an intermediate state between *hot* and *cold*. *Warm* slabs are given the grace period for T seconds before they are recognized as *cold* and reside where they are before the physical migration. The access history is tracked and recorded by the *hot/cold classifier*. Newly allocated slabs are considered *hot*. Figure 4.2 shows the state diagram.

Migration

After every slab has its temperature, *migration* is performed. The *migration* task promotes slabs from slower to faster devices if a slab has been identified as *hot*. On the other hand, if a slab becomes *cold*, it is downgraded to a slower device. *Migration* is not practiced to those who are shown as *warm*. *Warm* slabs are given T seconds grace period before being downgraded. Slabs that are in the uppermost or lowermost of the hierarchy are not the subjects of promotions/downgrades.

4.1.2 Storage Backend

Storage backend serves as a uniform abstraction for physical storage devices. Currently, RapidSwap recognizes 6 different storage types as shown in Table 4.1. During the *storage backend* driver provisioning, it automatically gathers the information about the device and calculates the maximum number of slabs that the device can accommodate.

Priority	Name	Details
0	DRAM	vmalloc'd memory region
1	PMEM	Persistent memory
2	RMEM	Remote memory
3	PCIE	NVMe in PCIe interface
4	SSD	SSD in SATA interface
5	HDD	HDD in SATA interface

Table 4.1: RapidSwap storage types.

Function	Details
alloc()	allocate a new slab from the device
dealloc()	free the slab from the device
read()	read from the specific page of the slab
write()	write to the specific page of the slab
sg_read()	(optional) read from N pages of the slab
sg_write()	(optional) write to N pages of the slab

Table 4.2: Storage backend APIs.

When the *storage backend* has been loaded, it registers itself to *storage frontend*. *Storage frontend* performs several checks to identify the device characteristics such as allocation, 4K read/write, and deallocation latencies. When the device is fully investigated, *storage frontend* puts the *storage backend* in the list of the device and sorts the list by the performance of the storage devices. *Storage frontend* fully manages *storage devices* during the RapidSwap lifetime. Each *storage backend* must implement the following mandatory APIs as shown in Table 4.2.

4.2 RapidSwap Implementation

4.2.1 Swap Handler

Memory allocations and all related requests for RapidSwap are given from the *swap handler*. To use *far memory*, the *swap handler* must call relevant functions given by the *swap frontend*. In this thesis, we implemented our custom *swap handler*. Our version of the *swap handler* retains the individual metadata to transparently manage the virtual memory area of the user-space applications. After user applications register their memory through the system-call interface, they can access the allocated memory through conventional load/store instructions. The *swap handler* is based on our previous work [14].

The memory area managed by the swap handler is equal to the size of an individual page in the Linux kernel. The swap handler retains a pre-defined number of pages that can reside in the local memory. It could allocate new pages until the number of allocated pages reaches the limit. Once the local pages are full, some pages will be selected and will be swapped out to *far memory*. Swap in/out interface is exposed by RapidSwap's *storage backend*.

User applications could request RapidSwap-managed virtual memory area by calling the mmap() system-call and providing the required memory size. To minimize the latency caused by selecting the wrong page and inducing its frequent swap-in/out to far memory, it precisely selects the victim page. *Swap handler* maintains 2 different page pool lists for pages: *active*, and *inactive*. Victim pages are selected from the *inactive* page pool. However, if the number of the required pages exceeds the number of reclaimable pages, victim pages will be selected from the *active* page pool. Active, or inactive classification is done in z-score based algorithm. The classification is transparently handled by the background kernel thread.

4.2.2 Storage Frontend

The main objective of the *storage frontend* is to expose *far memory* as a single storage device to the *swap handler*. The swap handler may request the storage frontend to swap in/out pages from/to the far memory. Storage frontend does not allocate space in the far memory until the relevant region is initially accessed. When the region is first accessed, the swap handler allocates 1 MiB-sized slab in the fastest among the available storage devices. Lazy-loaded slabs are initially allocated from the fastest storage devices among the available resource pool.

Hot/cold classification is done based on the access history. When a slab is accessed, the *I/O handler* turns on the *accessed bit*. *Classification* task will run every second and check whether the *accessed bit* is *true* in every single slab. If then, *classification* task marks the slab as *hot* and turns off the bit. If a specific slab is found accessed every time, it will continuously stay as *hot* slab. If the slab hasn't been accessed, it is considered *warm*. *Warm* slabs are given grace periods of T seconds and stay in the current storage tier. Every slab contains the metadata to record the time since the last access. If a slab hadn't been accessed for T seconds, *classification* task marks it as *cold*. *Hot* or *cold* slabs are subject to migration.

Migration task is responsible for the relocation of all slabs. It checks the list of all allocated slabs and migrates accordingly. If a slab is marked *hot*, it will be moved to the fastest among the storage hierarchy. On the other hand, if a slab is considered *cold*, *migration* task will move this slab to the slower and cheaper device. During the slab relocation, *migration* task will 1) read the target slab from the *source* device, 2) de-allocate the slab from the old device, 3) allocate a new slab to the target device, 4) lock the memory for synchronization

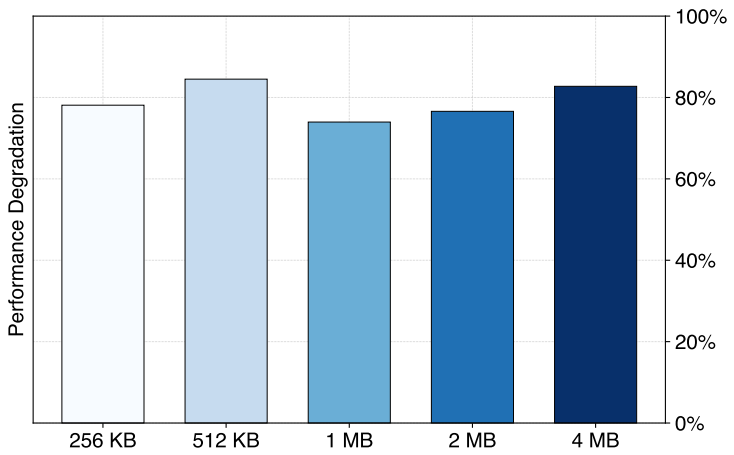


Figure 4.3: Average performance degradation by different slab sizes during when the local memory is limited to 30% of the benchmarks' RSS.

and relocate the memory contents, and finally 5) fix the relevant metadata and unlock the memory region. Slabs that are already at the top or bottom of the hierarchy will stay as long as it is hot/cold.

Slab Size

Slab size in RapidSwap determines the granularity for the allocation and migration. Smaller size is suitable for fine-grained control, but incurs more metadata overhead than the larger slab size. To select the best value, we have evaluated multiple slab sizes. In a strict environment where the local memory size is limited to 30% of the benchmarks' resident set size (RSS), 1MiB size allows us the best performance. Our observation is drawn in Figure 4.3.

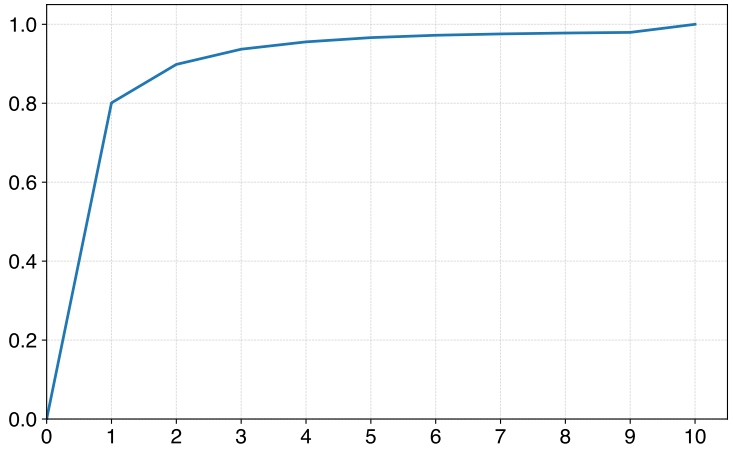


Figure 4.4: Miss-ratio curve as represented in CDF.

Migration Policy

We have selected the value T as 5 since the probability that the accessed page will be accessed again within 10 seconds is 98% for the workload that heavily accesses the memory. As RapidSwap retains intermediate *warm* temperature, it will take 10 seconds for a slab to become *cold*. Figure 4.4 shows miss-ratio curve in our experimental settings.

4.2.3 Storage Backend

The storage backend serves as an abstraction layer between the storage frontend and the physical storage devices. A single backend exists for each device. Storage backends must implement the following 4 APIs: *alloc()*, *dealloc()*, *read()*, and *write()*. *Storage backend* does not keep any metadata regarding the allocation or slab usage – *storage frontend* is responsible for keeping all the relevant records. Based on the device characteristics, *read* or *write* is implemented in either load/store (memcpy) or block-based approach.

Chapter 5

Results

5.1 Experimental Setup

Workloads' evaluations involve two entities – YCSB *server* and *client*. YCSB server node has Intel Xeon Silver 4215R CPU with 64GB DRAM. The node contains 1x960GB Intel 905P Optane NVMe PCIe (SSD) and 1x128GB Intel Optane Persistent Memory 200 Series (PMEM). The server runs Ubuntu Server 20.04 with modified QEMU-KVM v4.2 to use our custom *swap handler*. A virtual machine running redis database requests and allocates memory from the *swap handler* through custom QEMU. An external node runs YCSB *client* to generate queries for evaluations. *Persistent memory* and *PCIe SSD* comprise a two-tier far-memory hierarchy for RapidSwap. We evaluate 6 different core workloads provided by the Yahoo! Cloud Serving Benchmark (YCSB) [3].

Table 5.1 summarizes the evaluated workloads and their characteristics. Benchmark types `A`, `B`, `C`, and `F` are evaluated when their access patterns were given as both *zipfian* and *uniform*. For *zipfian* pattern, 80% of total accesses

Workload Type	Request Distribution	Details
A: Update Heavy	Zipfian / Uniform	50% Reads, 50% Writes
B: Read Mostly	Zipfian / Uniform	95% Reads, 5% Writes
C: Read Only	Zipfian / Uniform	100% Reads
D: Read Latest	Latest[1]	Read from the fresh data
E: Short Ranges	Zipfian & Uniform[2]	95% Scans, 5% Writes
F: Read-Modify-Write	Zipfian / Uniform	50% Reads, 50% Read-Modify-Writes

Table 5.1: YCSB core workloads. [3]

Phase	Properties	Value	Details
Loading	recordcount	1000000	Total number of records to insert in the dataset
Transactions	target	100000	Target throughput (operations/sec)
	threadcount	8	Number of client threads
	operationcount	1000000	Number of operations to execute
	operationcount[3]	50000	

Table 5.2: YCSB client parameters for evaluation.

will be intensively made on the 20% of total data. Workload `D` predominantly reads from freshly inserted data that is not physically contiguous. Workload `E` mostly scans the entire database since the initiating key is selected in *zipfian* and the length of scan is chosen by *uniform* distribution.

Table 5.2 shows our parameters to run the workloads with YCSB clients. YCSB workloads are separated into 2 phases, *loading* and *transactions*. In the *loading* phase, the YCSB client will prepare the redis database that is active in the VM by inserting records. When the database is ready in the *transactions*

[1]Requests lately inserted data

[2]Scan starting key selected as **zipfian**, scanning done in **uniform** distribution

[3]For workload E

phase, we begin the evaluation. The performance of RapidSwap is evaluated by measuring the query response latency as reported by YCSB client. Memory scarcity is simulated by artificially limiting the YCSB benchmark to a certain percentage of the benchmark's overall maximum memory requirements (resident set size, RSS). RapidSwap is compared to vanilla Linux paging to the NVMe PCIe SSD, RackMem [14] with an NVMe PCIe SSD Backend, and compressed DRAM to mock the prior work by Google [15].

5.2 RapidSwap Performance

5.2.1 Degradation over DRAM

Figure 5.1 and 5.2 presents the performance degradation of various far memory implementations with local memory limits 80, 70, 60, and 50 percent. As the amount of data kept in DRAM is reduced, all implementations experience higher performance degradations. In the average case, Linux vanilla paging to an SSD performs worst, followed by RackMem paging to the same SSD (thanks to its optimized page fault handling), and then *zswap*. RapidSwap outperforms the other approaches in almost all configurations.

Performance degradation of workloads `A`, `B`, `C`, and `F` are presented in Figure 5.1. The left half of the figure represents *zipfian* distribution (zip) where the right half stands for *uniform* distribution was given to the same workloads (uni). As expected, the distribution of the accesses has a significant effect on performance. The uniform access distribution of each workload causes significant slowdowns in restricted DRAM scenarios. However, even under these strict conditions, RapidSwap manages to maintain DRAM performance when the local memory is allowed more than 70% in almost all scenarios except workload `F`, where it shows degradation of less than 5%. The performance penalty induced

zipfian	Vanilla+SSD	RackMem+SSD	Compressed DRAM	RapidSwap
Local: 50%	25%	22%	19%	8%
Local: 60%	10%	9%	8%	2%
Local: 70%	7%	0%	3%	0%
Local: 80%	4%	0%	2%	0%

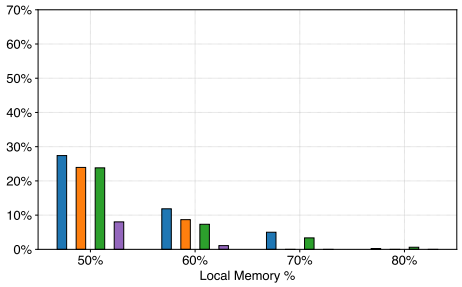
uniform	Vanilla+SSD	RackMem+SSD	Compressed DRAM	RapidSwap
Local: 50%	51%	49%	41%	22%
Local: 60%	29%	24%	18%	10%
Local: 70%	16%	2%	6%	1%
Local: 80%	13%	3%	4%	1%

Table 5.3: Average performance degradation when workloads are executed with zipfian and uniform distributions under different local memory sizes.

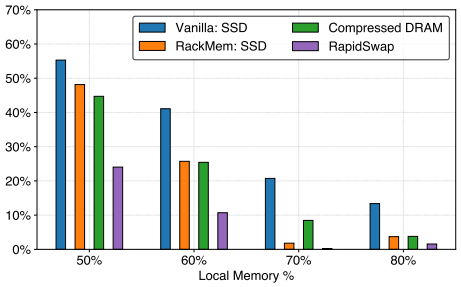
by RapidSwap in *zipfian* distribution is less than 13% even at 50% DRAM.

Figure 5.2 shows the performance degradation of workloads `D`, and `E`. These workloads retain special access patterns. Workload `D` simulates user status updates and yields minimal degradation for all designs in every configuration. The degradation patterns resemble similar trends shown in *zipfian* distributions. Threaded conversations scenario is represented by workload `E`, where it accesses most of the database and presents a larger working set. Workload `E` acts as *uniform* distributions. In the rest of the paper, we group these two workloads as *zipfian* or *uniform*.

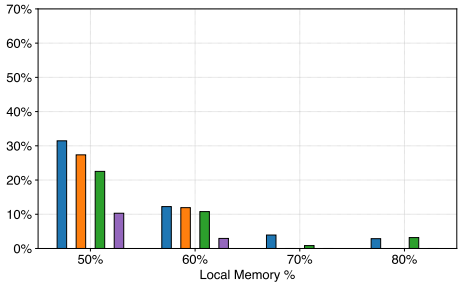
As shown in Table 5.3, RapidSwap's average performance degradation in *zipfian*+`D` scenarios is only 8% and 2% at 50% and 60% DRAM, whereas compressed DRAM shows 19% and 8% under the same conditions. In *uniform*+`E` scenarios, RapidSwap only suffers 23% and 10% slow down when the local memory limit is 50% and 60%. On the other hand, compressed DRAM presents 41% and 10% degradation.



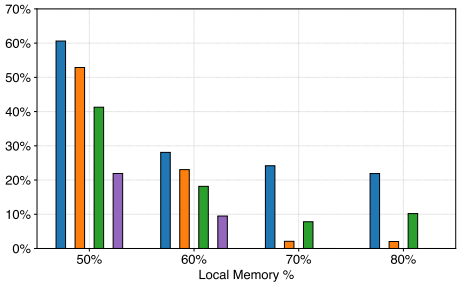
(a) YCSB A: Update Heavy (zip)



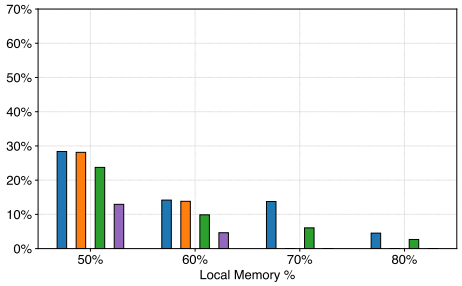
(b) YCSB A: Update Heavy (uni)



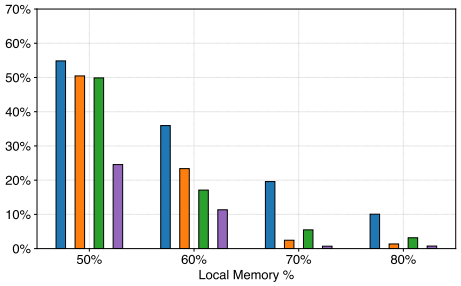
(c) YCSB B: Read Mostly (zip)



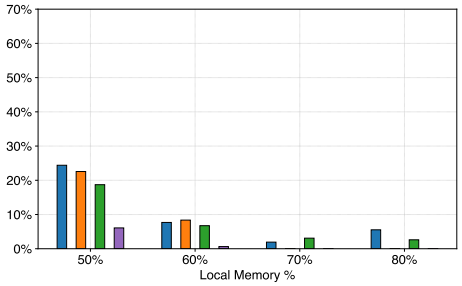
(d) YCSB B: Read Mostly (uni)



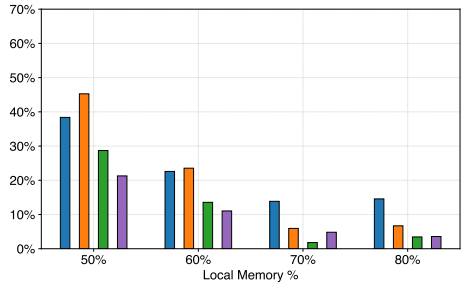
(e) YCSB C: Read Only (zip)



(f) YCSB C: Read Only (uni)

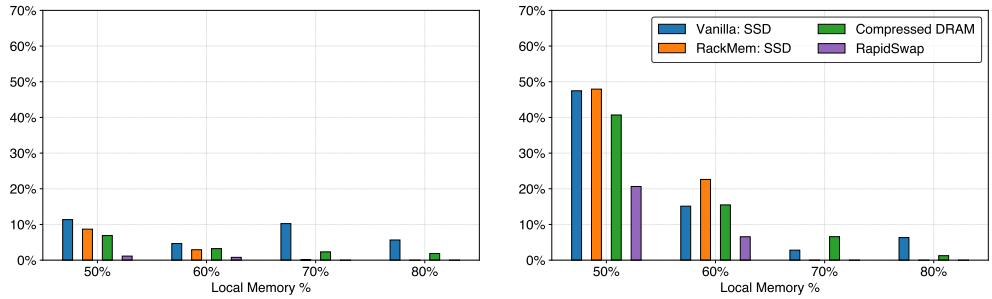


(g) YCSB F: Read-Modify-Write (zip)



(h) YCSB F: Read-Modify-Write (uni)

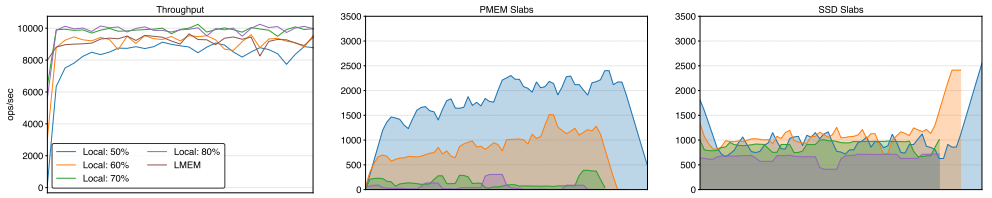
Figure 5.1: Performance degradation of RapidSwap and prior works over DRAM.



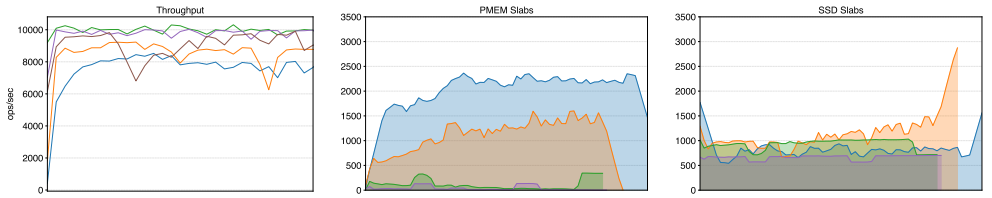
(a) YCSB D: Read Latest

(b) YCSB E: Short Ranges

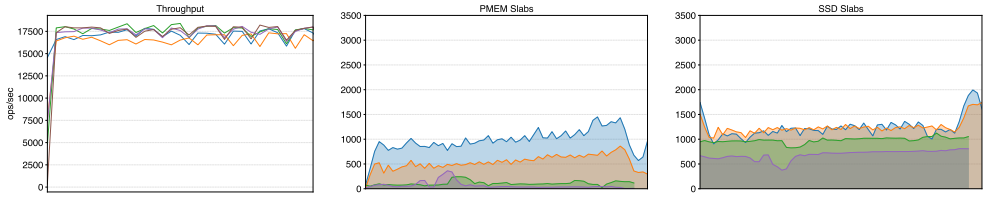
Figure 5.2: Performance degradation of RapidSwap and prior works over DRAM (continued).



(a) YCSB A: Update Heavy (zipfian)



(b) YCSB A: Update Heavy (uniform)



(c) YCSB D: Read Latest

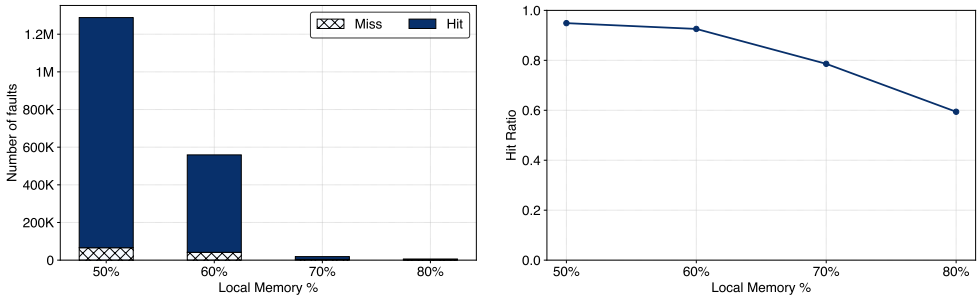
Figure 5.3: Throughput and the number of slabs resident in PMEM and SSD.

5.2.2 Tiered Storage Utilization

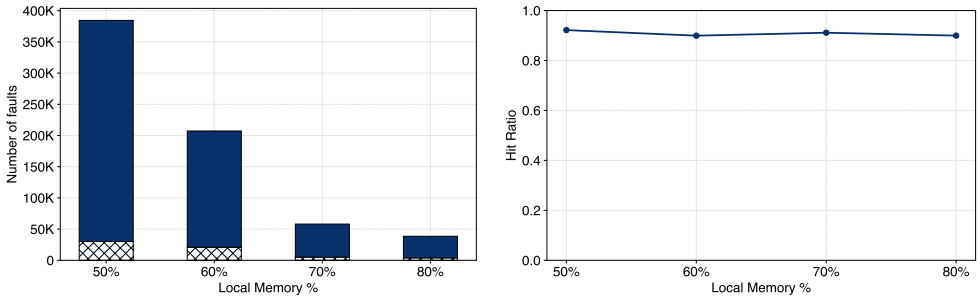
Figure 5.3 shows throughput and the utilization of the two storage tiers PMEM and SSD used in the experiments. Each different color represents the percentage of local memory over RSS of each benchmark, including `LMEM` where a sufficient amount of local memory is given during the execution. The leftmost figure in every row depicts the throughput for the first 30 seconds of execution of each workload. The other two figures present the number of slabs on each device over execution runtime.

Regardless of distributions, all workloads show near-DRAM throughput when the local memory is given more than 70%. Throughput drops when the local memory is limited for less than 60%. We observe that RapidSwap migrates most pages to the slowest storage level. Spikes in the second half of the `PMEM` graph indicate periods when slabs are brought in from slower storage due to page faults. This is due to the frequent migration between the storage tiers. As a consequence, the number of slabs over execution time in `PMEM` is larger than the number of slabs resident in SSD. Workloads with *uniform* distributions show similar trends.

Workload `D` with its latest access pattern shows a different picture. Limiting the local memory does not impact the throughput of the workload. As the workload only accesses the recently inserted data, the size of the working set is relatively small for workload `D`, therefore keeping smaller slabs in `PMEM` allows robust and cheaper operations.



(a) YCSB A: Update Heavy (zipfian)



(b) YCSB D: Read Latest

Figure 5.4: Number of hits/misses and hit ratio of each workload by RapidSwap.

5.2.3 Hit/Miss Analysis

The number of hits/misses and its ratio is shown in Figure 5.4. We consider slab *hit* if a requested data is available at our `PMEM` at the time of access. As expected, the number of page faults handled by RapidSwap dramatically increases as local memory became scarce. In all workloads except `D` shows similar trends as workload `A`. The hit ratio drops when the responsibility of DRAM increases as the page is mostly retained by DRAM and the request to the far memory rarely occurs. Hit ratio of workload `D` stays stable (90%-92%) in all local memory settings. More details are described in Table 5.4.

zipfian	Local: 50%	Local: 60%	Local: 70%	Local: 80%
Workload A	95%	93%	79%	59%
Workload B	94%	93%	78%	56%
Workload C	94%	93%	82%	60%
Workload D	92%	90%	91%	90%
Workload F	94%	92%	77%	56%

uniform	Local: 50%	Local: 60%	Local: 70%	Local: 80%
Workload A	96%	95%	75%	50%
Workload B	95%	95%	83%	61%
Workload C	96%	94%	80%	66%
Workload E	97%	92%	80%	64%
Workload F	95%	94%	79%	54%

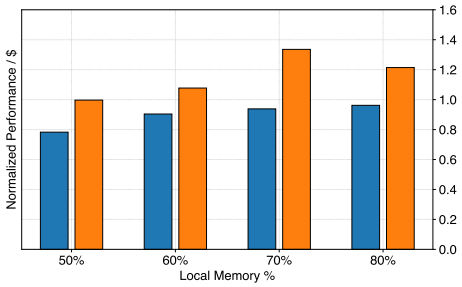
Table 5.4: Hit ratio by local memory limits of various YCSB workloads.

Storage Type	Baseline Model	$ per GB	Reference
Server DRAM	M386A8K40BM1-CPB (64 GB, LRDIMM)	9.36	[5]
Persistent Memory	NMB1XXD128GPSU4 (128 GB, DDR-T)	3.59	[19]
PCIe SSD	SSDPED1D015TAX1	1.66	[23]

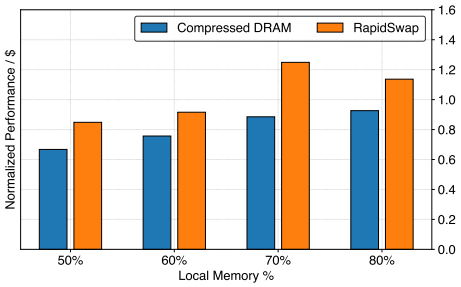
Table 5.5: Baseline storage prices.

5.3 Cost of Storage Tier

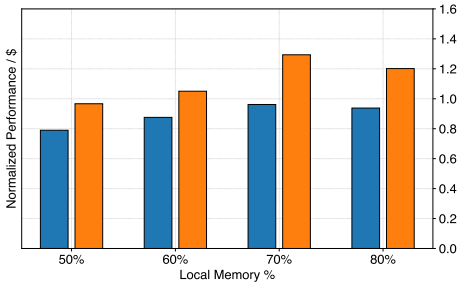
To analyze and compare the benefits of RapidSwap regarding the cost of the entire storage tier (DRAM, PMEM, and SSD), we surveyed the current market price of the different storage backends. Table 5.5 shows our findings. The cost is computed by first measuring the peak utilization (number of slabs) in the different storage tiers. Then, we multiply this peak utilization by the cost of the respective device to obtain the cost of the storage tiers. The total cost is obtained by adding the cost of the allocated DRAM.



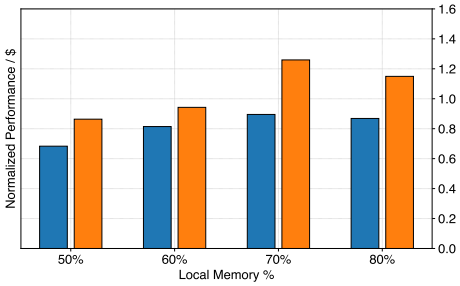
(a) YCSB A: Update Heavy (zip)



(b) YCSB A: Update Heavy (uni)



(c) YCSB B: Read Mostly (zip)



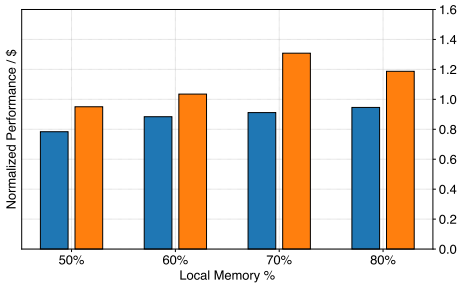
(d) YCSB B: Read Mostly (uni)

Figure 5.5: Cost-effectiveness of RapidSwap and Compressed DRAM.

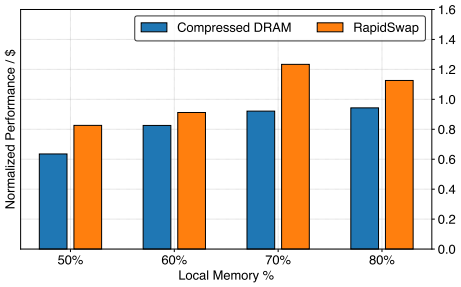
5.4 Cost-effectiveness

While the total cost of the storage tier is an important indicator, it does not consider the cost incurred by performance degradation. A more sensible metric is thus the *cost-effectiveness*, i.e., performance per cost.

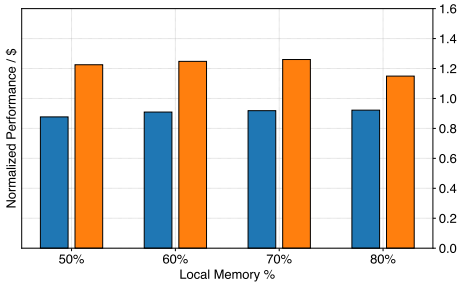
Figures 5.5 and 5.6 plot the cost-effectiveness of zswap and RapidSwap relative to a DRAM-only solution. The first observation is that RapidSwap achieves significantly better cost-effectiveness than zswap for all workloads and all configurations. Compared to DRAM, RapidSwap achieves an up to 40% higher cost-effectiveness with 70% of the data kept in DRAM and 30% paged out. As the amount of DRAM is reduced, workloads experience higher performance degradation and require larger amounts of storage.



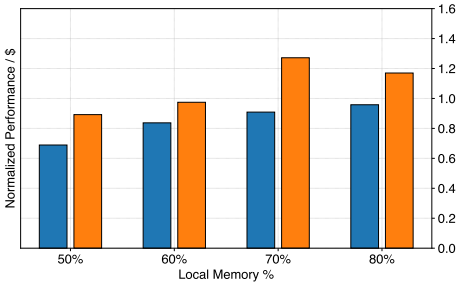
(a) YCSB C: Read Only (zip)



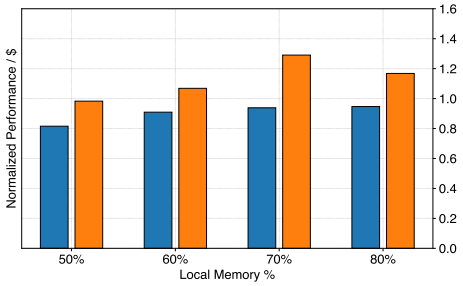
(b) YCSB C: Read Only (uni)



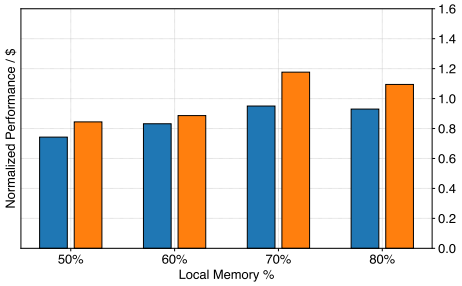
(c) YCSB D: Read Latest



(d) YCSB E: Short Ranges



(e) YCSB F: Read-Modify-Write (zip)



(f) YCSB F: Read-Modify-Write (uni)

Figure 5.6: Cost-effectiveness of RapidSwap and Compressed DRAM (continued).

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Motivated by the broad availability of novel storage technologies and the shortcomings of existing approaches to resource overcommitment, we have presented RapidSwap, an efficient hierarchical far memory implementation that is built for diverse storage tiers composed of faster and slower devices. Paging only to local devices, RapidSwap does not extend the failure domain such as disaggregated memory approaches, and its awareness of the storage hierarchy allows it to significantly outperform other techniques that swap out data locally.

Evaluated with a system equipped with Intel Optane memory and the Yahoo! Cloud Serving Benchmark, RapidSwap achieves a 30% improvement in cost-effectiveness at 70% local memory. RapidSwap demonstrates that proper management of new memory technologies can yield significant cost savings in data centers.

6.2 Future Work

There are some potential improvements in this research. First, we must consider application-specific memory management for practical research contributions. There are a number of solutions suggested exploiting the miss-ratio curve. By automatically adjusting the variable T in *storage frontend* with the assistance of methods proposed by the research community, we will be able to achieve more efficiency fine-tune the figure. Second, when a slab that is in the cold storage is accessed, it incurs *cold* miss penalty. Leveraging *bloom filter* seems favorable in grouping and identifying the spacial locality. In this way, we hope to apply prefetching to reduce performance penalties. Lastly, we did not yet compare RapidSwap against the memory mode configuration of PMEM, a hardware black-box approach that offers similar functionality as RapidSwap, implemented in hardware with cache-line granularity.

Bibliography

[1] Aws | sap hana. https://aws.amazon.com/ko/sap/solutions/saphana/. (Accessed on 06/06/2021).

[2] Pricing - linux virtual machines — microsoft azure. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/. (Accessed on 06/06/2021).

[3] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[4] 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.

[5] Samsung m386a8k40bm1-cpb 64gb ddr4-2133 4rx4 lp ecc lrdimm server memory at amazon.com. https://www.amazon.com/Samsung-M386A8K40BM1-CPB-DDR4-2133-LRDIMM-Server/dp/B017A8FJEG. (Accessed on 05/29/2021).

[6] Sk hynix expects chip supply shortages in 2021 as fourth-quarter profit surges | reuters. https://www.reuters.com/article/us-sk-hynix-results-idUSKBN29X322. (Accessed on 06/13/2021).

[7] Dram spot prices up 60 percent due to chip shortage | sourcengine. https://www.sourcengine.com/blog/dram-spot-prices-hit-two-year-high-due-to-global-chip-shortage-2021-03-25. (Accessed on 06/13/2021).

[8] Ec2 on-demand instance pricing - amazon web services. https://aws.amazon.com/ko/ec2/pricing/on-demand/. Last Accessed on 2 May 2021.

[9] Sap hana planning guide | google cloud. https://cloud.google.com/solutions/sap/docs/sap-hana-planning-guide. (Accessed on 06/06/2021).

[10] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, Boston, MA, Mar. 2017. USENIX Association.

[11] H. Herodotou and E. Kakoulli. Automating distributed tiered storage management in cluster computing. *Proc. VLDB Endow.*, 13(1):43–56, Sept. 2019.

[12] Intel® optane™ ssd 905p series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/optane-ssd-9-series/optane-ssd-905p-series.html. Last Accessed on 3 May 2021.

[13] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, Renton, WA, July 2019. USENIX Association.

[14] C. Jo, H. Kim, H. Geng, and B. Egger. Rackmem: A tailored caching layer for rack scale computing. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, page 467–480, New York, NY, USA, 2020. Association for Computing Machinery.

[15] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 317–330, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Y. Lee, H. A. Maruf, M. Chowdhury, and K. G. Shin. Mitigating the performance-efficiency tradeoff in resilient memory disaggregation. *CoRR*, abs/1910.09727, 2019.

[17] Memory management - the linux kernel documentation. `https://www.kernel.org/doc/html/latest/admin-guide/mm/index.html`. (Accessed on 06/04/2021).

[18] zswap - the linux kernel documentation. `https://www.kernel.org/doc/html/latest/vm/zswap.html`. (Accessed on 06/04/2021).

[19] Intel corporation nmb1xxd128gpsu4 intel optane 200 128gb ddr-t persistent memory module. https://www.itosolutions.net/Intel-Optane-200-128GB-DDR-T-Persistent-Memory-p/nmb1xxd128gpsu4.htm. (Accessed on 05/29/2021).

[20] Redis. https://redis.io/. (Accessed on 06/06/2021).

[21] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, Oct. 2018. USENIX Association.

[22] Apache spark™ - unified analytics engine for big data. https://spark.apache.org/. (Accessed on 06/06/2021).

[23] Intel optane 905p 1.50 tb solid state drive - internal - pci express nvme (pci express nvme 3.0 x4) - newegg.com. https://www.newegg.com/intel-optane-ssd-905p-series-1-5tb/p/0D9-002V-003X1. (Accessed on 05/29/2021).

[24] Y. Tan, B. Wang, Z. Yan, Q. Deng, X. Chen, and D. Liu. Uimigrate: Adaptive data migration for hybrid non-volatile memory systems. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 860–865, 2019.

[25] Z. Wang, T. Li, H. Wang, A. Shao, Y. Bai, S. Cai, Z. Xu, and D. Wang. Craft: An erasure-coding-supported version of raft for reducing storage cost and network cost. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 297–308, Santa Clara, CA, Feb. 2020. USENIX Association.

[26] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, page 8, USA, 1999. USENIX Association.

[27] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, Feb. 2016. USENIX Association.

[28] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 452–465, New York, NY, USA, 2017. Association for Computing Machinery.

[29] S. Zheng, M. Hoseinzadeh, and S. Swanson. Ziggurat: A tiered file system for non-volatile main memories and disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 207–219, Boston, MA, Feb. 2019. USENIX Association.

요약

컴퓨팅 환경이 클라우드 환경을 중심으로 변화하고 있어 클라우드 제공자는 고객이 제공하는 워크로드를 수용하기 위한 다양한 문제에 직면하고 있다. 오늘날 응용 프로그램은 일반적으로 많은 양의 메인 메모리를 요구한다. 기존 노드 중심 아키텍처에서 DRAM을 사용하면 빠르게 데이터를 제공할 수 있다. 그러나, 물리적으로 DRAM을 일정 수준 이상 확장하는 것은 하드웨어 제한과 비용으로 인해 현실적으로 불가능하다. 본 논문에서는 DRAM에 가까운 성능을 제공하면서도 총 소유 비용을 상당히 낮추는 효율적 far memory인 RapidSwap을 제시하였다. RapidSwap은 데이터센터 환경에서 상변화 메모리 (phase-change memory; persistent memory)를 활용하며 어플리케이션의 메모리 접근 빈도를 추적하여 자주 접근되지 않는 메모리를 느리고 저렴한 저장장치로 이송하여 이를 달성한다. 여러 저명한 클라우드 벤치마크 시나리오로 평가한 결과, RapidSwap은 순수 DRAM 대비 약 20%의 운영 비용을 절감하며 약 30%의 비용 효율성을 지닌다. RapidSwap은 새로운 스토리지 기술을 정교하게 활용하면 클라우드 데이터 센터 환경에서 운영비용을 상당히 저감할 수 있다는 사실을 보인다.

주요어: Far Memory, 가상 메모리, 스왑 시스템, 클라우드 데이터센터, 운영체제
학번: 2019-22487